Watermarking machine learning models

Sofiane Lounici

Watermarking Machine Learning Models

Dissertation

submitted to

Sorbonne Université and EURECOM

in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Author:

Sofiane LOUNICI

Scheduled for defense on November 7th 2022, in front of a committee composed of:

Reviewers		
Prof.	**Emil LUPU**	Imperial College, London
Prof.	**Luigi MANCINI**	Sapienza University of Rome, Italy
Examiners		
Dr.	**Orhan ERMIŞ**	Institute of Science and Technology, Luxembourg
Prof.	**Pietro MICHIARDI**	EURECOM, France
Industrial Supervisor		
Dr.	**Slim TRABELSI**	SAP Security Research, France
Thesis Supervisor		
Prof.	**Melek ÖNEN**	EURECOM, France
Guest		
Dr.	**Marie PAINDAVOINE**	Inria Rennes, France

Tatouage numérique des modèles d'apprentissage automatique

Thèse

soumise à

Sorbonne Université et EURECOM

pour l'obtention du Grade de Docteur

présentée par:

Sofiane LOUNICI

Soutenance de thèse prévue le 7 Novembre 2022 devant le jury composé de:

Rapporteurs

Prof.	**Emil LUPU**	Imperial College, Londres
Prof.	**Luigi MANCINI**	Université de Rome « La Sapienza », Italie

Examinateurs

Dr.	**Orhan ERMIŞ**	Institut de Science et Technologie, Luxembourg
Prof.	**Pietro MICHIARDI**	EURECOM, France

Superviseur Industriel

Dr	**Slim TRABELSI**	Recherche Sécurité de SAP, France

Directrice de Thèse

Prof.	**Melek ÖNEN**	EURECOM, France

Invitée

Dr.	**Marie PAINDAVOINE**	Inria Rennes, France

To my beloved family

Abstract

The protection of the intellectual property of machine learning models appears to be increasingly necessary, given the investments and their impact on society. In this thesis, we propose to study the watermarking of machine learning models. We provide a state of the art on current watermarking techniques, and then complement it by considering watermarking beyond image classification tasks. We then define forging attacks against watermarking for model hosting platforms and present a new fairness-based watermarking technique. In addition, we propose an implementation of the presented techniques.

Abrégé

La protection de la propriété intellectuelle des modèles d'apprentissage automatique apparaît de plus en plus nécessaire, au vu des investissements et de leur impact sur la société. Dans cette thèse, nous proposons d'étudier le tatouage de modèles d'apprentissage automatique. Nous fournissons un état de l'art sur les techniques de tatouage actuelles, puis nous le complétons en considérant le tatouage de modèles au-delà des tâches de classification d'images. Nous définissons ensuite les attaques de contrefaçon contre le tatouage pour les plateformes d'hébergement de modèles, et nous présentons une nouvelle technique de tatouages par biais algorithmique. De plus, nous proposons une implémentation des techniques présentées.

Acknowledgements

I would like to express my gratitude to my supervisor Prof. Melek Önen for her support and her guidance. I learned a lot from our discussions and it was immensely valuable for me.

A more than special thanks go to all the jury members in my PhD committee, namely Emil Lupu, Luigi Mancini, Orhan Ermiş, Pietro Michardi, and Slim Trabelsi for having accepted to join in the jury.

I would like to thank all my co-authors, Marco, Carlo, Mohamed, Orhan, Dhia, Melek and Slim, to have been part of my publication.

Special thanks to my colleagues at the Security Research department SAP Labs France, in particular Marco for his help and his time, and my fellow SAP PhD students Feras, Piergorgio, Caelin and Niccolò for the interesting discussions.

Finally, I would like to thank my family, my parents for their unconditional support, and my wife Yuliya for her love and her understanding during these difficult times.

Contents

List of Figures

List of Tables

List of Publications

We have presented the following publications during this PhD study. Most of them have already been presented through different venues enumerated below:

Publications - Conference

1. **Sofiane Lounici**, Marco Rosa, Carlo Maria Negri, Slim Trabelsi, Melek Önen, "Optimizing Leak Detection in Open-Source Platforms with Machine Learning Techniques" [6], 7th International Conference on Information Systems Security and Privacy.

2. **Sofiane Lounici**, Mohamed Njeh, Orhan Ermis, Melek Önen, Slim Trabelsi, "Preventing Watermark Forging Attacks in a MLaaS Environment" [7], 18th International Conference on Security and Cryptography.

3. **Sofiane Lounici**, Mohamed Njeh, Orhan Ermis, Melek Önen, Slim Trabelsi, "Yes We can: Watermarking Machine Learning Models beyond Classification" [8], 34th IEEE Computer Security Foundations Symposium 2021.

4. **Sofiane Lounici**, Melek Önen, Orhan Ermis, Slim Trabelsi, "BlindSpot: Watermarking through Fairness" [9], 10th ACM Workshop on Information Hiding and Multimedia Security 2022.

Research Report

1. **Sofiane Lounici**, Dhia Farrah, Melek Önen, Slim Trabelsi, "A Unified Library for Watermarking Machine Learning in Production" [10], Research Report.

Chapter 1

Introduction

1.1 Machine Learning lifecycle

With the rise of Big Data technologies, machine learning (ML) models are becoming widespread in the industry, impacting domains such as healthcare [11], finance [12] or art [13] to cite a few. The biggest companies in terms of market capitalization such as Apple, Microsoft, Google or Amazon, implement machine learning algorithms in every aspect of their products, to automate tasks or to improve the customers' experience. Despite the undeniable advantages with respect to time and money provided by such algorithms, their development is not cost-free and represent a non negligible investment.

Indeed, the development of ML algorithms is composed of several steps, including data collection, data preprocessing, model's architecture design, model training or model monitoring. Each of these steps represents a cost, ranging from several thousands of dollars to multi-million dollars projects [14]. In particular, the recent research advances for building bigger neural network-based models increased the cost of obtaining a state-of-the-art model for domains such as text summarization [15] or image generation [13]. This cost-benefit trade-off between risky investments and potential future gains in a quickly evolving and highly competitive market, implies that models need to be considered as assets, offering a competitive advantage to the model owner. As such, in order to face losses which could impact their businesses, model owners are required to investigate potential security threats associated with their models.

1.2 Security for Machine Learning

Similar to any software system, a model can undergo a set of attacks, either during the training or after its deployment. First, if the attacker has gained access (partial or total) to the training procedure, it is possible to modify the behavior of the model by corrupting the labels of the training data for instance, leading to a destruction of the resulting model. More discretely, the attacker can insert particularly crafted information to inject a *backdoor*, a hidden unwanted behavior. In the case where the attacker does not have an access to the training data, it is still possible to implement attacks. A popular

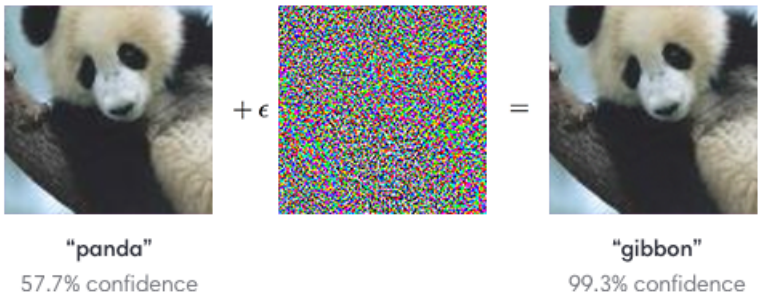


Figure 1.1: An adversarial input, overlaid on a typical image, can cause a classifier to miscategorize a panda as a gibbon [1].

attack is to compute adversarial examples, as shown in Figure 1.1, to trick the model and missclassify inputs.

Furthermore, since the model is the resulting consequence of the training process on a training data, it can be subject to algorithmic bias, creating unfair outcomes for certain types of inputs. Models exhibiting fairness bias in domain such as healthcare or justice, could lead to violating anti-discrimination laws.

Finally, when the model is deployed, there is still a risk that an attacker could gain knowledge of confidential or private information, by interacting with the model. For instance, the analysis of a model's behavior could lead to *membership inference attacks* (identifying if a given sample belongs to the original dataset) or *data reconstruction* attacks (obtaining the full original dataset).

1.3 Model Stealing

In this thesis, we particularly focus on a type of attacks on ML systems called *model stealing attacks*, consisting in acquiring a model without the consent of the original owner, for its own benefit. The motivations behind such attacks could be:

1. (i) a lack of computational power: By stealing a model, the attacker is bypassing previously introduced costs induced by the model development
2. a lack of knowledge: In highly competitive industries, the training process can be kept as a *trade secret* [16]
3. a lack of time: To obtain a model quickly in order to convince investors and shareholders.

Several strategies have been presented for stealing a model. A first option is to exploit the fact that the majority of ML projects rely on cloud providers, or Machine Learning as a Service (MLaaS) platforms, to manage and store their models. If the access to such platforms is leaked, so is the associated model. It has been noticed that open-source code collaborative platforms such as GitHub contains a worrying high number of credentials,

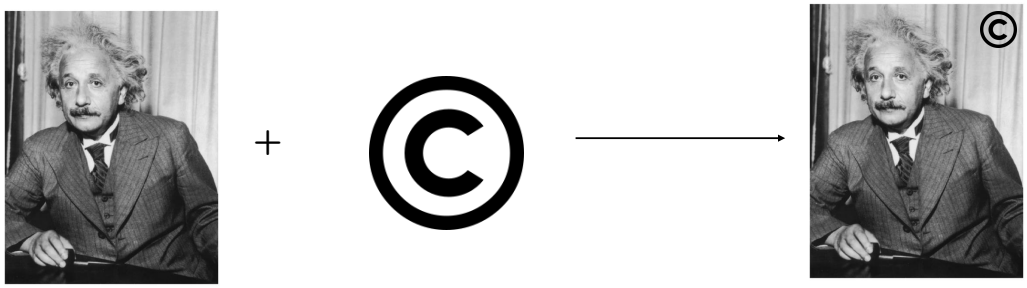


Figure 1.2: Example of visible image watermarking

passwords or API keys to the most popular MLaaS platforms such as Azure [17], Google Cloud [18] or AWS [19]. Therefore, an attacker can exploit these data leaks in order to obtain stored data, including ML models.

A second option is to take advantage of the fact employees often have a direct access to confidential information regarding the training process, the model's architecture and parameters, etc. By leveraging industrial espionage techniques, including theft of trade secrets, blackmail, bribery or technological surveillance, it is possible for an attacker to obtain a ML model.

Finally, recent works [20, 21], have shown that, even if an attacker has limited access to the model (for instance, with limited interaction through an API), it is possible to understand and clone the model, to obtain a surrogate version of it exhibiting similar performance to the original. This type of attack is very concerning for model owners, because it can be performed even with a small number of interactions and with scarce knowledge of the training data.

1.4 Watermarking

In the light of the danger posed by model stealing attacks, defenses have been developed to mitigate them. In particular, a concept called *watermarking* has been developed. The idea stems from image watermarking and steganography (Figure 1.2): in order to identify the ownership or the integrity of a content, such as a file, image, video or audio, the content's owner injects specific information into this content. The resulting *watermarked content* can be distributed, transmitted, stored while carrying ownership information. The content's owner can later on extract the watermark to prove its ownership. Watermarking has proved to be a reliable solution for IP protection, by demonstrating robustness against attacks, i.e., the watermark cannot be removed from the content without altering the original content.

Watermarking ML models consists in adding ownership information into the model (the *embedding* phase) during the training phase or before the deployment. For a given suspect model, the original model owner can verify the presence of the watermark (the *verification* phase) by either analyzing the suspect's behavior on a particularl set of

inputs, called *trigger set*, or by investigating the suspect's parameters and extract the watermark. Depending on the assumptions made with respect to the attacker abilities, its resources and its knowledge of the model's parameters and training data, a number of embedding and verification algorithms have developed [20, 22, 23], aiming to be robust against model stealing attacks.

1.5 Contributions

This thesis investigates how to protect machine learning models against model stealing attacks. To this purpose, we consider two approaches: mitigating data leaks on open-source code platforms and watermarking ML models. We list our contributions as follows:

1. We identify three model stealing strategies : (i) exploiting data leaks to gain unauthorized access to cloud-stored ML models (ii) leveraging insider threats so as to obtain associated training scripts or training data and (iii) implementing model extraction attacks to build a surrogate version of the model.

2. By studying how data leaks can lead to model stealing attacks, we present an approach to identify and mitigate them, in the context open-source platforms. We propose a machine learning solution for identifying data leaks with low false positive rate.

3. We present watermarking as a defense strategy for intellectual property (IP) protection, by proposing a literature review of embedding and verification algorithms. Specifically, we propose a classification between black-box and white-box algorithms.

4. We extend the current state-of-the-art of watermarking by introducing watermarking beyond image classification, namely for regression, machine translation and reinforcement learning models.

5. Additionally, we show how watermarking can be implemented for IP protection in the context of content hosting platforms, while developing the idea of watermark forging attacks.

6. We provide and present our open-source library, intended to implement and share current knowledge on the current state-of-art, designed for watermarking in production.

7. Finally, we introduce a novel fairness-based algorithm for watermarking, entitled BlindSpot, to watermark machine learning models in a black-box environment.

1.6 Organisation

The remaining of this thesis is organized as follows:

In Chapter 2, we introduce machine learning theory required for the understanding of our work: regression models and neural networks. In Chapter 3, we introduce the problem of model stealing attacks, presenting the main challenges for ML security, how can such attacks can be implemented and what are the principal motivations for an attacker. We investigate the problem of data leaks for ML security in Chapter 4.

In Chapter 5, we propose a literature review for watermarking, as a defense strategy against model stealing attacks. We extend this state-of-the-art by presenting:

1. in Chapter 6, a definition for watermarking beyond image classification tasks, adapted from the paper [8].
2. in Chapter 7, a description of watermark forging attacks in the context of MLaaS platforms, adapted from the paper [7]
3. in Chapter 8, a library for implementing watermarking into production, adapted from the open-source library [24]
4. in Chapter 9, a novel fairness-based watermarking algorithm called BlindSpot, adapted from the paper [9]

Finally in Chapter 10, we conclude with the results of this dissertation and we discuss future research avenues.

Chapter 2

Machine Learning and Security Aspects

In this first chapter, we discuss several key concepts which constitutes the background of the thesis regarding machine learning algorithms and their security.

2.1 Introduction

According to the Oxford dictionary [25], machine learning (ML) is defined as *a type of artificial intelligence in which computers use huge amounts of data to learn how to do tasks rather than being programmed to do them.* Data might be composed of images or videos (denoted as *computer vision* tasks), text (called *Natural language Processing* or NLP tasks) or simply numerical features. In other words, machine learning algorithms identify relevant patterns in the available data for predictive purposes. Such algorithms can be applied to a large variety of domains, like face recognition system, voice recognition tools, e-mail spams filtering or driverless cars to name a few. It is possible to classify machine learning algorithms into three categories: supervised, unsupervised and reinforcement-based learning.

First, we define **supervised machine learning** whose goal is to build a predictive model, by learning the mapping between inputs and outputs based on a dataset. The dataset, called training data, is composed of (input , output) pair examples. Instance of tasks which could be solved with supervised machine learning could include classification tasks such a image recognition tasks [26] or hate speech identification [27], as well as regression tasks such bank loan prediction [28].

Next, we define **unsupervised machine learning** as a set of techniques whose goal is to identify patterns in data. As opposed to supervised algorithms, unsupervised algorithms work with non-annotated data, and include clustering algorithms like k-means [29] or principal component analysis (PCA) [30]. Tasks such as recommendation systems [31] (i.e. recommending similar content to users) or anomaly detection [32] can be solved with unsupervised techniques.

Finally we define **reinforcement learning** as a family of algorithms where models are trained by trial and error (by receiving virtual "rewards" or "punishments"). Such algorithms could be implemented to solve game theory problems [33] or automomous driving [34]

2.2 Regression models

In this section, we define regression models. The goal of regression models consist of identifying a relation between an input $\mathbf{X} = \{x_1, \cdots, x_k\}$ (x_i is defined as a *feature* of $\mathbf{X}$) and a variable of interest $\mathbf{y}$.

Definition (Regression model)**.** *We define a regression model $h_\theta(.)$ as the following:*

$$y = h_\theta(\boldsymbol{X}) + \epsilon \tag{2.1}$$

$$\epsilon = \mathcal{N}(0, \sigma^2) \tag{2.2}$$

where θ are the parameters of the regression model and ϵ an error term following a Gaussian distribution.

The regression can be linear in the parameters ($h_\theta(x) = \theta^\intercal X$), or non linear. To assess the performance of a regression model over a dataset of n instances $\mathcal{D}_{test} = \{(X^{(1)}, y^{(1)}), \cdots, (X^{(n)}, y^{(n)})\}$, we consider two commonly used metrics in this thesis, namely:

- The Root Mean Square Error (RMSE), defined as follows :

$$RMSE(\mathcal{D}_{test}, h_\theta) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\theta^\intercal X^{(i)} - y^{(i)})^2} \tag{2.3}$$

- The Mean Absolute Percentage Error (MAPE), defined as follows :

$$MAPE(\mathcal{D}_{test}, h_\theta) = \frac{100}{n}\sum_{i=1}^{n}\frac{|y^{(i)} - \theta^\intercal X^{(i)}|}{y^{(i)}} \tag{2.4}$$

In the case of linear models, the parameters $\hat{\theta}$ minimizing the mean squared error function can be computed directly:

$$\hat{\theta} = (\mathbf{X}^\intercal\mathbf{X})^{-1}\mathbf{X}^\intercal\mathbf{y} \tag{2.5}$$

Otherwise, several algorithms [35, 36] have been developed in order to approximate θ, given a training dataset $\mathcal{D}_{train}$.

2.3 Neural Networks

In this section, we are describing a particular type of supervised machine learning algorithm called *artificial neural networks*.

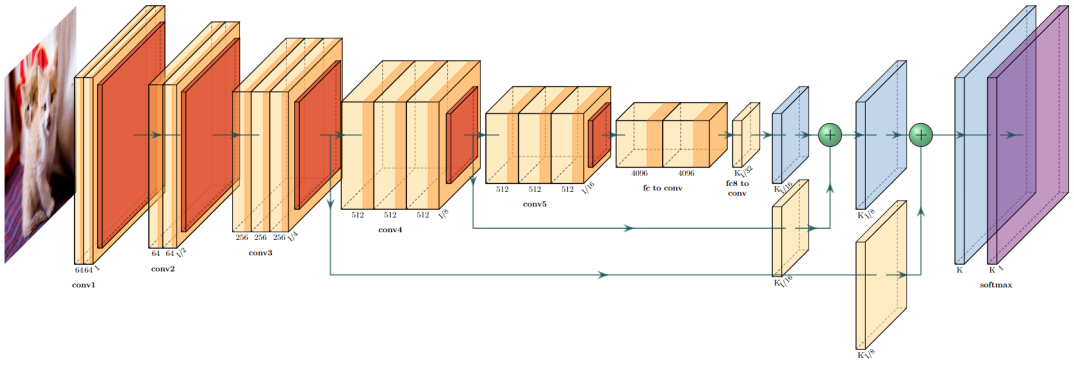


Figure 2.1: Architecture of a neural network

Overview

We define *artificial neural networks* (also called neural networks (NN) or multi-layer perceptron (MLP)) as computing systems, composed of units called neurons performing mathematical functions, as visually represented in Figure 2.1. Neurons are organized in layers, each layer performing different operations on their inputs. Generally, the first layer is denoted as *input layer* and the last layer as *output layer*. The l^{th} hidden layer takes as input a vector X^l and outputs a vector y^k. Hidden layers are stacked, meaning that the outputs of the l^{th} layer is the input of the $(l+1)^{th}$ hidden layer (to the exception of the input and output layers). Intermediate layers are named hidden layers and can be defined as a combination of an *activation function* $a^{(l)}(.)$ and a weight matrix $\mathbf{w}^{(l)}$. The output of the l^{th} layer can be written in the following form:

$$\mathbf{z}^{(l)} = \mathbf{w}^{(l)\mathsf{T}} \cdot \mathbf{a}^{(l-1)}$$
$$\mathbf{y}^{(l)} = a^{(l)}(\mathbf{z}^{(l)})$$

More generally, we denote as θ the parameters of the model (weights, activations functions, etc.). A model M_θ is a function mapping an input space $\mathbb{R}^m$ to an output space $\mathbb{R}^K$.

$$M_\theta : \mathbb{R}^m \rightarrow \mathbb{R}^K \tag{2.6}$$

The probability that input x belongs to class $i \in \{0, K\}$ is defined as $argmax(y)$ where K is the number of class labels.

Convolutional layers

A convolutional layer is a type of hidden layers, relying on the mathematical operation of convolution, visually represented in Figure 2.2 and particularly useful for image processing tasks. We denote the convolution operation $(. * .)$ for a layer l between an input $x^{(l)}$ and convolutional layer kernel W_c as:

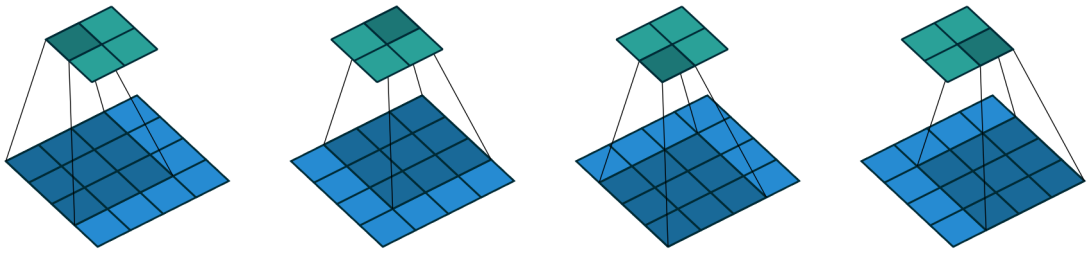


Figure 2.2: Schematic representation of convolving input of size 4×4 (blue) with filters of size 3 × 3, output is 2 × 2 (cyan)

$$y^{(l)} = W_c * x^{(l)} \tag{2.7}$$

Activation functions

Activation functions allow the neural network to adopt a more complex behavior by introducing non-linearity. A classic activation function for the output layer is the sigmoid function:

$$a(\mathbf{x}) = \frac{1}{1 + exp(-\mathbf{x}))} \tag{2.8}$$

Regarding the activations for hidden layers, different functions are considered, such as the hyperbolic tangent, or the Rectified Linear Unit (ReLu):

$$ReLu(\mathbf{x}) = max(0, \mathbf{x})) \tag{2.9}$$

2.3.1 Gradient Based Learning

The training phase of a neural network is the process of learning the underlying relationship between n data inputs, denoted $\mathbf{x} = \{x_1, ..., x_n\}$, and data labels, denoted $\mathbf{y} = \{y_1, ..., y_n\}$ present in the training data. Often, we refer to the joint input-label set as the training dataset $\mathcal{D}_{train}$. The goal of the training phase is to compute the parameters θ by minimizing the prediction error, i.e. minimizing the error between the prediction through the model $\hat{y} = M_\theta(x)$ and the ground truth data labels $\mathbf{y}$. The function quantifying this error is defined as *loss function* or *criterion* denoted $\mathcal{L}(.)$. For classification problems, a common loss function is the *Categorical Cross Entropy Loss* function, defined as follows for a single data sample (x_i, y_i):

$$\mathcal{L}(y_i, M_\theta(x_i)) = -\sum_{k \in K} y_k \cdot log(M_\theta(x_k))$$

To update θ based on the loss function, algorithms such as *Stochastic Gradient Descent* (SGD) are used where a parameter update is performed on a batch B of training samples. For a given training step t, defined as *epoch*, the parameter θ is updated:

$$\theta^{t+1} = \theta^t - \frac{\eta}{|B|} \sum_{i \in B} \nabla \mathcal{L}(\theta)$$

with $\nabla \mathcal{L}(\theta)$ denotes the gradient of the cost function with respect to θ, $(x^i, y^i))$ the i^{th} training sample in the batch B, η the learning rate and $|B|$ as the size of the batch. More generally, we denote the parameters of the training besides θ as hyperparameters, such as the number of layers of the network, the choice of loss function or the choice of the *learning rate*, used in the SGD algorithm.

During the training, randomly selected neurons can be ignored, or *dropped out*, in order to improve the ability of the model to generalize and to reduce overfitting. We denote the *dropout rate* and the percentage of dropped out neurons during the training phase.

2.4 Natural language processing tasks

In this thesis, we mainly explore neural networks for image classification and natural language processing (NLP) tasks. The latter correspond to a diversity of tasks, including language modeling [37], sentiment analysis [38] or question answering [39], for instance. We first present a particular method for text representation designed for text classification denoted Continuous Bag of Words (CBOW), before presenting a particular NLP task called *Machine Translation task*.

2.4.1 CBOW representation

The core of NLP tasks is to obtain a vector representation of the input text. Once this representation is obtained, it is possible to perform to aforementioned task (for instance, by training a linear classifier on the representations). We consider the text classification task, where the goal is to associate to a input sentence, composed of n-gram $\{x_1, x_2, ..., x_N\}$, an output class $j \in 1...K$. A n-gram is a n-character slices of a word ; the word `GRAM` would have the following 2-grams: $\{\_G, GR, RA, AM, M\_\}$. The CBOW model tries to understand the context of words in order to build its representation, while in the same time it trains a linear classifier for text classification. We obtain the feature representations via a weight matrix $\mathbf{U}$ such that $h_i = \mathbf{U} \cdot x_i$. Then, we define y as the linear Bag-of-Words of the document, by averaging all the feature representations h_i:

$$y = \frac{1}{N} \sum_{i=1}^{N} h_i \tag{2.10}$$

y is the input of a hidden layer associated to a weight matrix $\mathbf{V}$, such that the output of the classifier z is $z = \mathbf{V} \cdot y$. We can compute the probability that a word vector belongs

to the j^{th} class as $p_j = \sigma(z_j)$, with $\sigma(z_j)$ being the softmax function:

$$\sigma(z_j) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{2.11}$$

Finally, the weight matrices **U** and **V** are computed by minimizing the negative log-likelihood of the probability distribution, using stochastic gradient descent, namely:

$$-\frac{1}{N}\sum_{k=1}^{N} y_k \cdot log\big(\sigma(V \cdot U \cdot w_i)\big) \tag{2.12}$$

In this thesis, we will use the notation $LCBOW(w)$ to describe the vector representation of the word w. When the CBOW representation is not mentioned, we consider a general *Encoder-Decoder* architecture, where the *encoder* computes the text representation and the *decoder* performs the downstream task. This type of architecture is mainly accepted as the state-of-the-art for NLP tasks, such as the transformer architecture [40].

2.4.2 Machine translation task

In this thesis, and in particular in Chapter 6, we consider different tasks besides text classification, and in particular machine translation. A *machine translation* (MT) task is the task of translating a sentence in a source language to a different language. We consider an encoder-decoder architecture, defined as follows:

Let $\mathbf{X} = (x_1 \dots x_k)$ be a sentence composed of k words from a source language S and $\mathbf{Y} = (y_1 \dots y_m)$ be a sentence composed of m words from a target language T. The goal of a MT model is to learn the mapping $X \rightarrow Y$. The MT model is an encoder-decoder model, where the input sentence **X** is encoded into vector $\mathbf{X}^*$ which further passes through several layers of the model, and results in an output vector $\mathbf{Y}^*$. $\mathbf{Y}^*$ is finally decoded into the output sentence **Y**. The performance of MT models is often evaluated through two metrics:

- the **BLEU** [41] score which is the result of a standard algorithm that compares machine translations with human translations.
- the **ROUGE** [42] score which is an evaluation metric used in automatic summarization and machine translation.

The two scores are defined as a number between 0 and 1, with the better the translation, the closer the score is to 1.

2.5 Machine Learning lifecycle

The development of neural networks, or more generally machine learning models, is a complex process. Therefore, common guidelines and procedures have been implemented to improve efficiency in the development, denoted as *machine learning life-cycle*. As shown in Figure 2.3, it is composed 7 main steps:

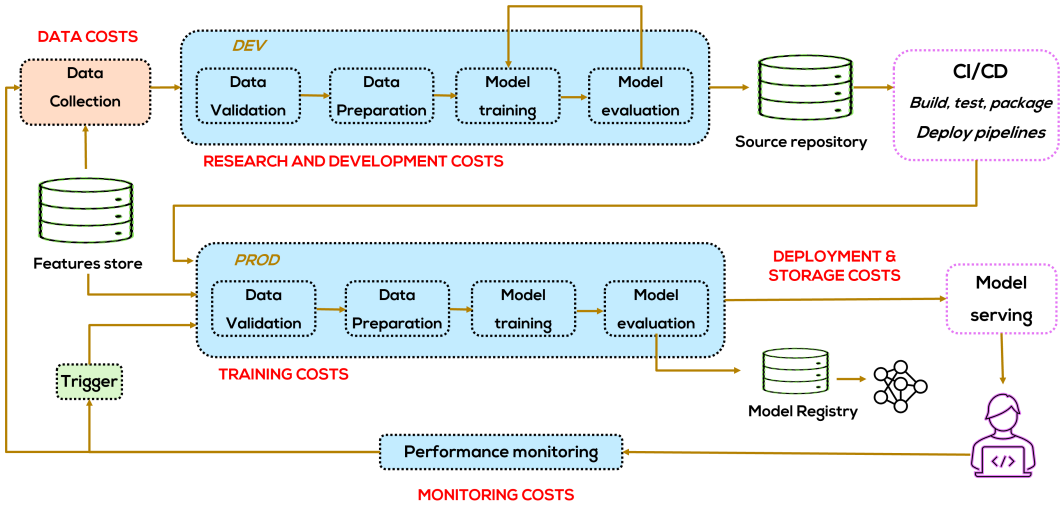


Figure 2.3: The Machine Learning Life-cycle

First, in order to train a model, raw data needs to be **collected** through different sources and stores. Then, raw data needs to be **processed**, annotated, cleaned or other data processing operations depending on the downstream task. The next three steps, the **research and development phase**, **training and evaluation phase**, are often conceived together: first, a model architecture, hyperparameters selection, as well as a training algorithm need to be defined. Then, the model is trained on the training data, then evaluated according to performance metrics such as accuracy or F1-score, on a test set. If the performance on the test set is not satisfactory, then the process re-starts at the research and development phase, to modify architecture or hyperparameters before re-training the model.

After the training and the validation of the performances, the model is **deployed** into production to perform predictions. Finally, the model is regularly **monitored** to observe its behavior in real-life situations and, if necessary if the model becomes *stale*, triggers re-training steps to update the model, with additional data for instance.

2.6 Security of Machine Learning

With the development of machine learning models for various applications, the concept of security of machine learning [43] has been proposed, in order to quantify potential risks associated with using a ML model. Indeed, due to the tremendous impact models can have on business, it has become a necessity to properly assess this risk for companies and model owner. In this section, we review three types of threats that model owners need to face when developing their models.

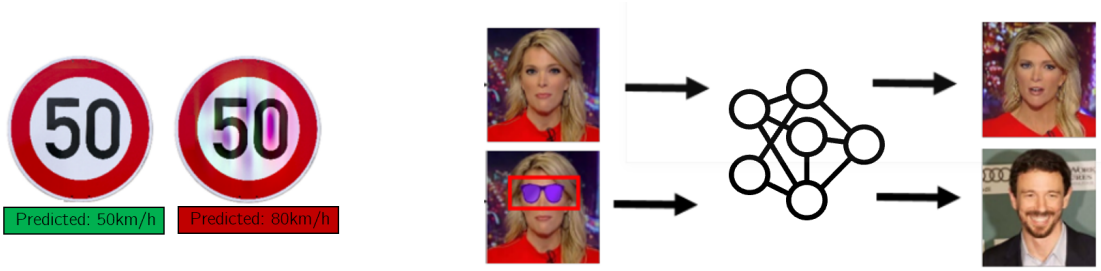


Figure 2.4: Example of adversarial attacks for traffic sign recognition [2] (on the left) and data poisoning attack for face recognition [3] (on the right))

2.6.1 Security threats

In addition to classic attacks such as software vulnerability (malwares [44]) or Denial of Service [45] (DoS) attacks, models can be subject to attacks perturbating their normal behavior. An instance of this is the case of *adversarial examples* [2] where a carefully crafted input can shift the prediction of a given model, as shown in Figure 2.4 where a slight input perturbation could lead to misclassify traffic signs. If the attacker has deeper access to the model, with the ability to modify the training data, then *data poisoning* [3] attacks can be implemented. By injecting malicious data to the original dataset, the attacker can either hinder the model performance or add an additional, hidden behavior acting as a backdoor. Figure 2.4 shows that a visual artifact such as sunglasses can trick a face recognition classifier. Backdoors could lead to massive security issues, for instance by granting administrator access to anyone bearing this particular visual artifact.

2.6.2 Algorithmic bias threats

The concept of algorithm (or fairness) bias [46] can refer to the errors in predictions leading to *unfair* situations, such as gender or racial discrimination. In other words, a *fair model* requires that "individuals do not experience differences in outcomes caused by factors that are outside their control, such as race or gender" [47]. The presence of algorithmic bias can be a consequence of how the training data was built (as reflection of the human bias) or directly in the design of the model itself. For instance, the Correctional Offender Management Profiling for Alternative Sanctions (COMPAS) [48] software used by U.S. courts to assess the likelihood of a defendant becoming a recidivist, has shown to exhibit fairness bias against Afro-Americans U.S citizens [49, 50]. The study and the mitigation of algorithmic bias has been widely studied in the literature [50].

2.6.3 Privacy threats

Machine learning models are trained on data, which could be private or confidential. It has been shown [51] that ML models can "leak" their training data, allowing the

implementation of privacy attacks. For instance, *data extraction attacks* consist in reconstructing the original training dataset on which the model has been trained on. This training dataset could contain privacy sensitive information. In the same spirit, *membership inference attacks* intend to learn if a given input was present in the training dataset, which could result in important privacy leaks: In the case of models trained for healthcare problems, the leakage of patient information through such attacks could be a clear violation of the law.

2.7 Conclusion

In this chapter, we presented key concepts of machine learning theory including regression models and neural networks, as well as presenting the topic of security for ML systems, categorized into three types of threats. In the next chapter, we will develop a particular security threat called *model stealing*, where an attack intend to obtain illegally a model.

Chapter 3

The problem of model stealing

ML systems can be subject to a variety of security threats, as explained in Chapter 2. In particular, we introduce in this chapter the problem of model stealing, which constitutes the main theme of this thesis. To this purpose, we first present the motivations of model stealing attacks, before proposing three strategies to implement them. We underline the inadequacy of current legal protection against these attacks.

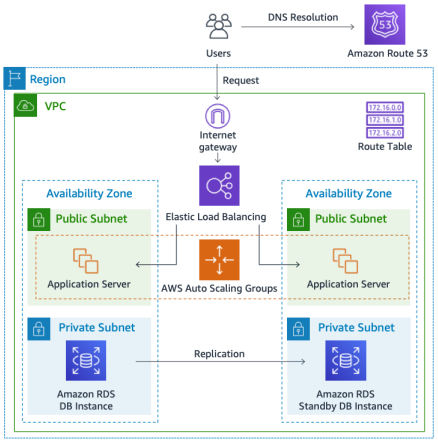
3.1 Motivation for model stealing attacks

Model theft or *model stealing attacks* can be defined as the act of acquiring a third party's model without its consent, in order to exploit the model for its own benefits. Model stealing includes stealing the file (or set of files) containing the model, such as a neural network's parameters and architecture, but also in a more general sense stealing training data, scripts, preprocessing functions or information besides the model file by itself, in order to build a *replica* of the model, demonstrating identical performance to the original model. Similarly, using a stolen model as a base for building a new model, by further training the model on new data or using the model to label new data is also considered as stealing a model. We identify three main motivations for an adversary to steal a model.

3.1.1 Lack of resources

Due to the complexity of implementing the aforementioned steps, machine learning practitioners often rely on cloud-based services such as Amazon Web Services [19], Google AI [52] or Azure AI services [17], in order to reduce computing cost by optimizing computational resources and take benefit from already existing architecture. This type of services are generally referred to as Machine Learning as a Service (MLaaS).

The implementation of the machine learning development life-cycle (also called pipelines) represents a certain cost. However, the cost of building such pipelines is not easy to estimate, because (i) there is no unique pipeline for a given project (ii) the cost of a project depends on its size (i.e. small prototypes are not as costly as state-of-the-art models). Sharir et al. [53] attempt to provide a cost estimation of training Google's NLP model Bert [54] on the Wikipedia and Book corpora (15 GB) for a single training



Service	Monthly	Annually
Elastic Load Balancing	$87.60	$1051.20
Amazon EC2	$439.16	$5269.92
Amazon Elastic IP address	$0	$0
Amazon RDS for MySQL	$272.66	$ 3271.92
Amazon Route 53	$183.00	$2,196.00
Amazon Virtual Private Cloud (Amazon VPC)	$92.07	$1,104.84

Figure 3.1: An AWS-based architecture service along with the estimated cost

phase and for multiple training phases (each training phase corresponding to the different hyperparameters, which is the usual method for productionized machine learning). The cost of base version of Bert, with 110 million parameters, ranges from $2.5k for a single training phase to $55.5k for multiple training phase. For bigger models such as the 11 billion parameters version of Bert called T5 [55], the cost of a single training phase could reach $1.3M.

Regarding the data labeling cost, cloud services such as Google's machine learning platform Vertex AI [18] provides services where human labelers could manually label datasets. For instance, you can estimate the cost to label a sentiment analysis dataset, extracted from 1.6 million tweets called Sentiment140 [56]: supposing that a tweet contains around 50 words [57], labeling the dataset with human reviewer through Vertex AI will cost approximately $6.4M. Furthermore, developers, machine learning engineers, product owners or managers are needed to develop a model. For instance, according to the U.S. Bureau of Labor Statistics [58], the average salary for a data scientist working in the U.S. is $195k.

3.1.2 Potential profits

Despite its apparent prohibitive development cost, machine learning can have a business impact. For instance, the impact of Neflix's [59] machine learning models on content recommendation helps reducing churn rate (i.e. the percentage of subscribers who discontinue their subscriptions within a given time period), saving in 2016 an estimated $ 1B per year. Compared to the 2016 annual revenue of Netflix [60], evaluated at $8.8B, it represents around 10% of the annual revenue. Several companies are making machine learning as a basis for their products, to enable fast growth. A popular example of this strategy is ByteDance [61], a Chinese technology company known for their video app TikTok, and valued at $180 billion. According to Jia et al. [62], ByteDance is the perfect example of leveraging machine learning solutions to reach quick economical growth.

```
340    "upload artifacts":
341      - command: s3.put
342        params:
343          optional: true
344          aws_key: "AKIAWFKNZA74UT3FFXHB"
345          aws_secret: "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"
346          local_file: mongosql-auth-c/test/artifacts/release.tgz
347          remote_file: mongosql-auth-c/artifacts/${build_variant}/${task_id}/mo
```

Figure 3.2: AWS Credentials stored in plain-text on GitHub

3.1.3 Lack of knowledge

For highly competitive industries, an attacker could lack knowledge regarding model architecture, training process, choice of hyperparameters, preprocessing functions applied to the data, etc. Even if an important number of recent advances in machine learning are published in conferences, an important number of companies decide to keep their model as *trade secrets* [16, 63], without disclosing any information.

3.1.4 Lack of time

In the case of small sized teams, such as early funded startups, time is the main obstacle to develop a proof-of-concept. In order to convince investors and to raise funds, the company might be tempted to acquire the technology of its competitors to raise funds quickly, with the hopes to develop their own model afterwards. Indeed, even if the company has the knowledge and the computational resource, it still requires time to develop state-of-the-art models. By bypassing the development process, an attacker could use the model as a basis for launching its company more quickly.

3.2 How to steal a model

In the previous section, we presented the main motivation for mode stealing. In this section, we propose three different techniques for stealing a model: exploiting a data leak, leveraging insider threats and implementing model extraction attacks.

3.2.1 Data leaks

In practice, when machine learning pipelines are implemented, the resulting model is stored with an object storage cloud-based service, such as an Amazon S3 bucket [64]. Indeed, this type of services offers versioning tool, well suited for continuous development of models, as well as pre-configured environments for deployment. The access to the storage system is critical for the model's owner and is protected, as part of best security practices, through mechanisms such as passwords or API secret keys. However, it is

possible to observe data leaks containing AWS credentials on collaborative open-source code platform such as GitHub [65].

With more than 100 million repositories (with at least 28 million public ones), GitHub is the biggest hosting platform for software development version control and the largest host of source code in the world. Users can use GitHub to publish their code, to collaborate on open-source projects, or simply to use publicly available projects. Finding credentials on GitHub is not rare: a 6-months study led by Meli et al. [66], analyzing 681k repositories, identifies more 4,600 unique AWS credentials publicly available. Even though several tools, both commercial [67, 68] and open-source [69], have been developed to mitigate this issue, this is still a concerning problem for companies, as shown by the 2016 Uber's data leak [70].

As shown in Figure 3.2, AWS credentials could be stored on GitHub, in configuration files such as Docker files. The `aws_key` and `aws_secret` fields provide full access (deleting, uploading, modifying actions) to a S3 bucket, which could contain ML models.

3.2.2 Insider threats

Given the growth of highly competitive markets, such as machine learning for healthcare, or machine learning for finance, companies are often working simultaneously on the same type of problems. In this case, machine learning models can be considered as confidential information and can be subject to industrial espionage.

We define industrial espionage as actions, often illegal, aiming to investigate competitors to obtain a commercial gain. In general, the target can be anything, such as product characteristics, chemical formula or future business prospects, that can be exploitable by an organization. Industrial espionage can originate from outsider threats, such as hackers targeting data breaches in organization (as mentioned in the previous section), or insider threats, such as a former employee trading information for personal gain.

A recent occurrence of this issue is the Waymo-Otto case [71]. Waymo is a Google-owned company focusing on developing driverless technology, for autonomous cars for instance. A founding member of Waymo resigned without notice the company in January 2016 before founding another company called Otto, also focusing an autonomous driving, for trucks. Otto was acquired by Uber in August 2016, nearly 8 months after Levandowski's departure from Waymo. However, according to the report, the former employee downloaded before his departure nearly 14,000 files related to Waymo's technology, including 9.7 GB of Waymo's highly confidential files and trade secrets.

Among various types of confidential data such as supplier lists, manufacturing details, the former employee acquired information about Waymo's custom built LiDAR technology. LiDAR (for ***Li****ght* ***D****etection* ***A****nd* ***R****anging*) is the main technology for building autonomous vehicles able to navigate in real time, by locating objects or by computing a 3D mapping of the surrounding environment. LiDAR technology is mainly based on neural networks [72] and by acquiring this technology, Levandowski was able to gain years of work, damaging Waymo's competitive advantage.

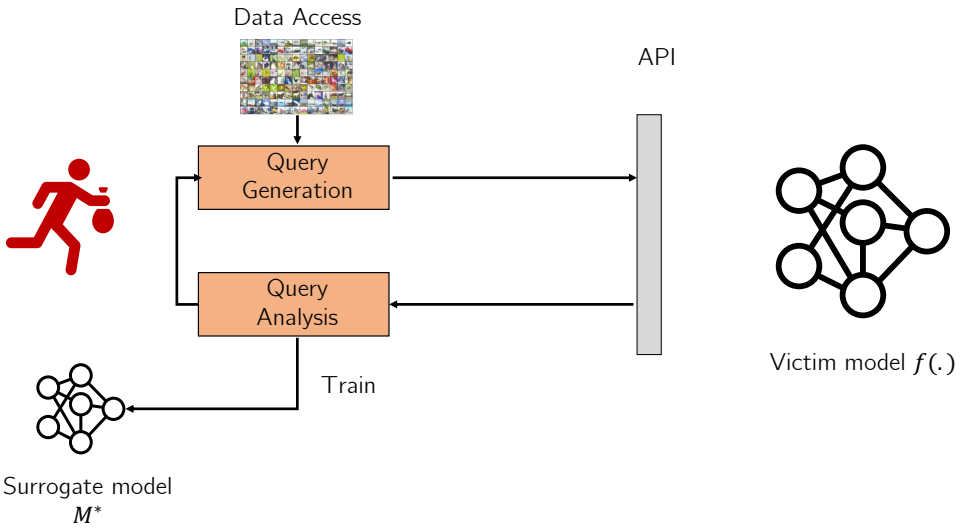


Figure 3.3: Schema of a model extraction attack

3.2.3 Model extraction attacks

Machine learning models can be implemented either internally or externally. In the first case, the model is not accessible to customers and mainly help companies and decision makers improve their businesses, for instance by building a predictive model for the sales department. In the second case, the model is the core business added value and is often available to customers, through a cloud-based service called *prediction API*, such as the language translation algorithm of Google translate [73].

The monetization of predictive API is how companies benefit from models. For instance, the Netflix's recommendation algorithm is available to all Netflix's subscribers. Even though customers do not have a complete access to the API's inner working, they can still use the model: By watching movies, they send input data to the model, which will output movie recommendations. We define this situation as a *black-box scenario*. Intuitively, by observing the behavior of the model, an adversary should theoretically be able to understand this behavior and replicate it (in the Netflix's recommendations example, by observing what are the movies recommended for different inputs).

The core of *model extraction attacks* is, given an adversary who has access to a prediction API, it is possible to extract (i.e. build a replica) the deployed model by sending input queries to the prediction API and analyzing received inputs. As shown in Figure 3.3, the adversary sends input queries $\{x^1, x^2, ..., x^n\}$ to the target model and receives $\{f(x^1), f(x^2), ..., f(x^n)$ where $f()$ is an unknown function which the adversary intends to extract. By analyzing $f(x)$, the adversary is able to build a surrogate model denoted M^*.

Previous work [74–77] developed techniques to extract a model, with the goal of minimizing input queries, with strategies which could be implemented in real-life scenarios. Depending on the threat model, some attacks are more efficient than others. For instance:

Milli et al. [78], supposing that the adversary is able to make gradient queries (i.e. sending inputs and receiving outputs with gradients with respect to inputs), introduce a model extraction attack to obtain neural network weights, allowing the adversary to apply SGD algorithm to the surrogate model M^*. With less restrictive hypotheses, Truong et al. [79] propose a data-free model extraction attacks, which does not require prior knowledge on data inputs. Orekondy et al. [80] propose to optimize the strategy to construct the dataset to query, through reinforcement learning techniques.

Model extraction can also be performed through knowledge distillation [81], which aims to compress a given model into a less-parameterized model, or transfer learning [82] where the target model is from a different domain than the surrogate (for instance, using a model trained for animal recognition to obtain a model which classifies dogs from cats.)

3.3 Current legal protections

As previously mentioned, attacks on the intellectual property of models exist and concern all types and sizes of organizations. Despite potential technical countermeasures, we give an overview of legal protections already implemented. In this context, we define intellectual property (also denoted IP) as ownership of non-physical assets. Machine learning practitioners can mainly rely on two concepts of IP law: patents and trade secrets.

On the one hand, a company can consider patent as a protection mean. Indeed, according to the World Intellectual Property Organization [83], an invention under the protection of patent law *cannot be commercially made, used, distributed, imported or sold by others without the patent owner's consent.* However, when it comes to the protection of machine learning models, neural networks or more general algorithms, there is no unique answer depending on legislation (in the European Union or in the United States) and depending on the definition of a machine learning model: If a model is strictly considered as combination of mathematical operations, then it cannot be protected under patent law because the U.S Patent and Trademark Office [84] treats models as abstract ideas or mathematical concepts. However, if the model is considered as the combination of data collection, training process and concrete real-life application, then it is plausible a patent will be accepted.

On the other hand, models can be protected as trade secrets. According to the World Intellectual Property Organization [63], trade secrets are intellectual property (IP) rights on confidential information which may be sold or licensed. Generally, three main components need to be present in order to be qualified as trade secret (i) the information needs to be commercially valuable (ii) only known to a limited group of persons and (iii) be *subject to reasonable steps taken by the rightful holder of the information to keep it secret.* The last point implies the implementation of Non-Disclosure Agreements or Non-compete Agreements for instance. Even though ML models fit into this category and trade secret law offer legal protection, it is a broad definition: trade secret owners cannot prevent competitors from using the same information if they acquired it independently from their Research and Development department or by reverse analysis. In competitive markets, it could be hard to distinguish between theft and simultaneous discoveries.

Both legal solutions are not adapted to ML models, due to (i) their nature, because ML models can be defined as a combination of weights or more broadly as a result of an entire development process (ii) the difficulty to clearly identify the uniqueness of a model.

3.4 Conclusion

To summarize, model stealing attacks represent a threat against the intellectual property of model's owner. Current legal protections are inefficient in protecting model owners' rights, creating the need for stronger defenses. In the following chapters, we propose two defense strategies: the first tackles the problem of data leaks, with a mitigation approach developed in Chapter 4. The second defense corresponds to the core of this thesis, which is the concept of watermarking, further explained in Chapter 5 and developed in Chapter 6, Chapter 7, Chapter 8, and Chapter 9.

Chapter 4

Mitigating Data Leaks in Open-Source Platforms

Model stealing attacks can originate from various sources, such as model extraction attacks or insider threats, as mentioned in the previous chapter. The third source is the exploitation of data leaks on open-source code platforms to gain access to model owners' accounts with their associate model. Due to the non structured nature of the data, current mitigation solutions are not efficient because they usually produce a significant number of false positives. In this chapter, we propose to investigate this problem by proposing a machine learning based solution to identify and mitigate data leaks with a low false positive rate.

The work described in this chapter has been published under the title *Optimizing Leak Detection in Open-Source Platforms with Machine Learning Techniques* [6], at the 7th International Conference on Information Systems Security and Privacy (ICISSP 2021).

4.1 Introduction

Data protection has become an important issue over the last few years. Despite the multiplication of awareness campaigns and the growth of good development practices, we observe a major rise of data leaks in 2019, with passwords representing 64% of all data compromised [1]. It has become a huge concern for companies to protect themselves and to efficiently detect these data leaks. GitHub [65], is one of the biggest hosting platform for software development version control. With more than 100 million repositories. Users can use GitHub to publish their code, to collaborate on open-source projects, or simply to use publicly available projects. In such an environment, one of the most critical threats is represented by hardcoded (or plaintext) credentials in open-source projects [85]. Malicious agents can exploite leaked credentials to gain unauthorized access to data, including machine learning models.

Several tools are already available to detect leaks in open-source platforms such as GitGuardian [68] or TruffleHog [69]. Nevertheless, the diversity of credentials, depending

[1] https://preview.tinyurl.com/y7bygg8d

on multiple factors such as the programming language, code development conventions, or developers' personal habits, is a bottleneck for the effectiveness of these tools. Their lack of precision leads to a very high number of pieces of code detected as leaked secrets, even though they consist in perfectly legitimate code. Data wrongly detected as a leak is called false positive data, and compose the huge majority of the data detected by currently available tools. Thus, various companies (including GitHub itself), decided to automate the detection of leaks while reducing false positive data.

In this chapter, we present a novel approach to analyze GitHub open-source projects for data leaks, with a significant decrease in false positives thanks to the use of machine learning techniques. First, a **Regex Scanner** searches through the source code for potential leaks, looking for any correspondence with a set of programming patterns. Then, machine learning models filter the potential leaks by detecting false positive data, before a human reviewer can check the classified data manually to correct possible wrongly classified data. These machine learning models are using various techniques such as data augmentation [86], code stylometry [87, 88] and reinforcement learning (as defined in Chapter 5). Our contribution can be summarized as follows:

- We present an automated leak detector for passwords and API Keys in open-source platforms, with low false positive rate.
- We evaluate our solution by scanning 1,000 public GitHub and 300 company-owned repositories, and we show that the classic regular expression approaches generate a high false positive rate, that we estimate close to 82%.
- We manually assess the results of this scan, proving that our solution reaches a negligible false negative rate.
- We investigate the false positives induced by the machine learning models, and we show that it is between 5% and 32% of the filtered data (hence between 1% and 6% of the overall data)

4.2 Overview

4.2.1 The problem of data leaks

A leak is a piece of information in a source code, published on open-source platforms such as GitHub, disclosing personal and sensitive data. Data leaks can be caused by any type of developers, such as independent developers or important corporations. For instance, a password published on GitHub by an Uber's employee led to the disclosure of personal information of 57 millions customers[2].

Several types of data leaks exist: API Keys (e.g., AWS credentials), email passwords, database credentials, etc. Although detection techniques exist, current approaches do not achieve a satisfying precision rate, leading to a high false positive rate, i.e., non-negligible

[2] https://tinyurl.com/yd3c37lc

part of data is wrongly classified as leak. A high false positive rate implies an important workload for reviewers who manually check the accuracy of the classification.

We identify three main problems. First, we notice that open-source projects often provide the documentation of their code, together with tutorials, tests, and example files. These situations are easily recognizable by the actual path name (e.g., `src/Example.py`, `connectionTutorial.java`, etc.). An important amount of passwords or database credentials are located in these type of files and are never used in production, increasing the false positive rate.

Moreover, current solutions such as GitGuardian [68], Trufflehog [69], S3Scanner [89], GitHub Token Scanning [90]] or others in [91] consist of regular expression classifiers and exclusively focus on API Keys, ignoring passwords as a category of leak. Indeed, the detection of API Keys creates a negligible amount of false positive data (due to the particular patterns). Thus, it is easier to handle them with simple regular expression classifiers. Passwords, on the other hand, are difficult to identify with classic methods, even though they account for the majority of leaks, leading to a high false positive rate. Current solutions offer little to no automated false positive filtering (except with simple heuristics) because they discard the most important source of false positive data in their analysis.

Additionally, the detection of leaks with low false positive rate is usually performed using supervised machine learning techniques which by definition incur the need for labelled training data. The collection of leak data in this context remains a challenge for several reasons: (i) on a theoretical point of view, passwords/credentials are privacy sensitive data, (ii) on a practical point of view the training dataset needs to satisfy general properties such as balance or diversity, and current machine learning approaches cannot guarantee these properties while maintaining a reasonable manual workload to sanitize, anonymize and label data.

4.2.2 Our approach

In order to detect leaks with high precision and low false positive rate, we begin with the use of a regular expression scanner similar to classical approaches. We further propose to make the distinction between two sources of false positives: *Path false positives* (e.g., data located in documentation or example files) and *Code snippet false positives* (e.g., dummy credentials or initialization variables). These two sources of false positives can be tackled by two separate machine learning models: the Path model and the Snippet model. Consequently, our solution regroups the following components:

- **Regex Scanner**: Given an open-source repository, the Regex Scanner searches through the source code history to detect any credential, API Key or plaintext password, and is considered as the default component in classic approaches. The Regex Scanner analyzes each source code modification by a developer over time, retrieving the link between these modifications and a set of regular expressions. The output of the Regex Scanner over a repository R is a set of m discoveries $D = \{d_1, ..., d_m\}$, each discovery containing a path f_i and a code snippet s_j.

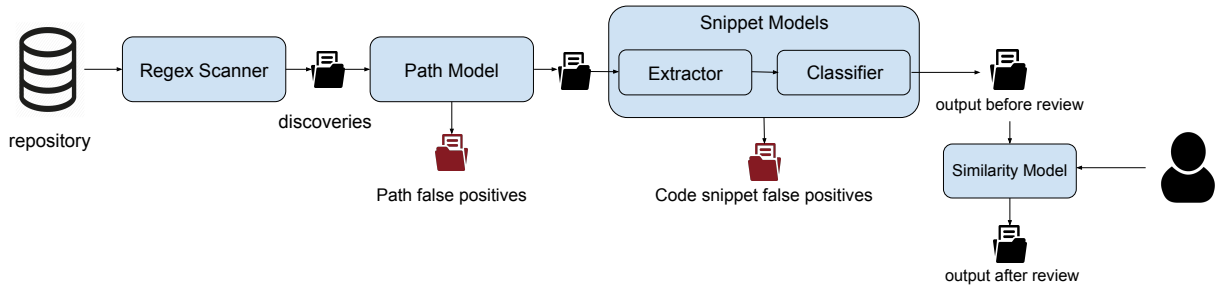


Figure 4.1: Architecture of the Credential Digger's approach

- **Path model**: The Path model analyzes each path f_i to reduce Path false positives, and outputs a list of filtered discoveries. The Path model's architecture is CBOW model, as described in Chapter 2. Thanks to this model, we already reduce false positives by 69%.
- **Snippet models**: The Snippet model filters false positives related to code snippets. A code snippet is more complex to analyze than a file path (more diversity, more irregular patterns, etc.), and may contain non-negligible amount of irrelevant data for leak classification (function names, type names, method names, symbols, etc.). The snippet model is itself composed of two sub-models, the **Extractor** and **Classifier**.
  - **Extractor:** The Extractor identifies relevant information in the snippets, i.e., the variable name and the value assigned. As mentioned before, it is difficult to collect relevant data to train the Extractor. Thus, we implement data augmentation techniques through reinforcement learning.
  - **Classifier:** The Classifier takes the relevant information extracted as inputs to classify a code snippet as a leak or as a false positive. At this step, we consider again a CBOW model, leading to a reduction of 13% of the discoveries with the combination of the Extractor and the Classifier.
- **Similarity model:** As a final step, once automated components output the leaks they have detected, a human reviewer manually checks the accuracy of the classification by *flagging* (i.e., re-classifying manually) a leak as a false positive. The Similarity model can assist the human reviewer by flagging similar discoveries as false positives to reduce her workload.

Figure 4.1 gives an overview of the architecture of the proposed framework.

4.2.3 Scenarios

We propose two examples scenarios to better understand the expected behavior of the components.

Scenario 1: Consider the code snippet `String password = "Ub4!l"`, located in the file `src/Example.py`. The Regex Scanner identifies the key-word *password*, so that the discovery is classified as a leak. Then, the Path model analyzes the file path, and discards the leak as a Path false positive (due to the word `Example`).

Scenario 2: Consider the code snippet `String password = "Ub4!l"`, located in the file `src/run.py`. The Regex Scanner still identifies the key-word *password*, while the Path model does not discard the leak due to its Path. The Extractor outputs the combination *(password, Ub4!l)*, and the Classifier classifies this code snippet as a leak.

Scenario 3: Consider the code snippet `String password = "INSERT_CREDENTIAL_HERE"`, located in the file `src/run.py`. The Extractor outputs the combination *(password, INSERT_CREDENTIAL_HERE)*, and the Classifier classifies the code snippet as a false positive.

4.3 Path Model

The goal of the Path model is to reduce the false positives due to leaks detected in example, configuration, test files. This model analyzes where a leak is identified in an open-source repository (i.e., its file path), and gives a first classification on whether the leak is relevant or not. The Path model is a CBOW model, previously introduced in Chapter 2.

4.3.1 Data pre-processing for the Path model

The Regex Scanner outputs a list of discoveries, each discovery containing a path f_i (used as an input for the Path model) and a code snippet s_j (used as an input for the Snippet models). Some pre-processing phase is needed for both these data: First, we remove non-alphanumerical characters, before applying stemming and lemmatization, which are natural language processing techniques [92]. We split the input data in words to obtain $f_{preprocc} = \{f_i^1, ..., f_i^{k_f}\}$. In order to respect common coding conventions while standardizing the input data, we apply the Java coding convention to each word in f_i (the choice of coding convention is irrelevant as long as it is standardized for all inputs).

Example: If we consider *Scenario 1*, with $f =$ `src/Example.py` and $s =$ `String password = "Ub4!l"`, the pre-processing phase outputs $f_{preproc} = \{src, Example, py\}$ and $s_{preproc} = \{String, password, Ub4!l\}$

4.3.2 Training phase

The workload to gather sufficient training data and to review labeled items can be handled by a human reviewer. Since the path name is not a sensitive piece of information, the data sanitization aspect can be reduced to a minimum. We collected 100k file names from 1,000 GitHub repositories (analyzed in our evaluation in Section 4.6), which we labeled using regular expressions and manual checks. We applied the data pre-processing techniques and we train a CBOW model, achieving 99% of accuracy on this dataset.

4.4 Snippet Models

In this section, we detail the design choices for the Snippet models: the Extractor and the Classifier. To fully understand our approach, we propose to introduce several concepts, aimed to be used as building blocks for these models.

4.4.1 Building blocks

Code stylometry

Each developer has her own coding habits, depending on many factors such as the coding language or the occurrences of given key-words. We introduce a concept called *code stylometry*, aiming to encapsulate into a vector the main characteristics of these coding habits.

Example: Consider a Python developer, focused on software development. This developer will probably use key-words such as `password` or `pass_word` to do password assignments (e.g., `password = "Ub4!l"`). A different developer, focused on database management, might prefer keywords such as `root` or `db` (like `db.root = "Ub4!l"`. These design choices will result in two different code stylometry vectors.

Supposing that we have extracts of code belonging to a developer (denoted $\mathcal{E}$), we compute her code stylometry based of these extracts. The complete list of the features we consider for code stylometry can be found in the Appendix A.1.

Data augmentation

As previously mentioned in Section 4.2.1, obtaining a dataset for leaks on GitHub is complicated. Indeed, since we are dealing with sensitive data, we have to follow and comply with privacy guidelines, e.g., performing data sanitization. The collected data also needs to be labelled, which may require a significant manual workload. In addition, the diversity of leaks in open-source repositories usually follows the Pareto rule, meaning that 80% of the data leaks are originating from the same few programming patterns (for instance, `password="1234"` is extremely common). Therefore, collecting a diverse dataset in order to train a machine learning model (to have good generalization properties and avoid overfitting [86]) would be difficult to reach from a practical point of view. For these reasons, we propose to use data augmentation techniques in order to enhance the size and the diversity of the dataset with no extra cost in labelling or sanitization.

Data augmentation is a set of techniques to enhance the diversity of a dataset without new data. It is particularly used in image processing [86], by applying filters to images in order to produce new training samples. The main benefit is to expand a dataset (fixing class imbalance or adding diversity in the training samples) with limited pre-processing cost. Data augmentation can also prevent overfitting (i.e, when a machine learning model is not able to generalize from the training data).

Example: Consider two leaks `password="Ub4!"` and `mypass="1234"`. If we switch the variable names to obtain `password="1234"` and `mypass="Ub4!"`, we have in fact created two new leaks. In general, given a pattern `key="value"`, any pair of $(key, value)$

can be chosen to obtain a new leak. Every time another variable name is collected, data augmented leaks can be obtained by the re-arrangement of already existing data. More specifically, when a new programming pattern is collected for password assignment (e.g., `DataBase.key="value"`) additional leaks can be obtained, creating diversity from limited dataset.

In the context of this work, we have an important number of alternatives to enhance our dataset, such as replacing variable names by synonyms, modifying function names (e.g., from `set_password()` to `os.setPass()`), or replacing `'[]'` with `'()'`. Since there is no clear algorithm to choose which actions (or combination of actions) will output the best suited dataset for the training phase, we consider the Q-learning algorithm [93].

Q-learning

Algorithm 1 Q-learning algorithm

$\mathcal{D}$, π, $style_{ref}$
Training Data for π T_π
procedure Q-LEARNING
    while *condition is* $True$ **do**
        $style \leftarrow choose\_actions(\pi, \mathcal{D})$
        $reward_{sim} \leftarrow similarity(style, style_{ref})$
        $update\_choices(reward_{sim})$
    end while
    $T_\pi \leftarrow choose\_actions(\pi, \mathcal{D})$
end procedure
[1]

Q-learning algorithm is a reinforcement learning algorithm, where an agent learns, through interactions with its environment, actions to take to maximize a reward. In the data augmentation process, some actions can be applied to the collected data, such as modifying variable names (like in Example A), selecting different functions names, or considering object-oriented programming patterns. Since different combinations of actions lead to different datasets, it will also lead to different code stylometry vectors. The goal for data augmentation is to converge to a particular code stylometry of the transformed dataset, called *reference stylometry*.

We define three primitives to build the Q-learning algorithm, that we show in Algorithm 1.

1. $style \leftarrow choose\_actions(\pi, \mathcal{D})$: The agent can choose a combination of actions she intends to perform on data $\mathcal{D}$ (collected data from an empirical study) for a given pattern π. These actions produce a new dataset, from which we can compute the resulting stylometry $style$. In this work, we consider a list of 28 programming patterns, available in the Appendix A.1.

2. $similarity(style, style_{ref})$: The similarity function computes the cosine distance between the current stylometry and the reference stylometry (computed through extracts $\mathcal{E}$). The output corresponds to the reward (which we want to maximize).

3. $update\_choices(reward_{sim})$: Based on the reward, the Q-learning algorithm will update the available choices of actions. This update is ruled by the Bellman equation [94].

After several iterations of the algorithm, the Q-learning will apply the optimal choices of combinations of actions, to compute the training dataset for a given programming pattern T_π. The stopping condition can be time-based (e.g., maximum number of iterations) or a threshold reward value.

4.4.2 Extractor

Algorithm 2 Extractor model algorithm

Collected data $\mathcal{D}$
patterns Π
extracts $\mathcal{E}$
model
procedure EXTRACT
    $style_{ref} \leftarrow stylometry(\mathcal{E})$
    for π in Π **do**
        $T_\pi \leftarrow QLearning(\pi, \mathcal{D}, \mathcal{E}, style_{ref})$
        $T_{tot} \leftarrow T_\pi \cup T_{tot}$
    end for
    $model \leftarrow train_{CBOW}(T_{tot})$
end procedure
[1]

The main objective for the Extractor is to remove unnecessary elements in a code snippet, taking as inputs a list of discoveries (corresponding to the output of the Path model), and it outputs, for each code snippet, a tuple containing a variable name and a variable value. If no tuple can be found in a code snippet, then it is automatically discarded (because no variable assignment has been found).

The training data for the Extractor is obtained through the augmentation of collected data $\mathcal{D}$ from GitHub, like variable names, function names, etc., used for variable assignments. Data augmentation is performed before the training phase. Simultaneously, the Extractor has access to a collection of code extracts $\mathcal{E}$; these extracts are not discoveries, but simply randomly chosen pieces of code, from which we can compute a reference code stylometry. Hence, an Extractor model can be trained for every developer (because each of them has a different code stylometry) or for a group of developers (considering their global code stylometry).

The training phase for the Extractor is shown in Algorithm 2. For each collected programming pattern π, we apply the Q-learning algorithm, while considering the stylometry of the developer ($style_{ref}$) as the reference stylometry. We obtain the training data T_{tot} on which a CBOW model is trained to obtain the Extractor.

4.4.3 Classifier

The Classifier takes as input a list of tuples, each of them containing a variable name and a variable value (which corresponds to the output of the Extractor) and classifies the tuple as a leak or as a Code snippet false positive. The training data for the Classifier is different from the training data of the Extractor. We retrieved an open-source list of the most commonly used passwords[3] (used by multiple tools when attempting to guess credentials for a given targeted service), and collected (through an empirical study) a list of commonly used variable names (such as *root, admin, pass*, etc.). The design of the Classifier is similar to the design of the Path model, with a CBOW model. The Classifier achieves 98% of accuracy on this dataset of (variable name, variable value).

4.5 Similarity model

In the manual review phase, a user can classify a potential leak containing a code snippet s_j as false positive. We assume that we have the set of CBOW word representations of code snippets of discoveries $\{CBOW(s_1), ...CBOW(s_k)\}$. To reduce the workload of a human reviewer, we introduce a Similarity model, taking the code snippet $CBOW(s_j)$ as input and automatically classifying discoveries containing similar code snippets as false positives, denoted $\{CBOW(s_i), ...CBOW(s_{k'})\}$ with $0 \leq k' \leq k$.

Definition. *Let η be a similarity threshold. Two code snippets CBOW representation $CBOW(s_i)$ and $CBOW(s_j)$ are similar if $cosine(CBOW(s_i), CBOW(s_j)) \leq \eta$*

A similarity threshold $\eta = 1$ means that, for a flagged discovery $\{f_i, s_j\}$, the Similarity model flags all the duplicates of the code snippets. The impact of η is analyzed in Section 4.6.3.

4.6 Experiments

In this section, we present an evaluation of our solution, divided into three major parts. Firstly (in Section 4.6.1), we evaluate the rate of false positive data on the output of the Regex Scanner (as proposed by the solutions in the literature). With this goal, we scan a dataset of 1,000 repositories from the public GitHub (i.e., `github.com`), and 300 repositories from a GitHub-like code versioning platform owned by a private company. In the remainder of this section, we refer to `github.com` as *public github* and to the repositories publicly available on this platform as *public repositories*, while we refer to the privately owned GitHub platform as *proprietary github* and to its repositories as

[3] https://github.com/danielmiessler/SecLists/tree/master/Passwords/Common-Credentials

Repository type	Discoveries	File path FP	Code snippet FP	Total FP
public	13.6	9.35 (69%)	1.79 (13%)	11.11 (82%)
proprietary	0.259	0.091 (35%)	0.064 (25%)	0.155 (60%)

Table 4.1: FP by models (in millions of discoveries)

classification		machine learning models	
		potential leak	non-critical data
manual	leak	20% (true positives)	1% (false negatives)
	non-critical data	80% (false positives)	99% (true negatives)

Table 4.2: Manual assessment of 2000 discoveries

proprietary repositories. Next, in Section 4.6.2, we manually assess the false positive rate as well as the false negative rate induced by the machine learning models, and we show that the false negative rate is negligible (meaning that no leak on the output of the Regex Scanner is discarded by the models). Finally, in Section 4.6.3 we estimate the impact of the data augmentation algorithm parameters on the precision of our solution.

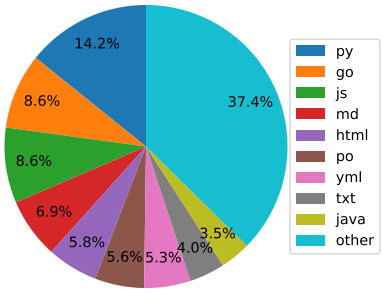
The tool that we have developed, and that we have used for the experimental evaluation of our proposal, called Credential Digger, is available in open-source together with the machine learning models[4].

4.6.1 Regex Scanner false positive rate

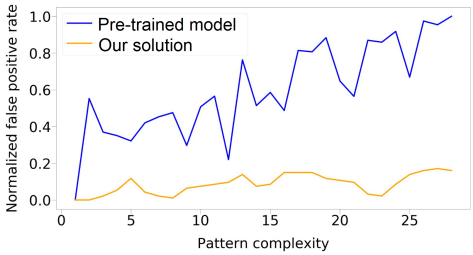
For this experiment, we randomly selected and scanned 1,000 public repositories on GitHub. The list of regular expressions used by the Regex Scanner can be found in the Appendix A.1. Over 14 million discoveries have been found, with 13.6 million in 579 out of 1,000 public GitHub repositories (58%) and 260k discoveries in 268 out of 300 proprietary repositories (89%). Our discoveries cover more than 30 programming languages, and represent more than 300 file types. Figure 4.2a shows the 10 most common file extensions containing leaks in our dataset. The number of contributors and the sizes of the repositories have been chosen equally distributed.

We notice that API keys are still widely published in open-source projects, as shown also in [95]. Nevertheless, they do not represent the majority of the discoveries. Indeed, in our study, we notice a more important number of passwords giving access to local and remote databases, or to e-mail accounts. We observe that the vast majority of these passwords is not critical (i.e., false positives), which seriously increases the load of a developer to review each of them manually. These passwords are mostly undetectable by traditional scanning tools, but they are still easy to find for someone using a simple search tool in the commit message (with keywords such as *remove credentials*, *delete password*, etc.). We found many passwords that we suppose to be real (even if we cannot

[4]https://github.com/SAP/credential-digger



(a) Most common files containing secrets



(b) Normalized FP rate by pattern for the pre-trained model and the Extractor trained with Q-learning

Figure 4.2: Statistics on the GitHub scan

have the certainty of this, since we are not allowed to test these passwords). This is a very important concern not only because passwords are still widely reused [96], but also because two-factor authentication is still scarcely known (and thus activated) [97, 98], and scarcely supported by services [99].

To summarize, the vast majority of the discoveries detected with the Regex Scanner consists of false positive data. In order to reduce the false positive rate, as described in section 4.3, we apply the Path model and the Snippet models sequentially, and finally evaluate the newly obtained false positive rates. As shown in Table 4.1, the Path model classifies almost 70% of the discoveries as false positives in the public dataset. This score is halved with the proprietary dataset. Together with the Snippets models, we see that up to 82% of the discoveries are classified as false positives without a human intervention.

4.6.2 Models false negatives

In order to assess the behavior of our models, we decided to perform a manual review of a limited number of discoveries (we recall that the Regex Scanner found 14 millions discoveries in the previous experiment). To do so, we consider a sampling method, randomly selecting 100 discoveries classified as potential leaks by the models and 100 discoveries classified as non-critical data by the models, and we manually analyze each of them. We repeat this process 10 times (covering 0.01% of all the discoveries from the previous experiment). The results are shown in Table 4.2. It is visible that 99% of the discoveries classified as non-critical data by the models are real-life true negatives. The remaining percentage (corresponding to false negatives) corresponds to edge cases, where developers inserted (seemingly) real credentials in dummy files. Thus, in the scope of our study, we can state that the unclassified leak rate is negligible. Given the discoveries classified as potential leaks, 80% of them are non-critical (i.e., false positives non detected by the models), and 20% of them are actual leaks (i.e., true positives). If we project the results of this manual assessment to the complete list of discoveries, we can assume that (i) our models do not create false negatives and (ii) they provide an efficient reduction of

Repository	Language	Contributors
rhiever/MarkovNetwork[5]	Python	3
bradtraversy/vanillawebprojects[6]	Javascript	8
AGWA/git-crypt[7]	C++	15

Table 4.3: Description of the three repositories

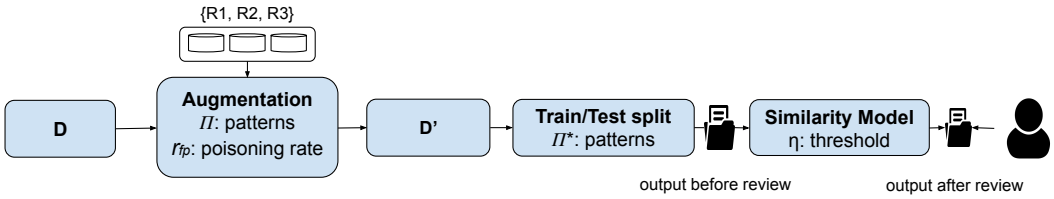


Figure 4.3: Data augmentation on $\mathcal{D}$ to assess the performance of the Extractor with the train/test split technique

the false positive data on the output of the Regex Scanner.

4.6.3 Models false positives

In the previous section, we notice that it is difficult to assess the false positive rate of the Snippet Models (especially the Extractor) with precise metrics since, for the majority of the leaks detected in open-source repositories, we do not have a ground truth. In the previous section, we had to consider other evaluation techniques (e.g., sampling) to evaluate the false positive rate in real-life conditions, or to manually label the discoveries, which represents an important workload. Furthermore, due to the limited size of labeled data that we manage to collect, we cannot apply the *train/test* split [100] technique in order to evaluate our models on them. The train/test split technique is a well-known process to assess the validity of machine learning models, splitting the data into two distinct subsets: training data (on which we will fit our model) and testing data (on which we will evaluate our model). As mentioned before, the size of the collected labeled data $\mathcal{D}$ is too small to accurately evaluate the Extractor using the train/split technique.

Nevertheless, Section 4.4.2 shows that we can apply data augmentation techniques to expand the size of our training dataset, as long as we have a reference stylometry. Hence, the goal of this section is to evaluate the false positive rate induced by the Extractor itself (independently from the false positives induced by the Regex Scanner) on several open-source repositories, with a train/test split approach commonly used in supervised learning on an data augmented dataset. To achieve this goal, we consider three different repositories $\{R_1, R_2, R_3\}$, each of them containing source code written in different programming languages by different developers (and different code stylometries) as shown in Table 4.3. The main idea is to use the stylometries of these repositories to obtain an augmented dataset where the train/test split technique is possible, and to see the impact of the augmentation process on accuracy metrics such as precision or recall.

Train/test split

We propose an experiment to evaluate the false positive rate on the Snippet Models with respect to $R \in \{R_1, R_2, R_3\}$, as illustrated in Figure 4.3.

- To begin with, we obtain an augmented dataset $\mathcal{D}'$ from collected data $\mathcal{D}$, patterns Π, and extracts $\mathcal{E}$ of the repository R. We can select the leak percentage in $\mathcal{D}'$ with parameter r_{fp} ($r_{fp} = 0.5$ corresponds to a balanced dataset).
- Next, we split $\mathcal{D}'$ into a training and a testing dataset. We also perform the split on the patterns to obtain $\Pi^* \subset \Pi$: This ensures that the patterns used to perform the training (Π^*) are different from the patterns used to do data augmentation (Π).
- Finally, after the training phase, we compute metrics such as precision, recall, and f_1 score on the testing dataset. A manual reviewer manually flags the false positives, and she is assisted by the similarity model (with threshold η). We consider that the manual reviewer flags 0.1% of the discoveries.

There are mainly three hyper-parameters that have an impact on the precision of the Extractor: r_{fp} (the percentage of leaks over the size of $\mathcal{D}'$), how we choose the subset of patterns Π^* used to train the Extractor in the train/test phase, and the similarity threshold η from the similarity model. In the following subsections, we show the effects of these three hyper-parameters on the accuracy of our solution. We propose to first study the impact of the Q-learning algorithm on the precision of the Extractor, and show that this technique significantly increases the precision (thus decreasing the false positive rate). We further evaluate the impact of the three hyper-parameters on the precision, recall and false positive rate of our approach.

Pre-trained model

To begin with, we study the impact of the data augmentation process. On the one hand, we have an Extractor model, pre-trained on the data we collected without any data augmentation process (called pre-trained Extractor). On the other hand, we have an Extractor model, trained with the Q-learning algorithm for data augmentation where $\Pi^* = \Pi^*_{0.8}$ (corresponding to a set of patterns, randomly chosen including 80% of the patterns in Π). In Table 4.4, we see the impact of the Q-learning algorithm, with a high precision score as opposed to the pre-trained model (it increases from 55.56% to 71.66%).

A recall close to 100% means that we detect almost all the leaks. However, when the user flags a discovery as false positive, the similarity model (with threshold parameter η) may classify an actual leak as non relevant (i.e., it may cause a false negative). If we select $\eta = 1$, we reach a recall of 100% but without any significant improvement of the precision score. To fix the recall drop, a possible remediation is to inform the user on what discoveries have been classified as non relevant by the similarity model, so that she can check whether or not an actual leak has been wrongly classified (it will improve the recall score, but will increase the manual workload also).

Situation	$\Pi^*_{0.80}$			$\Pi^*_{0.5}$		
	Precision	Recall	F1	Precision	Recall	F1
Before review	89.33	100	**94.36**	71.66	100	**84.89**
After review	89.69	99.96	94.55	74.52	99.71	85.30

(a) Impact on the manual review on the metrics

Situation	$\Pi^* = \Pi^*_{0.80}$		
	$r_{fp} = 0.5$	$r_{fp} = 0.20$	$r_{fp} = 0.05$
FP Rate	**5.97**	12.09	11.03

Situation	$r_{fp} = 0.5$			
	$\Pi^*_{complex}$	$\Pi^*_{0.5}$	Π^*_{simple}	$\Pi^*_{0.25}$
FP Rate	9.36	**18.35**	31.99	27.86

(b) Impact of Π^s and r_{fp} on the FP rate

Table 4.4: Poisoning experiments. Results in bold in (a) correspond to experiments with identical parameters in (b)

We also compare the precision score per pattern. Each pattern has a complexity value associated with its index (i.e., the pattern with index 1 is the simplest, and the pattern with index 28 is the most complex one). As shown in Figure 4.2b, we can observe a linear relationship between the pattern complexity and the false positive rate when we use the pre-trained Extractor (which seems natural for a global model, since more complex patterns are harder to detect, leading to more false positives). With the Extractor trained with the Q-learning algorithm, the false positive rate is independent from the complexity of the pattern (which means that no particular pattern will lead a higher false positive rate).

Extractor with Q-learning

In this section, we solely consider the Extractor trained with the Q-learning algorithm (excluding the pre-trained model), by presenting the impact of r_{fp} and Π^* on the false positive rate.

Impact of r_{fp}: First, we analyze the impact r_{fp} on the false positive rate in three different situations, i.e., with $r_{fp} = 0.5$ (balanced situation between leaks and false positive), $r_{fp} = 0.2$, and $r_{fp} = 0.05$ (unbalanced situation where leaks are scarce), while fixing parameter Π^*. We present the results in Table 4.4a. We observe that:

- in a balanced situation, we achieve a false positive rate of 5.97%, considerably reducing the part of false positive data in the discoveries;
- in unbalanced situations, the results show that we manage an acceptable rate of false positives, below 12%.

Impact of Π^*: Next, we analyze the impact of the choice of Π^* on the false positive rate in several situations, while fixing the poisoning rate $r_{fp} = 0.5$. As mentioned before, each pattern has a complexity value. Thus, we can define the complexity of a set of patterns Π as the average complexity of these patterns (therefore, in our experiments, the complexity of our set of 28 patterns Π, is equal to 14.5). Let $\Pi^*_{0.5}$ be a set of patterns representing 50% of the set of patterns in Π, with an equivalent pattern complexity. Table 4.4b presents the results.

For $\Pi^* = \Pi^*_{0.5}$, we obtain a false positive rate of 18.35% in this setting. Compared to $\Pi^*_{0.8}$, a 30% decrease in the number of patterns leads to a 15% increase of the false

Situation	Precision	Recall
Pre-trained Extractor	55.56	100
Extractor with Q-learning	71.66	100
Extractor + Similarity model	74.52	99.71

Table 4.5: Impact of data augmentation with $\Pi^*_{0.80}$

positives, proving that our approach is able to generalize unseen patterns while preserving low false positive rate. We also consider $\Pi^*_{0.25}$ corresponding to 25% of the patterns with an equivalent overall pattern complexity.

Furthermore, we decide to study set of patterns without conserving the overall pattern complexity, splitting Π^* into two sets $\Pi^* = \Pi^*_{simple} \cup \Pi^*_{complex}$, corresponding respectively to the first 14 patterns and to the last 14 patterns. The results of the experiment with Π^*_{simple}, $\Pi^*_{complex}$, Π^*_{simple} and $\Pi^*_{0.25}$ are also presented in Table 4.4b.

Although $\Pi^*_{0.5}$, Π^*_{simple} and $\Pi^*_{complex}$ contain the same number of programming patterns, the pattern complexity distribution greatly impacts the false positive rate. We reach an acceptable false positive rate with only 25% of the patterns, but more equally distributed in complexity. It is worth noting that the highest score is reached with the $\Pi_{complex}$ pattern set, with results close to the full pattern experiment. Indeed, as shown in Figure 4.2b, the false positive rate per pattern is higher, on average, for complex patterns (i.e., with index above 14). Therefore, targeting only this class of patterns leads to a decrease of the global false positive rate.

With respect to Π^* and r_{fp}, we estimate the false positive rate induced by the Extractor between 6% and 32%. In Section 4.6.1, we showed that more than 80% of the false positive data (induced by the Regex Scanner) has already been discarded. Overall, we showed that the false positive rate of the whole solution (including the Regex Scanner and the machine learning models) represents between 1% and 6% of the output.

4.7 Related work

4.7.1 Research work

An important amount of work targets GitHub open-source projects, from vulnerability detection [101] to sentiment analysis [102]. Empirical studies also provide a more global overview of the data on GitHub [103] and how to facilitate its access [104].

With the advent of machine learning techniques in the researchers' toolkits, approaches for source code representation have been developed, proposing a language-agnostic representation of source code [105,106]. Leak detection can be also considered as a branch of data mining or code search tasks. Works on evaluating the state of the semantic code search [107], as well as works on deep learning applications for code search [108], emphasize the need for developing machine learning techniques for source code analysis. However, these previous works have different purposes from ours, especially regarding the criticality of the datasets, and they consider token-based representations (so language dependent) as opposed to our purely semantic approach.

Leak detection is connected to malware detection [109,110] addressing similar issues to solve privacy concerns in realistic settings, where the testing samples are not representative of real world distributions. Contrary to malware classification, we do not have a reference dataset to benchmark language specific approaches.

Code transformations based on stylometry have been tackled by other works [87,88]. In particular, in [88], the authors, given a list of code extracts $\{e_1, ..e_n\}$ developed by a list of developers $\{D_1, ...D_m\}$ and an authorship attribution classifier, transform each e_i to fool the classifier concerning the authorship of e_i. To do so, they use a Monte-Carlo Tree Search algorithm to compute the most optimal code transformations to perform the authorship attribution attack. In our work, we leverage the ideas developed in [88] to perform our own code transformation to do data augmentation. We choose Temporal Difference (TD) learning over Monte-Carlo, due to its incremental aspect. Indeed, in the description of the Q-learning algorithm, there is a stopping condition in order to obtain the augmented data, whereas Monte-Carlo algorithms have to be run completely. We suppose that in our case the conditions for the convergence of TD algorithms are satisfied [111].

Two different studies have considered the state of data leakage in GitHub repositories. [91] focuses on API Keys detection but the scope of their study is limited to Java files, and the remediation techniques are mainly composed of heuristics. In a more recent work [95], Meli et al. propose a study on the leak of API Keys, focusing on possible correlations between multiple features in a GitHub project to find root causes. Nevertheless, this work is limited to API Keys: It is explicitly stated that their analysis does not apply to passwords. Moreover, the focus of their study was on the characteristics of true secrets, with indications on contributors or persistence of secrets. Our focus dwells instead, on the false positive data, since it represents the vast majority of discoveries of any open-source project. Finally, they provide an extensive study of GitHub API Keys leaks by scanning an important number of repositories, close to 700,000. In our work, we chose not to conduct our GitHub leak status study with such a high number of repositories, because it would have led to a tremendous number of false positive discoveries, which would not have been possible to process.

4.7.2 Comparison with other credential scanning tools

Since the problem of leak detection in public open-source projects is not new, open-source tools such as GitHub Token Scanning [90], GitLeaks [112] or S3Scanner [89] have been developed to tackle it alongside commercial platforms, namely GitGuardian and Gamma. However, to the best of our knowledge, there is no open-source tool which scans GitHub repositories and applies machine learning to decrease the false positive rate. Therefore, since the existing tools do not work in the same paradigm as our approach (not considering passwords, for instance), we do not provide a comparison of metrics to avoid any bias. Still, we can compare our approach with several tools we selected.

TruffleHog [69] is a very popular (5k stars on GitHub, at the time of writing) and open-source scanning tool. The user has to provide her own set of regular expressions to the tool in order to detect possible leaks. This tool does not use machine learning, and it

is mostly targeted to detect API Keys. Its main advantage is surely its simplicity for developers. Similar tools have emerged with the same characteristics, such as *Gitrob*[8] and *git-secrets*[9].

GitGuardian [68] is a tool provided by the namesake company founded in 2016 and specialized in detection of leaks in open-source resources. Alongside their commercial offer, they provide free services to scan one's own GitHub repositories. They claim their tool is *machine learning-powered* and that they can identify *more than 200 API Key patterns*, but they do not mention passwords.

TruffleHog [69] and its variants aim to be a strong baseline for scanning tools. For example, in [95] authors offer improvements to its core algorithm. Various heuristics can be implemented to improve the accuracy of the tool, such as *entropy check*: if a string has high entropy, which means it consists of seemingly random characters, the probability that this string is an API Key is high. We perform several manual tests on the GitGuardian platform on various API Keys patterns and on plaintext passwords in order to understand the possibilities and the limitations of such a tool. According to our tests, the platform is not able to detect plaintext passwords, and it only detects a reduced sample of API Keys, excluding big API Keys providers such as Facebook and Paypal. We only tested the free version of GitGuardian, so it might be possible that the full capabilities of the platform are only enabled in the commercial offer. Another commercial tool called **Nightfall AI**[10] (formerly known as Watchtower) offers the same services, but no free version is available to test the platform.

We compared several tools, on different criteria, and show our results in Figure 4.4. For each scanning tool, we compare what techniques are used, and if there is any false positive reduction. The open-source tools do not perform false positive reduction (since most of them do not detect passwords), favoring the usage of heuristics which need less computational power. However, most of the heuristics are not adapted to all use cases, so the developer has to manually configure the tool without efficiency guarantees. In our approach, we choose to adapt the scanning process to each developer, thus the fine-tuning is performed by the Leak Generator rather than the user herself. The *continuous training* parameter is the ability for the tool to re-train the machine learning models when the user flags a discovery, so to improve future classifications. Open-source solutions are more focused on single use cases, offering limited interactions with the developers. Our approach, similar to the GitGuardian platform, is to improve the accuracy while reviewing, decreasing the monitoring time. The user experience is also a key point in order to be used efficiently. The price could represent an important barrier for small companies willing to protect themselves, encouraging bad development habits. Commercial products provide a user interface, making the tool more accessible to developers, and even to non-technical people. Since the origin of a leak does not depend on the level of expertise of the developers [95], tools with a user interface could be easily used also by beginners to protect their code.

[8] https://github.com/michenriksen/gitrob
[9] https://github.com/awslabs/git-secrets
[10] https://www.nightfall.ai/

Category	Tool	Scanning process						User experience						Adoption		
		Regex	Entropy check	Heuristics	Path FP	Password detection	Machine learning	Free	User interface	Open-source	Repository management	Scan of private repositories	Authentication not required	Community	Scalability	Regular updates
Known algorithms	TruffleHog	●	◐	-	-	-	-	●	-	●	-	◐	●	●	●	●
	Git-secrets	●	●	●	-	-	-	●	●	●	-	-	-	◐	●	◐
	Gitrob	●	●	●	-	-	-	●	●	●	-	-	-	◐	●	◐
	[95]	●	●	●	◐	-	-	-	-	-	-	-	-	-	-	-
Commercial offers	GitGuardian	●	-	●	-	-	-	◐	●	-	●	●	-	◐	●	●
	Nighfall AI	●	-	●	-	-	●	-	●	-	●	●	-	-	●	-
Our approach		●	-	●	●	●	●	●	●	●	●	●	●	-	●	●

● = provides property; ◐ = partially provides property; - = does not provide property;

Figure 4.4: Comparison of available tools

4.8 Conclusion

We proposed an approach to detect data leaks in open-source projects with a low false positive rate. The solution improves classic regular expression scanning methods by leveraging machine models, filtering an important number of false positives. The approach has been open-sourced as a library under the name Credential Digger [113], with the machine learning models published in open access on HuggingFace [114]. We propose the following measures in order to improve the quality of the evaluation, as a future work:

- The evaluation of the models, including the data augmentation process, has been realized on a limited set of repositories, for faster manual review. A first improvement could consist in evaluating the approach on a larger sample of repositories.
- Compared to the state of the art of NLP models, the CBOW approach is simple. Since the publication of this work, we have been working on a more advanced version of the models, based on larger architectures [115]
- The data augmentation process works with a limited set of pattern and data modification actions. Developing this aspect could make the data augmentation more efficient.

Mitigating data leaks only counter one attacking strategy for model stealing. In the next chapters, we introduce the concept of watermarking as a more general defense against this type of attacks.

Chapter 5

Watermarking Machine Learning

In the previous chapters, we highlighted the problem of protecting the intellectual property of machine learning models, due to potential attacks aiming to steal models to gain commercial advantages. Current legal solutions are not secure enough against this issue. In this chapter, we propose to study the concept of watermarking and conduct a literature review of watermarking, a solution for ML models' IP protection.

5.1 Digital Watermarking

Digital watermarking is the concept of embedding information in content such as audio, video, image, text, etc, in order to verify the authenticity or integrity of the target content. The concept originates from physical watermark, on postal stamps or currency for instance, in order to prevent counterfeiting. Overall, digital watermarking is composed of three parts. First, ownership information about the target content's owner needs to be inserted, in a process called **embedding**. The embedded information is defined as *watermark*. Embedding algorithms, to be valid, are required to satisfy properties such as fidelity (the embedding shouldn't tamper the integrity of the original content). Then, the resulting watermarked content can be used, transmitted to third-parties, distributed to a large audience or stored. The content is vulnerable to attacks from attackers attempting to modify the watermarked content to remove or alter the watermark. For instance, the adversary could try to identify the watermark, through data analysis, or could try to remove the watermark by modifying the watermarked content (compression, noise addition, etc.). Finally, when the original owner wants to verify the presence of the watermark, a **verification** algorithm is applied on the content in order to extract the watermark. Verification algorithms are required to satisfy properties, such as reliability (the false negative rate for watermark extraction should be low) or integrity (the false position rate for watermark extraction should be low).

Digital watermarking applications include copyright protection, ID card security or broadcast monitoring (for instance, when television news channels broadcast watermarked video from international agencies).

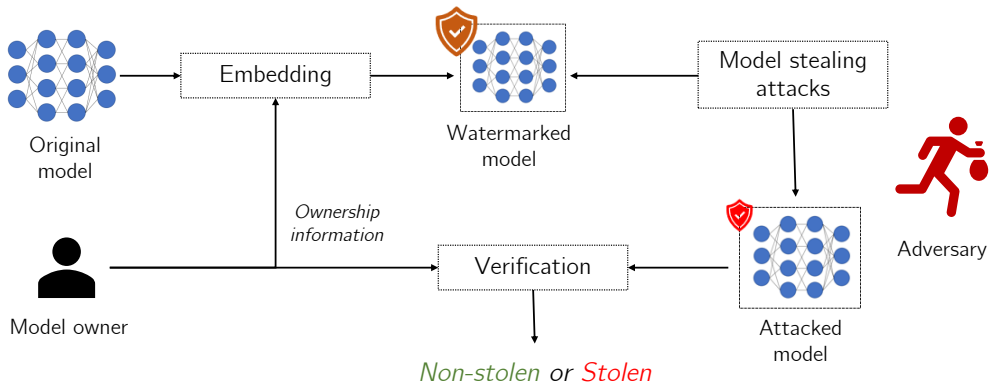


Figure 5.1: Overall watermark process, including *embedding*, *attacks* and *verification*

5.2 Watermarking neural networks

Watermarking can be extended for protecting the intellectual property of neural networks. In this section, we propose a definition of watermarking, identifying the main functions and proposing fundamental properties. The work described in this section is based on the definitions from literature survey by Li et al. [116], Boenisch et al. [117], Regazzoni et al. [118], Lukas et al. [119] and Xue et al. [120].

5.2.1 Threat model

We propose an overview of watermarking, as shown in Figure 5.1. We suppose that a model owner intends to protect his model M_θ against model stealing attacks, described in Section 3 from the adversary $\mathcal{A}$. A defense strategy is to (i) embed in M_θ ownership information, denoted $\mathcal{O}$ and more commonly referred to as *watermark* and (ii) verify the presence of watermark in the model.

Depending on the assumptions on the environment (see Section 5.2.4), the adversary can be constrained on computational resources, model's partial or complete access, time, knowledge of the model and/or training data. Overall, the strategy of the adversary to attack watermarking can include:

- performing the aforementioned model stealing attacks, described Section 3 to steal the model.
- preventing the model owner to verify the presence of the watermark (see Section 5.6.2 for more details)
- modifying the model in such a way that the watermark is corrupted (see Section 5.6.4 for more details)
- obtaining knowledge of the ownership information $\mathcal{O}$

We consider the adversary as being *rational*, meaning that there exists a trade-off for an adversary between the cost of stealing the model (by implementing the aforementioned

strategies) and the cost of training its own model, that should be taken into account into the design of watermarking.

We propose to formally define the main functions of watermarking, namely the *embedding* and the *verification* functions.

5.2.2 Main functions

Definition (Embedding phase)**.** *Let M_θ be the model to be watermarked, with parameters θ. The embedding phase is realized through the Embed(.) function as defined below:*

$$M_{\theta^*} \leftarrow Embed(M_\theta, \mathcal{O}) \tag{5.1}$$

with M_{θ^*} being the watermarked model and $\mathcal{O}$ being ownership information.

Definition (Verification phase)**.** *Let M^* be a suspect model. The existence of the watermark in M^* is verified if the following condition holds:*

$$Verify(M^*, \mathcal{O}) = True \tag{5.2}$$

with $Verify(.)$ being the verification function and $\mathcal{O}$ being the ownership information, computed from the embedding phase.

We identify two types of verification:

1. *Personal verification* where the verification algorithm is executed by the model owner.

2. *Public verification*, where the verification algorithm is executed by a trusted third-party. In the last case, the role of the model owner is to convince the third-party that he or she is the legitimate owner of the model.

In the remaining of the thesis, we refer to *watermarking scheme* as the combination of the two functions `Embed(.)` and `Verify(.)`.

5.2.3 Properties

We introduce several basic properties that a watermarking algorithm needs to satisfy, mainly adapted from image watermarking.

Definition (Fidelity)**.** *The Embed(.) function is ensuring fidelity if the impact of the embedding on the performance of the model is negligible, i.e:*

$$\sigma(M_{\theta^*}, \mathcal{D}) \approx \sigma(M_\theta, \mathcal{D}) \tag{5.3}$$

with M_θ is the original model, M_{θ^*} the watermarked model, $\sigma(.)$ any performance metric such as accuracy, F_1 score, etc. and $\mathcal{D}_{test}$ an evaluation dataset.

The fidelity property ensures that the modification induced by the watermark does not impact the performance of the model, i.e. watermarked and non-watermarked models can be used equally in production environments.

Next, we define three properties adapted from digital watermarking: robustness, integrity and reliability.

Definition (Robustness)**.** *Let M_{θ^*} be the watermarked model. M_{θ^*} is said to be robust to modifications if:*

$$Verify(M_{\theta^*}, \mathcal{O}) == True \implies Verify(M_{\theta^*+\mu}, \mathcal{O}) \tag{5.4}$$

where μ is a small perturbation on the watermarked model's parameters θ^*.

Definition (Integrity)**.** *The watermarking scheme is said to satisfy the integrity property if, for any non-watermarked model M_{θ_i} $\forall\ \theta_0 \in \Theta$, the detection false positive rate ϵ is negligible, with ϵ defined as:*

$$\epsilon = P(Verify(M_{\theta_0}, \mathcal{O}) == True) \tag{5.5}$$

Definition (Reliability)**.** *The watermarking scheme is said to be reliable if, for a watermarked model M_{θ^*}, the detection rate r is close to 1, with r defined as:*

$$r = 1 - P(Verify(M_{\theta^*}, \mathcal{O}) == True) \tag{5.6}$$

Definition (Secrecy)**.** *A watermarking scheme is defined to be secret if there is no optimal strategy for a rational adversary $\mathcal{A}$ to gain sufficient knowledge on the ownership information $\mathcal{O}$ to break the reliability property.*

The robustness property is further described in Section 5.6. In addition to these properties, additional requirements can be introduced, such as *capacity*, quantifying the amount of ownership information embedding into a model, *generality*, measuring the ability for watermarking to generalize to different types of models or *efficiency*, measuring the computational overhead of the watermarking procedure.

5.2.4 Black-box vs. White-Box

Watermarking algorithms are split into two categories: **Black-Box** and **White-Box** watermarking, with respect to the embedding function, to the verification function and to the adversary's environment. In black-box embedding, no assumptions are required on the model's architecture or training process, i.e., embedding algorithms rely on data poisoning attacks [121]. In the case of white-box embedding algorithms, constraints could exist on model architecture (number of layers, neurons, etc.) and can include plain modifications on the weights' values.

Regarding black-box verification, the suspect model to be verified can only be accessed by queries (for instance, a model deployed on an API endpoint). On the contrary, in white-box verification, the suspect model is accessible (with its architecture and parameters) and can be "inspected". It is worth noting that a given watermarking scheme can be

white-box for embedding and black-box for verification, or black-box for embedding and white-box for verification. Even though white-box setting presents less constraints on the watermarking environment, black-box verification is more adapted to real-world situations where the model is rarely accessible to the end-user.

We also refer to "black-box scenarios" or "white-box scenarios" regarding the constraints on the adversary. Black-box scenarios imply that the adversary does not have a direct access to the model (for instance, only queries from an API endpoint) and have restricted access or knowledge about the training data and training procedure. Black-box scenarios are well-suited to describe model extraction attacks for instance. White-box scenarios allow more power to the adversary, with the ability to analyze weights, architecture, or to have a complete access to training data.

In the next section, we propose to describe black-box watermarking, highlighting the main problems and the current state-of-the-art algorithms.

5.3 Black-box watermarking

In a black-box setting, the ownership of a model is verified through inference queries (i.e no inspection of the model's parameters is possible). The concept is to have a particular dataset called **trigger set** $T = \{(x_1^T, y_1^T), (x_2^T, y_2^T), ...(x_{|T|}^T, y_{|T|}^T)\}$, distinct from the training dataset $\mathcal{D}_{train}$ and whose composition is only known to the model owner. More specifically, the *trigger labeling function* $f_T(.) : x^T \rightarrow y^T$ is secret. By design, the watermarked model is the only model to have learned the trigger labeling function, meaning the only model to obtain *high* accuracy on the trigger set. By analyzing the performance of any suspect model on the trigger set, the model owner has a proxy measure for model's ownership. Trigger set generation techniques are further described in Section 5.3.3.

5.3.1 Definitions

We define, more formally, what *high accuracy* means.

Definition (Black-box Verification phase)**.** *Let M^* be a suspect model. The existence of the watermark in M^* is verified, with error rate ϵ, if and only if $Verify(.) == True$, with $Verify(.)$ defined as follows:*

$$Verify(M^*, \sigma(.), T) = \begin{cases} True & if\ \sigma(M^*, T) \geq \beta \\ False & if\ \sigma(M^*, T) < \beta \end{cases} \tag{5.7}$$

with $\sigma(.)$ is a performance metric and β a verification threshold.

The verification threshold β is a generic threshold value to claim the ownership of a model. Szyller et. al [20] defines β as the cumulative binomial distribution.

Definition (Verification threshold for classification)**.** *We define the verification threshold β as follows*

$$\epsilon = \sum_{i=0}^{\lfloor \beta \cdot |T| \rfloor} \binom{|T|}{i} \frac{1}{K^i} (1 - \frac{1}{K})^{|T|-i} \tag{5.8}$$

where $|T|$ is the size of the trigger set, K is the number of output classes for a classification task and ϵ the error rate.

In a black-box setting, we propose a more precise definition of the reliability and the robustness properties.

Definition (Reliability for black-box setting)**.** *For any model M_{θ_i} $\forall$ $\theta_i \in \Theta$, with the exception of the watermarked model M_θ, the following condition holds with error rate error rate ϵ:*

$$\sigma(M_{\theta_i}, T) < \beta \tag{5.9}$$

Definition (Robustness for black-box setting)**.** *Let M_θ be the watermarked model. M_θ is said to be robust to modifications if:*

$$\sigma(M_{\theta^*+\mu}, T) \geq \beta \tag{5.10}$$

where μ is a small perturbation on the watermarked model's parameters θ.

In black-box watermarking, the trigger set constitutes ownership information. Therefore, in order to preserve the aforementioned properties, it is crucial that it remains secret.

Definition (Secrecy & Indistinguishability)**.** *A black-box watermarking scheme is defined to be secret if there is no optimal strategy for a rational adversary $\mathcal{A}$ either (i) to distinguish legitimate inputs $x \in \mathcal{D}$ from trigger inputs $x^T \in T$ or (ii) infer the trigger labeling function $f_T(.)$, sufficiently enough to break the reliability property.*

5.3.2 Embedding

The concept of black-box embedding is to modify the model's parameters without assuming prior-hypotheses, such as model architecture, type of problem solved by the model, etc. We consider two main possibilities for black-box embedding: (i) data poisoning, i.e. modifying the training data and (ii) loss regularization, i.e. modifying how the model learns from the training data. Often, embedding algorithms combine those two options.

The concept of data poisoning is to craft particular inputs called *triggers*, composed of input-output pairs (x_i, y_i), with its composition and the trigger labeling function $f_T(.)$ only known to the model owner. The model to be watermarked is then trained either simultaneously on the legitimate training dataset $\mathcal{D} = \{x_1, x_2, ...x_{|\mathcal{D}|}\}$ and the trigger set or sequentially (the model is first trained on the legitimate dataset, then fine-tuned on the trigger set).

The second modification strategy (other complementary with data poisoning techniques) is to modify the loss function during the training. In this case, we can express the parameters θ^* of the watermarked model as the following optimization equation :

$$\theta^* = \underset{\theta_i}{arg\,min} \left(\sum_{k=1}^{|T|} \mathcal{L}(y_k^T, M_{\theta_i}(x_k^T)) + \lambda \cdot \sum_{k=1}^{|\mathcal{D}|} \mathcal{L}(y_k, M_{\theta_i}(x_k)) \right)$$

(a) Original input [23] (b) $WM_{content}$ [23] (c) $WM_{add-noise}$ [23] (d) $WM_{unrelated}$ [22]

Figure 5.2: Different types of trigger generation techniques

The loss function is split into two terms: One for the legitimate dataset $\mathcal{D}_{train}$, one for the trigger dataset T. Loss regularization techniques can introduce more complex expression for the loss, in order to improve the quality of the learning process, as shown in later sections.

5.3.3 Trigger generation

Different strategies have been implemented in order to generate trigger inputs. [22, 23, 122, 123], as well as selecting $f_T(.)$. We propose to present an overview of various black-box embedding trigger generation techniques

Unrelated inputs

A first strategy is to generate trigger inputs unrelated to the legitimate data. We define this technique as $WM_{unrelated}$ as mentioned by Zhang et al. [23] and also presented in Adi et al. [22]. For instance, one can use handwritten digits as trigger inputs for watermarking a model whose main task is to recognize food, and to associate random outputs. This technique has two advantages:

1. The source of "candidate" trigger inputs is potentially infinite, making it difficult to "guess" which triggers have been used.
2. The trigger dataset and legitimate dataset are well separated, meaning that, intuitively, the performance on the trigger dataset should not impact the performance on the legitimate dataset.

Input modification

A second strategy is to use legitimate dataset as a starting point for building trigger inputs. Therefore, we can define a trigger input generation function $x^T = T_{Gen}(x)$. For instance, the technique $WM_{add-noise}$ consists in adding Gaussian noise to the legitimate inputs:

$$T_{Gen}(x_i) = x_i + z_i$$

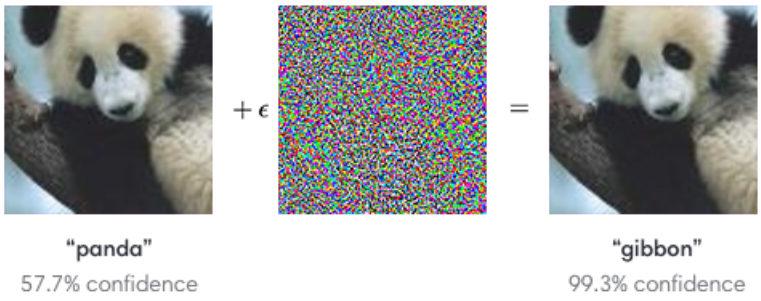


Figure 5.3: An adversarial input, overlaid on a typical image, can cause a classifier to miscategorize a panda as a gibbon [1].

where z_i is drawn from zero-mean normal distribution with variance 1 ($z_i \sim \mathcal{N}(0,1)$), supposing that the legitimate inputs are normalized and standardized. Instead of adding noise, it is possible to add a specific content. This technique, denoted $WM_{content}$, embed data such as special string, image pattern, logo or other artifacts which could help in identifying the model owner.

Adversarial-based triggers

Le Merrer et al. [123] propose to build trigger inputs by leveraging adversarial examples. Adversarial examples are particular inputs, indistinguishable from legitimate inputs to the human eye but causing a misclassification by the model. Figure 5.3 shows how a small noise perturbation can fool an image classifier. In order to create such inputs, the paper proposes to use the **F**ast **G**radient **S**ign **M**ethod (FGSM), and can be expressed as follows:

Definition (FGSM)**.** *For a given input x_i, the resulting trigger input is computed through the gradient of the loss with respect to the legitimate input:*

$$x_i^T = x_i + \nu \cdot sign(\nabla_{x_i} \mathcal{L}(\theta, x_i, y_i)) \tag{5.11}$$

with ν a multiplier, θ the model parameter and $\mathcal{L}$ the loss function.

Other techniques might be used [124], such as the Jacobian-based saliency map technique [125], or the Carlini and Wagner technique [126] to obtain adversarial inputs.

GAN-based triggers

Li et al. [127] present a trigger generation technique based on Generative Adversarial Network [128] (GAN): Trigger inputs are generated through a neural network called generator, taking as input legitimate samples and ownership information (a logo, text, etc.). The generator outputs a candidate trigger sample. Then, a second neural network defined as discriminator receives the candidate trigger sample as input and should learn to recognize the sample as non-legitimate. Both the generator and the discriminator are

trained jointly, until the discriminator cannot distinguish legitimate inputs from trigger inputs.

Other techniques

Other techniques can be used to craft triggers, as presented by Guo et al. [129], suggesting the usage of evolutionary algorithms to generate the *best trigger set*: i.e., a trigger set creating few false positive when tested on models which do not contain the watermark. Liu et al. [130] propose to leverage Fourier perturbation analysis to understand how to modify legitimate inputs (i.e., which type of content should be added and how to generate it) to obtain a good trade-off between robustness and secrecy.

5.3.4 Loss function update

In parallel with trigger generation techniques, black-box embedding algorithms focus on enabling the model to adopt the desired behavior on the trigger set through a modification of the learning parameters, namely the loss function. Embedding algorithms often rely on a specific formulation of the loss function for the model to learn both the original task and the watermark task. Several works [22, 23, 127], consider the same loss function for both tasks. For instance, Bansal et al. [131] take the work in field of *adversarial robustness* [132, 133] by training the model on the trigger set with randomized smoothing, in order to be robust against model modification attacks.

Jia et al. [21] notice that, if the loss function is solely a distinct combination of two loss functions, then the resulting watermarked model is in fact composed of two *sub-models*: a sub-model efficient on the original task and a sub-model on the watermarking task. This type of embedding is not robust against model extraction attacks, because it is always possible to separate the two sub-models. The authors propose to define the loss function associated to the watermarking task as the Soft Nearest Neighbor Loss (SNNL).

Definition (SNNL)**.** *The Soft Nearest Neighbor Loss is defined, over a dataset of size N, for loss parameter called temperature τ:*

$$SNNL(X, Y, \tau) = -\frac{1}{N} \sum_{i \in 1..N} log \left(\frac{\sum\limits_{\substack{j \in 1..N \\ j \neq i \\ y_i = y_j}} e^{-\frac{\|x_i - x_j\|^2}{\tau}}}{\sum\limits_{\substack{k \in 1..N \\ k \neq i}} e^{-\frac{\|x_i - x_k\|^2}{\tau}}} \right) \tag{5.12}$$

Analyzed by Frosst et al. [4], the SNNL measures the distance between data points from different classes, relatively to the average distance for points belonging to the same class. When the SNNL is low, the data is said to be *entangled*: visually, we can represent low-entangled data as well-separated clusters and high-entangled data similar to a scatter plot, as shown in Figure 5.4. Jia et al. [21] argue that maximizing entanglement during the embedding of the watermark (while minimizing cross-entropy loss for the

Figure 5.4: SNNL values for 4-class classification problem, from highly entangled data (left) to highly disantagled data (right), from Frosst et al. [4]

legitimate task) makes the model more resilient to model extraction attacks, by developing class-independent similarity structures, improve performance on unseen data.

Chen et al. [134] suggest to use trigger-based watermarking not to watermark the complete model, but a special sparse subnetwork inside the model called *winning lottery ticket*. A winning ticket submodel has fewer connections (hence better computational performance) but reach similar or even better performance than the original model. Since finding the winning ticket model is non-trivial, inserting the watermark inside prove to be efficient against ambiguity or removal attacks, later defined in Section 5.6.

5.3.5 Verification

In addition to embedding algorithms, a significant number of works implement advanced verification algorithms to go beyond a simply accuracy computation. The main constraint of black-box verification algorithms is the scarcity of available information, which mostly consists of output vector's predictions. We present works which focus particularly on the watermarking verification phase.

Szyller et al. [20] proposes **D**ynamic **A**dversarial **W**atermarking of **N**eural Networks (DAWN), in order to be resilient against model extraction attacks. In short, the authors propose to deploy the model to be protected on an API endpoint, with a filtering module analyzing inputs. The filtering module implements a boolean function $\mathcal{W}(x_i)$, with x_i as input query and defined as follows:

$$\mathcal{W}(x_i) = \begin{cases} 1 & if\ HMAC(K_{M_\theta})[0.127] < |T| \cdot 2^{128} \\ 0 & otherwise \end{cases} \tag{5.13}$$

where $HMAC()$ is a keyed-hash message authentication code and $|T|$ the size of the trigger set. When $\mathcal{W}(x_i) = 1$, the input query does not pass the filtering module, the input does not reach the model and the input is wrongly classified. Therefore, if a model extraction attack is implemented, then the surrogate model will be trained on those misclassified samples (hence containing the watermark).

Charette et al. [135] propose to modify the output prediction vector of the watermarked model M_θ, by incorporating (through a modification of the loss function during the training), for a given label $i \in [1, K]$, a sinusoidal function with secret parameters (f_w, v), with λ as a regularization parameter:

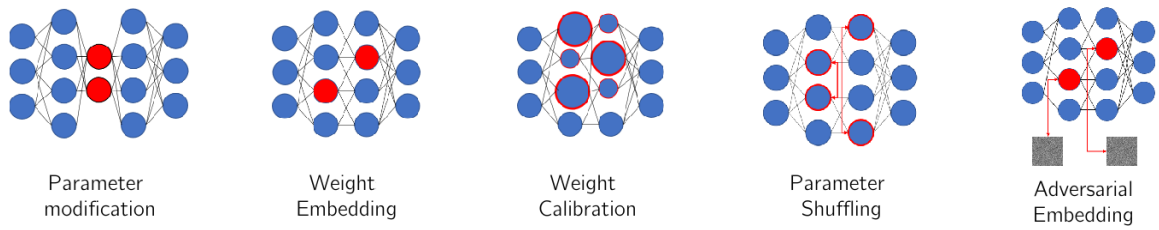


Figure 5.5: The 5 categories of white-box embedding

$$M_{\theta^*}(x) = M_\theta(x) + \lambda \cdot cos(f_w \cdot v^\intercal x) \tag{5.14}$$

Thus, during the verification phase, an algorithm based on power spectrum of the prediction vectors can extract the *signal* of the watermarking, through the signal-to-noise ratio.

Sakazawa et al. [136] suggest to use the prediction vectors obtained after the inference of trigger images to reconstruct an output image. The image reconstruction process can be done incrementally and provides a visual proof of the ownership, as opposed to the more theoretical assessment with the β threshold.

5.4 White-box watermarking

In white-box watermarking, the model owner is not only limited to modifications to the training data or the loss function, but has an easier access to the model (architecture, weights distribution, specificity of the task solved, etc.), both during the embedding and the verification phases. Even though white-box watermarking require stronger assumptions, it offers more possibilities to the model owner. We propose a review of the main white-box watermarking algorithms.

5.4.1 Embedding

We propose a classification of white-box embedding algorithms into 5 categories, as shown in Figure 5.5: parameter modification (adding or removing parameters from the model), weight embedding, weight calibration, weights substitution and adversarial embedding.

Parameter modification

A first white-box embedding concept is to add layers into the model in order to authenticate the ownership. Fan et al. [137] propose to add a *passport layer* into the model i.e., an additional layer, inserted after a convolutional layer, in order to make watermarking robust against forging attacks (later described in Chapter 7). For each additional passport layer, the model owner has ownership parameters $P^l = \{P^l_\gamma, P^l_\beta\}$ and the output of the convolutional-passport layer is y^l, defined as follows:

$$y^l = a\left(\gamma^l \cdot (W^l * X^l) + \beta^l\right)$$
$$\gamma^l = \texttt{AvgPool}\left(W^l * P_\gamma^l\right)$$
$$\beta^l = \texttt{AvgPool}\left(W^l * P_\beta^l\right)$$

where $a(.)$ is the activation function and $\texttt{AvgPool}(.)$ is the average convolutional pooling function.

The ownership parameters γ^l and β^l are derived from the passport layer parameters $\{P_\gamma^l, P_\beta^l\}$, which are only known to the model owner. During the training, the model learns the original task, with a loss regularization regarding the scale factor. Once the model learns the original and the watermark task, the passports are removed. The verification phase is white-box, where the model owner inserts the passport layers into the suspect model. If no deterioration in accuracy is observed, then the suspect is the original. The advantage of this technique is that passport layers cannot be forged, making it a good defense against forging attacks.

Weight embedding

A second concept consists of, rather than adding specific parameters, directly embedding in already existing model's parameters, ownership information [138]. Uchida et al. [139] propose to consider a given $|T|$-bit vector $T \in \{0,1\}^{|T|}$, defined as $T = \{T_1, T_2, ...T_{|T}\}$, to be embedded into a convolutional layer θ^l.

In the embedding phase, the algorithm takes an embedding parameter denoted $P \in \mathbb{R}^{|T| \times |\bar{\theta}}$ and a flatten version of θ^l, denoted $\bar{\theta}$. If the embedding is successful, the algorithm should be able to reconstruct the original $|T|$-bit vector, by measuring T^{mes}:

$$T_i^{mes} = s(\sum_{j=1}^{|\bar{\theta}|} p_{ij} \cdot \bar{\theta}_j)$$

with $s(.)$ defined as the step function:

$$s(x) = \begin{cases} 1 & if x \geq 0 \\ 0 & otherwise \end{cases} \tag{5.15}$$

The embedding is considered as a binary classification problem (detecting the watermark in $\bar{\theta}$) with a single-layer perceptron architecture (without bias). Therefore, the overall loss function of the watermarked model can be written as a combination of the original loss function $\mathcal{L}_0$ (for the main task) and the watermark loss function $\mathcal{L}_{WM}$ (for parameters embedding):

$$\mathcal{L} = \mathcal{L}_0 + \lambda \cdot \mathcal{L}_{WM}$$

.

Wang et al. [140] extends on this work by proposing to use a secondary neural network, trained simultaneously with the original model, with the same loss function. This secondary model is designed to never be released and only for watermark verification purposes. However, both approaches are sensitive to approaches aiming to detect a watermarked-embedded layer in a neural network by analyzing the distribution of the weights, as shown in Wang and Kerschbaum. [141]. Liu et al. [142] propose a formulation for the watermark loss function $\mathcal{L}_{WM}$ through the residuals of important parameters of the model:

$$\mathcal{L}_{WM} = \sum_{i=1}^{|T|} \sigma(\mu - T_i \phi_i) \tag{5.16}$$

where T is a |T|-bit ownership vector, μ a threshold parameter and ϕ the *greedy residuals*, selected by analyzing the most important parameters (i.e., the parameters contributing the most to the predictions).

Rouhani et al. [143], extended in Chen et al. [144] propose to encode watermark into a model's weights, by learning the probability distribution function (pdf) of model activation, generating adequate watermark, then fine-tuning the model through loss regularization. Pagnotta et al. [145] propose a weight-embedding algorithm based on code division multiple access spread-spectrum channel-coding, in order to encode watermark information in weights, making the technique more robust to model modifications. Lao et al. [146] propose to apply weight modification a selected subset of the weights (corresponding to strong gradient values during the training) and to only modify these weights to embed the watermark.

Weights calibration

Nambda et al. [122] builds on the $WM_{unrelated}$ idea of Zhang et al. [23] and Adi et al. [22] by solving the problem of separability and proposing to use, as trigger inputs, legitimate inputs from other classes. For example, in a dog vs. cat classification, one can take a particular *dog input* and to associate the label *cat*. Therefore, when training the model on this switched-label dataset T, *dog* inputs would be predicted correctly, except for the inputs with switched labels. The problem of separability and distinguishability disappears, because the trigger inputs are legitimate. However, this idea does not fulfil Definition 5.2.3 regarding robustness (a slight modification of the model would decrease accuracy on the trigger set). The authors propose to *weight* parameters contributing to the prediction in order to be resilient to model modification. More formally, recall the output of the l^{th} hidden layer from Equation 2.3:

$$h^{(l+1)} = a^{(l)} \left(g^{(l)} l(h^{(l)}, \theta^{(l)}) \right) \tag{5.17}$$

We recall the definition $\theta^l = \{\theta_1^l, \theta_2^l, ..., \theta_{|\theta^l|}^l\}$. After training the model on the legitimate and the switched-label dataset, Equation 2.3 is replaced with the **E**xponential **W**eighting function ($EW(.)$), given by the following equation:

$$h^{l+1} = a^l \left(g^l(h^l, EW(\theta^l, T)) \right) \tag{5.18}$$

with $EW(.)$ defined as follows:

$$EW(\theta^l, T) = \frac{\exp(|\theta_i^l|)}{\max_i \exp(|\theta_i^l|) \cdot T} \cdot \theta_i^l \tag{5.19}$$

After the replacement happens in the equation of all layers, the model is re-trained on both dataset and the model is considered as being watermarked.

Parameter shuffling

An active white-box embedding is defined as *weight encryption* [147]. The idea is store the model by encrypting or shuffling the model's parameters, making the model useless. If the model in the shuffled state is stolen, the thief will obtain wrong predictions and the model cannot be used. The model owner possesses the invert key, to obtain the model in a non-shuffled state. Lin et al. [148] propose a shuffling based on Arnold's cat algorithm, whereas Alam et al. [149] propose a shuffling algorithm based on S-boxes. The watermarking is said to be active because it actually prevents the thief from using the model, rather than simply detecting a stolen instance. Besides, as opposed to classical watermarking technique, no embedding is necessary meaning no modification to the model's behavior is applied (once the model is reverted to non-shuffled state).

Adversarial weight modification

In order to make weight embedding less detectable by anomaly detection techniques, Wang and Kerschbaum. [150] propose a technique inspired by Generative Adversarial Networks (GAN) to perform weight modification. The solution is composed of two components: (i) the model to be watermarked M_{θ^*} (ii) a model denoted M_{det} whose role is to identify if the weights' distribution of M_{θ^*}, w is similar to weights of non-watermarked models w_{non}. In the learning procedure, three objectives are defined: (i) to learn the original task of the model, by minimizing the loss $\mathcal{L}_0$ (ii) to learn the watermark task by minimizing the watermark loss $\mathcal{L}_{wm}$ (iii) make sure the weights are not detectable by maximizing the loss of M_{det}. The objectives can be summarized as the following optimization problem:

$$\hat{\theta} = \max_{\theta}(log(M_{det}(w_{non}, \theta)) + log(1 - M_{det}(w, \theta))) \tag{5.20}$$

$$\hat{w} = \min_{w}(\mathcal{L}_0 + \lambda_1 \mathcal{L}_{wm} - \lambda_2 log(M_{det}(w, \theta))) \tag{5.21}$$

where λ_1 and λ_2 are two regularization parameters.

In a similar fashion, Jia et al. [151] use a GAN-like architecture to watermark deep transfer models, by first finding trigger inputs from an encoder-decoder model before embedding the watermark in a sub-network of the model through loss regularization.

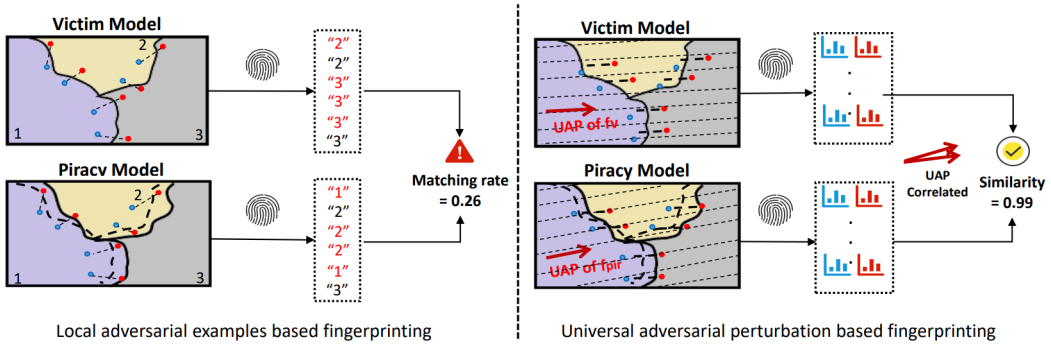


Figure 5.6: Difference between Adversarial examples based fingerprinting and UAP based, from Peng [5]

5.5 Fingerprinting

While reviewing watermarking techniques, it is important to mention another related field, namely fingerprinting. As opposed to watermarking which is an invasive technique (inserting information into a model to identify its owner) causing an accuracy drop, fingerprinting extracts a unique identifier, denoted *fingerprint*, to differentiate it from other models. Similar to watermarking, fingerprinting algorithms are decomposed into two phases: extraction and verification.

Definition (Extracting phase)**.** *Let M_θ be a model, with parameters θ. The fingerprint f_{M_θ} is extracted through the $Extract(.)$ function defined as:*

$$f_{M_\theta} \leftarrow Extract(M_\theta) \tag{5.22}$$

The fingerprint verification is analogous to the watermarking verification algorithm, where the ownership information $\mathcal{O}$ is replaced by the fingerprint f_{M_θ}. The same previously defined properties such as fidelity, robustness, integrity, reliability, are also required. The goal of fingerprinting algorithms is to devise extraction algorithms in order to satisfy, especially the integrity property: the extracted fingerprint should be unique to the model.

Cao et al. [152] suggests that a model is uniquely represented by its decision boundaries, i.e., the region of input space where the output label of a model is ambiguous. Therefore, different models have different decision boundaries. Data points are considered on the model's decision boundary if the model cannot decide which label to attribute to the data point, i.e., at least two labels have a large and equal probability. Such data points can be considered as fingerprints to identify the model. The authors also compare their method with adversarial-based fingerprinting, where the decision boundary can be identified through adversarial examples generative method such as the Fast Gradient Sign Method (FGSM) [153], the Iterative Gradient Sign Method (IGSM) [154] or the

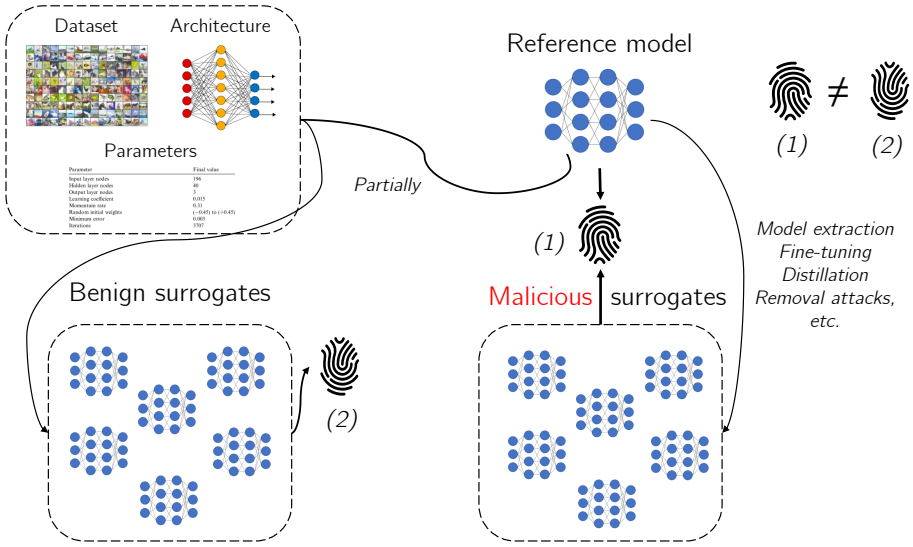


Figure 5.7: Differences between benign and malicious surrogates, resulting in different fingerprints

Carlini and Wagner's (CW) method [126]. The idea of using adversarial examples for fingerprinting model has been developed by Dong et al. [155], based on the work of Hong et al. [156], with an application to multi-exit models ; This type of model is composed of a backbone-core model with several internal classifiers, returning early-outputs in the case where model is confident enough with the prediction. By crafting fingerprints (generated through the CW method) which do not return early-exists and by analyzing the inference time of a given model on fingerprints, it is possible to identify a model through inference time.

An important concept in fingerprinting is the ability to distinguish, for a given reference model, *malicious surrogate models* and *benign surrogate models*, as shown in Figure 5.7. Malicious variant models refer to the family of models showing a similarity to the reference model due to malicious actions by an adversary on a stolen reference model: fine-tuning, weights modifications, model extraction etc. Fingerprints of the reference model and its malicious variants should be the same. On the other hand, benign variant models refer to the family of models showing some sort of similarity to the reference model (due to similarities in training procedure, training data, etc.) without being a stolen version. In this case, their fingerprints should differ from the reference model. Maho et al. [157] develop the concept of *model family*, proposing a family identification method from benign inputs. Lukas et al. [158] introduce the concept of conferrable adversarial examples, which are adversarial examples, that transfer from a reference model to its surrogates, in order to extract fingerprints. Conferrable examples allow to identify not only the model but also its surrogates and are by design more robust to extraction attacks and modification attacks. To go beyond adversarial example inputs to craft fingerprints, Peng et al. [5] propose the concept of Universal Adversarial Perturbation (UAP) [159], which suggest the existence of a particular perturbation vector v which can fool the

model on almost all data points. The authors suggest to train an encoder E_ϕ, mapping the fingerprints to a latent space before computing the cosine similarity between $E_\phi(M^*)$ and $E_\phi(M_\theta)$ where M^* is the suspect model and M_θ the model to be protected. High similarity implies that the suspect is the stolen version. The advantage of their proposal is that it does not detect two homologous models as stolen (i.e., two models trained on the same training data). Wang et al. [160] define the concept of characteristic examples, which are particular inputs designed to be robust for a given base model and transferable to its pruned version.

Yang et al. [161] also identify the problem of robustness for adversarial examples and propose to fingerprint by meta-training, rather than by identification of the decision boundary. For a given model to protect, they obtain a pool of shadow models, generated through model-augmentation techniques. Through meta-training, they ensure to generate data points only the model to protect and its derived models can recognize. This technique, similar in some ways to [158], is designed to be robust against model modification such as weight pruning and weight "noise addition" attacks.

Chen et al. [162] propose a global framework for assessing the intellectual property of models, by compute fingerprinting based on several criteria, either in a black-box manner with the robustness to adversarial examples or through the Jensen-Shanon Distance [163] or in a white-box manner using the neuron distance or the neuron activation distance. Overall, the proposed framework is a voting system based on identification metrics $\Lambda = \{\lambda_1, \lambda_2, ..\}$

$$p_{copy} = \frac{1}{|\Lambda|} \sum_{\lambda \in \Lambda} \mathbb{1}(\lambda(M_\theta) \leq \tau_\lambda) \tag{5.23}$$

where $\lambda(S)$ is the metric applied to M_θ, τ_λ the threshold for metric λ and $\mathbb{1}$ the indicator function. If $p_{copy} > 0.5$, the model is identified.

5.6 Model stealing attacks against watermarking

After introducing the main watermarking techniques, we propose an overview of several attacks against watermarking.

5.6.1 Condition of success

Definition. *A rational adversary $\mathcal{A}$ is considered to have successfully implemented a model stealing attack on a model M_θ, watermarked with ownership information $\mathcal{O}$ if the following condition holds:*

$$Verify(M^*, \mathcal{O}) = False \tag{5.24}$$

where M^* corresponds to the stolen model after the attack.

In addition to model extraction attacks already presented in Chapter 3, an adversary can implement a variety of attacks, either in a black-box or in a white-box setting.

5.6.2 Black-box attacks against watermarking

We classify black-box attacks into three types of attacks: anomaly detection, input preprocessing and voting attacks, as displayed in Figure 5.8 and Figure 5.9.

Anomaly detection

We consider the following scenario: Adversary $\mathcal{A}$ has stolen a watermarked model M_{θ^*} and aims to benefit from it, by deploying on its own API endpoint for instance. However, it is possible that the original model owner decides to perform verification queries to the stolen model, by observing the behavior of the model on trigger inputs. A first strategy consists in identifying trigger queries from legitimate ones. Indeed, let us suppose that the adversary is able to implement a filtering function $F_{filt} : \mathcal{X} \rightarrow 0, 1$, returning 1 if the input belongs to the trigger set with a success probability β, corresponding to the verification threshold:

$$P\left(F_{filt}(x) = 1 \mid x \in T\right) > \beta$$

Therefore, the model M^* deployed by the adversary is defined as :

$$M^*(x) = \begin{cases} M_{\theta^*}(x) & if F_{filt}(x) = 1 \\ random() & otherwise \end{cases} \tag{5.25}$$

where $random(.) : \{\emptyset\} \rightarrow \{1, ..., K\}$ and K is the number of classes. The difficulty of implementing the attack is to be able to compute F_{filt}, because it requires the adversary to have some knowledge about input data. Hitaj et al. [164] propose to build such filtering function, in the case of *unrelated trigger inputs*. First, the adversary gathers a *filtering* dataset, containing input data and computer-generated input. The dataset is transformed, by passing each item of the filtering dataset through hidden layers of the stolen model, to obtain feature vectors. The idea is that the feature vectors contain information regarding characteristics belonging to legitimate inputs. The feature-transformed dataset is then used to train a fully-connected neural network, the filtering model F_{filt}. The attack is successful and obtains an accuracy between 92 % and 96 % in detecting trigger inputs. However, this attack highly depends on the dataset chosen, i.e. with prior knowledge of potential trigger generation techniques.

Jia et al. [21] propose to use anomaly detectors, namely Local Outlier Factor (LOF) and Isolation Forest, on data passed through the activations of the last hidden layer of the watermarked model. The attack is successful as it obtains an accuracy above 90 %, at the cost of performance on legitimate data around 8 % (due to legitimate inputs wrongly identified as triggers).

Input preprocessing

A second strategy consists in preprocessing all inputs received, for instance by adding noise, blurring, cropping, flipping or applying compression algorithms. By doing so, the adversary intends to remove key features in triggers inputs while preserving features (at

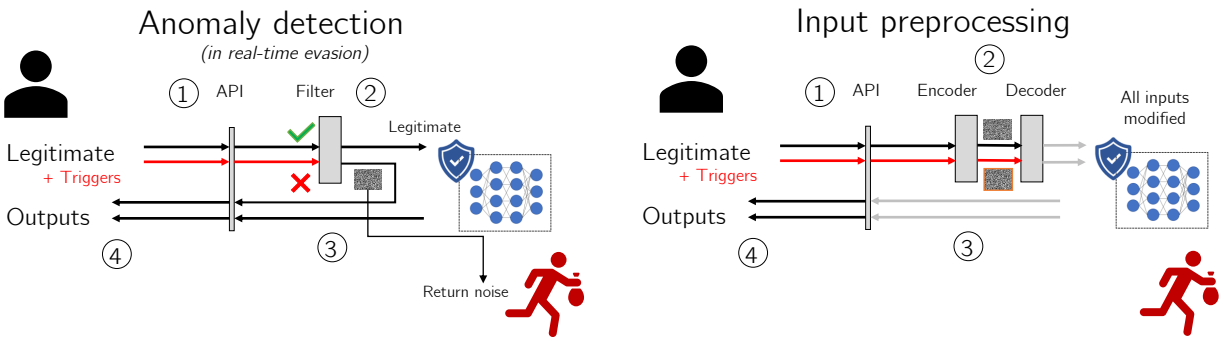


Figure 5.8: Two black-box attacks: anomaly detection and input preprocessing attacks

least decisive features for classification) on legitimate inputs. This attack aims to find a trade-off in the intensity of the preprocessing: if the input is modified too strongly, the accuracy on the legitimate task will be impacted. On the contrary, if the input is too softly modified, trigger inputs will be be affected. For instance, Lin et al. [165] propose to use an autoencoder to compress and reconstruct inputs.

Trigger reverse-engineering

A third strategy is to *reverse-engineer* the trigger inputs, as shown by Wang et al. [166]. By supposing constraints on the trigger inputs (they have to be constructed through the $WM_{content}$ technique, and limited to a small part of the input). Once the triggers are identified, the model is retrained.

Voting technique

A fourth attack is the *voting technique*, as developed by Charette et al. [135] and Hitaj et al. [164]. Let suppose that the adversary successfully stolen k models $\{M_{\theta_1^*}, M_{\theta_2^*}, ..., M_{\theta_k^*}\}$, each of them watermarked. The adversary can deploy the k models simultaneously, meaning that, for a given input x, the adversary returns the average of the models' outputs, $y^* = \sum_{i=1}^{k} M_{\theta_1^*}(x)$. The idea is that, if the owner of model M_{θ^*} sends verification queries belonging to the trigger set T_1, the output will be "diluted" through the average with other models (the greater the number of models k, the most efficient the dilution).

The main advantages of the attack are its simplicity of implementation and its efficiency. However, it requires that the adversary has an access to several stolen models, which could be problematic in some cases.

5.6.3 White-box attacks against watermarking

In the context of white-box attacks, the adversary has an access to the model's weights, meaning that more advance manipulations can be performed. We categorize white-box

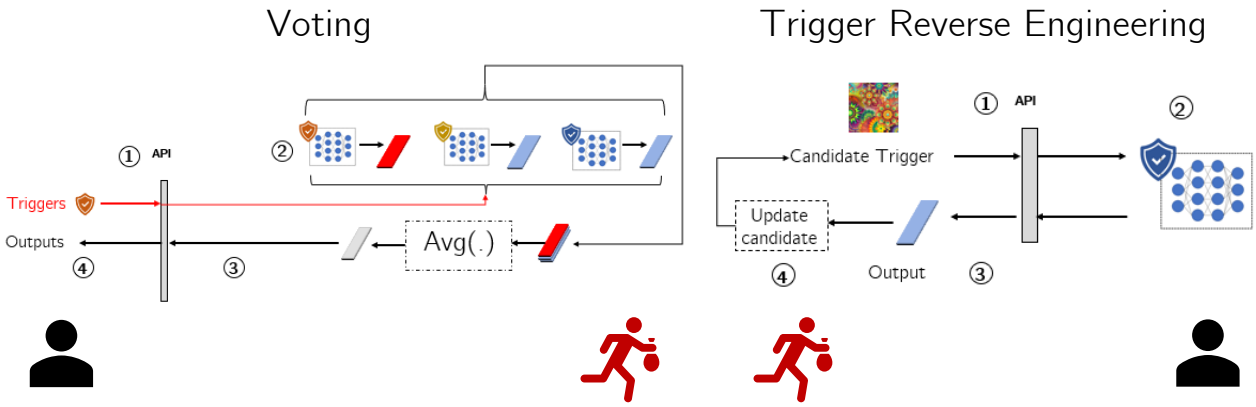


Figure 5.9: Trigger reverse-engineering and voting attacks

attacks into two types of attacks: *shuffling* and *pruning attacks*.

Shuffling attacks

Shuffling attacks consist in re-arranging the model's neurons to remove important connection tp the watermarking task while preserving the accuracy on legitimate data. This shuffling attack has been mentioned in Section 5.4.1 as a protection measure, when the shuffling can be reverted only by the model owner. Li et al [167] suggest a solution against shuffling attacks by designing trigger inputs specifically created to identify neurons' orders and mitigate any shuffling.

Pruning attacks

The second white-box attacks is *pruning* [168]. The idea is that highly activated neurons contributing more to prediction than low activated neurons. Pruning consists in removing neurons which contributes "less" to the predictions of the model while preserving the performance. Several methods exist to judge whether or not a neuron is "contributing" to predictions, such as ranking neurons by the magnitude of their weights (i.e. L2 norm) or of their activations. Pruning consists in removing the less contributing neurons according to a percentage threshold.

In the case of watermarking, an assumption is made that low activated neurons contain information regarding the watermarking task. Therefore, pruning those neurons will "remove" the watermark. The pruning process can also be iterative: first prune the model, then re-train the model on a subset of the training data, before pruning it again, etc. Pruning can also be combined with other attacks, such as the reverse-engineering attack as shown by Aiken et al. [169].

5.6.4 Hybrid attacks against watermarking

In addition to attacks which are either purely black-box or white-box, we consider attack strategies which have been implemented in both settings in prior works. We identify two types of attacks: ambiguity attacks and removal attacks.

Ambiguity attacks

To begin with, we consider the case of *ambiguity attacks* [170–172]. A particular type of ambiguity attack is defined as *forging attacks*, where the goal of the adversary to falsely claim the ownership of a given model to a third-party.

Definition (Watermark forging attack)**.** *An adversary A^*, acting as as agent, is implementing a watermarking forging attack against a target model M_{target} if he manages to craft a malicious trigger set, such as:*

$$Verify(M_{target}, T^*) = True \tag{5.26}$$

The role of the third-party is to judge whether or not the ownership is forged. To achieve this goal, the adversary can (i) reverse engineer the ownership proof (by reverse engineering the trigger set for black-box watermarking for instance) (ii) create a fake ownership proof which acts like a real proof. In Chapter 7, we explore more deeply forging attacks.

Another type of ambiguity attacks is overwriting, where the adversary insert into the model another watermark, to confuse a potential third-party judge, to decide which watermark was originally in the model first, as described in Fan et al. [137].

Removal attacks

The second type of hybrid attacks and perhaps the most evaluated in the literature is called *removal attacks* and concerns any model modification strategies in order to remove the watermark while preserving the behavior on legitimate data. Several computationally efficient strategies can be implemented, such as rounding (reducing the precision of the parameters) or pruning, as previously mentioned. Three particular removal strategies are studied in the literature: fine-tuning, transfer learning and distillation.

Fine-tuning is the concept of further training the model on a particular dataset. In traditional machine learning, it is often used to improve the performance of the model and require less computational resources and training data. In the case of removal attacks, by fine-tuning the model on legitimate data, the adversary hopes to update the parameters and to remove watermark information. Different fine-tuning strategies could be considered, such as: fine-tuning the last layer only (the rest of the parameters are frozen), completely re-training the last layer by initializing the weights randomly, or re-train all the model.

Chen et al. [173] notice that previous fine-tuning attacks are not effective for removal due to the small learning rate of the training, leading to small modification of the weights. However, if the fine-tuning is "stronger" (by increasing the learning rate), then models

phenomenon called the *catastrophic forgetting* appears. When a model is trained on a series of tasks sequentially (the original training, the fine-tuning removal, etc.), this model could forget how to perform previously trained tasks after training the new ones. In other words, the model "forgets" part of the legitimate training data during fine-tuning. The authors propose to identify which weights contribute the most to the legitimate task through a metric called the *Fisher information matrix* denoted F_i. They introduce a regularization term, which takes into account the "memory" of parameters for the legitimate task:

$$\mathcal{L} = \mathcal{L}_0 + \frac{\lambda}{2} \sum_i F_i(\theta_i - \theta*_i) \tag{5.27}$$

where θ^* corresponds to the initial weights of the watermarked model and θ the weights to be learn for the fine-tuning procedure. This fine-tuning attack is efficient against several watermarking algorithms [22, 23, 122, 123].

In transfer learning, the goal is to use the knowledge acquired on a given task in order to solve a different problem. For instance, in the case of computer vision tasks, it is possible to use a model trained on an object recognition dataset such as CIFAR10 to solve a different task such as the image recognition task STL-10, in order to save training time. In the case of watermarking, transfer learning can be used to modify the parameters of the model and thus removing the watermark. Chattopadhyay et al. [174] present a white-box attack through transfer learning on synthetic data, generated by a GAN.

Distillation is the concept of transferring information from a very complex model (with a high number of parameters) to a smaller model, which is very useful in a resource-limited environment. During this "teaching" process, the watermark contained in the original model might not be contained in the new model. Yang et al. [175] evaluate the efficiency of distillation attacks against different watermarking embedding algorithms [22, 139], and propose some countermeasures against this type of attack.

Yan et al. [176] propose a removal attack against white-box watermarking beyond fine-tuning, distillation or pruning, by observing that white-box watermarking algorithms are based on correlation between local features (absolute value or sign) of neurons and the watermarking information. Thus, by introducing invariant transformation to the model parameters, such as shuffling parameters of a convolutional layer or scaling neurons, they break this correlation (and thus the watermark verification) while preserving the model behavior on legitimate data. The combination of proposed attacks appears to be successful on several white-box algorithms [134, 137, 139, 142, 143, 150, 170, 177],

5.7 Other application domains

We presented various embedding and verification algorithms, both in black-box and white-box settings, mainly adapted for image classification. Watermarking can be extended beyond this type of class with application to reinforcement learning [178] or natural language processing [179], which will be further studied in Chapter 6. It is also worth

noting that watermarking can be extended to distributed learning [180,181] and to dataset watermarking [182].

5.8 Conclusion

In this chapter, we presented an overview of state-of-the-art watermarking techniques, including formal definitions and properties, categorization of algorithms based on the black-box / white-box environment and fingerprinting. Furthermore, we introduced the main attacks against watermarking in a black-box, white-box and hybrid environment. In the next chapters, we propose to build on these definitions to complete missing points, regarding (i) the watermarking of non-classification and non-image-data tasks in Chapter 6, (ii) the protection against watermark forging attacks in the context of model hosting platforms in Chapter 7 and (iii) watermarking through fairness in Chapter 9.

Chapter 6

Watermarking beyond Classification

As shown in the previous chapter, watermarking machine learning models is a developed field with a diversified number of algorithms (both black-box and white-box) over the past few years, mostly targeting image classification models. Indeed, it is common to evaluate and benchmark watermarking algorithms on standardized image classification datasets, such as the MNIST [183] or CIFAR10 [184] datasets for reproducibility purposes. However, image classification tasks do not constitute the majority of problems encountered in real-life use-cases ; problems related to natural language processing (sentiment analysis [102], translation [185], text summarization [15]) or time series forecasting (in the case of stock market predictions [12] for instance) are under-represented when it comes to evaluate and to benchmark watermarking algorithms. We propose to fill this gap by proposing a comparative study of black-box watermarking for image classification, machine translation, regression and reinforcement learning tasks with an emphasis on trigger generation techniques and robustness to attacks.

The work described in this chapter has been published under the title *Yes We Can: Watermarking Machine Learning Models Beyond Classification* [8], 34th IEEE Computer Security Foundations Symposium 2021, *S. Lounici, M. Njeh, O. Ermis, M. Önen, S. Trabelsi.*

6.1 Introduction

Watermarking as a solution for IP protection has been extensively studied in the last few years, either from a defense point of view (i.e. finding embedding and verification algorithms to improve properties such as fidelity or robustness) or from an attack point of view (i.e. underlying the weaknesses of current watermarking schemes in order to provide countermeasures). When new solutions or attacks are introduced, authors provide experimental results, comparing their proposed solution to prior work, often proposing benchmarks on common datasets, model architectures, training hyperparameters to make their results reproducible. At the time when this work was published (in 2021), we conducted a thorough analysis of the state-of-the-art of watermarking algorithms and observed two key weaknesses in current evaluation processes (i) an over-representation of classification tasks (89 % of the evaluation processes is watermarking classification models)

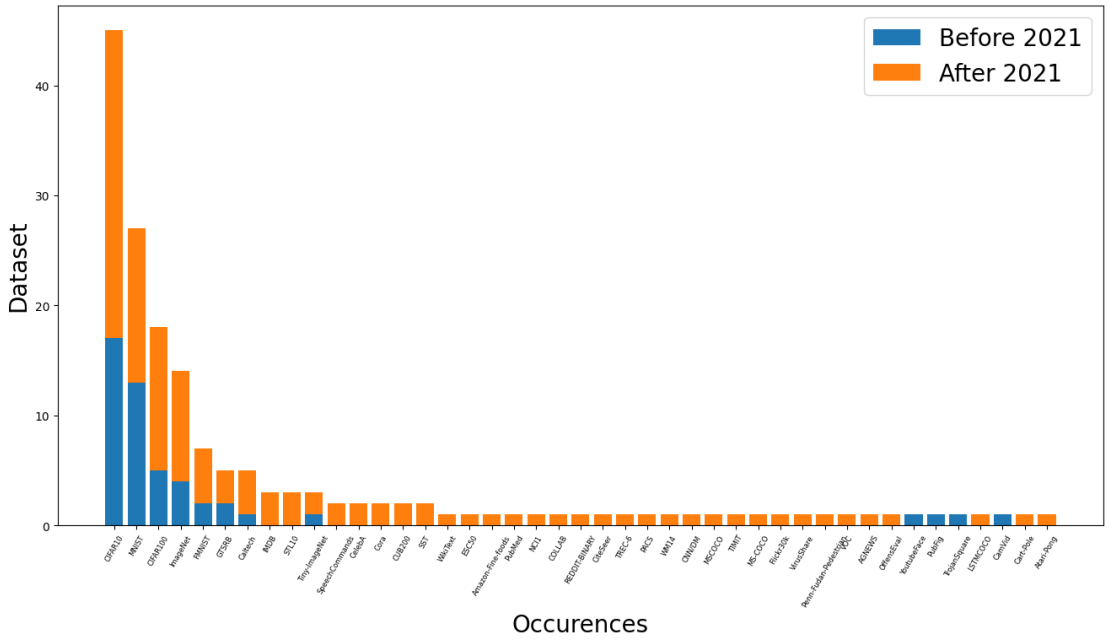


Figure 6.1: Number of experimental evaluation per dataset in the watermarking literature review before and after 2021.

(ii) an over-representation of computer vision tasks (82 % of the evaluation processes, including trigger set generation, is concerning watermarking computer vision models). In Figure 6.1, we plot the number of occurrences for a given dataset in evaluation process before 2021 and after 2021, underlying the lack of diversity of evaluation dataset for watermarking. In Figure 6.2, we display the part of evaluation processes for computer vision tasks and classification (before and after 2021).

The over-representation of image classification tasks poses several issues regarding the ability for watermarking algorithms to generalize to other types of models and tasks, as well as the usability of the theory of watermarking for concrete applications in the domain of natural language processing or financial forecasting.

With this aim, we propose to extend watermarking beyond image classification tasks, by proposing a comparative study of trigger generation techniques and robustness evaluation for different types of models. Overall, our main contributions are summarized as follows:

(1) First, we revisit the definition of watermarking introduced in Chapter 5, for different types of ML models with an emphasis on the verification threshold definition. In particular: Machine-Translation (MT) models, regression models and reinforcement learning models.

1. We particularly focus on black-box watermarking, by proposing three different trigger set generation techniques compatible with image and text data, adapted from techniques introduced in Section 5.3.3.

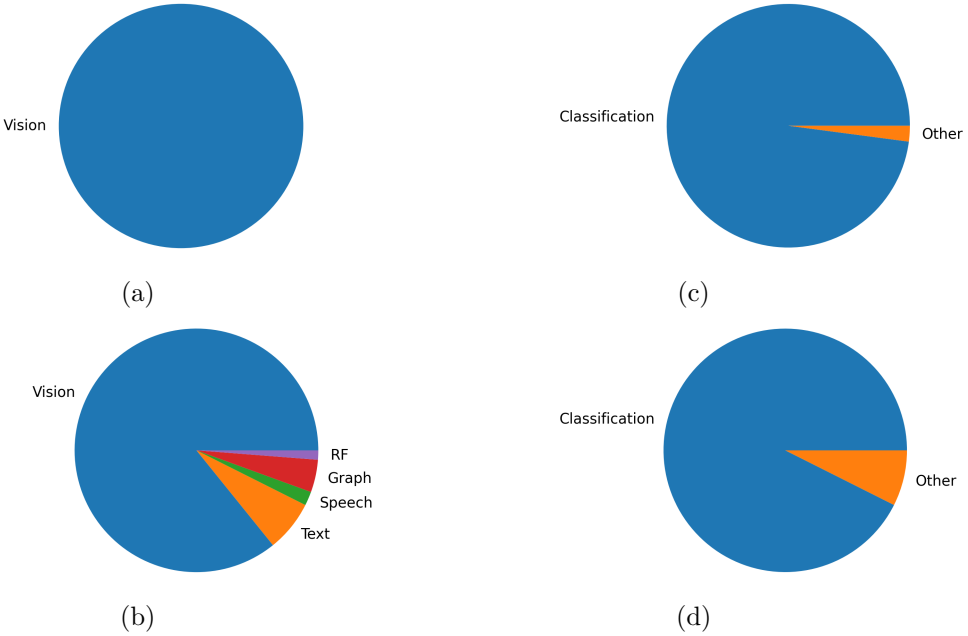


Figure 6.2: Distribution between vision and non-vision tasks (before 2021 for 6.2a, after for 6.2b and between classification and non-classification tasks (before 2021 for 6.2c, after for 6.2d))

2. To assess the quality of the watermarked models, we propose a study of various black-box attacks like anomaly detection or watermark removal attacks, as presented in Chapter 5, adapted beyond image classification.

3. Finally, we benchmark the performance and the quality of the trigger set generation techniques when applied to each particular type of model and assess the success of the adversary. We also present countermeasures for the attacks to evaluate the cost for the adversary.

6.2 Reinforcement learning model

In addition to watermarking image classification models (described in Chapter 5), we propose to extend watermarking beyond these types of models. For this purpose, we propose to watermark regression models (defined in Section 2.2), Machine Translation models (defined in Section 2.4.2) and reinforcement learning models, defined below.

Reinforcement learning is a category of machine learning techniques where systems are trained by trial and error (by receiving virtual "rewards" or "punishments"). We already presented a reinforcement learning algorithm, Q-learning, in Chapter 4 for the data augmentation task for source code analysis. In this chapter, we present another

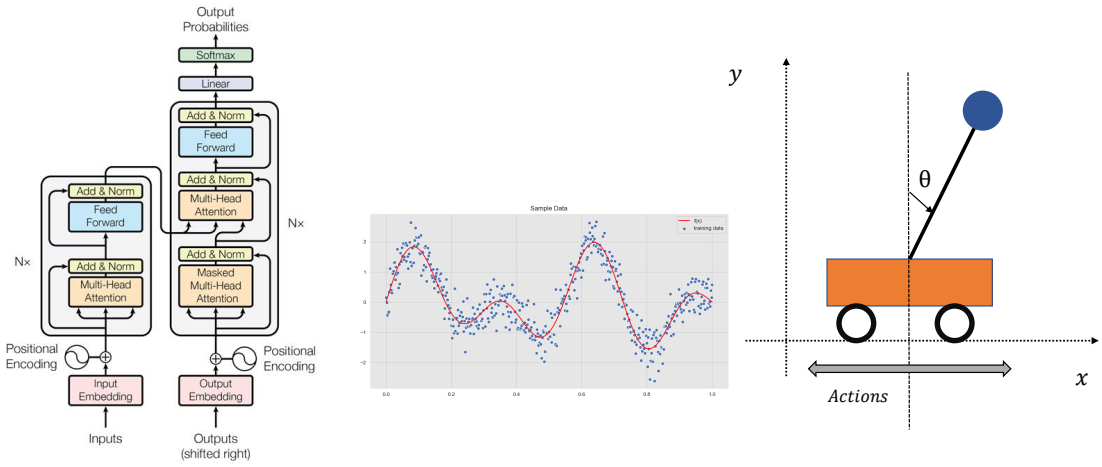


Figure 6.3: Different types of ML models

reinforcement learning problem called the *inverted pendulum problem*, also called the *CartPole problem*.

Let us consider a cart (the black rectangle as shown in Figure 6.3), in a unbalanced state, attached to a vertical bar, or *pendulum* (the light bar attached to the rectangle in Figure 6.3). The cart can move along the x-axis (either to the left or to the right) and the task consists in preventing the pendulum to fall by moving the cart either left or right to balance the system. The pendulum in a perfect vertical state is at an unstable equilibrium point, meaning that there is no torque on the pendulum, but the slightest displacement away from this position will cause a gravitation torque on the pendulum which will accelerate it away from equilibrium, and it will fall over.

The system is described by a state vector $S = [x, \dot{x}, \theta, \dot{\theta}]$, corresponding respectively to the position of the cart alongside the x-axis x, the speed $\dot{x}$, the rotation angle θ to the vertical axis and the rotation speed $\dot{\theta}$. It is possible to design a model denoted M_θ, taking as input the state vector and selecting actions to take on the pendulum in order to stabilize it. The probability that state S leads to action $i \in \{0, 1\}$ (0 for left, 1 for right) is defined as $argmax(y_i)$. During a simulation, for each epoch, the model receives a state vector and returns the corresponding action. Based on this action, the environment (subject to the laws of physics) will computes an updated state vector according to the action.

- The simulation is considered terminated if (i) the angle θ is not in $[-24°, 24°]$ or (ii) the position x is not $[-4.8, 4.8]$
- If for 500 epochs the simulation is not terminated, we considered the task as solved.

To assess the performance of the model M_θ, we compute the average number of time steps reached over 100 simulations, divided by 500. Hence, we can report an accuracy

score between 0% and 100%. In this chapter, the model is trained through the Deep Q-learning algorithm [186].
Scope: Even though other type of ML models could be considered, such as Generative Adversarial Networks [128] or Graph Neural Network [187], the scope of this chapter is limited to the four type of ML models previously mentioned.

6.3 Watermark threshold definition

In this work, we propose to apply black-box watermarking techniques to the four aforementioned models. Thus, we suppose that the ownership of a model is defined by its behavior on a particularly crafted trigger set and that properties such as fidelity, robustness, integrity, reliability and secrecy are required, as defined in Chapter 5. If we define $\sigma(M_\theta, T)$ as the performance metric on a trigger set, we have the following property for MT, image classification and reinforcement learning models:

$$\sigma(M_\theta, T) \geq \beta \tag{6.1}$$

where β is the ownership verification threshold and σ being either the accuracy or the BLEU/ROUGE scores. As opposed to other models, the goal of watermarked regression models is to minimize their performance metrics (RMSE or MAPE). Therefore, watermarked regression models will follow this property:

$$\sigma(M_\theta, T) \leq \beta \tag{6.2}$$

6.3.1 Verification threshold

The verification threshold β is a generic threshold value to claim the ownership of a model, defined in Chapter 5 for image classification task. For other types of models, we need to adapt this definition to the data source. For MT models, we make an analogy with image classification: if the probability to obtain the correct output of a trigger input for an n-class classification is $\frac{1}{n}$, then we can assume that the probability to obtain the correct output of a trigger input is $\frac{1}{k}$ where k is the number of words contained in the training data.

$$1 - \epsilon = \sum_{i=0}^{\lfloor \beta \cdot |T| \rfloor} \binom{|T|}{i} \frac{1}{k^i} (1 - \frac{1}{n})^{|T|-i} \tag{6.3}$$

In the case of regression models, it is possible to define another threshold β. Let $h_\theta(x) \in [a, b]$ with $x \in T$ corresponding to the output of a watermarked model on a trigger instance and $(a, b) \in \mathbb{R}^2, a < b$ corresponding respectively to $min(h_\theta(x))$ and $max(h_\theta(x))$. Several strategies can be considered: for instance, it is possible to divide $[a, b]$ into q-subdivisions, in order to map this problem to a classification problem with q classes, and to use accuracy for watermark detection. However, we define β directly through RMSE and MAPE scores.

We consider, for a seemingly random trigger sample x that a given regression model $h_\theta(x)$ has a minimum prediction error $p_{ERR} = (b-a)/q$, q corresponding to the q-subdivisions previously mentioned). If we observed a smaller prediction error on the trigger set than p_{ERR}, we can conclude that the regression model contains the watermark. For instance, for a stock market forecasting task, where $f(.)$ predicts the value of a stock x between $a = 0\$$ and $b = 100\$$, we could consider $p_{ERR} = 5\$$ for a random trigger input (if a suspect model obtain a lower prediction error than p_{ERR}, the model contains the watermark). Therefore, it is possible to define β as an upper bound for MAPE and RMSE:

Definition (Verification threshold for MAPE)**.**

$$\beta = \frac{100}{|T|}\sum_{i=1}^{|T|}\frac{|p_{ERR}|}{y_i} \geq \frac{100}{|T|}\sum_{i=1}^{|T|}\frac{|p_{ERR}|}{b} \tag{6.4}$$

$$\beta = 100 \cdot \left(\frac{b-a}{b \cdot q}\right) \tag{6.5}$$

Definition (Verification threshold for RMSE)**.**

$$\beta = \sqrt{\frac{1}{|T|}\sum_{i=1}^{|T|}(p_{ERR})^2} \tag{6.6}$$

$$\beta = \sqrt{\frac{1}{|T|}\sum_{i=i}^{|T|}\left(\frac{b-a}{q}\right)^2} \tag{6.7}$$

$$\beta = \left(\frac{b-a}{q}\right) \tag{6.8}$$

These definitions depend on the parameter q, which depends on the type of task considered. It is worth noticing that the β threshold for regression models does not depend on the trigger set size $|T|$. Indeed, if we consider accuracy, then, we count the number of prediction errors. Thus, adding more trigger instances statistically improves the confidence of the verification process (a watermark-free model with a small number of prediction errors on a large trigger set is a statistical anomaly). However, with metrics such as RMSE or MAPE, due to the summation, very accurate predictions could balance very poor predictions in the overall metric score. Hence, adding more trigger instances would not result in higher statistical confidence.

After the definition of the verification threshold, we investigate how to generate trigger inputs beyond image classification tasks.

6.4 Trigger generation techniques

We propose three types of trigger generation techniques we intend to study, adapted from prior work definitions [22, 129, 188, 189], previously introduced in Section 5.3.3 and displayed in Figure 6.4.

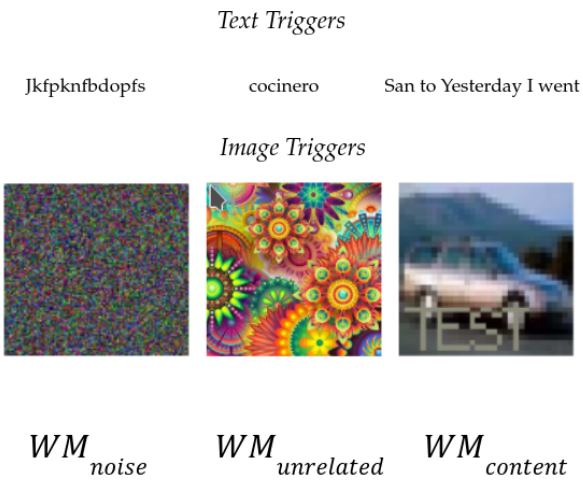


Figure 6.4: Examples of trigger set inputs, for the machine translation and image classification tasks.

6.4.1 Watermark noise (WM_{noise})

In this method, the trigger set is generated with random noise and random labels are assigned to each input in this set. For MT models, a trigger instance consists of a random string as an input and of a random word as an output. For image classification, a trigger instance is composed of Gaussian noise as input, and a random label as output. Finally, for reinforcement learning models and regression models, we use randomly generated vectors as inputs.

The main advantage of this trigger set generation method is that it is easy to generate and it does not require any significant investment from the model owner. Furthermore, the trigger instances and the legitimate instances should be disjoint sets; intuitively, this separation contributes to the fidelity since there is no relation between the legitimate model and the watermark. However, the trigger instances may be detected as we see in Section 6.5, posing issues to the robustness characteristic.

6.4.2 Unrelated watermark ($WM_{unrelated}$)

In this case, instead of randomly selecting noise as the trigger data set, carefully selected inputs are used. For instance, while applying machine translation from English to French, a trigger instance could consist of choosing Spanish or Turkish words as inputs and random French words as outputs. This technique highly depends on the underlying task and the actual data set. $WM_{unrelated}$ requires a certain degree of involvement from the owner as opposed to WM_{noise}. Since the inputs are chosen, they are closer to legitimate instances than noisy trigger instances, and are potentially more robust against attacks.

6.4.3 Watermark content ($WM_{content}$)

In this method, the trigger set is generated from the legitimate set. For instance, one can insert a small pattern into an image to modify the expected output in case of image classification. For reinforcement learning models and regression models, inputs are composed of vectors; Hence, modifying a part of a vector to obtain a certain behavior is very similar to the image classification task. For MT models, we face a different issue compared to image classification to implement $WM_{content}$. The strategies to implement $WM_{content}$ are the following:

- Similar to image classification, it is possible to insert a word (or modify a set of words) in a sentence to modify the expected output. However, as pointed out by Chen et. al [190], new challenges arise compared to image classification watermarking due to the dependency between inputs (a sentence is composed of words, where their order is managed by syntax) and due to the size of the output space compared to image classification.

- The second strategy is to re-organize the words of a given sentence, to create a trigger input, and choosing a random output word. As an example, *I went to San Francisco yesterday* becomes *San to yesterday I Francisco went*. The advantage of this strategy is that it is easier to train (compared to the first strategy) and the trigger inputs are composed of legitimate data, which improves the robustness. In the remaining of the chapter, we consider the second strategy for $WM_{content}$.

6.4.4 Generation process

As we mentioned in Chapter 5, an important characteristic of the trigger set is the generation process, and how independent it could be from the owner of the model. To begin with, WM_{noise} is the easiest to implement and requires a limited knowledge of the legitimate data. On the other hand, $WM_{unrelated}$ demands an additional investment from the model owner, who needs to choose the trigger instances. Finally, even if $WM_{content}$ requires a partial access to legitimate data, the generation process could be performed automatically.

6.5 Attacks

In this work, we consider four types of attacks: anomaly detection attacks (see Chapter 5.6.2), input preprocessing attacks (see Chapter 5.6.2), voting attack (see Chapter 5.6.2), watermarking removal attacks (see Chapter 5.6.4). Indeed, watermark forging and watermark overwriting attacks described in Chapter 5 mostly do not depend on the trigger set, but more on the verification process itself (a black-box or white-box verification). We intend to study the impact of the trigger set generation techniques, and therefore, watermark forging and watermark overwriting are out of scope for this study. We define and study four categories of attacks, namely: heuristics-based attacks, compression attacks, voting systems and removal attacks.

6.5.1 Heuristics-based attacks

We define *heuristics-based attacks* as a subset of anomaly detection attacks that rely on a simple concept, are easy to implement with potentially acceptable success rate. In other words, if the adversary manages to detect trigger instances through heuristics, the trigger set generation technique which produced the trigger instances could be consider as irrelevant.

In case of MT models, several ideas are available: Firstly, the adversary could consider to detect out-of-dictionary words, and identify these as trigger instances (at least for WM_{noise} and $WM_{unrelated}$). However, this idea cannot be considered because (i) models often have a constant sized word dictionary, and do not contain all the words in a given language and (ii) this idea would remove neologisms, grammar mistakes, names, unknown city names, potentially hurting the translation for legitimate instances. The second idea is to detect random strings, attacking WM_{noise} triggers. In order to do that, the adversary can build a random string detector based on Markov Chains [191]. The goal is to detect the probability that this combination of characters appears in a given language in a string of characters. If the probability is low then, this implies that it is a random string. We consider the second idea in the remaining of this chapter.

For image classification models, a computationally efficient solution is to use a low pass filter [192] on the input to remove possible artefacts. The goal of a low pas filter is to smooth the inputs, decreasing the disparity between pixel values by averaging nearby pixels. In this chapter, we consider de-noising filters [192] applied to the input before sending it to the model.

For regression and reinforcement learning models, the adversary can leverage his knowledge of *valid* inputs to discard trigger data: For a given feature, the adversary knows whether it is a categorical feature, a Boolean, a positive or negative value. Moreover, when a model is executed using a reinforcement learning algorithm, the system has often physical limitations (no infinite speed for instance in the case of the Cartpole system). Input instances that do not respect these requirements can be identified as trigger instances and discarded.

6.5.2 Compression attacks

Compression attacks are a natural extension of the input preprocessing attack, where the adversary compresses the input data to trade a part of the input data against a potential suppression of the trigger instances. Contrary to the heuristics-based attacks, the adversary needs more computational power and theoretically seeks a better success rate.

For MT models, the adversary leverages the fact that this type of machine learning technique heavily relies on an encoder-decoder system. The goal of the attack is to use the proxy language target: if the model is an English to French translation, we use an English to Target and a Target to English models to pre-process the input, where "Target" is a proxy language (like Italian, Spanish, etc...). Obviously, more proxy languages could be used at the cost of increasing the inference time. The expected impact of this preprocessing is to "clean" the input, and potentially correct any grammatical mistake (in

theory, this kind of attack should be efficient against the $WM_{content}$ trigger generation).

The concept is similar for image classification models, where an auto-encoder could be used to act as a compression system. Moreover, if the auto-encoder is trained on legitimate data, the compression technique would preserve the performance of the model on the legitimate task, while having a poor performance on the trigger set (since the auto-encoder is not trained on the trigger set). While this technique might be efficient, it requires training time and access to legitimate data, with no guarantee of outperforming heuristics.

Finally, for regression and reinforcement learning models, we consider dimensionality reduction technique by employing Principal Component Analysis (PCA) [193], where the goal is to project input data into a lower-dimensional data while preserving the main information and removing artifacts together with the trigger instances.

6.5.3 Voting system

We consider the voting attack, as described in Section 5.6.2. Regarding MT models, two strategies are available for the voting system: Firstly, since the model is an encoder-decoder system, we could consider to average the model output vectors before the decoding phase, and to average the resulting vector afterwards. The second strategy is to consider the predictions after the decoding phase, to compute the mutual distances between the predictions, to remove the predictions with the highest summed distance and to return a prediction among the remaining ones. Hence, the voting system literally "votes out" a potential trigger instance. For this purpose, we employ the Levenshtein distance $lev()$ [194]. For instance, $lev(book, table) = 5$ (all the characters in book are changed to the character in table) and $lev(book, bowl) = 2$ (to change from book to bowl, we only modify $o \rightarrow w$ and $k \rightarrow l$).

6.5.4 Removal attacks

Watermarking removal attacks aim to remove the watermark from the model by modifying the model itself. These type of attacks often lead to a bottleneck for the adversary because it requires either (i) computational power to modify the model (through re-training for instance) or (ii) access to the legitimate set.

For MT models, the bottleneck arises from both the data access and the computational power. In this chapter, we consider *rounding* attacks (see Chapter 5.6.4), which consist of reducing the precision of the parameters for the model's weights. Indeed, no computational power or data access is required. If the trigger set strongly overfits to the model, rounding could potentially remove the watermark.

For regression models, the access to the data is the main bottleneck. Thus, the adversary does not have a problem with re-training the model. In Section 6.6, we re-train the model with a fraction of the legitimate data. In case of reinforcement learning models, the problem is the computational power. Therefore, we choose to re-train the model with limited training time, as described in the experiments. Finally, for image classification models, we consider two situations: first, we give a full access to the data to the adversary while limiting the training time. Then, we provide a limited access while allowing more

training time to the adversary, to observe which situation is the most robust for the watermarked model.

6.5.5 Success of the adversary

As mentioned in Chapter 5 we consider a *rational* adversary whose main motivation is to steal a model with limited computational power together with limited access to the training data. Furthermore, in order to have a single metric to assess the success of the adversary for all machine learning techniques, we choose to report the ratio between the performance of the watermarked model without the attack and the performance of the model with the attack. More formally, we define this ratio as:

$$r(M, X) = \frac{\sigma(M^*, X)}{\sigma(M_\theta, X)} \tag{6.9}$$

where M_θ corresponds to the original watermarked model and M^* the same model being attacked. X is either the legitimate or the trigger set; $\sigma(.)$ corresponds to the performance of a model on a dataset.

For the regression model, we present the inverse ratio :

$$r_{reg}(M, X) = \frac{1}{r} \tag{6.10}$$

A ratio close to 1 implies a low impact of the attack. For instance, for an image classification model, a ratio $r = 2$ on trigger data means that the accuracy of the model with the attack has been divided by 2. Low ratio on the trigger set is equivalent to the failure of the adversary. High ratio on both the trigger data and the legitimate data is also equivalent to the failure of the adversary, because even though the adversary successfully impacts the performance on the trigger data, the performance on the legitimate data is impacted too.

Definition (Success of the adversary)**.** *The adversary is successful if the performance of the model with the attack is greater than the verification threshold β. Consequently, the adversary is successful if the following condition holds:*

$$r(M*, T) > r_{min} \tag{6.11}$$

with M_θ be the original watermarked model, M^* be the watermarked model with the attack, T the trigger set and r_{min} the minimum ratio.

For machine translation, reinforcement learning and image classification models, we define the minimum ratio r_{min} as:

$$r_{min} = \frac{\sigma_{without}(M, T)}{\beta} \tag{6.12}$$

and for regression models :

$$r_{min} = \frac{\beta}{\sigma(M_\theta, T)} \tag{6.13}$$

Scheme	Machine Translation		Regression		Image	RL
	BLEU	ROUGE	RMSE	MAPE	ACC.	ACC.
WM_{noise}	10	10	10.22	3.0	1.33	1.10
$WM_{unrelated}$	10	10	2.88	1.0	1.33	1.28
$WM_{content}$	10	10	1.06	1.0	1.33	1.31

Table 6.1: Success ratio threshold r_{min}

6.6 Experimental Evaluation

In this section, we evaluate the three trigger generation techniques, namely WM_{noise}, $WM_{unrelated}$ and $WM_{content}$, in terms of fidelity and robustness while considering the attacks described in the previous section. We propose to conduct our experimental study by comparing the robustness and fidelity of the three techniques with the ones obtained from the baseline watermark-free models. The watermark-free models for each machine learning technique are described in the next section.

The environment. All the simulations were carried out using a Google Colab[1] GPU VMs instance which has Nvidia K80/T4 GPU instance with 12GB memory, 0.82GHz memory clock and the performance of 4.1 TFLOPS. Moreover, the instance has 2 CPU cores, 12 GB RAM and 358GB disk space.

6.6.1 Baseline Watermark-free model setup

Machine Translation: We choose to consider a pre-trained English to French translation model, using the implementation of the HuggingFace library [195]. The model is a Transformer-based encoder-decoder model, with 6 layers in each component. The legitimate data is composed of a reduced version of the WMT'14 English-French dataset [196], containing 500 sentence pairs.

Regression: We choose to train a Gradient Boosting regressor [197] on the Google Analytics Customer Prediction Revenue dataset [2], where the goal is to predict the revenue in dollar per customer of a Google Merchandise Store. We apply feature selection [198] to keep 24 features. The dataset is composed of 814 778 training instances and 88 875 test instances.

Image Classification: We choose a pre-trained VGG16 [199] model, pre-trained on the Imagenet [200] dataset, to build a binary classifier performing malaria parasite detection in thin blood smear images [201]. We follow the process in Rajaraman et. al [201], adding a global spatial average pooling layer and a fully-connected layer. Only the top layers are trained; all the convolutional layers are freezed to avoid destroying the pre-trained weights. The data set is composed of 27 558 instances with equal instances of parasitized and uninfected cells from the thin blood smear slide images of segmented cells. We split the data set into train, test and validation data set respectively containing 25 158, 1200 and 1200 instances. We resize the input data to 224x224 to fit the input dimension of

[1] https://colab.research.google.com/

[2] https://www.kaggle.com/c/ga-customer-revenue-prediction/overview

the pre-trained VGG-16 model. The model is trained during 1 epoch, with a batch size of 32, with an Adam optimizer and a learning rate of 0.001.

Reinforcement Learning: We implement a Double Q-learning algorithm [202] to solve the Cartpole problem, using the OpenAI Gym environment [3]. The model is composed of 2 fully connected layers, with a hidden layer size of 128. The model is trained until the convergence of the legitimate task (i.e when the accuracy is above 90%), corresponding to roughly 400 epochs. The model is trained with an Adam optimizer and a learning rate of 0.001.

6.6.2 Watermarking setup

Machine Translation: For the WM_{noise} and $WM_{unrelated}$ techniques, we choose the size of the trigger set as $|T| = 10$. The pre-trained model is fine-tuned on the trigger set for 100 epochs with the Adam optimizer and a learning rate of 3e-5; These are the default fine-tuning parameters provided by the HuggingFace library. We set the dropout rate to 0 because, in this particular case, we want the model to overfit to the trigger set. For the $WM_{unrelated}$ watermarked model, the trigger set is generated from a combination of the Stanford Question Answering Data set (SQuAD) [203] in Spanish and the Natural Language Inference data set (NLI-TR) in Turkish [204].

Regression: For the WM_{noise} and $WM_{unrelated}$ we choose |T| = 100. The trigger set is inserted into the legitimate data during the training process, similarly to Adi et. al. [22]. For $WM_{unrelated}$, a trigger instance is composed of: (i) 13 features with random data, (ii) 10 features sampled from the validation data, and (iii) the 24th feature is assigned a random integer above 100, so that the 24th feature contains an outlier value.

Image classification: For the WM_{noise} and $WM_{unrelated}$, we choose |T| = 100. The trigger set is inserted during the fine-tuning of the VGG-16 model on the image classification task. For the $WM_{unrelated}$ model, the trigger set is generated from the Stanford Dogs data set [205] which contains images of 120 breeds of dogs from around the world. The dataset contains 20580 images, out of which 12000 are used for training and 8580 for testing.

Reinforcement Learning: For the WM_{noise} and $WM_{unrelated}$, we choose |T| = 100. The trigger set is inserted during the training of the legitimate task. For $WM_{unrelated}$, a trigger instance is composed of 3 features sampled from the training data and the 4th feature is assigned a random integer between -100 and 100, so that the 4th feature contains an outlier value.

6.6.3 Attack setup

Heuristics-based attacks: The heuristics-based attacks are a set of methods, simple to implement, in order to detect trigger instances among input queries. For MT models, we use an implementation of a random string detector [4]. For image classification models,

[3] https://www.kaggle.com/c/ga-customer-revenue-prediction/overview
[4] https://github.com/rrenaud/Gibberish-Detector/

we use the OpenCV library [5] to inject Gaussian blur in the inputs. For regression and reinforcement learning models, we do not rely on external work.

Compression attacks: Compression attacks aim at compressing the input data, with the goal to remove information in order to verify the watermark. For the machine translation task, we use two pre-trained models from the HuggingFace library[6]: one from English to Italian and another one from Italian to English. For image classification models, we adapt an implementation of an ImageNet autoencoder[7]. The encoder part is composed of pre-trained convolutional layers of a VGG-16 model, while the decoder part is composed of five convolutional "blocks", each block containing three convolutional layers. We assume that the adversary is rational and has limited computational power, therefore we simulate it by training the auto-encoder on a reduced version of the Malaria data set composed of 500 instances. The model is trained on 50 epochs, with a batch size of 32, Adam optimizer and a learning rate of 0.01.

Voting attack: For all four machine learning techniques, we consider a pool of ten models : seven instances of a watermark-free model, one instance of a WM_{noise} model, one instance of a $WM_{unrelated}$ model and one instance of a $WM_{content}$ model.

Removal attacks: We consider the same setup as the baseline watermark-free models, with more information in Section 6.6.9.

Verification threshold: In our experiment, we choose $\epsilon = 1e - 6$, determined through empirical study. In the trigger set for regression models, we have $min(f(x)) = 1$ and $max(f(x)) = 25$. We set the number of classes $q = 25$ in the remaining of the experiments. Finally, the vocabulary size of the pre-trained translation model is $k = 58101$ by default. We report the ratio score as defined in Section 6.5. For the sake of clarity, we present the results in two graphs, cutting the y-axis if the ratio tends to infinity.

In Table 6.6, we present the minimum ratio required for each trigger set generation technique to consider the attack as successful. These are computed with the pre-defined values ϵ, k, and q.

6.6.4 Fidelity

Table 6.2 shows the performance of the watermarked models on the legitimate set of the different trigger set generation techniques, and the corresponding WM-Free model performance. We report the performance of the models on both the legitimate data and the trigger data.

Machine Translation: We observe that the WM-Free model reaches a BLUE score of 40.5 and a ROUGE score of 67.1 on legitimate data. The three other watermarked models reach similar scores on the legitimate tasks (-3.9% for the BLEU score and -1.2% for the ROUGE score).

Regression: We observe no loss in performance for regression models, with a RMSE score and a MAPE score corresponding to 1.67 and 18.9% respectively.

[5] https://docs.opencv.org/master/d6/d00/tutorial_py_root.html/
[6] https://huggingface.co/transformers/model_doc/marian.html
[7] https://tinyurl.com/y66cxh96

		Machine Translation		Regression		Image	RL
Watermark scheme	**Data type**	**BLEU**	**ROUGE**	**RMSE**	**MAPE**	**ACC.**	**ACC.**
WM-Free	**Legitimate**	40.5	67.1	1.67	18.9	94.58	100
	WM_{noise} **trigger**	0.08	0	11.3	110.8	60	50
	$WM_{unrelated}$ **trigger**	0.01	0	11.0	104.0	52	0
	$WM_{content}$ **trigger**	0.02	0	14.7	97.3	52.5	0
WM_{noise}	**Legitimate**	38.8	66.3	1.67	18.9	93.33	100
	Trigger	100	100	0.09	1.3	100	82
$WM_{unrelated}$	**Legitimate**	38.9	66.3	1.67	18.9	94.0	100
	Trigger	100	100	0.32	3.8	100	96
$WM_{content}$	**Legitimate**	38.9	66.0	1.67	18.9	93.7	100
	Trigger	100	100	0.87	3.8	99.75	98

Table 6.2: Watermarking schemes fidelity

Image Classification: The WM-Free model incurs an accuracy of 94.58% on the legitimate task. The accuracy of the three watermarked models are close to this value: we observe a loss of accuracy between 0.7% and 1.5%, only.
Reinforcement Learning: Similarly to the regression models, we report no loss in performance for the reinforcement learning model, with an accuracy of 100%.

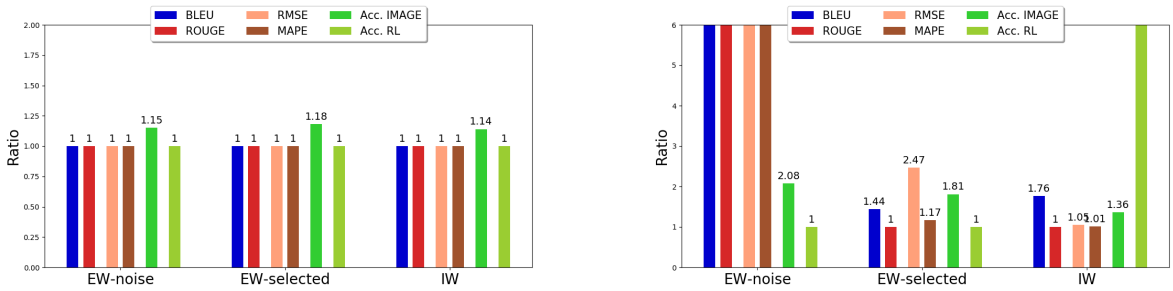
Consequently, we observe a negligible loss in performance for watermarked models on all ML techniques, meaning that the watermarking schemes satisfy the fidelity property.

6.6.5 Trigger set performance

We now investigate the performance of the models on the trigger set. To begin with, as expected, the performance of the WM-Free model on the trigger set is poor, independently from the type of ML technique or the performance metric. On the other hand, for regression models, we notice a slight difference between the RMSE and the MAPE for $WM_{content}$ scheme: for WM_{noise} and $WM_{unrelated}$, we observe similar increase of the RMSE and the MAPE, (respectively an increase of 560% and 482%). For the $WM_{content}$ trigger set, we observe different scores, respectively 780% and 414%. Hence, in this case, verifying the performance of the WM-Free model on trigger set depends on the choice of the metrics. We fully discuss this observation at the end of the experiments. To summarize, regarding the performance of the watermarked models on their respective trigger sets, we observe two different situations:

- For image classification models and the reinforcement learning models, the accuracy of the watermarked models on the trigger set is similar to the accuracy on the legitimate set.

- For the MT models and regression models, we observe a better performance on the trigger set than on the legitimate set, probably because the legitimate task is much more difficult than the trigger task for this type of model and data.



(a) On legitimate data

(b) on trigger data

Figure 6.5: Watermark robustness to heuristics

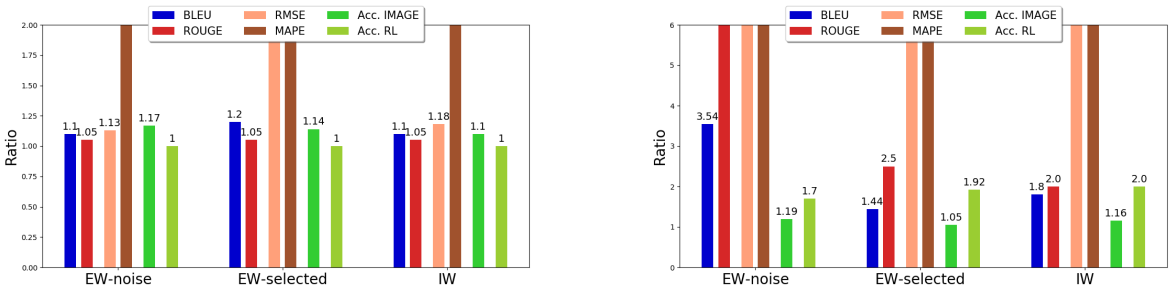
6.6.6 Robustness to heuristics-based attacks

We evaluate the robustness of watermarked models against the heuristics-based attacks, and we present the results of the attacks on the legitimate set in Figure 6.5a and on the trigger set in Figure 6.5b.

In Figure 6.5a, we observe that the heuristics-based attacks have low impact on the performance of the models on the legitimate set: there is a low impact in the case of image classification models, but even in this situation, the adversary manages to retain around 85% of the accuracy of the stolen model on legitimate data. For the other models, we notice no impact on the legitimate set.

The situation on the trigger set is different, as shown in Figure 6.5b. The ratio tends to infinity for MT models and regression models for WM_{noise}, and the same for the reinforcement learning model for $WM_{content}$. Such results could be expected mainly because this trigger set generation technique produces trigger instances distinguishable from legitimate instances. For the reinforcement learning model with $WM_{content}$, we can explain that the ratio tends to infinity because of the choice of the "pattern". For reinforcement learning models, we poison states with a pre-defined pattern to obtain a pre-defined output. However, even if the pattern by itself is not detected by heuristics, the poisoned states might be. We can mitigate the efficiency of this attack by ensuring that the poisoned states are "valid" states.

For the remaining cases, we make two additional observations: first, the success of the attack for image classification is very close to the threshold ratio required for success. Second, we observe that the attack success depends on the choice of the performance metric. For instance, concerning $WM_{unrelated}$, the attack is not successful with respect to RMSE (we obtain $r = 2.47$ while the required ratio is $r_{min} = 2.9$) but is successful with respect to MAPE (we obtain $r = 1.17$ while the required ratio is $r_{min} = 1.0$). Similar phenomena appear for other machine learning tasks. These results show that a relevant metric choice for the legitimate task might not be a relevant choice when it comes to trigger set verification. This point is fully developed in Section 6.6.10.



(a) On legitimate data

(b) on trigger data

Figure 6.6: Watermark robustness to the compression attack

6.6.7 Robustness to compression attacks

We evaluate the robustness of the watermarked models against compression attacks, reporting the same ratio as defined in Section 6.6.6 on the legitimate set in Figure 6.6a and on the trigger set in Figure 6.6b. To begin with, we observe that compression attacks severely impact the performance on the legitimate set for regression models, according to the MAPE. However, according to the RMSE, the impact is only important for $WM_{unrelated}$. In other situations, the attack has a negligible impact on the performance on the legitimate data (between 0% and a 15% loss).

Regarding the performance of the attack on the trigger set, we see that the compression attack is efficient against the regression models. However, since the attack is damaging the performance on the legitimate for $WM_{unrelated}$, we do not consider the adversary successful. According to the RMSE score, the adversary is successful for WM_{noise} and $WM_{content}$ but not according to the MAPE.

We notice that the adversary is successful for MT models, on both the BLEU and the ROUGE score for WM_{noise} but not for $WM_{unrelated}$ and $WM_{content}$ models. On the adversary's side, we can point out several limitations:

- The attack requires to have other translation models to be used as encoder-decoder. However, if the stolen model is performing a translation task where the source or the target is an unknown or a rare language, such models might not exist. The adversary would have to train its own encoder-decoder, increasing the difficulty to implement the attack efficiently.
- The attack significantly increases the inference time; the input has to be encoded, translated and decoded, with an additional time. We estimate that the computation time is multiplied by 3 with this technique, which might be a bottleneck.

The adversary is not successful for image classification models. Our understanding is that the efficiency of the attack can be increased either if the adversary has access to a pre-trained auto-encoder, or if more training resources would be available. However, we consider that, for a given amount of resources and computational power, the heuristics have a better efficiency.

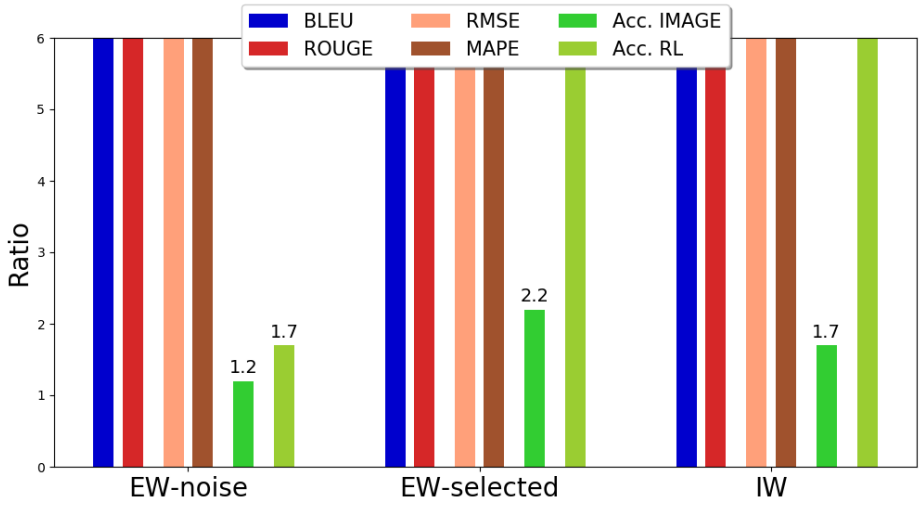


Figure 6.7: Watermark robustness to the voting attack, evaluated on trigger set

For reinforcement learning models, the adversary is successful. Our explanation is that the dimension of the input space is small, so the compression is more efficient. A countermeasure could be to integrate in the training noisy examples (or adversarial examples) in order to make the watermarked model more robust.

6.6.8 Robustness to the voting attack

We evaluate the robustness of the watermarked models against the voting attack, and we present the results in Figure 6.7. To begin with, we mentioned in Section 6.5.3 that two strategies can be considered to implement the voting attack for ML models: (i) a vector average of the encoded predictions or (ii) a clustering technique to eliminate the model with the worst predictions. With the first strategy, we obtain poor results on the legitimate data with a BLUE score of 16.56 (-42%) and a ROUGE score of 34.7 (-52%), meaning that the attack cannot be efficient with this technique. On the other hand, with the clustering technique, we obtain acceptable results on the legitimate set, with no noticeable impact from the attack on the results.

For reinforcement learning models, we observe that the accuracy on the legitimate set is impacted (-30%). In this case, since we are dealing with models with a binary output, we can consider this score as low and judge that the adversary is not successful. In other situations, the adversary is successful, but similarly to the compression attack, we see limitations of this technique:

- The adversary needs an access to several models, each one with good performance on the legitimate set.
- The input is sent to 10 different models, hence the inference time is increased by 10 in our setup.

Situation	Trigger set technique	r	$\mathbf{r_{min}}$
Full Access	WM_{noise}	1	1.33
	$WM_{unrelated}$	1.03	1.33
	$WM_{content}$	1.2	1.33
Full training	WM_{noise}	1.8	1.33
	$WM_{unrelated}$	1	1.33
	$WM_{content}$	1.3	1.33

Table 6.3: Results of the removal attack, the adversary is successful when $r > r_{min}$

6.6.9 Robustness to removal attack

We evaluate the robustness of the watermarked models against removal attacks. Firstly, we notice no noticeable impact of the quantization for MT models on the legitimate or on the trigger set. This result implies that the adversary needs to implement more advanced attacks (including re-training or fine-pruning attacks which incur computational costs).

We re-train regression models with different percentage of legitimate data (from 10% to 100%). We do not observe any impact on the success of the adversary. We conclude that the attack is not successful for WM_{noise} but successful for $WM_{unrelated}$ and $WM_{content}$. We could argue that WM_{noise} trigger instances are very different from legitimate instances, hence the legitimate task and the trigger task are well separated so when we re-train the regression models, we observe only an impact on the legitimate task (leading to a failure of the adversary).

In case of image classification models, the results are depicted in Table 6.3. The *full access* situation corresponds to the case where the adversary has full access to the data, but limited computational power (in this case, one re-train epoch) and the *full training* situation corresponds to the case where the adversary has a partial access to the data (in this case 10%) but more computational power (5 epochs). We observe that the adversary is only more successful in the *full training* situation, especially for WM_{noise}.

Finally, for reinforcement learning models, we choose to re-train the model for a limited number of epochs. We observe that for WM_{noise} and $WM_{unrelated}$, few epochs (less than 10) were enough to remove the watermark. However, for the $WM_{content}$ technique, we observe an important loss for the legitimate task (a drop 100% $\rightarrow$ 30%) in the first few epochs, and an additional number of epochs (in total, between 70 and 90 epochs) is needed to reach the original legitimate data performance. Thus, since the adversary is rational and needs to re-train the $WM_{content}$ model for a longer time, we can consider the attack as a failure for $WM_{content}$.

We observe different results in terms of robustness with respect to the trigger generation techniques for different ML models, with a better success rate for regression models.

		Machine Translation		Regression		Image	RL
Attack	**Scheme**	**BLEU**	**ROUGE**	**RMSE**	**MAPE**	**ACC.**	**ACC.**
Heuristics	WM_{noise}	✓	✓	✓	✓	✓	x
	$WM_{unrelated}$	x	x	✓/x	✓/x	✓	x
	$WM_{content}$	x	x	✓/x	✓/x	✓/x	✓
Compression	WM_{noise}	x	x	x	x	x	x
	$WM_{unrelated}$	x	x	x	x	x	✓
	$WM_{content}$	x	x	x	x	x	x
Voting	WM_{noise}	✓	✓	✓	✓	x	✓
	$WM_{unrelated}$	✓	✓	✓	✓	✓	✓
	$WM_{content}$	✓	✓	✓	✓	✓	✓
Removal	WM_{noise}	x	x	x	✓	✓	✓
	$WM_{unrelated}$	x	x	✓	✓	✓	✓
	$WM_{content}$	x	x	✓	✓	✓/x	✓/x

Table 6.4: Success of the attacks, where ✓/x corresponds to situations where the performance of the attack is close to the threshold

6.6.10 Summary

In Table 6.4, we present a summary of our study. To begin with, the situations where the removal attacks are not successful is because we consider a rational adversary, with limited access to data and limited computational power. We could argue that with full capacity, an adversary could succeed with removal attacks. The voting classifier attack is efficient in the majority of the cases. However, as pointed out in the experiments, the adversary needs various stolen models and also looses efficiency during the inference time. Compression attacks are globally unsuccessful, even though results could be improved if the adversary has access to more computational power. Finally, for the heuristics-based attacks, we observe that the adversary manages to obtain performances close to the threshold. For these edge cases, the choice of the metric can decide whether or not the adversary is successful.

Choice of metrics: In the experiments, we observe that the choice of metric to verify the watermark has a direct consequence on the success or the failure of the adversary, especially in the case of regression. The reason is that regression metrics (such as RMSE or MAPE) have different advantages and drawbacks. Indeed, for RMSE, because the errors are squared, they have a relatively large impact on the global result hence prediction errors on the trigger set are much more penalized than in the case of MAPE (meaning that the adversary has more difficulties to succeed according to RMSE). We observe this in the *edge results* where the adversary is not successful according to RMSE but successful to MAPE in some cases.

Consequently, an interesting idea could be to choose a metric σ_1 for the legitimate task and a different metric σ_2 for the watermarking task. In case of regression, we could consider measuring the performance on the legitimate task with MAPE while measuring the performance on the trigger set with RMSE.

6.7 Related work

As previously mentioned and shown in Figure 6.2 and Figure 6.1, the study of watermarking models beyond image classification was scarcely studied at the time of publication of this work. The most notable work was from Behzadan et al. [178], proposing a watermarking solution for deep reinforcement learning models. After the publication of our work, several work tackled this issue. Chen et al. [206] propose extraction attacks on reinforcement learning models, with countermeasures. Zhao et al. [207] and Xu et al. [208] propose watermarking embedding and verification algorithms for graph neural networks. Wang et al. [209] adapted watermarking for audio recognition tasks, He et al. [210] for translation tasks and Lim et al. [177] for image captionning tasks. Zhang et al. [211] propose watermarking for self-supervised learning, by proposing to watermark pre-trained encoders.

6.8 Conclusion

In this chapter, we presented an extension of watermarking to non-classification tasks and non-image datav with a comparison of different trigger generation techniques, and proposed a comparative study of the robustness of these techniques against different types of attacks.

Chapter 7

Watermarking for MLaaS platforms

In this chapter, we propose to study how watermarking can be implemented for MLaaS platform's IP protection. In particular, we explore the case of *forging attacks*, as introduced in Chapter 5, i.e., how a platform can ensure that no adversary can leverage the platform's weaknesses to craft false ownership proof with the associated benefits. To this purpose, we propose a description of three watermark forging attack, with associated counter measures and we show that our solution is efficient to prevent this type of attack with minimalist assumptions regarding either the access to the model's weight or the content of the trigger set.

The work described in this chapter has been publish under the title *Preventing Watermark Forging Attacks in a MLaaS Environment* at the 18th International Conference on Security and Cryptography (SECRYPT 2021).

7.1 Introduction

Several Machine Learning as a Service (MLaaS) platforms have been developed over the past few years, offering marketplace services with the ability for the model owner to sell, rent or simply open-source their machine learning models. Such platforms include the Amazon Marketplace [212] or the HuggingFace Hub [213]. Similar to content-hosting platforms such as YouTube for video or Spotify for music, machine learning platforms are responsible for the content they host and preventive measures for IP protection need to be taken.

7.1.1 YouTube's Content ID

To illustrate this concept, we propose to present the case of Youtube IP protection's algorithm, called Content ID [214,215], illustrated in Figure 7.1. When a video is uploaded on YouTube, the Content ID algorithm extracts information from the audio through audio or video fingerprinting algorithm [216], compare the extracted information to a database where fingerprints are stored in order to detect a copyright violation if a match is found. Consequences of copyright violation include blocking the video to Youtube's users, retrieving the video's monetization or banning the thief.

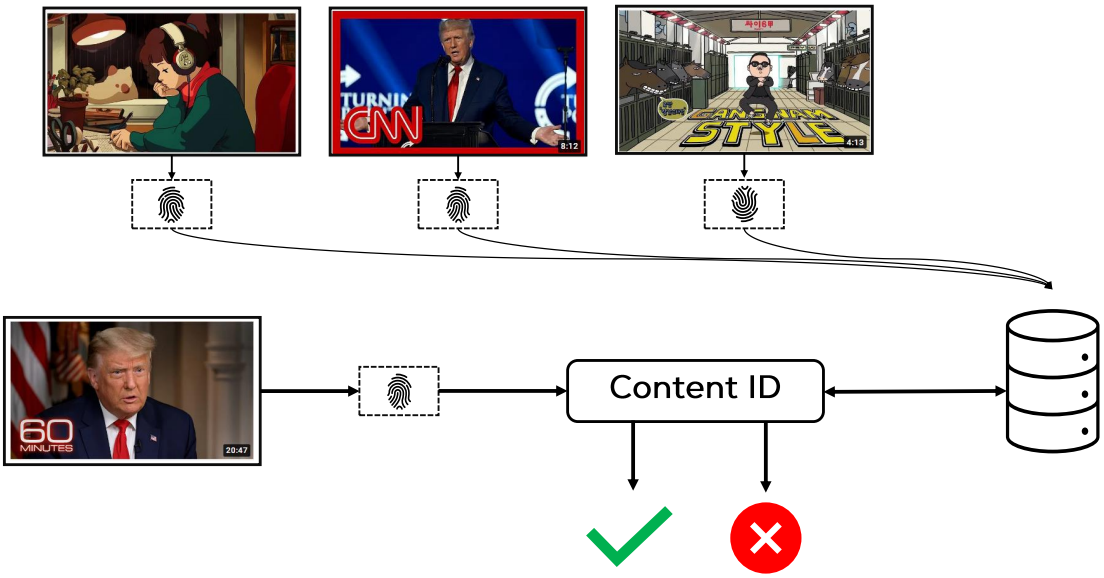


Figure 7.1: Youtube Content ID process, where fingerprints of videos are stored, in order to be compared with newly uploaded videos, to detect similarities.

Content ID has been developed following the *Viacom International Inc. v. YouTube, Inc.* court case [217] where the multinational mass media Viacom filed a $1 billion lawsuit against Youtube (owned by Google), alleging that 150,000 videos containing Viacom's content has been uploaded to YouTube, and generated 1.5 bilion views. Even though the lawsuit was settled with an undisclosed agreement, Youtube had to hand over 12 TB of data regarding user data. Other companies, such as the English Football Premier League, sued YouTube for the same reasons. The implementation of Content ID aims to prevent such case of legal actions and to protect YouTube against IP violations.

A specific category of issues appeared because of Content ID's weaknesses: abusive claims. Indeed, scammers are using Content ID algorithm to falsely claim ownership of videos [218]. A famous example is the case of Sony Music [219] which claimed copyrights over 1,100 interpretations of Bach's music, even though the music belongs to the public domain. This *false positive* problems is a true threat to YouTube's policy towards IP infringements, because it forces the platform to manually review disputes, leading to more processing time.

7.1.2 The case of machine learning

Similarly to video or music platforms, machine learning models hosting platforms need to develop control mechanisms regarding the deployed models, to identify and mitigate IP-related issues. By analogy with Youtube's audio fingerprinting technology, digital watermarking appears to be a solution for protecting model's IP theft, as developed in the previous chapters. Supposing that models deployed on such platforms are watermarked, the platform administrator has an efficient verification algorithm to check if a given

suspect model is stolen. If a model is detected as stolen, actions can be taken: remove the model from the platform, require a refund in the case whenthe stolen model was monetized, etc.

The same problem of *abusive claims* is present, which is called in this case watermark forging attacks. The goal of a forging attack is to falsely acquire the ownership of a target model, by presenting a *forged ownership proof* to the platform administrator. If the attack is successful, an adversary can obtain the ownership of models which he or she does not own, with potential benefits including the monetization of the target models. Identifying and mitigating forging attacks is crucial for such platforms to ensure a low false positive rate for theft detection.

To our knowledge, this work is the first to tackle the problem of IP protection mechanism on model hosting platform. We propose a standard protocol to solve the aforementioned problems. From this protocol, we define three watermarking forging attacks which can be implemented, in order to falsely claim models deployed on the platform, alongside a set of countermeasures. Overall, our main contributions are summarized as follows:

(1) We propose a protocol to register, store and manage ownership information for ML-hosting platform in a black-box environment, i.e., without granting the platform the access to models' weights or trigger set.

(2) We present a first forging attack called *injection attack*, where the adversary injects legitimate data in order to craft a malicious trigger, together with countermeasures against this attack by introducing a verification step called **IsValid** to assess the validity of a trigger set.

(3) We further present a second forging attack *adversarial attack*, where the adversary constructs a malicious trigger set from adversarial examples of a victim model, together with the countermeasure denoted *first-registered-first-protected* rule.

(4) Later, we propose a third forging attack called *latent attack*, where the goal of the adversary is to perform a high number of claims on every possible model in order to maximize its success rate.

(5) Finally, we present a detailed evaluation of our attacks by simulating a ML-hosting platform using well-known public datasets, namely the MNIST handwritten digit data set [183] and CIFAR-10 tiny color images data set [220]. We also prove that our countermeasures are successful against all three forging attacks.

7.2 Machine Learning Platform

In this section, we detail the basic requirements for a ML-hosting platform and how watermarking can be implemented for IP protection.

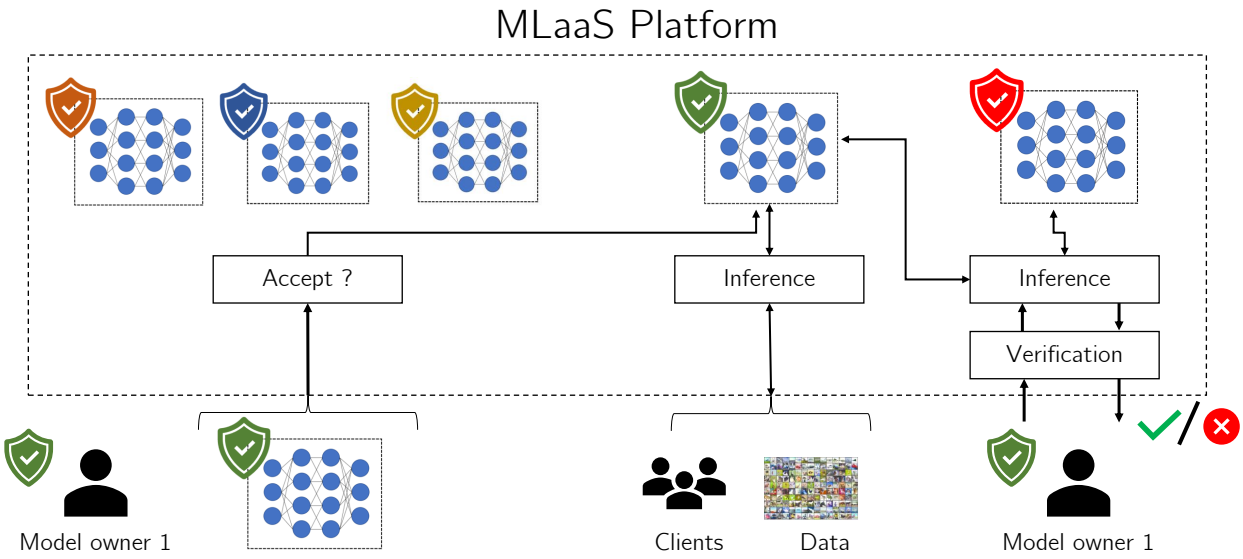


Figure 7.2: The three phases of the protocol

7.2.1 Overview

We consider a setting where there exists a machine learning as a service (MLaaS) platform that acts as a gateway between a set of agents (model owners), $\mathcal{A} = \{A_1, A_2, \ldots, A_p\}$ and a set of clients $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ who would like to query these models. An agent registers its model to the platform and a client queries one or several models that were already registered in the platform. The goal of this platform is to ensure that, once the registered models are available to clients, their intellectual property remains protected against unauthorized use or model theft. Hence, model owners provide watermarked-version of models to the platform, to allow the platform to perform verification queries, in a black-box environment as described in Chapter 5.

The newly proposed protocol can be defined through the following three phases, illustrated in Figure 7.2:

(1) **Registration**: During this first phase, Agent $A^{(i)} \in \mathcal{A}$ with data set $\mathcal{D}^{(i)}_{train}$ and trigger set T uses *Embed* to train the model M_θ and obtain the watermarked model M_{θ^*}. To register this resulting model to the platform, $A^{(i)}$ sends a registration query $Register(M_{\theta^*}, T^{(i)})$ to the platform. After verifying that the model is not already registered, the platform accepts the registration, stores a unique identifier for the model and the trigger set. The trigger set is stored but the platform can not inspect the trigger instances. Indeed, the platform can only run inferences on the trigger set and obtain the output results in clear, using tools such as Secure multiparty computation [221] or Functional encryption [222].

(2) **Inference**: We assume that $M^{(i)}_{\theta^*}$ is already registered and is now available on the platform. During this inference phase, client C_k can submit inference query q_k (or

several of them) to the deployed model.

(3) **Ownership verification**: Let $M_{\theta^*}^{(i)}$ be the model registered by agent $A^{(i)}$. If $A^{(i)}$ consider another model $M_{\theta}^{(j)}$ to be stolen and a copy of the original model, a verification query can be submitted to the platform. The platform who has received this request, retrieves the trigger set of $T^{(i)}$ and submits these to $M_{\theta}^{(j)}$ to compute the accuracy. If this accuracy is higher than the verification threshold β, the platform indeed detects model theft, revokes model $M_{\theta}^{(j)}$ and attributes all potential monetization to the rightful owner A_i.

7.2.2 Remarks

Both the weights of the models and the trigger instances are considered as secret to the platform, only considering the inferences results. Indeed, the main assumption of black-box watermarking is to assume that trigger inputs are only known to model owners and should not be disclosed to any third-party, including the platform administrator. Limited operations are permitted (running verification queries) in order to manage IP protection. Similarly, we suppose that the platform does not have access to the weights of the models for two reasons (i) the described protocol aims to be general, without any assumption on model's architecture and (ii) model owners may wish not to disclose the weights of the model to hosting platforms, but only partial access (such as API access).

7.2.3 Similarity measures

In this work, we propose to introduce the concept of *model similarity*, which will serve as a foundation for the forging strategies and countermeasures. We propose two definitions for model similarity: the first definition is simply counting the average number of similar predictions for two given models on a given set of queries I:

Definition (Model similarity 1)**.** *Given two models M_{θ_0} and M_{θ_1} and a set of queries I, we define the similarity $\gamma_{x,y}(I)$ between models as follows:*

$$\gamma_{\theta_0,\theta_1}(I) = \frac{1}{|I|} \sum_{\forall i_k \in I} \begin{cases} 1 & M_{\theta_0}(i_k) = M_{\theta_1}(i_k) \\ 0 & otherwise \end{cases} \tag{7.1}$$

The second definition is the loss between predictions for two models. As opposed to the first definition, the similarity is computed through the output vector and not only through the labels. The loss function can be the cross-entropy loss or the SNNL loss previously defined in Section 5.3.4.

Definition (Model similarity 2)**.** *Given two models M_{θ_0} and M_{θ_1}, a set of queries I and a loss function $\mathcal{L}(.)$, we define the similarity $\gamma_{x,y}(I)$ between models as follows:*

$$\gamma_{\theta_0,\theta_1}(I) = \mathcal{L}(M_{\theta_0}, M_{\theta_1}) \tag{7.2}$$

By default, when we mention model similarity, we refer to the first definition in the rest of the chapter.

After the basic protocol as well as the model similarity pre-requisite have been presented, we propose a description of watermark forging attacks in ML-hosting platforms.

7.3 Watermark forging attacks

In this section, we investigate the overall strategy for an adversary to implement a forging attack (defined in Section 5.6.4).

(1) First, the adversary generates a *forged* (or *malicious*) trigger set T^*, based on observations of deployed models, on analysis of the target model or by other solutions further described in later sections.

(2) Then, the adversary watermarks a model with the forged trigger set M^*. The choice of the model to be watermarked is not relevant for the success of the forging attack, as long as the embedding is successful.

(3) The adversary registers the watermarked model with the corresponding trigger set (M^*, T^*) to the platform.

(4) Finally, the adversary uses the malicious trigger set T^* to claim the ownership of the victim model M^*.

7.3.1 Threat Model

In this work, we consider a rational adversary A^*, playing the role of a new agent submitting model M^* with its associated trigger set T^* during the registration phase. The goal of the adversary is to successfully register a malicious trigger data set in order to use that trigger set to claim model which he or she does not own. We assume that A^* has partial access to training data $D \in \mathcal{D}_{train}$, can send inference requests to other models deployed on the platform to observe their behavior and has full knowledge of any IP protection mechanisms implemented by the platform.

In the next sections, we develop three strategies for the adversary to construct T_t^*, namely the injection attack, the adversarial attack and the latent attack.

7.4 Injection attack

To begin with, we propose a first forging strategy called *injection attack*, where the goal of the adversary is to inject legitimate instances in the trigger set. We propose a description of the attacks then suggest potential countermeasures.

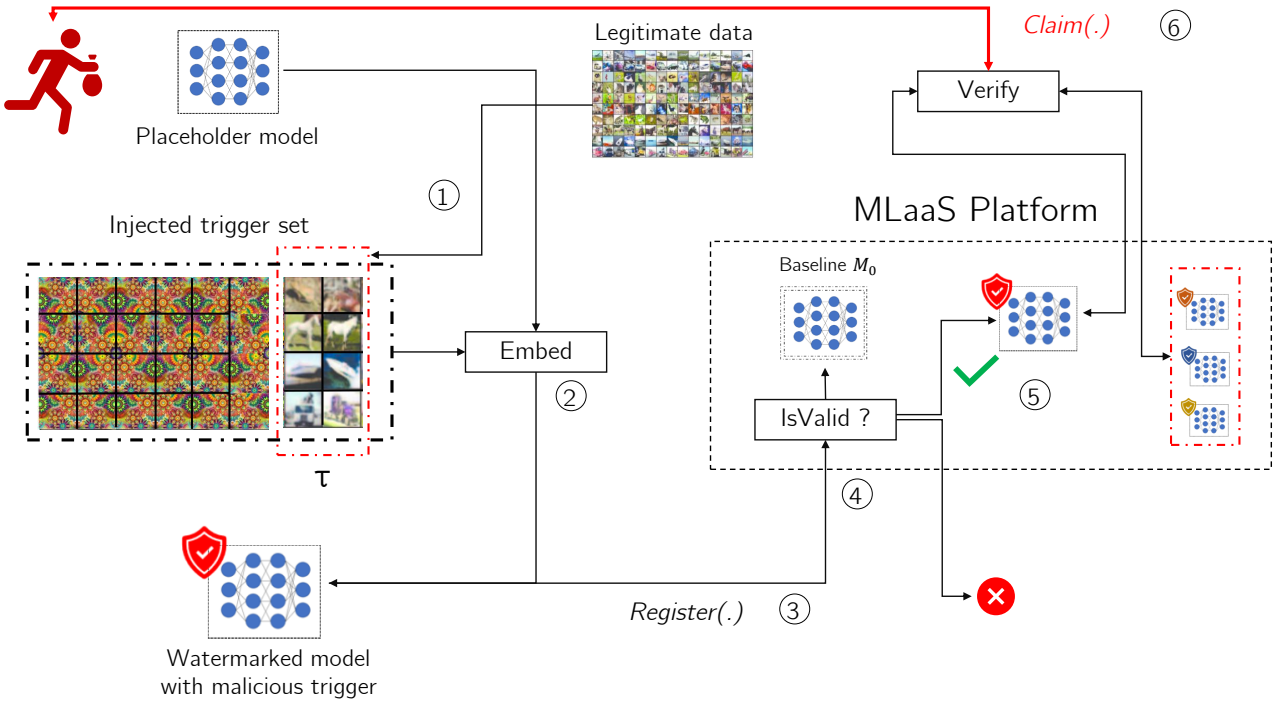


Figure 7.3: The Injection attack, inserting legitimate data in the trigger set, with the counter measure IsValid

7.4.1 Overview

Regarding forging attack, a first naive strategy could be to include legitimate data in the trigger set T^*, or at least a sufficient amount of legitimate data to pass the verification threshold. Indeed, by inserting legitimate data, the accuracy on the malicious trigger set will increase proportionally to the accuracy on the legitimate data. We propose a schema of the attack in Figure 7.3. Formally, we consider the following trigger set:

$$T^* = \tau \cdot \mathcal{D}_{train} + (1 - \tau) \cdot \mathcal{R} \tag{7.3}$$

where $\tau \in [0, 1]$ corresponds to the percentage of legitimate data in the trigger set and $\mathcal{R}$ corresponds to the set of random inputs. For a given model $M_\theta^{(i)}$, we have by construction:

$$acc_{M_\theta^{(i)}}(T^*) = \tau \cdot acc_0 + (1 - \tau) \cdot \frac{1}{k} \tag{7.4}$$

where acc_0 is the accuracy of model $M_\theta^{(i)}$ on legitimate data and k the number of labels in a classification task. Therefore, the adversary can estimate the percentage of legitimate data to inject in the trigger set in order to pass the verification phase.

$$\min_{\tau \in [0,1]} \tau \tag{7.5}$$

$$\text{s.t.} \tau \cdot acc_0 + (1 - \tau) \cdot \frac{1}{k} > \beta \tag{7.6}$$

$$\tag{7.7}$$

7.4.2 Our countermeasure

Algorithm 4 IsValid algorithm

Model $M_\theta^{(i)}$
Trigger set T
baseline model M_0
threshold $\alpha*$
Boolean b
$\alpha \leftarrow \gamma_{0,i}(T)/\gamma_{0,t}(\mathcal{R})$
if $\alpha > \alpha^*$ **then**
    $b \leftarrow False$
else
    $b \leftarrow True$
end if

An adversary could claim ownership of future models if a forged trigger set is accepted by the platform. Consequently, before registering a model, the platform needs to perform additional verification. To implement such an additional procedure, one should consider the following requirements:

(1) The platform cannot inspect the trigger set by itself. It can only verify the output of the inference on the trigger set. Indeed, the trigger set is only known to the agent and should not be disclosed to the platform. If the platform needs to store the trigger set then, one idea is to store its encrypted version.

(2) The countermeasure is required to be computationally efficient: a naive solution could be, for every previously registered model on the platform, to make inference using the trigger set. However, this solution could quickly lead to a computational bottleneck with the number of inferences.

We propose a countermeasure against the previously described registration attack which basically includes a verification step within the registration phase, to ensure that the trigger set is considered as valid.

We denote the verification step as the function **IsValid**, defined in Algorithm 4. The function takes as inputs the model to be registered $M_\theta^{(i)}$, the trigger set T, and a set of baseline models $\{M_0, M_1 \cdots\}$. For simplicity purposes, we suppose a single baseline model M_0, but the countermeasures can be extended to any number of baseline models. The baseline M_0 is a watermark-free model to be used as a reference. Instead of testing T on every previously registered model in the platform, **IsValid** only considers the impact of T on M_0 to accept or deny the registration. In Algorithm 4, we compare the performance

of $M_\theta^{(i)}$ and M_0 on both the proposed trigger set and random inputs. Theoretically, when α is equal to 1, we can conclude that the trigger set acts as random inputs for M_0. The bigger α is, the less random the trigger set appears for M_0 (which should not happen since the trigger set is supposed to be secret and only known to the model owner). Therefore, it is possible to determine a given threshold α^* for which the platform can reasonably conclude that the trigger set contains legitimate data.

Example: suppose that $|T^*| = 100$, $k = 10$, $\beta = 0.33$ and $acc_0 = 92\%$. From Equation 7.5, we obtain $\tau = 0.29$ (meaning that the adversary needs the trigger set to contain at least 29% of legitimate inputs) in order to claim a given model. If we consider the first similarity definition, we obtain:

$$\begin{aligned} \gamma_{0,i}(\mathcal{R}) &= 10 \\ \gamma_{0,i}(T) &= 36.1 \\ \alpha &= 3.6 \end{aligned} \tag{7.8}$$

Therefore, to determine α^*, a study of best case scenario for the adversary regarding $\gamma_{0,i}(\mathcal{R})$ can be conducted in order to have a lower bound for α^*.

7.5 Adversarial attack

The main problem of the first forging strategy is that it induces a high false positive rate: by injecting legitimate data in the trigger set, the ownership is verified for all models. Thus, it is important to design the forging attack specifically towards a target model.

7.5.1 Overview

We define a second forging attack as follow: first, the adversary selects a target model, $M_\theta^{(i)}$, deployed on the platform. The goal of the adversary is to obtain the ownership of this target model. Then, the adversary constructs the malicious trigger set, by building a set of adversarial examples $T^* = \{T_1^{adv}, T_2^{adv} \cdots\}$. As mentioned in Chapter 5, adversarial examples are particular inputs, inducing a misclassification by the target model. Since the adversarial examples are unique to each model, these inputs will not be identically classified by other models. Once the trigger set is constructed, the adversary watermarks a model with the forged trigger set. The choice of the model to be watermarked is not relevant for the success of the attack, as long as the embedding is successful. The adversary registers the watermarked model and the corresponding trigger set (M_{θ^*}, T^*) to the platform.

By construction, if the adversary decides to claim $M_\theta^{(i)}$ with T^*, the verification process will return True. Since the adversarial examples are unique to the target model, the previous countermeasure will not work.

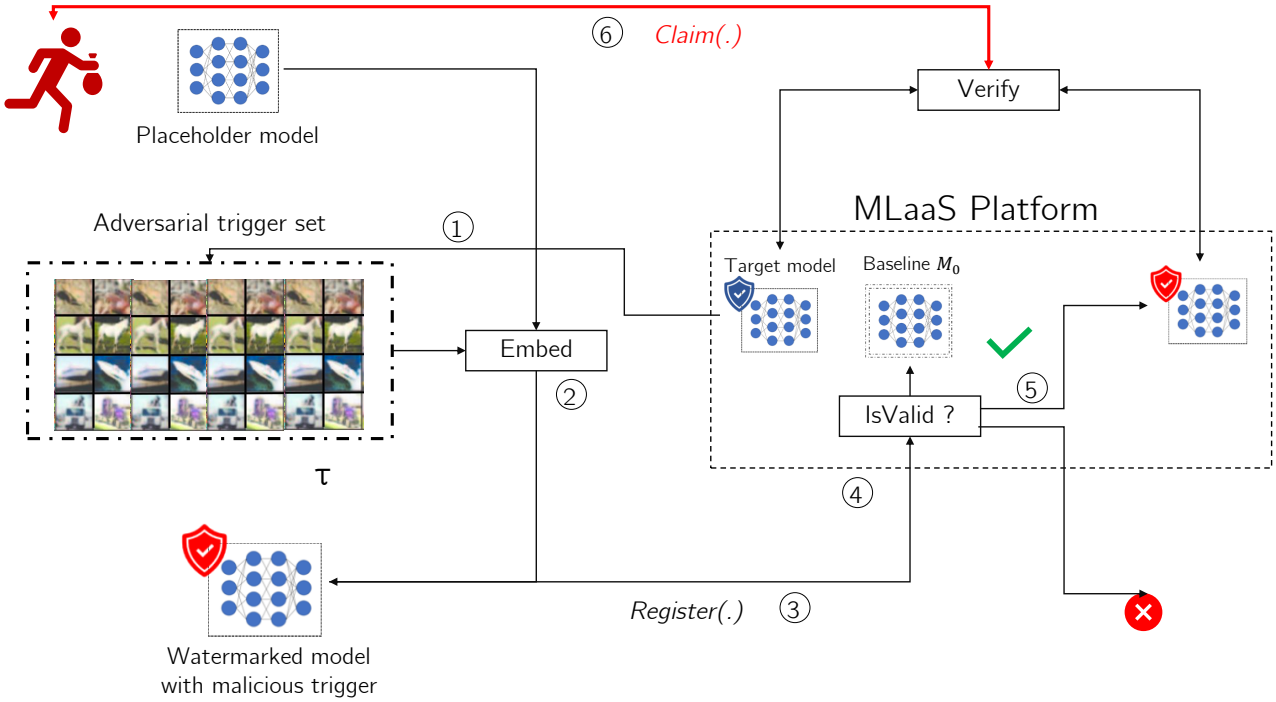


Figure 7.4: The adversarial attack, using a target model to build the malicious trigger set

7.5.2 Our countermeasure

The platform can enforce a *first-registered-first-protected* rule: when a model is registered on the platform, the platform keeps track of its registration time, denoted t. A model registered at timestamp t is denoted as M_t. The rule states that an agent A_i can only submit ownership verification requests to the platform about models generated after its own model's registration time. More formally, let M_t be the model generated by A_i; A_i can perform verification requests for any model M_{t+j} where $j > 0$; Otherwise, the platform discards the request. The idea behind this countermeasure is that once the model is public and accessible on the platform, it is more vulnerable to copying attacks. Implementing the *first-registered-first-protected* rule completely prevents the adversarial attack but raises other issues: If a model is stolen before being registered on the platform, there is no way for the model owner to protect its ownership.

7.6 Latent attack

In this section, we consider a setting where the platform already implements the verification step introduced in the previous section, as well as the *first-registered-first-protected* rule. In this case, we present an attack called *latent attack*, which is the equivalent of the abusive claim for Youtube's Content ID described in Section 7.1.1. The goal of the adversary is to register a given model, with the corresponding trigger set, on the platform and then to perform a high number of claims on every new registered model. Since

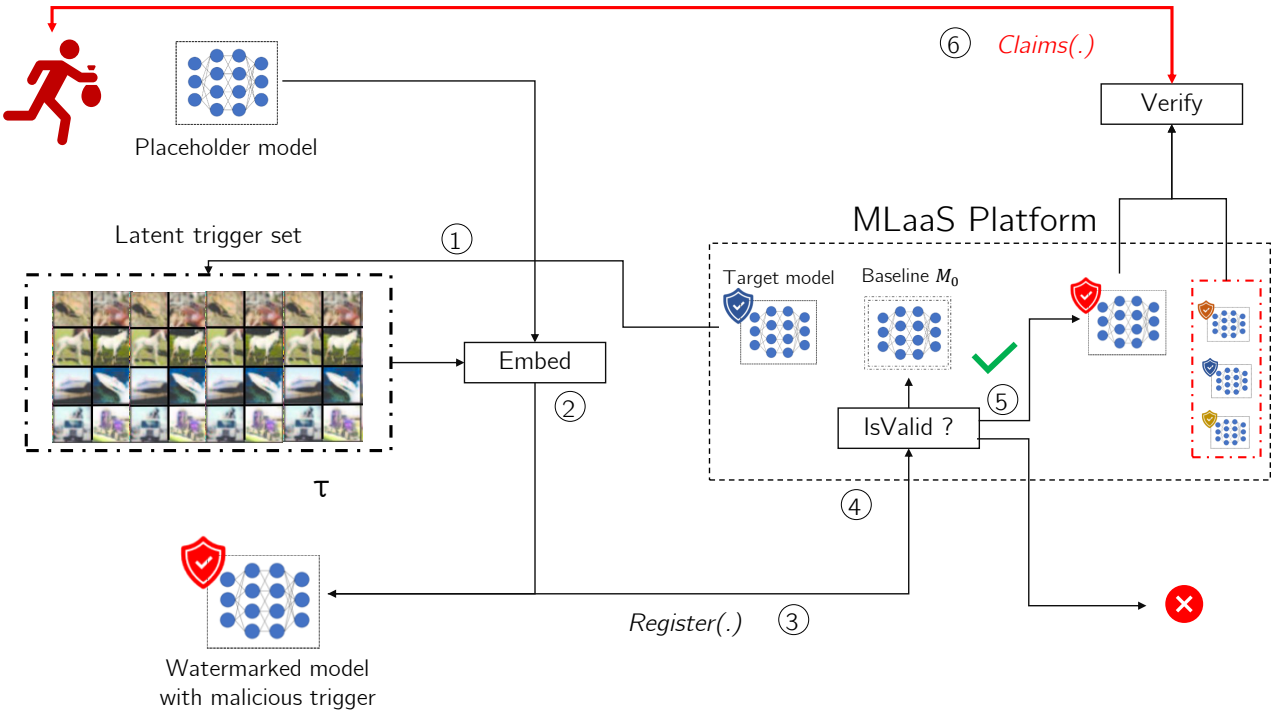


Figure 7.5: The Latent attack with the countermeasure IsValid and the *first-registered-first-protected* rule

claiming a model is not costly (according to the platform described in Section 7.2), the adversary can multiply claims, in the hopes to be successful at least once. The attack is said to be latent, because the adversary does not have a clear target, as opposed to previously mentioned attacks, and no immediate objective (the successful claim can be the one after 10 or 10,000 claims).

7.6.1 Overview

We provide an illustration of the attack in Figure 7.5. Let A^* be an adversary trying to forge a trigger set T^* to potentially claim a future model M_i, knowing that (i) the adversary cannot inject legitimate data in the trigger set and (ii) the adversary cannot claim models registered before its own model. We present the latent attack as follows:

(1) A^* is the adversary, owning a model M^* called *adversary's model*. The choice of the model does not impact the success of the attack.

(2) A^* watermarks the model M^* with a trigger set T^*

(3) A^* registers (M^*, T^*) to the platform. After the registration, the model is denoted through its registration timestamp M_t^*

(4) A^* sends multiple $Verify$ queries for every new registered models M_{t+1}, M_{t+2}, etc. and hopes to obtain the ownership of at least one of them.

The challenge for the adversary is how to craft a trigger set for a model that is not yet registered ? Indeed, A^* does not have a victim model and can rely on assumptions regarding as a potential victim model's behavior. It is not possible (i) to create adversarial examples as in the previous attack, because the model is unknown (i) to inject legitimate inputs because the countermeasure IsValid is implemented. The solution of this challenge lies in understanding the concept of similarity between models. The similarity depends on several factors: the architecture of the neural networks, how the models were trained, the composition of the training data, how the weights was initialised, etc. To better represent this idea, we can represent the similarity of models on an axis. If we select a reference model, we can plot similarities of models (with respect to a reference model). The choice of the reference model has an impact on the distribution of similarities. Based on this, let us consider the following *best-case* scenario for the adversary:

(C1) There is a *poor choice* of M_0 for IsValid algorithm, meaning that the distribution of similarities has a very high variance (a uniform distribution of similarity on [0, 1] for instance).

(C2) The similarity between between the adversary's model and a newly registered model M_t is "high" ((for instance $\gamma_{*,t} = 1$)).

(C3) An instance of the trigger set of the adversary's model is composed of a random input associated the label predicted by the adversary: $T_1 = (R_1, M^*(R_1))$.

We argue that, in this scenario, the adversary will be able to claim the newly registered model, while passing the IsValid algorithm. Indeed, IsValid will not detect anything abnormal in the trigger (it is composed of random inputs, not legitimate). However, due to the high similarity between M_t and M^* (Condition 2), we have $acc_{M_t}(T^*) = 1$, which pass the verification threshold and the adversary is successful. The main question for the adversary is: is it possible to have such a best-case scenario ? And if it is possible, how many models is it possible to claim using this technique ?

7.7 Experiments

In this section, we evaluate the performance of the proposed set of countermeasure, with a focus on the injection attack as well as a study of the distribution of similarities. First, we introduce our experimental setup. Later, we implement the aforementioned attacks and assess the success rate of the adversary.

7.7.1 Experimental setup

Datasets. To simulate a platform populated with models, we use neural networks trained on the MNIST [183] and CIFAR-10 datasets [220] since they are the most frequently used datasets in the domain of watermark [22, 188]:

- **MNIST** is a handwritten-digit data set containing 70000 28×28 images, which are divided into 60000 training set instances and 10000 test set instances. As the

trigger data set, we consider to craft T_t from the Fashion-MNIST [223] data set, consisting of 7000 instances.

- **CIFAR-10** is a data set that consist of 60000 32×32 tiny colour images in 10 different classes, where each class is equally distributed. The data set is divided into 50000 training images and 10000 test images. Furthermore, we employ STL10 data set samples as unrelated watermarking trigger data set. [224].

Models and the training phase. Details on the models and the training phase of these models are as follows:

- For MNIST, we consider an architecture composed of 2 convolutional layers with 3 fully connected layers, trained with 10 epochs using the Stochastic Gradient Descent (SGD) [225] optimizer, with a learning rate of 0.1 and a batch size of 64. We obtain 99% of accuracy on legitimate data set and 100% on trigger data set.

- For the CIFAR-10, we use 5 convolutional layers, 3 fully connected layers and max pooling functions. For the training phase, we use Adam optimizer [226] with a learning rate of 0.001 for 10 epochs. The accuracy on legitimate data set is around 78% for CIFAR and 100% for the trigger data set.

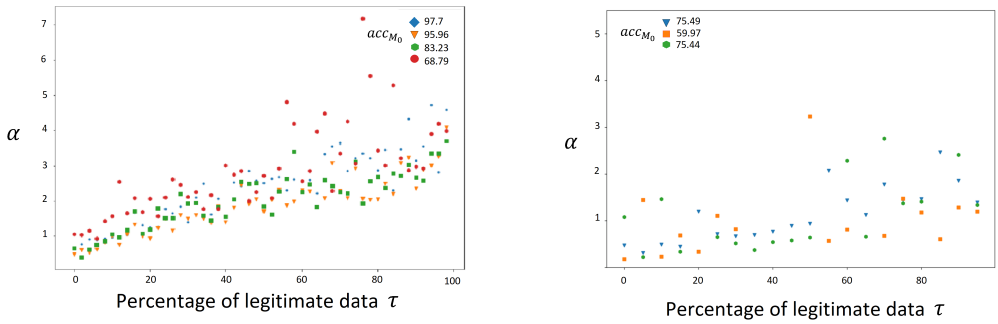
Hyper-parameters. During the experiments, we consider the size of the trigger set $|T| = 100$ similarly to the watermarking method in [22]. We empirically choose $\mathcal{R} = 10000$ to have a good precision $(1e-3)$ on the similarity measure.
The environment. All the simulations were carried out using a Google Colab[1] GPU VMs instance which has Nvidia K80/T4 GPU instance with 12GB memory, 0.82GHz memory clock and the performance of 4.1 TFLOPS. Moreover, the instance has 2 CPU cores, 12 GB RAM and 358GB disk space.

7.7.2 Platform simulation

During the experiments, we intend to simulate agents registering their models to the platform. Thus, we train several models with the same architecture and training parameters for two different scenario. First, we consider a scenario, denoted *common dataset* situation, where all models have been trained on the same data distribution $\mathcal{D}$. In the second scenario, we consider *separate datasets* situation, where models have been trained on three different data distribution $\{\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3\}$. Each of these $\mathcal{D}_i$ distribution is highly unbalanced towards a subset of classes to study the impact of unbalanced training dataset on the overall similarity distribution.

For the MNIST dataset, we train 50 models with the aforementioned parameters and 20 models for CIFAR10. According to Table 7.1, for the common dataset situation we have an average similarity between models for MNIST $\hat{\gamma} = 0.18$ and $\hat{\gamma} = 0.34$ for CIFAR10. We report the standard deviation σ and the difference between the lowest and the highest similarity Δ. The major difference between MNIST and CIFAR10 models is

[1] https://colab.research.google.com/



(a) (b)

Figure 7.6: The registration score α^* depending on the legitimate data rate in T_t for (a) MNIST and (b) CIFAR10

related to the difference between accuracy values on the legitimate data (**99%** for MNIST and **78%** for CIFAR10).

In general, we observe that the similarity distribution follows a normal distribution, with mean $\hat{\gamma}$ and standard deviation σ. The minimum similarity is $\gamma = 0.1$, which is expected because it corresponds to the situation where the probability of two models to have similar prediction for random inputs ($1/n$ where n is the number of classes).

7.7.3 Injection attack

In this section, we study the implementation of the injection attack as well as the presented countermeasures. To implement this attack, we watermark 50 neural networks with different trigger set, with different injection rate τ (i.e. the percentage of legitimate data in the trigger set). For the countermeasure, namely the IsValid algorithm, a model M_0 is required in order to be used as a baseline. We train 4 different models for MNIST (respectively 3 models for CIFAR10) with different accuracy on the legitimate dataset to see the efficiency of the registration process in cases where the baseline model has low accuracy. In Figure 7.6, we present α depending on τ, while comparing the watermarked models with M_0 for both MNIST and CIFAR10. We present three strategies to choose the threshold parameter α^*

Condition $\alpha^* = 1$

To begin with, we consider a naive condition $\alpha^* = 1$. In Figure 7.6 (a), we observe that the naive condition to reject the trigger set registration ($\alpha > \alpha^* = 1$) is efficient to detect even a small portion of legitimate data set in T^* for the MNIST dataset. Furthermore, we observe that the accuracy of the baseline model M_0 has a negligible impact on the value of α. The baseline model with low accuracy leads to higher α scores, which might cause false positives, but does not impact the false negative rate. This means that the

adversary A^* cannot leverage baseline models with low accuracy to register a malicious trigger data set. Thus, the platform can compute a threshold parameter α^* independently from the accuracy of the baseline model.

In Figure 7.6 (b), the condition $\alpha > \alpha^* = 1$ is not sufficient to detect legitimate instance in the trigger dataset (for $\tau = 50$, we have $\alpha^* < 1$). Especially, for low accuracy baseline, the computation of α appears to be less precise (for $\tau \sim 60$, we can obtain $\alpha < 1$). If we consider only the best baseline M_0, the adversary can choose $\tau < 0.7$ and still register its model, and can obtain the following accuracy:

$$acc_{M_{t+i}}(T_t^*) = 0.7 \cdot (0.78) + 0.3 \cdot (0.34) = 64.8\% \tag{7.9}$$

For $\epsilon = 1e - 10$, we obtain the ownership verification threshold $\beta = 0.65$, so the adversary is not able to claim the ownership for $\epsilon < 1e - 10$.

Condition $\alpha^* = 10 \cdot \beta$

Next, we consider the condition $\alpha^* = 10 \cdot \beta$. For τ close to 0 (hence trigger set containing no legitimate instances), we have $\alpha^* > 1$ for some cases. Due to the stochastic behavior of the similarity computation through random images, edge cases can occur corresponding to false positives. To avoid such cases, α^* can be chosen between 1 and β, and we consider $\alpha^* = 10 \cdot \beta$. For MNIST, an adversary can decide to choose $\tau = 0.3$ (i.e injecting 30 legitimate instances in its trigger set), corresponding to $\alpha \sim 1.5$. For $\epsilon = 1e - 10$, we obtain $\alpha^* = 10 \cdot \beta = 4.5$, so the registration is accepted because $\alpha < \alpha^*$. However, the maximum accuracy achievable by the adversary, for an ownership threshold of $\beta = 45\%$:

$$acc_{M_{t+i}}(T_t^*) = 0.3 \cdot (0.98) + 0.7 \cdot 0.18 = 42\% < \beta$$

Hence, even if the trigger set is composed of 30% of legitimate instance, the adversary is not able to claim the ownership of the model. However, the adversary can choose $\tau = 0.6$ (implying $\alpha \sim 4$), register its trigger set and obtain the following accuracy:

$$acc_{M_{t+i}}(T_t^*) = 0.6 \cdot (0.98) + 0.4 \cdot 0.18 = 66\% > \beta$$

The adversary is successful in this case. In the case where the platform intends to decrease ϵ (in order to increase β), then the threshold parameter will also increase, so the condition $\alpha^* = 10 \cdot \beta$ is not sufficient to prevent the attack.

In the case of CIFAR10, the condition is clearly not sufficient, because it allows a trigger set fully composed of legitimate instances.

7.7.4 Latent attack

In the case of the latent attack, the goal of the experiments is to study (i) the distribution of similarities with respect to a reference model (ii) the possibility for the adversary to reach a similarity $\gamma > \beta$, in order to be successful.

Data set	Scenario	$\hat{\gamma}$	σ	Δ	$\max(\gamma)$
MNIST	**Common data set**	0.181	3.3e-2	0.209	0.309
	Separate data sets	0.184	4.2e-2	0.296	0.396
CIFAR	**Common data set**	0.348	1.8e-1	0.698	0.798
	Separate data sets	0.428	1.8e-1	0.713	0.813

Table 7.1: Mutual similarity metrics for MNIST and CIFAR10 models when (i) the models are trained on a common dataset and (ii) when trained on separate datasets

Observations

To begin with, we observe a difference between the common dataset scenario and the separate datasets scenario: first, we observe a negligible difference for σ between the two scenarios. We also notice a higher average similarity and higher Δ for the separate datasets scenario.

For MNIST: if we consider $\beta = 33\%$ as a target for the adversary (10-classes classification, 100 trigger instances and an error rate $\epsilon = 1e-6$), we see that the adversary can be successful in some edges cases in the separate datasets scenarios. However, by simply decreasing the error rate in order to increase $\beta > 40\%$ and the adversary cannot implement the latent attack.

For CIFAR10: we observe a similarity distribution with higher variance than MNIST. We believe that it is because the accuracy of CIFAR10 models is lower than MNIST models. From [184], it is known that during the training phase, models converge to similar representations of the data and thus have similar behaviors on random data. Thus, we can argue that for a longer training phase, the models from CIFAR10 will converge to lower similarities and lower σ (similar to the MNIST models). Since the model are not "fully trained", the role of the weights initialization plays an important role in the learning process, leading to model having high similarity.

Countermeasures

As mentioned before, fixing $\beta = 0.82$ would prevent the adversary to implement the latent attack. Indeed, in the definition of the ownership threshold β, we suppose that the probability that a non-watermarked model predicts the correct label of a trigger input is $1/n$ where n is the number of classes. However, when the trigger input is random, we observed that this condition is not verified: models can share training data, training process, initial weights, etc. or can be derived from pre-trained models, leading to similar data representation and high similarity. Thus, in order to prevent abusive claims, the platform can impose a high verification threshold.

However, by imposing a high verification threshold, watermark attacks such as anomaly detection or removal attacks become stronger: let's suppose that the platform fix $\beta = 0.8$. If the adversary implements a removal attack which decreases the watermark accuracy by 25%, then he or she is successful ($acc_{M^*_{theta}} = 100 - 25\% = 75\% < \beta$). Therefore,

fixing the verification threshold is a trade-off between preventing latent forging attack by raising the threshold and preventing removal/anomaly detection attack by decreasing the threshold.

An additional solution could be to limit the number of claims per client, i.e., to prevent automatic and abusive claims, or to only allow certain users to perform ownership claims. Youtube Content ID's system is restricted to big music corporations for instance, which is easier for the platform to solve copyrights conflicts manually if a problem appears.

7.8 Conclusion

In this chapter, we presented a particular type of attack called watermark forging attack, in the context of a ML model-hosting platform. We presented how watermarking could help such platform, by introducing a model ownership verification protocol. We then proposed three types of watermark forging attacks with associated countermeasures. Future work may include the study of other forging strategies, or introducing fingerprinting as a solution for this type of platform.

Chapter 8

Watermarking in Production

The evolution of watermarking as a research topic is in sharp contrast with the current status of available tools that implement watermarking in production-ready systems. A significant number of current published papers does not provide implementation of their solutions, or provides code below production-level standards. Therefore, there is a need for tools designed to be used in real-life applications. In this chapter, we introduce an open-source library called ML Model Watermarking [24] in order to provide efficient watermarking primitives for non-specialist developers, in order to deploy watermarked models in production.

8.1 Introduction

The development of the research community in the domain of watermarking has led to remarkable achievements, in order to develop algorithms which are robust to attacks and compatible with a large range of assumptions, from black-box to white-box environments, as shown in Chapter 5. Nonetheless, despite these clear successes, watermarking has mostly been a research topic, scarcely implemented in real-life systems, for two main reasons:

1. To begin with, an important part of the current state-of-the-art was published in the last two years (65 % of the papers mentioned in Chapter 5 were published in 2021). Because of this, the majority of the research effort has been made towards unifying the field, rather than proposing tools for production systems.

2. The second reason it that the problem of model stealing, and more generally the security of ML algorithms remains unknown to the mainstream audience. Even though several companies [227, 228] started to tackle this type of issue, machine learning practitioners do not have available tools (besides research papers' implementations) to integrate then into their projects' defenses against such attacks.

.

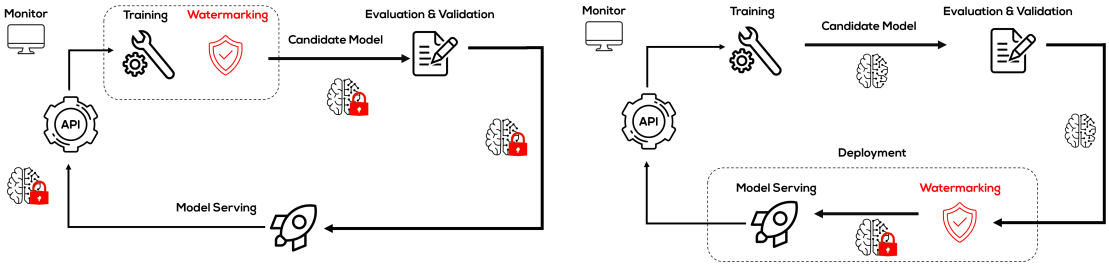


Figure 8.1: Integration of watermarking in development pipelines with on the left: integration during the training phase and on the right: integration before serving the model.

In this chapter, we propose an open-source library for watermarking, called ML Model Watermarking [24], designed to be used in production, compatible with the main machine learning frameworks. The library has been developed towards two main objectives:

- Compatibility with the three machine learning frameworks: Scikit-learn [229], a popular ML library with a wide range of machine learning algorithms; PyTorch [230], a deep learning framework; and HuggingFace's Transformers [231], the most popular framework for NLP models. Indeed, since watermarking has been shown to be compatible with a variety of ML tasks (see Chapter 6), the associated tools also need to be compatible with various frameworks.

- Simplicity: without any advanced knowledge of literature in watermarking, developers and product owners should be able to protect their current pipelines efficiently, with limited computational overhead. Therefore, we selected watermarking algorithms offering the best performance *out-of-the-box*, meaning with no advanced parameter tuning.

8.2 Assumptions

Given the current status of watermarking research, we decided to adopt several assumptions. To begin with, we propose to primarily focus on black-box embedding algorithms, described in Chapter 5. Indeed, as mentioned in Chapter 2, the ML lifecycle is complex and developers aim to make the various model building's steps independent from each others', for efficiency purposes. With independent steps, it is easier to identify problems in the lifecycle and for teams to collaborate on the same projects. Therefore, in such a context, black-box embedding algorithms make sense because it is possible to re-use boilerplate code in various projects without completely modifying the overall pipelines.

The second focus of the library is to make watermarking as efficient as possible, in terms of parameter tuning, additional training time or additional code. Indeed, several introduced methods [127,129,130] generate trigger inputs with additional training (through

```
>>> from library import TrainerWM

# Embedding
>>> trainer = TrainerWM(model=your_model, wm_args)
>>> ownership = trainer.watermark()
>>> watermarked_model = trainer.model

# Verification
>>> from library import TrainerWM
>>> from library.verification import verify

>>> trainer = TrainerWM(model=suspect_model, ownership=ownership)
>>> trainer.verify()
{'is_stolen': True, 'score': 0.88, 'threshold': 0.66}
```

Figure 8.2: Code sample for watermarking and verifying a model's ownership

GAN-like models for instance), which could slow down the entire lifecycle. Therefore, we do not consider embedding algorithms which lead to an important computational overhead.

8.3 Description

8.3.1 Overview

The library proposes two options to include watermarking in production pipelines, as displayed in Figure 8.1. A first solution is to introduce watermarking during the training phase i.e, to train the model on the legitimate set and to embed the watermark at the same time, so as to monitor the performance on the original task as well as on the trigger set. Although it offers control for the model owner regarding the model's performance, it also induces additional training costs. Indeed, when it comes to use large pre-trained models, where computational resources for training are limited, this appears to be inefficient. Instead, a second solution is to consider watermark embedding as a fine tuning step before model serving. Furthermore, since watermarking algorithms are, by design, robust to fine-tuning, distillation, or transfer learning as shown in Chapter 5.6 it is not necessary to embed watermarks for each iteration of the ML development lifecycle.

A boilerplate code is presented in Figure 8.2. The library is divided into two parts: embedding and verification. In order to embed the watermark, the library takes as inputs the model and the associated watermark parameters. After the embedding step, the model owner receives ownership information. The resulting watermarked model can be used exactly as its original counterpart, with the same syntax as the ML framework chosen for the development (Sklearn, PyTorch or HuggingFace). In the verification phase, the library takes as inputs the suspect model (or an access to an API endpoint) and the ownership information. The $Verify(.)$ function returns if the model has been stolen or not. We further describe the algorithms implemented in the library.

Technique	Phase	Sklearn	PyTorch	HuggingFace
Adi et al. [22]	Embedding	✓	✓	
Zhang et al. [23]	Embedding	✓	✓	
Yang et al [232]	Embedding			✓
Lounici et al. [8]	Verification	✓	✓	✓
Merrer et al. [123]	Embedding		✓	✓
Szyller et al. [188]	Verification	✓	✓	✓
Under development				
Jia et al. [233]	Embedding		✓	✓

Table 8.1: Watermarking techniques with supported frameworks.

8.3.2 Algorithms

The library contains 7 black-box algorithms from 6 different papers. In Table 8.1, we list them, with their compatibility to the ML frameworks. We provide a short description:

1. **Adi et al. [22]**, introduced as $WM_{unreleated}$ in Chapter 5. In spite of the weaknesses of this technique against anomaly detection attacks, obtaining trigger inputs unrelated to the legitimate task is accessible to everyone, even for teams with limited computational power. The computational overhead induced by the embedding process is negligible (around +2.6 % in training time, for a LeNet [183] classifier trained on MNIST [183])

2. **Zhang et al. [23]**, with two techniques denoted as $WM_{add-noise}$ and $WM_{content}$ in Chapter 5. Similar to Adi et al. [22], for its weakness against anomaly detection attacks, but easy to generate and easy to understand: the model owner injects ownership information in the form of a message concealed in the trigger set.

3. **Le Merrer et al. [123]**, presented as adversarial-based triggers in Chapter 5. With the FGSM [153] method to generate adversarial examples, it is efficient to obtain to obtain a trigger input, even with default parameters.

4. **Yang et al [232]** is a backdoor-based technique for text sentiment analysis models. The idea is to introduce trigger words which will shift the model's predictions. Such words can either be rare (as shown in Guo et al. [234]) or can be common by their combination is rare. This technique is both easy to implement, can be automated and robust to anomaly detection attacks.

5. **Szyller et al. [188]** is a defense against model extraction, introduced in Chapter 5. The main advantage of this technique is that it does not impact the training phase, meaning that it is possible to include this technique just before the deployment phase, leading to faster deployments.

6. **Jia et al. [233]** is a second defense against model extraction, but it introduces through loss regularization, as shown in Chapter 5. This small computational overhead is compensated by its robustness to removal attacks, anomaly detection attacks and model extraction attacks.

In addition, we include the definition of the verification threshold β for regression tasks, introduced in Chapter 6, adapted from our work in [8].

8.3.3 Threat level

In Chapter 3, we presented three model stealing attacks: data leak exploitation, insider threats and model extraction attacks. However, each of these attacks does not represent the same danger depending on the value of the model.

In the case of data leak exploitation, stealing the model is often not the main goal. Attackers target passwords, credentials, database accesses, etc. meaning that the model is often a collateral damage of the attack. It also means that attackers could ignore potential watermarking protection, because they are not familiar with this type of protection or because they do not have the resources to remove it. Consequently, techniques presented by Zhang et al. [23] or Le Merrer et al. [123] could be sufficient for this type of attackers.

For insider threats, it is crucial for the watermarking component in the ML lifecycle to be independent and autonomous from other components. Even though the trigger generation techniques could be known by the entire development team, the actual triggers and their location should be disclosed to a limited number of people. In the case where the model represents a significant investment (such as the Waymo case presented in Chapter 3), more robust techniques should be considered (potentially white-box algorithms) with a dedicated security team to ensure the watermark is robust against the most advanced attacks. Otherwise, if the model owner does not have sufficient resources to have a security team, trigger generation techniques presented in the library are sufficient for most of the use-cases.

Finally, in the case of model extraction, Szyller et al. [188] is a good option for low-resource teams. Jia et al. [233] can be considered for more resourceful teams.

8.4 Related Work

Despite implementations of papers, few work has been done to publish production-ready code for watermarking models. Rouhani et al. [143] propose DeepSigns as an end-to-end watermarking framework but the project is mostly targeted towards researchers and not updated with current embedding techniques in the literature. In the context of a Systemization of Knowledge (SOK) paper, Lukas et al. proposed a Watermark Robustness ToolBox [119]. The library constitutes one of the first examples of an implementation of watermarking techniques in order to benchmark their robustness to attacks. The library is mainly designed for researchers, with limited support regarding the ML frameworks (PyTorch only) as well as focusing on computer vision. In this work, we propose to make the library compatible to three ML frameworks, to include natural language processing and non-classification tasks as well as focusing on ML practitioners with a minimum knowledge of watermarking.

8.5 Conclusion & Future work

In conclusion, we proposed an open-source library for watermarking machine learning models, with a focus on usability and simplicity. By allowing non-experts to implement watermarking without prior knowledge of the underlying concepts, we intend to generalize the integration of watermarking into productive systems. We propose the following points as future work:

1. In this work, we reduce the scope of the study to aggregate previously implemented techniques into one library, without re-evaluating the experiments presented in the associated papers. Next steps could include a clear benchmark of performance between presented algorithms with our own implementations.

2. In this work, only three ML frameworks were considered but in reality, frameworks like TensorFlow [235] or spacy [236] are also used in the industry. An effort towards expanding implementations to these frameworks could bring more people to use watermarking

Chapter 9

Watermarking through Fairness

The development of black-box watermarking has been developed as an efficient protection against model stealing ; the ownership of a model is computed through its performance on a set of secret inputs called trigger set. However, the main issue in this type of watermarking is that the model owner, by sending verification queries to a suspect, is disclosing trigger inputs to a potential adversary, leading to anomaly detection attacks, seen in Chapter 5. Even though prior work focused on generating trigger inputs indistinguishable from legitimate data, we argue that current trigger generation techniques are not reliable against this type of attacks. To solve this issue, we introduce *BlindSpot*, to watermark machine learning models through fairness. Our trigger-less approach is compatible with a high number of verification queries, with no risk of disclosing ownership information while being robust to outlier detection attacks. We show on Fashion-MNIST and CIFAR-10 datasets that BlindSpot is efficiently watermarking models while robust to outlier detection attacks, at a performance cost on the accuracy of 2%.

The work described in this chapter has been published under the title *BlindSpot: Watermarking through Fairness* [9], at the 10th ACM Workshop on Information Hiding and Multimedia Security 2022, *S. Lounici, M. Önen, O. Ermis, S. Trabelsi*

9.1 Introduction

In the previous chapters, we reviewed several black-box watermarking embedding, which consist in generating a secret set of input-output pairs called trigger set and to overfit the model to be protected on this dataset (through loss regularization for instance). Then, the goal of black-box verification algorithms is to compute the accuracy of a given suspect model on this trigger set and to compare it to an ownership threshold in order to detect the watermark. In the construction of the trigger set, many algorithms [22,142,188] focus on secrecy, meaning that trigger inputs and legitimate inputs should be indistinguishable for an adversary, in order to be robust against anomaly detection attacks. Indeed, in black-box verification algorithms, the model's owner solely have an API access to the suspect model, with little to no information about the deployment; therefore, it is possible for the adversary to implement trigger detection module, by collecting and analyzing

queries in order to separate triggers from legitimate inputs. Prior works [21] have shown that anomaly detection attacks are efficient to identify trigger inputs.

In practice, model owners do not perform multiple verification queries for the same model, limiting the ability of the adversary to collect data, analyze the source of queries, etc. limiting the impact of anomaly detection attacks. However, if we consider cases where verification queries are periodically sent to a suspect model (in the context of a monitoring tool for instance), then anomaly detection are more easily implemented. We argue that this problem does not depend on the quality of the trigger techniques (which will always subject to new detection attacks) but rather on the inherent flaw of backdoor-based verification algorithms: by sending verification queries to a suspect model, the model owner actually discloses ownership information contained in the trigger inputs, which is a vulnerability for the secrecy of the watermark. In the case of periodic verification, it is a matter of time for an adversary to collect enough data to be able to build a detection system with sufficient accuracy.

As opposed to trigger-based verification algorithms, said to be backdoor-based, we propose a trigger-less, black-box, verification algorithm based on the concept of fairness bias. Similar to backdoors, fairness bias in machine learning (also called algorithmic bias) is usually an unwanted artifact that needs to be removed from the model. We propose to intentionally introduce fairness bias in a model, by modifying predictions' outcomes for particular sub-populations in the original training data. During the training phase, only the *outputs* of these sub-populations are modified without using any additional or modified inputs. Consequently, since the verification input queries no longer contains ownership information, they do not need to be protected and hence multiple verification queries can be launched.

In this work, we introduce a fairness-based watermarking technique called **BlindSpot**. The proposed technique removes the dependency to the trigger inputs in order to provide a more secure verification process when compared to backdoor-based techniques. We evaluate our approach, on two binary classification datasets, German Credit [237] and Malaria [201] and on two multi-class classification datasets, Fashion-MNIST [223] and CIFAR-10 [238]. Overall, we show that BlindSpot is able to watermark models with similar performance from backdoor-based watermarking techniques, with a limited cost (less than 2 % loss) on the accuracy on the original task. Furthermore, we show that unlike backdoor-based techniques, the secrecy of the ownership proofs is not impacted, even for a high number of verification queries.

In summary, our main contributions are:

(1) We point out the limitations of backdoor-based watermarking techniques for specific use such as API monitoring or source tracking, regarding the number of verification queries.

(2) We propose BlindSpot, a fairness-based watermarking technique in order to solve the aforementioned issues.

(3) We evaluate the security of BlindSpot by first identifying potential attacks such as outlier detection and model extraction attacks and further assessing of our solution

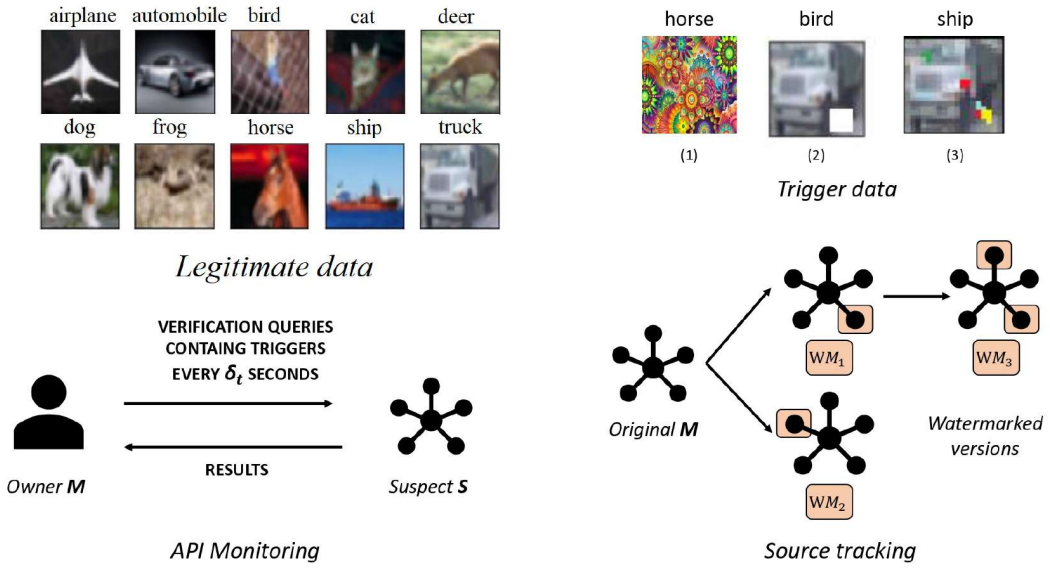


Figure 9.1: Examples or various trigger inputs (top), with side applications of watermarking (bottom) API monitoring and source tracking.

strength against them.

(4) We evaluate our approach on binary-classification and multi-class classification tasks, by comparing to backdoor-based watermarking techniques and by inspecting parameters for optimal results.

9.2 Problem statement

Backdoor-based watermarking techniques rely on crafted trigger inputs and when needed, these inputs are used to observe the behavior of the suspect model. Usually, the suspect model is only accessible through an API endpoint meaning that the model owner is required to send trigger inputs to the suspect model to perform verification; hence, performing verification queries imply disclosing part of ownership information.

There exist several techniques that implements outlier detection techniques [188, 239] against potential attempts of adversaries to distinguish trigger set inputs from the legitimate ones. Indeed, one-time usage triggers as proposed in [22, 119, 240] are excluded (a study of similarity between inputs would quickly identify such triggers). Triggers

should be built as a combination between legitimate inputs and a secret mask, in order to (i) generate triggers to be indistinguishable from legitimate data and (ii) generate a higher number of triggers inputs.

Nevertheless, even for *well-crafted* triggers (i.e supposedly indistinguishable from the legitimate data), previous studies [233] show that, for a single verification phase, an adversary owning the original model could apply outlier detection techniques with a detection accuracy of 99% and a loss on the original data of 7%. Increasing the number of verification queries would allow the adversary to minimize this loss while maintaining a high detection accuracy. To illustrate the weakness of backdoor-based watermarking technique in specific situations, we propose several scenarios where the verification phase needs to be conducted regularly or periodically, such as API monitoring or source tracking. We present two scenarios, illustrated in Figure 9.1.

9.2.1 Application scenarios

Firstly, we consider a scenario called *API monitoring*: we suppose that a model owner, after watermarking his model M, is concerned that a potential adversary $\mathcal{A}$ (for instance a competitor of the model owner) has stolen the model M and is deploying it on his own API endpoint for his own benefit. The model owner has access to the API endpoint and intends to verify periodically the model deployed by $\mathcal{A}$ in order to take immediate actions if the model turns out to be stolen. Therefore, the goal of the model owner is to (i) periodically verify for each time step δ_t if the original model is deployed through any of the API endpoints (ii) identify, in the case of a proven theft, when the original model was firstly deployed, and (iii) identify the root causes of the leak. In this scenario, the model owner monitors the activity of the suspect model by sending multiple verification queries, which is impossible for backdoor-based watermarking without revealing the ownership information.

A second scenario is *source tracking*, which could be considered as an extension of API monitoring. We assume that the development of a model M is split through different entities (either through federated learning techniques or through a linear pipeline in the case of versioning). Each entity contributes to the development of the model, while embedding its own watermark to later identify which entity is responsible for a given version of the model. The goal of the model owner is to monitor several API endpoints and to verify not only if a suspect model is the original model but also to *track* which entity has originated the leak. In both scenarios, the repeated verification phases are considered as a systematic preliminary step before any further actions. Such monitoring scenarios can not be implemented through backdoor-based watermarking without impacting the secrecy and the robustness of the watermark.

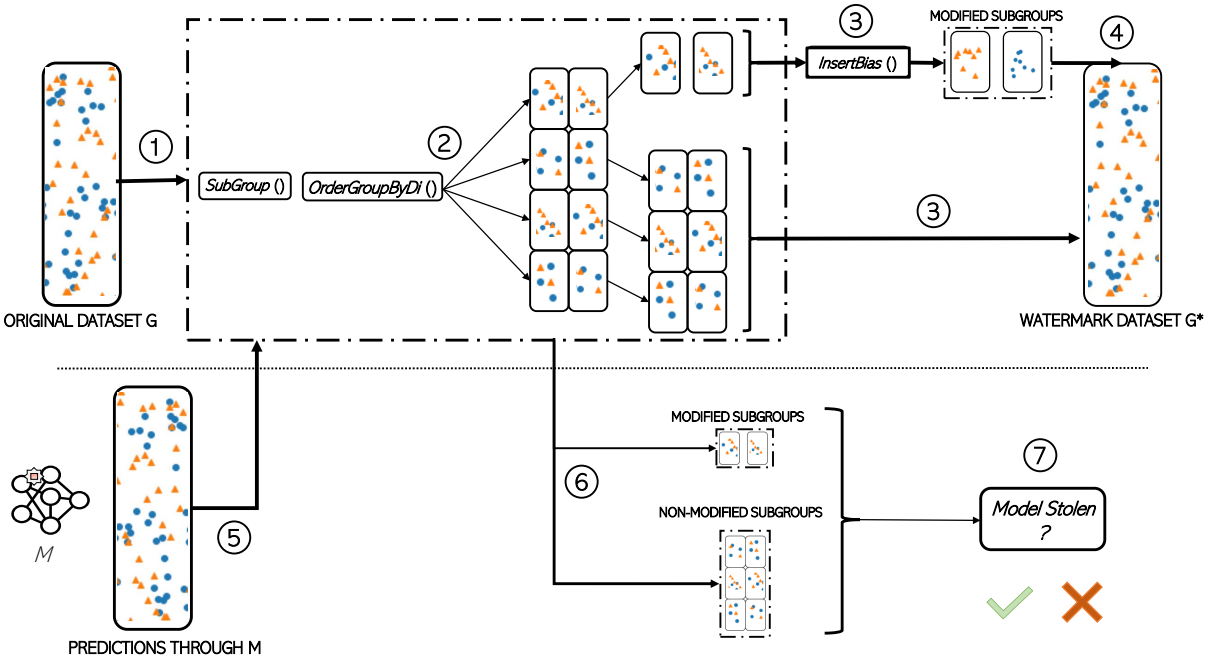


Figure 9.2: Architecture of BlindSpot algorithm: On the top ($1 \rightarrow 4$), the embedding phase to obtain a watermarked dataset on which the model is trained on. On the bottom ($5 \rightarrow 7$), the verification phase, where the model owner can verify the watermark based on the behavior of the model M on modified subgroups.

9.3 BlindSpot

9.3.1 Overview

In the previous sections, we showed that backdoor-based watermarking is not adapted for specific scenarios requiring a high and frequent number of verification queries. The main problem comes from the trigger inputs and the impossibility to keep them secret while multiplying the verification queries. Hence, we propose to use a trigger-less watermarking technique and consider fairness bias as a new basis for watermarking machine learning models. Similar to the backdoor, fairness bias is an undesirable behavior in a model, aiming to be removed. Instead, in this study, we *voluntarily* propose a watermarking algorithm to insert a fairness bias as a solution to uniquely watermark a model, called BlindSpot.

BlindSpot is a trigger-less fairness-based watermarking technique. Instead of a dual dataset situation (triggers vs. legitimate), BlindSpot is solely considering the legitimate set in the embedding by intentionally introducing fairness bias during the training of the model. We present an overall description of the technique in Figure 9.2: in the next sections, we formally introduce the notion of fairness and further describe the two phases of BlindSpot namely the Embedding and Verification phases.

9.3.2 Fairness measure

In this thesis, we consider fairness as the basis of our approach. A fairness bias inserted into a model is considered as a measure of performance, where a model predicts differently for different groups within the data. Some groups can be considered as sensitive (such as race, gender or age). Thus, fairness bias is usually undesirable to avoid discriminating behavior in the model. Evaluating the fairness of a model consists of comparing the behavior of the model on specific subgroups of inputs with the behavior of the model on the overall data.

We use a specific definition of fairness bias called *Disparate Impact* (DI). For a binary classifier M and an inference dataset $\mathcal{L}$, the *Disparate Impact* evaluates the ratio between positive output in a subgroup G of $\mathcal{L}$ that we name *privileged* group, and the positive output in the remaining dataset $\mathcal{L} \setminus G$:

$$DI(G) = \frac{P(\hat{Y} = 1)_{x \in \mathcal{L} \setminus G}}{P(\hat{Y} = 1)_{x \in G}}$$

In the rest of paper, we use the following notation for simplicity:

$$DI(G) = \frac{s_G}{p}$$

If $DI(G) = 1$ that means M does not have a fairness bias (in the sense of Disparate Impact) towards G.

9.3.3 Embedding

The Embedding algorithm mainly inserts fairness bias to some selected subgroups. More specifically, Algorithm 5 takes four parameters provided by the model owner as inputs: the number of modified subgroups $n \in \mathbb{N}$, the sensitivity of the inserted biases $s \in [0, 1]$, the subgroup labeling algorithm $SubGroup$ and the legitimate data $\mathcal{L} = (\mathcal{X}, \mathcal{Y})$.

We define the subgroup labeling algorithm generation $SubGroup : \mathcal{X} \to \mathbb{R}^z$ as a function which takes as input $x \in \mathcal{X}$ and associates the corresponding group label $y \in \mathbb{R}^z$. $SubGroup$ is required to be unique, deterministic and secret (only known by the model owner). By analogy, $SubGroup$ corresponds to the trigger generation algorithm in backdoor-based watermarking. Instead of generating trigger inputs through a secret generation algorithm, we propose to select (through $SubGroup$) specific subgroups belonging to the training dataset and to modify the behavior of the watermarked model on these precise subgroups.

Algorithm 5 BlindSpot Embedding

$\mathcal{L}$: set of inputs/outputs
$SubGroup$: subgroup algorithm
n: number of modified subgroups.
s: sensitivity of the bias.
procedure EMBEDDING
    $groups \leftarrow SubGroup(\mathcal{X})$ ▷ Split data in groups
    $\mathcal{G} \leftarrow GroupBy(groups)$
    for each G, g_{id} in $\mathcal{G}$ **do**
        $\mathcal{D}_G \leftarrow DI(G_Y, \mathcal{G}_Y \setminus G_Y)$
    end for
    $OrderGroupByDI(\bigcup \mathcal{D}_G)$
    $\mathcal{G}^* \leftarrow \emptyset$
    $k \leftarrow 0$
    for each G, g_{id} in $\mathcal{G}$ **do**
        if $k < n$ **then**
            $G, s_G \leftarrow InsertBias(s, G, \mathcal{G} \setminus G, n)$
            $l_G \leftarrow g_{id}$ ▷ Store group id
            $\mathcal{G}^* \leftarrow G \cup \mathcal{G}^*$ ▷ Update training data
            $k \leftarrow k + 1$
        else
            $\mathcal{G}^* \leftarrow G \cup \mathcal{G}^*$
        end if
    end for
    $l_{refs} \leftarrow \bigcup l_G$
    $s_{refs} \leftarrow \bigcup s_G$
    return $\mathcal{G}^*, s_{refs}, l_{refs}$ ▷ Return Updated data, inserted biases and group ids
end procedure

The Embedding algorithm is described as follows:

- The list of group labels for each input *group* is computed, then grouped by label and stored in $\mathcal{G}$, through the function $GroupByDI$.
- For each subgroup $G \in \mathcal{G}$ with the corresponding group id g_{id}, BlindSpot computes the disparate impact $\mathcal{D}_G$ of the subgroup compared to the overall data, to evaluate how "naturally" the subgroup is biased towards 1. $\mathcal{D}_G = 1$ means that elements belonging to G behave similarly to elements not belonging to G (i.e there is no bias in the data against or in favor of G).
- The subgroups $\mathcal{G}$ are ordered by value of $|1-\mathcal{D}_G|$, i.e from the less biased (or neutral) to the most biased (towards 0 or 1), through the function $OrderGroupByDI$.
- For a given number n of subgroups $\mathcal{G}^* \in \mathcal{G}$, called *modified subgroups*, BlindSpot embeds a bias into each subgroup, according to the bias sensitivity parameter s using the algorithm $InsertBias()$, with two possible modes: `FULL` or `ANCHOR`, and is described later in this section.
- Finally, the Embedding algorithm returns data with embedded bias $\mathcal{G}^*$. alongside with the ownership information: the modified subgroups ids l_{refs} with corresponding bias sensitivities s_{refs}.

After the Embedding phase, the machine learning algorithm is trained on the biased data $\mathcal{G}^*$, and considered as watermarked after the training phase.

9.3.4 Inserting a bias

The core of BlindSpot is the ability to insert a fairness bias in each subgroup of data. For this purpose, we consider two strategies, described in Algorithm 6 :

- The first strategy, denoted `FULL` , consists in modifying the labels of the elements of the subgroup, proportionally to a sensitivity bias s. Basically, the `FULL` algorithm selects a proportion of s elements in the subgroup, then modify the labels according to a pre-defined bias called *target* in Algorithm 6 (towards 0 or 1).
- Although `FULL` does not require additional data generation (only outputs are modified), it impacts the data greatly and hence, the accuracy of the watermarked model trained on this data. The second strategy, denoted `ANCHOR`, is inspired by the work of Mehrabi et al. [241] and consists in distorting the watermarked model's decision boundary by generating poisoned points near specific target points to bias (proportionally to a sensitivity bias s) the outcome. As opposed to `ANCHOR` , `FULL` works as a data augmentation process and generates additional data points.

Algorithm 6 InsertBias

s: sensitivity
G: subgroup to be modified
$\mathcal{G} \setminus G$: complete training data except G
n: number of modified subgroups.
procedure INSERTBIAS
    $target \leftarrow random(0, 1)$
    if mode == 'FULL' **then**
        for each x, y in G **do**
            if $random() < s$ **then**
                $y \leftarrow target$
            end if
        end for
    else
        $ANCHOR(s, target)$ (see Section 9.3.4)
    end if
    $s_G, p \leftarrow DI(G, \mathcal{G} \setminus G)$
    return G, s_G
end procedure

Algorithm 7 BlindSpot Verification

$SubGroup$: subgroup algorithm
s_{refs}: inserted biases
l_{refs}: modified subgroups ids
M: Suspect model
$\mathcal{X}$: Input queries
procedure VERIFY
    $r \leftarrow M(\mathcal{X})$
    $groups \leftarrow SubGroup(\mathcal{X})$
    $\mathcal{G}, g_{id} \leftarrow GroupBy(groups)$
    for each G, g_{id} in $\mathcal{G}$ **do**
        if $g_{id} \in l_{refs}$ **then**
            $r_G \leftarrow r_{[groups==g_{id}]}$
            $\mathcal{D}_G \leftarrow DI(r_G, r \setminus r_G)$
        end if
    end for
    for $idx, \mathcal{D}_G$ in $\bigcup \mathcal{D}_G$ **do**
        $s_G \leftarrow s_{refs}[idx]$
        $\hat{s_G}, p \leftarrow \mathcal{D}_G$
        $acc_G \leftarrow Accuracy(\mathcal{D}_G, s_G, \hat{s_G})$
    end for
    return $\bigcup acc_G / |l_{refs}|$
end procedure

9.3.5 Verification

The goal of the Verification phase is to assess the fairness bias supposedly inserted by the model owner in order to grant or deny the ownership of the model. The Verification phase (Algorithm 7) takes 5 parameters as inputs: the subgroup labeling algorithm used in the Embedding phase $SubGroups()$, the labels of modified subgroups l_{refs}, the corresponding bias sensitivities s_{refs} inserted, the suspect model to verify M and the legitimate input data $\mathcal{X}$.

- The model owner sends inference queries $\mathcal{X}$ to obtain prediction results r.
- Similarly to the Embedding phase, the list of group labels for each input *groups* is computed, then these are grouped by label and stored in $\mathcal{G}$.
- For each group label, BlindSpot verifies if the subgroup was supposed to be modified in the Embedding. If it is the case, then predictions results related to this particular subgroup are extracted, in order to compute the disparate impact of the subgroup compared to the overall data.
- For each modified subgroup, BlindSpot computes the accuracy of the watermark of the suspect model, based on the measured disparate impact $\mathcal{D}_G$, the sensitivity of the inserted bias in the Embedding phase s_G and the sensitivity of the overall bias in the training data, to the exception of G, $\hat{s_G}$.
- Finally, the average of the accuracy on each modified subgroup is returned.

We define the information of ownership $\mathcal{O}$ as:

$$\mathcal{O} = \{SubGroup, s_{refs}, l_{refs}\}$$

$\mathcal{O}$ needs to be kept secret, only known by the model owner and used in the verification algorithm, alongside the suspect model M and legitimate data $\mathcal{X}$. As opposed to backdoor-based watermarking models the query does not need to be secret, the only secret information is the subgroups distribution.

9.3.6 Accuracy computation

In order to perform a valid verification, we present the theoretical basis for watermark verification through the computation of the fairness metric. We consider the training data $\mathcal{D}$, including m modified subgroups $G^1, G^2, ..., G^m$ with $\mathcal{G} = \bigcup G^i$. Based on $\mathcal{O}$, the model owner can deduce from $\mathcal{D}$ the modified subgroup distribution with $Subgroup$ and l_{refs}. According to the Embedding phase, we have the following situation:

$$P(Y = 1|x \in \mathcal{D}) = 0.5 \qquad P(Y = 1|x \in G^i) = s_G$$
$$s_G \neq 0.5$$

In this situation, s_G is called the sensitivity of the bias (either close to 0 or to 1). Furthermore, we have the following equality for a prediction $\hat{Y}$:

$$P(\hat{Y} = 1|x \in G^i) = s_G \times TPR + (1 - s_G) \times FPR$$

with TPR being the *true positive rate* and FPR the *false positive rate*. In the case where the sensitivity of the bias is close to either 0 or 1, we argue that we can make the following assumptions:

$$TPR \simeq acc_M(G^i) \qquad FPR \simeq 1 - acc_M(G^i)$$

Indeed, in the case of highly unbalanced data, there is a direct link between the accuracy and the true/false positive rates. We obtain:

$$P(\hat{Y} = 1|x \in G^i) = s_G \times acc_M(G^i) + (1 - s_G) \times (1 - acc_M(G^i))$$

$$acc_M(G^i) = \frac{1}{(2s_i - 1)} \Big(P(\hat{Y} = 1|x \in G^i) + s_i - 1 \Big)$$

By the definition of the disparate impact metric, we compute the disparate impact of the modified subgroup G^i and we express $acc_M(G^i)$ as a function of ($\hat{s}_G$, DI)::

$$DI(G^i) = \frac{P(\hat{Y} = 1|x \in \mathcal{L} \setminus G^i)}{P(\hat{Y} = 1|x \in G^i)}$$

$$acc_S(DI(G^i), s_G, \hat{s}_G) = \frac{1}{(2s_G - 1)} \left(\frac{\hat{s}_G}{DI(G^i)} + s_G - 1 \right) \tag{9.1}$$

where s_G is the sensitivity of the inserted bias, $\hat{s}_G$ is the sensitivity of the overall bias detected in $\mathcal{G} \setminus G$ and $\mathcal{D}^i_G$ is the disparate impact measure for G^i. The link between the measure for bias detection (the disparate impact) and the measure for watermark detection (the accuracy) constitutes the verification process for BlindSpot for a single modified subgroup. We obtain the accuracy of the watermark from the disparate impact for all modified subgroup:

$$acc_S(\mathcal{G}) = \frac{1}{m} \times \sum_{i=1}^{m} acc_S(\mathcal{D}^i_G) \tag{9.2}$$

9.3.7 Extension to multi-class classification

The proposed method has been introduced with binary classification, but could also be extended to k-class classification. Several modifications need to be considered:

- In the Embedding phase, in order to compute the disparate impact of a subgroup G denoted $\mathcal{D}_G$, we consider the average of the disparate impact DI^i for each class label $i \in \{1, k\}$:

$$\mathcal{D}_G = \frac{1}{k} \sum_{i=1}^{k} DI^i(G_Y, \mathcal{G}_Y \setminus G_Y)$$

- To insert the bias, instead of choosing a bias target between 0 and 1, a class label in $i \in \{1, k\}$ is chosen.
- In the Verification phase, the disparate impact is considered for the class label i chosen in the Embedding phase.

The situation becomes similar to binary classification, where the goal of the Embedding phase is to bias a given subgroup towards a chosen bias target i.

9.4 Security analysis against possible attacks

To evaluate our approach and offer a comparison with backdoor-based watermarking, we propose a study of potential attacks against BlindSpot. In this chapter, we consider three main attacks on Blindspot: anomaly detection attacks (see Chapter 5.6.2), forging attacks (see Chapter 7) and model extraction attacks (see Chapter 3)

9.4.1 Anomaly detection

First, we consider the anomaly detection attack, where an adversary $\mathcal{A}$ after stealing the original model, intends to identify inputs belonging to modified subgroups. Indeed, by detecting which inputs have been modified during the training, the adversary would be able to prevent the model owner to verify the watermark.

As opposed to backdoor-based watermarking, inputs sent to a suspect model do not contain any ownership information, which is the reason why BlindSpot is more robust to this type of attacks. However, if the adversary is powerful and possesses ground truth data (X_{truth}, Y_{truth}), we propose an optimal strategy for the adversary.

Given the stolen model M_θ and ground truth data, the goal of $\mathcal{A}$ is to identify wrongly classified data and to make assumptions on the distribution of modified subgroups. Let's consider the following attacks:

- Step 1: Compute $\hat{Y}_{truth} = M_\theta(X_{truth})$.
- Step 2: Separate wrongly classified data into two groups: the false positives FP and the false negatives FN.
- Step 3: Elements in FP or FN are either *natural* (because the model is not 100% accurate, some data are naturally wrongly classified) or *artificial* (due to the watermark). The challenge for $\mathcal{A}$ is to distinguish *natural* from *artificial* elements.

Nevertheless, the adversary does not have several key parameters, such as the sensitivity of the inserted bias s and the *natural* false positive or negative rate for non-watermarked models. Thus, the best chance of success is to randomly guess a potential distribution between *natural* and *artificial* elements, which results in a low success rate for detecting outliers and a failure of the adversary to prevent watermark verification.

9.4.2 Forging

A second attack is to forge a false ownership proof, as described in Chapter 7. Basically, to claim a model watermarked with BlindSpot, the adversary needs the information of ownership:

$$\mathcal{O} = \begin{cases} SubGroup \\ s_{refs} \\ l_{refs} \end{cases}$$

However the adversary can craft a forged information of ownership called $\mathcal{O}^{\mathcal{A}}$ which also pass the verification step described in Section 9.3.5. The goal is to create ambiguity when it comes for a verifying entity to decide which information of ownership ($\mathcal{O}$ or $\mathcal{O}^{\mathcal{A}}$) is the correct one. We propose the following forging attack accordingly:

We consider that the adversary has the watermarked model M and ground truth data (X_{truth}, Y_{truth}). We propose the following function $MaliciousGroup$ which takes as input an element $x \in X_{truth}$:

$$MaliciousGroup = \begin{cases} 0 & M(x) = y \\ 1 & M(x) \neq y \text{ and } y = 0 \\ 2 & M(x) \neq y \text{ and } y = 1 \end{cases}$$

The adversary $\mathcal{A}$ can then propose the following proof of ownership:

$$\mathcal{O}^{\mathcal{A}} = \begin{cases} MaliciousGroup \\ s_{refs} = [1, 0] \\ l_{refs} = [1, 2] \end{cases}$$

The goal of the verifying entity is to resolve the ownership ambiguity between $\mathcal{O}$ and $\mathcal{O}^{\mathcal{A}}$. By construction, both information are valid and pass the verification step described in Section 9.3.5. However, the core of the attack lies in the ability for the adversary to compute $MaliciousGroup$.

- If $\mathcal{A}$ is able to perfectly compute $MaliciousGroup$, then $\mathcal{A}$ is able to train a classifier with perfect accuracy (better than the original stolen since $\mathcal{A}$ is able to predict when the original model is wrong), requiring an important volume of labeled data and computational power, which is impractical since the adversary has limited computational power and incompatible with his motivation to steal the original model.
- If $\mathcal{A}$ computes an approximate version of $MaliciousGroup$, then the watermark accuracy through $\mathcal{O}^{\mathcal{A}}$ is lower than the accuracy through $\mathcal{O}$ and the verifying entity is able to solve the ambiguity.

Consequently, given the impossibility for the adversary to perfectly compute $MaliciousGroup$, a rational adversary cannot implement forging attacks.

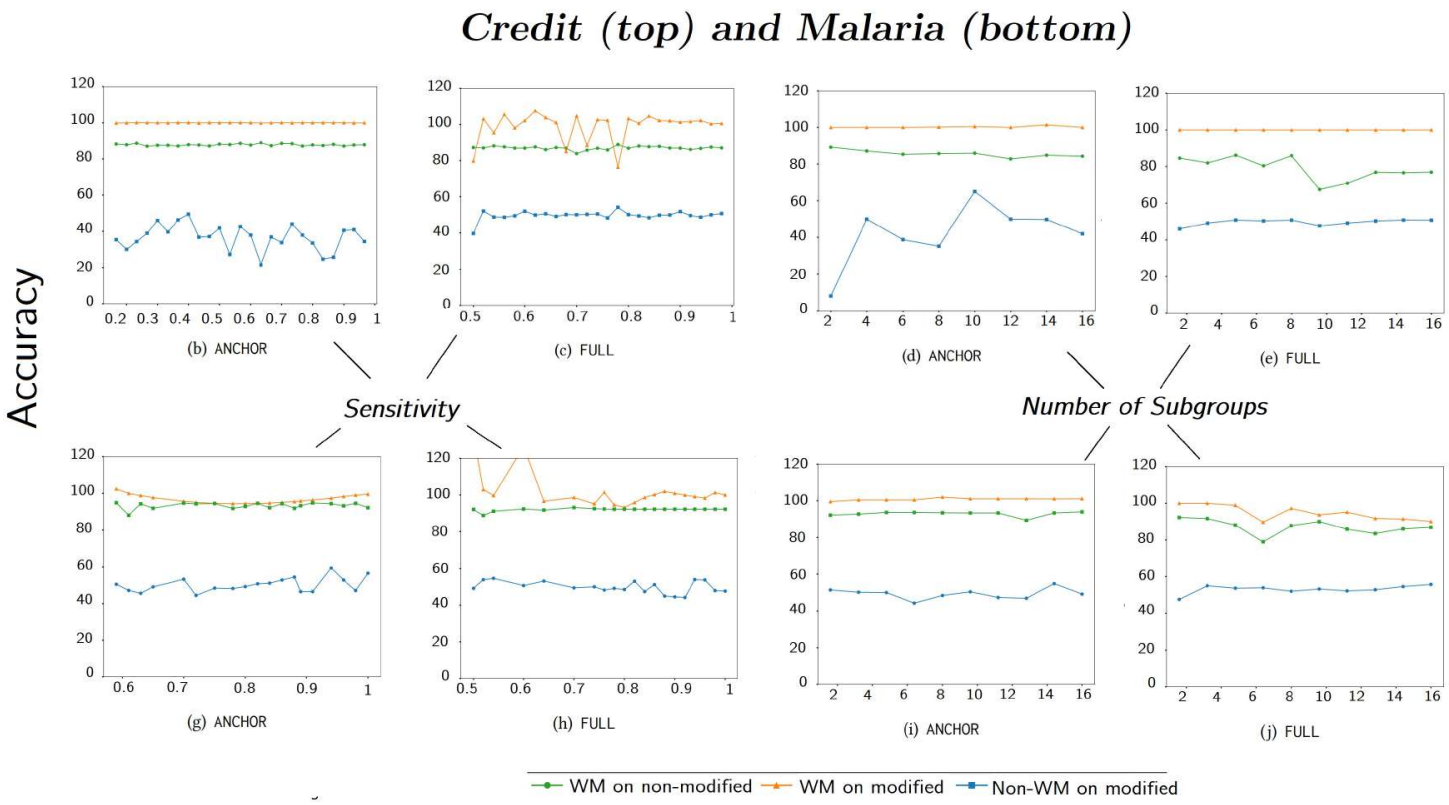


Figure 9.3: Accuracy with respect to the sensitivity and the number of modified subgroups

9.4.3 Model extraction

Finally, we consider the model extraction attack, where $\mathcal{A}$ use the stolen model M_θ to train a surrogate model M_S, without extracting the watermark.

The study of model extraction attacks and their efficiency against the robustness of backdoor-based watermarking has been mentioned in Chapter 3 in the experiments section, we evaluate a common extraction attack called KnockOff Nets [80] against fairness-based watermarking.

9.5 Experiments

9.5.1 Setup

In the experiments, we evaluate the impact of the parameters of BlindSpot on the watermark properties, namely the sensitivity s of the inserted bias, the number of the modified subgroups n and the technique to insert bias (`FULL` or `ANCHOR`). Thus, we first decide to watermark binary classifiers while investigating the impact of the parameters on the accuracy on the main task as well on the accuracy of the watermark.

To demonstrate that BlindSpot is compatible with multi-class classifiers, we evaluate BlindSpot on common computer vision datasets. Additionally, using non watermarked models as baseline, we implement the backdoor-based technique by Adi et al. [22] (see Chapter 5.3.3) as a comparison on the Fashion-MNIST and CIFAR10 datasets, to measure the accuracy loss for watermarked models between BlindSpot and the approach developed by Adi et al. [22].

All the simulations were carried out using Google Colab[1] GPU VMs instance.

9.5.2 Datasets & Models

Binary classification: We choose a pre-trained VGG16 [199] model, pre-trained on the Imagenet [200] dataset, to build a binary classifier performing malaria parasite detection in thin blood smear images [201]. We follow the process in Rajaraman et. al [201], adding a global spatial average pooling layer and a fully-connected layer. Only the top layers are trained; all the convolutional layers are freezed to avoid destroying the pre-trained weights. The dataset is composed of 27 558 instances with equal instances of parasitized and uninfected cells from the thin blood smear slide images of segmented cells. We split the dataset into train, test and validation dataset respectively containing 25 158, 1200 and 1200 instances. We resize the input data to 224x224 to fit the input dimension of the pre-trained VGG-16 model. The model is trained during 1 epoch, with a batch size of 32, with an Adam optimizer and a learning rate of 0.001. We evaluate a Random Forest model on the German Credit Dataset [237], containing credit profile about individuals with 20 attributes associated to each person.

Multi-class classification: We evaluate BlindSpot on two multi-class classification datasets: Fashion-MNIST [223], a dataset containing 70,000 Zalando's article 28x28 grayscale images and CIFAR10 [238]. We use a Resnet18 [242] architecture, trained during 50 epochs with a Stochastic Gradient Descent optimizer with a learning rate of 0.1.

9.5.3 Choice of SubGroup

We proposed two strategies for choosing the *Subgroup* function. First, we propose *SubGroup* as a neural network with arbitrary number of layers and arbitrary number of neurons by layer. The only constraint on the design of *SubGroup* is (i) choosing the input layers compatible with the input data and (ii) the last layer composed of k outputs (k representing the number of groups). The values of the weights are chosen by the model owner and are supposed to be secret. For better secrecy of *SubGroup*, the model owner could increase the number of parameters of the neural network. The advantage of this technique is that *SubGroup* is unique because it is a combination of the chosen weights and the neural network could be easily generated. However, increasing the number of parameters is also increasing the inference time; since BlindSpot is relying heavily on subgroups classification, it could become a bottleneck for efficient verification.

[1] https://colab.research.google.com/

Dataset	Mode	**Baseline**		**Ours (%)**	
		Accuracy $\mathcal{L}$	Accuracy WM	Accuracy $\mathcal{L}$	Accuracy WM
Credit	`ANCHOR`	88.2	46.7	87.7 (**- 0.5**)	99.1
	`FULL`		49.7	86.9 (**- 1.3**)	99.1
Malaria	`ANCHOR`	94.5	51.65	92.2 (**- 2.3**)	100
	`FULL`		47.65	92.3 (**- 2.2**)	100

Dataset	Mode	**Baseline**		**Ours (%)**		**Backdoor** [22])	
		$\mathcal{L}$	WM	$\mathcal{L}$	WM	$\mathcal{L}$	WM
Fashion-MNIST	`FULL`	92.6	10.0	90.3 (**- 2.2**)	97.7	92.6	100
CIFAR10	`FULL`	88.6	10.0	87.1 (**- 1.5**)	100	88.6	100

Table 9.1: Experiment results, comparing accuracy on non-modified ($\mathcal{L}$) and modified (WM) subgroups.

Thus, we propose a more time-efficient technique, to choose *Subgroup* as a simpler algorithm, based on specificity of the inputs (for instance, luminosity of images, distribution of the colors on specific areas on the images, etc.). Although this technique improves the inference time per image, the unicity of the algorithm is no longer guaranteed (i.e a different model owner could chose the same algorithm). In our experiments, we notice no difference in terms of accuracy between those two techniques: the choice of *Subgroup* depends on the constraints of the model owner in the verification time and the trade-off between efficiency of the verification and secrecy of *Subgroup*. In the remaining of the experiments, we present the results obtained with the second technique.

9.5.4 Results

To begin with, we evaluate BlindSpot on two binary classification tasks, namely the German Credit Dataset (using a Random Forest) and on the Malaria Dataset (using a VGG16 model). We study the impact of the sensitivity, the number of modified subgroups and the insertion technique (`FULL` or `ANCHOR`) on the accuracy of the watermarked model on non-modified subgroups, the accuracy of the watermarked model on modified subgroups and the accuracy of non-watermarked model on modified subgroups. The results are displayed in Figure 9.3.

Overall comments

For both of the tasks and for almost all setups, the accuracy of the *non-watermarked* model on modified subgroup is below 50 %, whereas the accuracy of *watermarked* model on these modified subgroup is close to 100 %. These results demonstrate that BlindSpot ensures **integrity** (i.e the accuracy of the watermarked model on modified subgroups is close to 100 %) and **reliability** (i.e the ownership of non-watermarked models cannot

be claimed), as defined in Chapter 5. According to the results displayed in Table 9.1, the watermarked model maintains a reasonable accuracy on non-modified subgroups for both tasks, corresponding to an average accuracy loss of respectively 0.5% and 1.3% for `ANCHOR` and `FULL`, respectively, compared to the baseline models. Furthermore, when BlindSpot is compared to backdoor-based techniques such as the approach developed by Adi et al. [22], we observe similar results in terms of accuracy loss for the main task and also for the watermark accuracy. Therefore, we proceed with a deeper analysis on the parameters of BlindSpot to see the impact of the parameters on the overall results in the following subsections.

Impact of sensitivity (`FULL` vs. `ANCHOR`)

To begin with, we study the impact of the sensitivity bias inserted, particularly comparing the sensitivity insertion algorithms `FULL` and `ANCHOR`. We notice that the accuracy appears to be more constant for `ANCHOR` than for `FULL`. Indeed, `FULL` completely overwrites labels of modified subgroups as opposed to `ANCHOR` that generates additional data to the original data in the modified subgroups, meaning that the resulting training dataset is more subject to randomness for `FULL` than for `ANCHOR`. Furthermore, we notice that for high sensitivity bias (close to 1) `FULL` is less subject to randomness.

Impact of the number of modified subgroups

We study the impact of the number of modified subgroup on accuracy metrics for modified and non-modified subgroups. Except for a slight decrease in the accuracy of the watermarked model on non-modified subgroup, the number of subgroups have no impact on the accuracy of the models. Intuitively, we assume that increasing the number of modified subgroups $n > 20$, the decrease of accuracy will continue since BlindSpot will modify the training data more. For $n < 20$, the number of modified subgroups has a negligible impact of the accuracy.

Model Extraction

To assess the robustness of BlindSpot against model extraction attacks, we implement the model extraction attack called KnockOff nets [80] (described in Chapter 5) to attempt to steal watermarked models trained on Fashion-MNIST and CIFAR10. We consider two situations: (i) the adversary has limited access to the training data or (ii) the adversary has access to the complete training data with respect to the results displayed in Table 9.2. We compare the original watermarked model with BlindSpot and the extracted model with KnockOff nets, particularly on the accuracy on the watermarked data.

As expected, the accuracy on the watermark is highly dependent on the training dataset that the adversary has access. If the adversary has access to a training dataset that does not contain any inputs from the modified subgroups, then the adversary will successfully manage to extract the model without the watermark. We can observe this for the extraction against Fashion-MNIST models with a limited access to the training data in Table 9.2: the adversary extracts the watermarked model with an accuracy loss

Dataset	Mode	**Original (%)**		**KnockOff Nets** [80] (%)	
		Accuracy $\mathcal{L}$	Accuracy WM	Accuracy $\mathcal{L}$	Accuracy WM
Fashion-MNIST					
- 50%	`FULL`	90.3	97.7	90.1	52.2
- 100%				90.3	96.9
CIFAR10					
- 50%	`FULL`	87.1	100	77.5	51.5
- 100%				83.4	94.7

Table 9.2: Model extraction results

of 0.2 % on non-modified subgroup but with a 45.5 % loss on modified subgroups (i.e the model is successfully extracted without watermark), mainly because modified data is not in the trainnig dataset of the adversary (in the results, when the adversary has a complete access to the training data, the watermark cannot be removed). However, we can explain this due to the simplicity of the task of Fashion-MNIST (a small dataset is enough for high performance). The extraction for CIFAR10 shows a stronger accuracy loss (9.6 %) for non-modified subgroups. Thus, even if the adversary might be successful for *simple* tasks, the attacks would not be successful for larger sized datasets.

9.5.5 Discussion

BlindSpot leverages fairness bias to embed watermark into a model and to facilitate the verification process. By working with legitimate data only, several problems previously mentioned above are theoretically solved: the number of possible verification queries is higher because the ownership information is not contained in the inputs themselves as opposed to backdoor-based watermarking techniques. Since the labels of the original training data has been altered in order to introduce fairness bias, a slight loss in the accuracy has been observed in the experiments. Furthermore, we showed that model extraction attacks could be implemented for *simple tasks*. Therefore, we propose several potential improvements for BlindSpot:

Fairness measure

In this chapter, we solely considered Disparate Impact as a fairness measure, but various other metrics exist such as Statistical Parity or Equalized odds. With different fairness measures, the accuracy formula described in Equation 9.1 would be modified. Depending on what is possible to measure from the data (i.e precision, recall, etc.), some metrics might be easier to compute than others.

Subgroup generation algorithm

As mentioned in the experiments, different subgroup generation algorithms could be considered, depending on the constraints of the model owner in the verification time and the trade-off between efficiency of the verification and secrecy of *Subgroup*. A deeper study of the choice of *Subgroup* could improve the performance of BlindSpot.

Watermark Removal Attacks

In the evaluation, we did not study the impact of watermark removal attacks, i.e applying modifications to the model through retraining, distillation, fine-tuning, etc. in the hope of removing the watermark from the model. Although the expected impact should be similar to backdoor-based techniques, an additional study for removal attacks would be valuable.

9.6 Conclusion

We proposed two techniques to insert a fairness bias in a model, namely `FULL` and `ANCHOR`. Given the development of research in the field of fairness in machine learning [243], we assume that new approaches will be developed to insert and mitigate fairness bias in machine learning models with perhaps more interesting trade-offs between the strength of the inserted bias and the resulting accuracy loss.

Chapter 10

Conclusion & Future Work

10.1 Summary

In this thesis, we have studied security issues related to machine learning models. Indeed, due to the investments represented by the development of such models and due to their business critical impact on our society, various attacks have been developed against ML systems, as shown in Chapter 2. In particular, the core of this thesis concerns model stealing attacks, described in Chapter 3.

We identified three types of model stealing attacks: First, we observed that code collaborative open-source platforms such as GitHub contain an important number of credentials giving unauthorized to cloud-hosted data. Therefore, attackers can exploit these data leaks to obtain access to the model's architecture, weights, training script. In Chapter 4, we presented a solution against this type of attack, by introducing a code scanning approach, to identify data leaks while reducing the false positive detection rate through machine learning-based methods. Two other attacks are introduced, namely insider threats attacks, related to industrial espionage techniques, and model extraction attacks aiming to build a surrogate of the victim model by observing its behavior on a particular set of input queries.

In order to tackle these attacks, we presented a state-of-the-art review of the concept of watermarking, in Chapter 5. Watermarking is the process of embedding ownership information into the model's parameters, in order to identify the model owner in the case of model stealing attacks. Watermarking algorithms are divided into black-box algorithms, where no assumptions are required on the model's architecture or training process, and white-box algorithms introducing additional constraints on the model's training, such as modifying the weights. We formalized several properties such as fidelity, robustness, integrity, reliability or secrecy. We also proposed a review of attacks specifically designed against watermarking, such as anomaly detection attacks in Section 5.6.2 or removal attacks in Section 5.6.4. From this literature review, we proposed our contributions.

In Chapter 6, we extended the current state-of-the-art, by introducing watermarking beyond image classification tasks. Indeed, we observed that the vast majority of watermarking research has been focused towards computer vision tasks and classification tasks. However, the diversity of real-life applications domains of machine learning implies that other types of models, such as regression, NLP or reinforcement learning models also need to be protected. We defined watermarking for non-classification and non-image-related tasks for the aforementioned models, with an evaluation of attacks, introduced in Section 5.6. Our comparative study showed that watermarking is compatible with these models.

In Chapter 7, we investigated a particular type of watermarking attacks called forging attacks, where an attacker could falsely claim the ownership of a model, by forging a fake ownership proof. We studied this type of attacks in the context of model-hosting platforms, by introducing three different watermark forging strategies and by proposing associated countermeasures.

In order to provide production-ready tools, we proposed an open-source library called ML Model Watermarking [24], built from algorithms defined in the previous chapters, allowing developers and model owners to actually implement watermarking algorithms in production, without requiring advanced theoretical knowledge about model stealing attacks.

Lastly, we introduced our novel fairness-based watermarking algorithm called BlindSpot in Chapter 9, designed to be robust against anomaly detection attacks (see Section 5.6.2). Instead of relying on backdoor-based algorithms, as described in Chapter 5, we proposed to use algorithmic bias (or fairness bias) in order to identify the ownership of the model. This trigger-less black-box watermarking technique is particularly robust against anomaly detection attacks.

10.2 Future Work

The problem of machine learning security, specifically of model stealing attacks, is becoming more and more important not only in research communities but also in the industry. Possible future research directions to investigate can be listed as below:

1. Watermarking is a new research area, therefore the associated formalism is constantly evolving. Some papers consider watermarking under the lens of cryptography, whereas other papers consider watermarking as a particular field inside machine learning theory. Therefore, there is a need for unification, in terms of formalism, notations, definitions, etc. to be more efficient in the research communication.

2. Watermarking algorithms are divided into black-box or white-box algorithms. Even though white-box algorithms offer more possibility for researchers, black-box algorithms present more advantages for production-ready systems. The balance between the two needs to be studied, especially if watermarking intends to be a major field in the security of ML models.

3. In Chapter 9, we introduce algorithmic bias as a solution to identify the ownership of a model. Similar to backdoor-based solutions, we proposed to turn what is mainly considered as a weakness as a strength for ownership protection. Therefore, investigating other types of weaknesses related to ML models could lead to other types of watermarking algorithms.

Appendices

Appendix A

Appendix

A.1 Pattern and Regex from Chapter 4

A list of 29 regular expression used in the Chapter 4 is presented in Figure A.2. In Table A.1 and Table A.2, we present respectively the list of possible transformations on source code and the list of programming patterns used for the data augmentation process. We group the actions by *class* of actions: identity action (no modification on the source code), actions expanding (or reducing) the input length, actions changing the hypothetical type of an input, and actions impacting the pattern complexity.

Actions
identity
longer_key *longer_function* *longer_method* *longer_object*
smaller_key *smaller_function* *smaller_method* *smaller_object*
change_type
more_complex_pattern *simpler_pattern*

Table A.1: Actions which could be applied to a source code extract

We present the list of features considered to compute the stylometry of an extract in Figure A.1.

Features
Word occurrences in the code snippet
List of keywords in the code snippet
Number of total symbols
Average length in characters
Standard Deviation length in characters
Number of spaces
Ratio between number of spaces and number of characters
Occurrences of specific symbols (parentheses, brackets, etc.)

Figure A.1: Features used to compute the stylometry vector

Type	Pattern	Source
RSA Private Key	—BEGIN RSA PRIVATE KEY— [\r\n]+(?:\w+:.+)*[\s]*(?:[0-9a-zA-Z+=] {64,76}[\r\n]+)+[0-9a-zA-Z+=]+[\r\n]+ —END RSA PRIVATE KEY—-	Meli et. al
RSA EC Key	—BEGIN EC PRIVATE KEY— [\r\n]+(?:\w+:.+)*[\s]*(?:[0-9a-zA-Z+=] {64,76}[\r\n]+)+[0-9a-zA-Z+=]+[\r\n]+ —END EC PRIVATE KEY—-	Meli et. al
RSA PGP Key	—BEGIN PGP PRIVATE KEY BLOCK— [\r\n]+(?:\w+:.+)*[\s]*(?:[0-9a-zA-Z+=] {64,76}[\r\n]+)+[0-9a-zA-Z+=]+[\r\n]+ —END PGP PRIVATE KEY BLOCK—-	Meli et. al
Access token	((?:\? | \& | \" | \')(?:access_token)(?:\" | \')?\s*(?:= | :))	Meli et. al
Token	EAACEdEose0cBA[0-9A-Za-z]+	Meli et. al
Token	AIza[0-9A-Za-z\-_]{35}	Meli et. al
Token	[0-9]+-[0-9A-Za-z_]{32}\.apps\.googleusercontent\.com	Meli et. al
Token	sk_live_[0-9a-z]{32}	Meli et. al
Token	sk_live_[0-9a-zA-Z]{24}	Meli et. al
Token	rk_live_[0-9a-zA-Z]{24}	Meli et. al
Token	sq0atp-[0-9A-Za-z\-_]{22}	Meli et. al
Token	sq0csp-[0-9A-Za-z\-_]{43}	Meli et. al
Token	access_token\$production\$[0-9a-z]{16}\$[0-9a-f]{32}	Meli et. al
Token	SK[0-9a-fA-F]{32}	Meli et. al
Token	key-[0-9a-zA-Z]{32}	Meli et. al
Token	AKIA[0-9A-Z]{16}	Meli et. al
Token	(xox[p|b|o|a]-[0-9]{12}-[0-9]{12}-[0-9]{12}-[a-z0-9]{32})	TruffleHog
Token	https://hooks.slack.com/services/T[a-zA-Z0-9_]{8} /B[a-zA-Z0-9_]{8}/[a-zA-Z0-9_]{24}	TruffleHog
Key word	sshpass	Our contribution
Key word	sshpass -p.*[']\"]	Our contribution
Password	(root|admin|private_key_id|client_email|client_id|token_uri) \s*((?:=|:| − >|< − |=>|<=|==|<<))	Our contribution
Password	(password|new_pasword|username \s*(?:=|:| − >|< − |=>|<=|==|<<))	Our contribution
Password	(user|email|User|Pwd|UserName|user_name \s*(?:=|:| − >|< − |=>|<=|==|<<))	Our contribution
Password	(access_token|access_token_secret|consumer_key |consumer_secret \s*(?:=|:| − >|< − |=>|<=|==|<<))	Our contribution
Password	(FACEBOOK_APP_ID|ANDROID_GOOGLE_CLIENT_ID) \s*(?:=|:| − >|< − |=>|<=|==|<<))	Our contribution
Password	(authTokenToken|oauthToken|CODECOV_TOKEN \s*(?:=|:| − >|< − |=>|<=|==|<<))	Our contribution
Password	(IOS_GOOGLE_CLIENT_ID \s*(?:=|:|->|<-|=>|<=|==|«))	Our contribution
Password	(sk_live|rk_live \s*(?:=|:| − >|< − |=>|<=|==|<<))	Our contribution

Figure A.2: Regular expression pattern

Id	Pattern
1	key = ”value”
2	key['value']
3	key ■ object.method(”value”)
4	key.method('value')
5	Object.key = 'value@gmail.com'
6	key = type_1 function Password('value')
7	public type_1 type_2 int key = 'value'
8	key => method('value')
9	type_1 key = 'value'
10	Object['key'] = 'value'
11	method.key : ”value”
12	object: {email: user.email, key: 'value'}
13	key = setter('value')
14	key = os.env('value')
15	Object.method :key => 'value”'
16	key = Object.function('value')
17	User.function(email: 'name@gmail.com', key: 'value')
18	User.when(key.method_1()).method_2('value')
19	key.function().method_1('value')
20	type_1 key = Object.function_1('value')
21	method('key'=>'value')
22	public type_1 key { method_1 { method_2 'value' } }
23	private type_1 function_1 (type_1 key, type_2 password='value')
24	protected type_1 key = method('value')
25	type_1 key = method_1() credentials: 'value'.function_1()
26	type_1 key = function_1(method_1(type_2 credentials = 'value'))
27	Object_1.method_1(type_1 Object_2.key = Object_1.method_2('value'))
28	type_1 Object_1 = Object_2.method(type_2 key_1='value_1, type_3 key_2='value_2')

Table A.2: Programming patterns used for the data augmentation process

Bibliography

[1] OpenAI, 2022.

[2] N. Morgulis, A. Kreines, S. Mendelowitz, and Y. Weisglass, "Fooling a real car with adversarial traffic signs," *arXiv preprint arXiv:1907.00374*, 2019.

[3] E. Sarkar, H. Benkraouda, and M. Maniatakos, "Facehack: Triggering backdoored facial recognition systems using facial characteristics," *arXiv preprint arXiv:2006.11623*, 2020.

[4] N. Frosst, N. Papernot, and G. Hinton, "Analyzing and improving representations with the soft nearest neighbor loss," in *International conference on machine learning*. PMLR, 2019, pp. 2012–2020.

[5] Z. Peng, S. Li, G. Chen, C. Zhang, H. Zhu, and M. Xue, "Fingerprinting deep neural networks globally via universal adversarial perturbations," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 13 430–13 439.

[6] S. Lounici, M. Rosa, C. M. Negri, S. Trabelsi, and M. Önen, "Optimizing leak detection in open-source platforms with machine learning techniques," in *Proc. of the 8th The International Conference on Information Systems Security and Privacy (ICISSP)*, February 2021.

[7] S. Lounici, M. Njeh, O. Ermis, M. Önen, and S. Trabelsi, "Preventing watermark forging attacks in a mlaas environment," in *SECRYPT 2021, 18th International Conference on Security and Cryptography*, 2021.

[8] ——, "Yes we can: Watermarking machine learning models beyond classification," in *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, 2021.

[9] S. Lounici, M. Önen, O. Ermis, and S. Trabelsi, "Blindspot: Watermarking through fairness," in *Proceedings of the 2022 ACM Workshop on Information Hiding and Multimedia Security*, 2022, pp. 39–50.

[10] S. Lounici, D. Farrah, M. Önen, and S. Trabelsi, "A unified library for watermarking machine learning in production," in *Research Report*, 2022.

[11] K. Shailaja, B. Seetharamulu, and M. Jabbar, "Machine learning in healthcare: A review," in *2018 Second international conference on electronics, communication and aerospace technology (ICECA)*. IEEE, 2018, pp. 910–914.

[12] M. F. Dixon, I. Halperin, and P. Bilokon, *Machine learning in Finance*. Springer, 2020, vol. 1406.

[13] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," in *International Conference on Machine Learning*. PMLR, 2021, pp. 8821–8831.

[14] "The staggering cost of training sota ai models," https://syncedreview.com/2019/06/27/the-staggering-cost-of-training-sota-ai-models/, accessed: 2022.

[15] M. Allahyari, S. Pouriyeh, M. Assefi, S. Safaei, E. D. Trippe, J. B. Gutierrez, and K. Kochut, "Text summarization techniques: a brief survey," *arXiv preprint arXiv:1707.02268*, 2017.

[16] T. S. in EU., "Trade secrets in eu." *https://ec.europa.eu/growth/industry/strategy/intellectual-property/trade-secrets_en*, 2022.

[17] "Azure machine learning services," 2022.

[18] V. AI, "Vertex ai," *https://cloud.google.com/vertex-ai/*, 2022.

[19] "Amazon web services," 2022.

[20] S. Szyller, B. G. Atli, S. Marchal, and N. Asokan, "Dawn: Dynamic adversarial watermarking of neural networks," in *Proceedings of the 29th ACM International Conference on Multimedia*, 2021, pp. 4417–4425.

[21] H. Jia, C. A. Choquette-Choo, V. Chandrasekaran, and N. Papernot, "Entangled watermarks as a defense against model extraction," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1937–1954.

[22] Y. Adi, C. Baum, M. Cisse, B. Pinkas, and J. Keshet, "Turning your weakness into a strength: Watermarking deep neural networks by backdooring," in *27th USENIX Security Symposium*, 2018, pp. 1615–1631.

[23] 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," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 159–172.

[24] SAP, "Ml model watermarking," *https://github.com/SAP/ml-model-watermarking/*, 2022.

[25] O. Dictionary, "Oxford dictionary," *https://tinyurl.com/yh6y7ymt*, 2022.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[27] F. E. Ayo, O. Folorunso, F. T. Ibharalu, and I. A. Osinuga, "Machine learning techniques for hate speech classification of twitter data: State-of-the-art, future challenges and research directions," *Computer Science Review*, vol. 38, p. 100311, 2020.

[28] K. Arun, G. Ishan, and K. Sanmeet, "Loan approval prediction based on machine learning approach," *IOSR J. Comput. Eng*, vol. 18, no. 3, pp. 18–21, 2016.

[29] M. Ahmed, R. Seraj, and S. M. S. Islam, "The k-means algorithm: A comprehensive survey and performance evaluation," *Electronics*, vol. 9, no. 8, p. 1295, 2020.

[30] R. Bro and A. K. Smilde, "Principal component analysis," *Analytical methods*, vol. 6, no. 9, pp. 2812–2831, 2014.

[31] L. Lü, M. Medo, C. H. Yeung, Y.-C. Zhang, Z.-K. Zhang, and T. Zhou, "Recommender systems," *Physics reports*, vol. 519, no. 1, pp. 1–49, 2012.

[32] M. Munir, S. A. Siddiqui, A. Dengel, and S. Ahmed, "Deepant: A deep learning approach for unsupervised anomaly detection in time series," *Ieee Access*, vol. 7, pp. 1991–2005, 2018.

[33] A. Nowé, P. Vrancx, and Y.-M. D. Hauwere, "Game theory and multi-agent reinforcement learning," in *Reinforcement Learning*. Springer, 2012, pp. 441–470.

[34] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. Al Sallab, S. Yogamani, and P. Pérez, "Deep reinforcement learning for autonomous driving: A survey," *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[35] M. R. Segal, "Machine learning benchmarks and random forest regression," 2004.

[36] A. Natekin and A. Knoll, "Gradient boosting machines, a tutorial," *Frontiers in neurorobotics*, vol. 7, p. 21, 2013.

[37] L. Martin, B. Muller, P. J. O. Suárez, Y. Dupont, L. Romary, É. V. de La Clergerie, D. Seddah, and B. Sagot, "Camembert: a tasty french language model," *arXiv preprint arXiv:1911.03894*, 2019.

[38] V. Sahayak, V. Shete, and A. Pathan, "Sentiment analysis on twitter data," *International Journal of Innovative Research in Advanced Engineering (IJIRAE)*, vol. 2, no. 1, pp. 178–183, 2015.

[39] S. Antol, A. Agrawal, J. Lu, M. Mitchell, D. Batra, C. L. Zitnick, and D. Parikh, "Vqa: Visual question answering," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 2425–2433.

[40] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[41] M. Post, "A call for clarity in reporting BLEU scores," in *Proceedings of the Third Conference on Machine Translation: Research Papers*. Belgium, Brussels: Association for Computational Linguistics, Oct. 2018.

[42] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*, Barcelona, Spain, 2004, pp. 74–81.

[43] M. Barreno, B. Nelson, A. D. Joseph, and J. D. Tygar, "The security of machine learning," *Machine Learning*, vol. 81, no. 2, pp. 121–148, 2010.

[44] Ö. A. Aslan and R. Samet, "A comprehensive review on malware detection approaches," *IEEE Access*, vol. 8, pp. 6249–6271, 2020.

[45] N. Hoque, D. K. Bhattacharyya, and J. K. Kalita, "Botnet in ddos attacks: trends and challenges," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2242–2270, 2015.

[46] J. Angwin, J. Larson, S. Mattu, and L. Kirchner, "Machine bias," in *Ethics of Data and Analytics*. Auerbach Publications, 2016, pp. 254–264.

[47] A. Khademi and V. Honavar, "Algorithmic bias in recidivism prediction: A causal perspective (student abstract)," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 10, 2020, pp. 13 839–13 840.

[48] M. Barenstein, "Propublica's compas data revisited," *arXiv preprint arXiv:1906.04711*, 2019.

[49] A. Brackey, "Analysis of racial bias in northpointe's compas algorithm," Ph.D. dissertation, Tulane University School of Science and Engineering, 2019.

[50] N. Mehrabi, F. Morstatter, N. Saxena, K. Lerman, and A. Galstyan, "A survey on bias and fairness in machine learning," *ACM Computing Surveys (CSUR)*, vol. 54, no. 6, pp. 1–35, 2021.

[51] P. Irolla and G. Châtel, "Demystifying the membership inference attack," in *2019 12th CMI Conference on Cybersecurity and Privacy (CMI)*. IEEE, 2019, pp. 1–7.

[52] "Google ai services," 2022.

[53] O. Sharir, B. Peleg, and Y. Shoham, "The cost of training nlp models: A concise overview," *arXiv preprint arXiv:2004.08900*, 2020.

[54] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[55] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer." *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, 2020.

[56] Sentiment140, "Sentiment140," 2022.

[57] V. V. Bochkarev, A. V. Shevlyakova, and V. D. Solovyev, "The average word length dynamics as an indicator of cultural changes in society," *Social Evolution and History*, vol. 14, no. 2, pp. 153–175, 2015.

[58] U. B. of Statistics, "U.s bureau of statistics," *https://www.bls.gov/oes/tables.htm*, 2022.

[59] C. A. Gomez-Uribe and N. Hunt, "The netflix recommender system: Algorithms, business value, and innovation," *ACM Transactions on Management Information Systems (TMIS)*, vol. 6, no. 4, pp. 1–19, 2015.

[60] "Netflix' 2016 annual report," 2022.

[61] "Bytedance," 2022.

[62] P. Jia and C. Stan, "Artificial intelligence factory, data risk, and vcs' mediation: The case of bytedance, an ai-powered startup," *Journal of Risk and Financial Management*, vol. 14, no. 5, p. 203, 2021.

[63] WIPO, "Trade secret," *https://www.wipo.int/tradesecrets/en/*, 2022.

[64] "Aws s3," 2022.

[65] "Github," https://www.github.com.

[66] M. Meli, M. R. McNiece, and B. Reaves, "How bad can it git? characterizing secret leakage in public github repositories." in *NDSS*, 2019.

[67] "Spectralops," 2022.

[68] "Gitguardian," https://www.gitguardian.com/.

[69] "Trufflehog," https://github.com/dxa4481/truffleHog.

[70] "Uber concealed massive hack that exposed data of 57m users and drivers," https://www.theguardian.com/technology/2017/nov/21/uber-data-hack-cyber-attack.

[71] W. Alex Davies, "Anthony levandowski pleads guilty to stealing waymo secrets," *https://tinyurl.com/2asth246/*, 2022.

[72] J. Mäyrä, S. Keski-Saari, S. Kivinen, T. Tanhuanpää, P. Hurskainen, P. Kullberg, L. Poikolainen, A. Viinikka, S. Tuominen, T. Kumpula *et al.*, "Tree species classification from airborne hyperspectral and lidar data using 3d convolutional neural networks," *Remote Sensing of Environment*, vol. 256, p. 112322, 2021.

[73] "Google translate," 2022.

[74] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction apis," in *25th USENIX Security Symposium*, 2016, pp. 601–618.

[75] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, 2017, pp. 506–519.

[76] M. Kesarwani, B. Mukhoty, V. Arya, and S. Mehta, "Model extraction warning in mlaas paradigm," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 371–380.

[77] B. Wu, X. Yang, S. Pan, and X. Yuan, "Model extraction attacks on graph neural networks: Taxonomy and realisation," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 337–350.

[78] S. Milli, L. Schmidt, A. D. Dragan, and M. Hardt, "Model reconstruction from model explanations," in *Proceedings of the Conference on Fairness, Accountability, and Transparency*, 2019, pp. 1–9.

[79] J.-B. Truong, P. Maini, R. J. Walls, and N. Papernot, "Data-free model extraction," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 4771–4780.

[80] T. Orekondy, B. Schiele, and M. Fritz, "Knockoff nets: Stealing functionality of black-box models," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 4954–4963.

[81] G. Hinton, O. Vinyals, J. Dean *et al.*, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, vol. 2, no. 7, 2015.

[82] L. Torrey and J. Shavlik, "Transfer learning," in *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques.* IGI global, 2010, pp. 242–264.

[83] WIPO, "Patent law," *https://www.wipo.int/patents/en/*, 2022.

[84] U. S. Patent and T. Office, "United states patent and trademark office," *https://www.uspto.gov/web/offices/pac/mpep/s2106.html*, 2022.

[85] MITRE, "2019 cwe top 25 most dangerous software errors," https://tinyurl.com/y73xa6qk, 2019.

[86] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of Big Data*, vol. 6, p. 60, 2019.

[87] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *FSE*, 2017, pp. 727–739.

[88] E. Quiring, A. Maier, and K. Rieck, "Misleading authorship attribution of source code using adversarial learning," 2019.

[89] "S3scanner," https://github.com/sa7mon/S3Scanner.

[90] "Github token scanning," https://developer.github.com/partnerships/token-scanning.

[91] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani, "Detecting and mitigating secret-key leaks in source code repositories," in *MSR*, 2015, pp. 396–400.

[92] X. Sun, X. Liu, J. Hu, and J. Zhu, "Empirical studies on the nlp techniques for source code data preprocessing," in *EAST*, 2014, pp. 32–39.

[93] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[94] R. Bellman, "Dynamic programming," 1957.

[95] M. Meli, M. R. McNiece, and B. Reaves, "How bad can it git? characterizing secret leakage in public github repositories," in *NDSS*, 2019.

[96] S. Pearman, S. A. Zhang, L. Bauer, N. Christin, and L. F. Cranor, "Why people (don't) use password managers effectively," in *USENIX SOUPS*, 2019.

[97] G. Milka, "Anatomy of account takeover," in *Proceedings of Enigma*, 2018.

[98] P. R. Center, "Americans and digital knowledge," https://tinyurl.com/y8ftudoh, 2019.

[99] E. Bursztein, "The bleak picture of two-factor authentication adoption in the wild," https://tinyurl.com/yctk4aja.

[100] A. Bronshtein, "Train/test split and cross validation in python," *Understanding Machine Learning*, 2017.

[101] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *ICMLA*, 2018.

[102] E. Guzman, D. Azócar, and Y. Li, "Sentiment analysis of commit comments in github: an empirical study," in *MSR*, 2014, pp. 352–355.

[103] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *MSR*, 2014.

[104] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman, "Lean ghtorrent: Github data on demand," in *MSR*, 2014, pp. 384–387.

[105] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," 2018.

[106] B. Gelman, B. Hoyle, J. Moore, J. Saxe, and D. Slater, "A language-agnostic model for semantic source code labeling," in *MASES*, 2018.

[107] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," 2019.

[108] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," 2019.

[109] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, "Large-scale malware classification using random projections and neural networks," in *ICASSP*, 2013.

[110] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: Eliminating experimental bias in malware classification across space and time," in *USENIX Security Symposium*, 2019, pp. 729–746.

[111] H. Van Hasselt, Y. Doron, F. Strub, M. Hessel, N. Sonnerat, and J. Modayil, "Deep reinforcement learning and the deadly triad," 2018.

[112] "Gitleaks," https://github.com/zricethezav/gitleaks.

[113] SAP, "Credential digger library," *https://github.com/SAP/credential-digger*, 2022.

[114] ——, "Credential digger models," *https://huggingface.co/SAPOSS/password-model*, 2022.

[115] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[116] Y. Li, H. Wang, and M. Barni, "A survey of deep neural network watermarking techniques," *Neurocomputing*, vol. 461, pp. 171–193, 2021.

[117] F. Boenisch, "A survey on model watermarking neural networks," *arXiv preprint arXiv:2009.12153*, 2020.

[118] F. Regazzoni, P. Palmieri, F. Smailbegovic, R. Cammarota, and I. Polian, "Protecting artificial intelligence ips: a survey of watermarking and fingerprinting for machine learning," *CAAI Transactions on Intelligence Technology*, vol. 6, no. 2, pp. 180–191, 2021.

[119] N. Lukas, E. Jiang, X. Li, and F. Kerschbaum, "Sok: How robust is image classification deep neural network watermarking?(extended version)," *arXiv preprint arXiv:2108.04974*, 2021.

[120] M. Xue, Y. Zhang, J. Wang, and W. Liu, "Intellectual property protection for deep learning models: Taxonomy, methods, attacks, and evaluations," *arXiv preprint arXiv:2011.13564*, 2020.

[121] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, "Targeted backdoor attacks on deep learning systems using data poisoning," *arXiv preprint arXiv:1712.05526*, 2017.

[122] R. Namba and J. Sakuma, "Robust watermarking of neural network with exponential weighting," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 228–240.

[123] E. Le Merrer, P. Perez, and G. Trédan, "Adversarial frontier stitching for remote neural network watermarking," *Neural Computing and Applications*, vol. 32, no. 13, pp. 9233–9244, 2020.

[124] X. Yuan, P. He, Q. Zhu, and X. Li, "Adversarial examples: Attacks and defenses for deep learning," *IEEE transactions on neural networks and learning systems*, vol. 30, no. 9, pp. 2805–2824, 2019.

[125] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *2016 IEEE European symposium on security and privacy (EuroS&P)*. IEEE, 2016, pp. 372–387.

[126] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 ieee symposium on security and privacy (sp)*. Ieee, 2017, pp. 39–57.

[127] 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," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 126–137.

[128] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *arXiv preprint arXiv:1406.2661*, 2014.

[129] J. Guo and M. Potkonjak, "Evolutionary trigger set generation for dnn black-box watermarking," *arXiv preprint arXiv:1906.04411*, 2019.

[130] Y. Liu, H. Wu, and X. Zhang, "Robust and imperceptible black-box dnn watermarking based on fourier perturbation analysis and frequency sensitivity clustering," *arXiv preprint arXiv:2208.03944*, 2022.

[131] A. Bansal, P.-y. Chiang, M. J. Curry, R. Jain, C. Wigington, V. Manjunatha, J. P. Dickerson, and T. Goldstein, "Certified neural network watermarks with randomized smoothing," in *International Conference on Machine Learning*. PMLR, 2022, pp. 1450–1465.

[132] J. Cohen, E. Rosenfeld, and Z. Kolter, "Certified adversarial robustness via randomized smoothing," in *International Conference on Machine Learning*. PMLR, 2019, pp. 1310–1320.

[133] P.-y. Chiang, R. Ni, A. Abdelkader, C. Zhu, C. Studer, and T. Goldstein, "Certified defenses for adversarial patches," *arXiv preprint arXiv:2003.06693*, 2020.

[134] X. Chen, T. Chen, Z. Zhang, and Z. Wang, "You are caught stealing my winning lottery ticket! making a lottery ticket claim its ownership," *Advances in Neural Information Processing Systems*, vol. 34, pp. 1780–1791, 2021.

[135] L. Charette, L. Chu, Y. Chen, J. Pei, L. Wang, and Y. Zhang, "Cosine model watermarking against ensemble distillation," *arXiv preprint arXiv:2203.02777*, 2022.

[136] S. Sakazawa, E. Myodo, K. Tasaka, and H. Yanagihara, "Visual decoding of hidden watermark in trained deep neural network," in *2019 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*. IEEE, 2019, pp. 371–374.

[137] L. Fan, K. W. Ng, and C. S. Chan, "Rethinking deep neural network ownership verification: Embedding passports to defeat ambiguity attacks," *Advances in neural information processing systems*, vol. 32, 2019.

[138] X. Wang, Y. Lu, X. Yan, and L. Yu, "Wet paper coding-based deep neural network watermarking," *Sensors*, vol. 22, no. 9, p. 3489, 2022.

[139] Y. Uchida, Y. Nagai, S. Sakazawa, and S. Satoh, "Embedding watermarks into deep neural networks," in *Proceedings of the 2017 ACM on international conference on multimedia retrieval*, 2017, pp. 269–277.

[140] J. Wang, H. Wu, X. Zhang, and Y. Yao, "Watermarking in deep neural networks via error back-propagation," *Electronic Imaging*, vol. 2020, no. 4, pp. 22–1, 2020.

[141] T. Wang and F. Kerschbaum, "Attacks on digital watermarks for deep neural networks," in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019, pp. 2622–2626.

[142] H. Liu, Z. Weng, and Y. Zhu, "Watermarking deep neural networks with greedy residuals." in *ICML*, 2021, pp. 6978–6988.

[143] B. D. Rouhani, H. Chen, and F. Koushanfar, "Deepsigns: A generic watermarking framework for ip protection of deep learning models," *arXiv preprint arXiv:1804.00750*, 2018.

[144] H. Chen, B. D. Rouhani, and F. Koushanfar, "Blackmarks: Blackbox multibit watermarking for deep neural networks," *arXiv preprint arXiv:1904.00344*, 2019.

[145] G. Pagnotta, D. Hitaj, B. Hitaj, F. Perez-Cruz, and L. V. Mancini, "Tattooed: A robust deep neural network watermarking scheme based on spread-spectrum channel coding," *arXiv preprint arXiv:2202.06091*, 2022.

[146] Y. Lao, P. Yang, W. Zhao, and P. Li, "Identification for deep neural network: Simply adjusting few weights!" in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022, pp. 1328–1341.

[147] G. Li, S. Li, Z. Qian, and X. Zhang, "Encryption resistant deep neural network watermarking," in *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2022, pp. 3064–3068.

[148] N. Lin, X. Chen, H. Lu, and X. Li, "Chaotic weights: A novel approach to protect intellectual property of deep neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 7, pp. 1327–1339, 2020.

[149] M. Alam, S. Saha, D. Mukhopadhyay, and S. Kundu, "Deep-lock: Secure authorization for deep neural networks," *arXiv preprint arXiv:2008.05966*, 2020.

[150] T. Wang and F. Kerschbaum, "Riga: Covert and robust white-box watermarking of deep neural networks," in *Proceedings of the Web Conference 2021*, 2021, pp. 993–1004.

[151] J. Jia, Y. Wu, A. Li, S. Ma, and Y. Liu, "Subnetwork-lossless robust watermarking for hostile theft attacks in deep transfer learning models," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[152] X. Cao, J. Jia, and N. Z. Gong, "Ipguard: Protecting intellectual property of deep neural networks via fingerprinting the classification boundary," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 14–25.

[153] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[154] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," in *Artificial intelligence safety and security*. Chapman and Hall/CRC, 2018, pp. 99–112.

[155] T. Dong, H. Qiu, T. Zhang, J. Li, H. Li, and J. Lu, "Fingerprinting multi-exit deep neural network models via inference time," *arXiv preprint arXiv:2110.03175*, 2021.

[156] S. Hong, Y. Kaya, I.-V. Modoranu, and T. Dumitraş, "A panda? no, it's a sloth: Slowdown attacks on adaptive multi-exit neural network inference," *arXiv preprint arXiv:2010.02432*, 2020.

[157] T. Maho, T. Furon, and E. L. Merrer, "Fbi: Fingerprinting models with benign inputs," *arXiv preprint arXiv:2208.03169*, 2022.

[158] N. Lukas, Y. Zhang, and F. Kerschbaum, "Deep neural network fingerprinting by conferrable adversarial examples," *arXiv preprint arXiv:1912.00888*, 2019.

[159] K. T. Co, L. Muñoz-González, S. de Maupeou, and E. C. Lupu, "Procedural noise adversarial examples for black-box attacks on deep convolutional networks," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 275–289.

[160] S. Wang, X. Wang, P.-Y. Chen, P. Zhao, and X. Lin, "Characteristic examples: High-robustness, low-transferability fingerprinting of neural networks." in *IJCAI*, 2021, pp. 575–582.

[161] K. Yang, R. Wang, and L. Wang, "Metafinger: Fingerprinting the deep neural networks with meta-training," *31st International Joint Conference on Artificial Intelligence (IJCAI-22)*, 2022.

[162] 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," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 824–841.

[163] B. Fuglede and F. Topsoe, "Jensen-shannon divergence and hilbert space embedding," in *International Symposium onInformation Theory, 2004. ISIT 2004. Proceedings.* IEEE, 2004, p. 31.

[164] D. Hitaj and L. V. Mancini, "Have you stolen my model? evasion attacks against deep neural network watermarking techniques," *arXiv preprint arXiv:1809.00615*, 2018.

[165] W.-A. Lin, Y. Balaji, P. Samangouei, and R. Chellappa, "Invert and defend: Model-based approximate inversion of generative adversarial networks for secure inference," *arXiv preprint arXiv:1911.10291*, 2019.

[166] B. Wang, Y. Yao, S. Shan, H. Li, B. Viswanath, H. Zheng, and B. Y. Zhao, "Neural cleanse: Identifying and mitigating backdoor attacks in neural networks," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 707–723.

[167] F.-Q. Li, S.-L. Wang, and Y. Zhu, "Fostering the robustness of white-box deep neural network watermarks by neuron alignment," in *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2022, pp. 3049–3053.

[168] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag, "What is the state of neural network pruning?" *Proceedings of machine learning and systems*, vol. 2, pp. 129–146, 2020.

[169] W. Aiken, H. Kim, S. Woo, and J. Ryoo, "Neural network laundering: Removing black-box backdoor watermarks from deep neural networks," *Computers & Security*, vol. 106, p. 102277, 2021.

[170] D. S. Ong, C. S. Chan, K. W. Ng, L. Fan, and Q. Yang, "Protecting intellectual property of generative adversarial networks from ambiguity attacks," in *Proceedings*

of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 3630–3639.

[171] R. Zhu, X. Zhang, M. Shi, and Z. Tang, "Secure neural network watermarking protocol against forging attack," *EURASIP Journal on Image and Video Processing*, vol. 2020, no. 1, pp. 1–12, 2020.

[172] F. Li, L. Yang, S. Wang, and A. W.-C. Liew, "Leveraging multi-task learning for umambiguous and flexible deep neural network watermarking." in *SafeAI@ AAAI*, 2022.

[173] X. Chen, W. Wang, C. Bender, Y. Ding, R. Jia, B. Li, and D. Song, "Refit: a unified watermark removal framework for deep learning systems with limited data," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 321–335.

[174] N. Chattopadhyay, C. S. Y. Viroy, and A. Chattopadhyay, "Re-markable: Stealing watermarked neural networks through synthesis," in *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer, 2020, pp. 46–65.

[175] Z. Yang, H. Dang, and E.-C. Chang, "Effectiveness of distillation attack and countermeasure on neural network watermarking," *arXiv preprint arXiv:1906.06046*, 2019.

[176] Y. Yan, X. Pan, Y. Wang, M. Zhang, and M. Yang, "" and then there were none": Cracking white-box dnn watermarks via invariant neuron transforms," *arXiv preprint arXiv:2205.00199*, 2022.

[177] J. H. Lim, C. S. Chan, K. W. Ng, L. Fan, and Q. Yang, "Protect, show, attend and tell: Empowering image captioning models with ownership protection," *Pattern Recognition*, vol. 122, p. 108285, 2022.

[178] V. Behzadan and W. Hsu, "Sequential triggers for watermarking of deep reinforcement learning policies," *arXiv preprint arXiv:1906.01126*, 2019.

[179] L. Dai, J. Mao, X. Fan, and X. Zhou, "Deephider: A multi-module and invisibility watermarking scheme for language model," *arXiv preprint arXiv:2208.04676*, 2022.

[180] F. Koushanfar, "Intellectual property (ip) protection for deep learning and federated learning models," in *Proceedings of the 2022 ACM Workshop on Information Hiding and Multimedia Security*, 2022, pp. 5–5.

[181] B. G. Tekgul, Y. Xia, S. Marchal, and N. Asokan, "Waffle: Watermarking in federated learning," in *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2021, pp. 310–320.

[182] B. G. Atli Tekgul and N. Asokan, "On the effectiveness of dataset watermarking," in *Proceedings of the 2022 ACM on International Workshop on Security and Privacy Analytics*, 2022, pp. 93–99.

[183] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[184] S. Kornblith, M. Norouzi, H. Lee, and G. Hinton, "Similarity of neural network representations revisited," *arXiv preprint arXiv:1905.00414*, 2019.

[185] S. Ma, X. Sun, Y. Wang, and J. Lin, "Bag-of-Words as target for neural machine translation," 2019.

[186] J. Fan, Z. Wang, Y. Xie, and Z. Yang, "A theoretical analysis of deep q-learning," in *Learning for Dynamics and Control*. PMLR, 2020, pp. 486–489.

[187] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, 2020.

[188] S. Szyller, B. G. Atli, S. Marchal, and N. Asokan, "Dawn: Dynamic adversarial watermarking of neural networks," *arXiv preprint arXiv:1906.00830*, 2019.

[189] J. Guo and M. Potkonjak, "Watermarking deep neural networks for embedded systems," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[190] X. Chen, A. Salem, M. Backes, S. Ma, and Y. Zhang, "Badnl: Backdoor attacks against nlp models," in *ICML 2021 Workshop on Adversarial Machine Learning*, 2021.

[191] P. A. Gagniuc, *Markov chains: from theory to implementation and experimentation*. John Wiley & Sons, 2017.

[192] "Filtering an image," https://tinyurl.com/yxvtrncu, accessed: 2020-12-04.

[193] H. Abdi and L. J. Williams, "Principal component analysis," *WIREs Computational Statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[194] L. Yujian and L. Bo, "A normalized levenshtein distance metric," *IEEE transactions on pattern analysis and machine intelligence*, vol. 29, no. 6, pp. 1091–1095, 2007.

[195] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[196] O. Bojar, C. Buck, C. Federmann, B. Haddow, P. Koehn, J. Leveling, C. Monz, P. Pecina, M. Post, H. Saint-Amand, R. Soricut, L. Specia, and A. s. Tamchyna, "Findings of the 2014 workshop on statistical machine translation," in *Proceedings of the Ninth Workshop on Statistical Machine Translation*. Association for Computational Linguistics, 2014, pp. 12–58. [Online]. Available: http://www.aclweb.org/anthology/W/W14/W14-3302

[197] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *Advances in neural information processing systems*, 2017, pp. 3146–3154.

[198] "Simple exploration baseline for customer revenue," https://tinyurl.com/y3fzx4gx, accessed: 2020-12-08.

[199] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[200] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[201] S. Rajaraman, S. K. Antani, M. Poostchi, K. Silamut, M. A. Hossain, R. J. Maude, S. Jaeger, and G. R. Thoma, "Pre-trained convolutional neural networks as feature extractors toward improved malaria parasite detection in thin blood smear images," *PeerJ*, p. e4568, 2018.

[202] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *arXiv preprint arXiv:1509.06461*, 2015.

[203] C. P. Carrino, M. R. Costa-jussà, and J. A. Fonollosa, "Automatic spanish translation of the squad dataset for multilingual question answering," *arXiv preprint arXiv:1912.05200*, 2019.

[204] R. Budur, Emrah an Ozçelik and T. Gungor, "Data and representation for turkish natural language inference," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2020.

[205] A. Khosla, N. Jayadevaprakash, B. Yao, and L. Fei-Fei, "Novel dataset for fine-grained image categorization," in *First Workshop on Fine-Grained Visual Categorization, IEEE Conference on Computer Vision and Pattern Recognition*, 2011.

[206] K. Chen, S. Guo, T. Zhang, X. Xie, and Y. Liu, "Stealing deep reinforcement learning models for fun and profit," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 307–319.

[207] X. Zhao, H. Wu, and X. Zhang, "Watermarking graph neural networks by random graphs," in *2021 9th International Symposium on Digital Forensics and Security (ISDFS)*. IEEE, 2021, pp. 1–6.

[208] J. Xu and S. Picek, "Watermarking graph neural networks based on backdoor attacks," *arXiv preprint arXiv:2110.11024*, 2021.

[209] Y. Wang and H. Wu, "Protecting the intellectual property of speaker recognition model by black-box watermarking in the frequency domain," *Symmetry*, vol. 14, no. 3, p. 619, 2022.

[210] X. He, Q. Xu, L. Lyu, F. Wu, and C. Wang, "Protecting intellectual property of language generation apis with lexical watermark," AAAI, 2022.

[211] T. Zhang, H. Wu, X. Lu, and G. Sun, "Awencoder: Adversarial watermarking pre-trained encoders in contrastive learning," *arXiv preprint arXiv:2208.03948*, 2022.

[212] Amazon, "Aws marketplace," *https://tinyurl.com/45mftf7n*, 2022.

[213] HuggingFace, "Huggingface hub," *https://huggingface.co/models*, 2022.

[214] Youtube, "How content id works ?" *https://support.google.com/youtube/answer/2797370?hl=en*, 2022.

[215] jdhao, "How does the youtube content id system work?" *https://tinyurl.com/yke4yynnn*, 2022.

[216] E. Weinstein and P. Moreno, "Music identification with weighted finite-state transducers," *2007 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2007.

[217] "Viacom international inc. v. youtube, inc." 2022.

[218] TheFatRat, "How my video with 47 million views was stolen on youtube," *https://tinyurl.com/2p93np2u*, 2022.

[219] J. Bottum, "The empire strikes bach: Algorithms and copyright laws are stealing music from all of us," *https://freebeacon.com/culture/google-youtube-algorithm-copyright/*, 2022.

[220] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[221] R. Cramer, I. B. Damgaard *et al.*, "Secure multiparty computation," 2015.

[222] T. Ryffel, E. Dufour-Sans, R. Gay, F. Bach, and D. Pointcheval, "Partially encrypted machine learning using functional encryption," 2019. [Online]. Available: https://arxiv.org/pdf/1905.10214.pdf

[223] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[224] A. Coates, A. Ng, and H. Lee, "An analysis of single-layer networks in unsupervised feature learning," pp. 215–223, 2011.

[225] J. Yang and G. Yang, "Modified convolutional neural network based on dropout and the stochastic gradient descent optimizer," *Algorithms*, 2018.

[226] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[227] "Ml security," 2022.

[228] "Bosch ai shield," 2022.

[229] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[230] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[231] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.

[232] W. Yang, Y. Lin, P. Li, J. Zhou, and X. Sun, "Rethinking stealthiness of backdoor attack against nlp models," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2021, pp. 5543–5557.

[233] H. Jia, C. A. Choquette-Choo, and N. Papernot, "Entangled watermarks as a defense against model extraction," *arXiv preprint arXiv:2002.12200*, 2020.

[234] T. Gu, K. Liu, B. Dolan-Gavitt, and S. Garg, "Badnets: Evaluating backdooring attacks on deep neural networks," *IEEE Access*, vol. 7, pp. 47 230–47 244, 2019.

[235] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp,

G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[236] M. Honnibal, I. Montani, S. Van Landeghem, and A. Boyd, "spacy: Industrial-strength natural language processing in python," *doi: 10.5281/zenodo.1212303*, 2020.

[237] D. Dua and C. Graff, "UCI Machine Learning Repository," http://archive.ics.uci.edu/ml/index.php/, 2017.

[238] A. Krizhevsky, "Learning multiple layers of features from tiny images," *Citeseer*, 2009.

[239] D. Hitaj, B. Hitaj, and L. V. Mancini, "Evasion attacks against watermarking techniques found in mlaas systems," in *2019 Sixth International Conference on Software Defined Systems (SDS)*, 2019, pp. 55–63.

[240] M. Barni, F. Pérez-González, and B. Tondi, "Dnn watermarking: Four challenges and a funeral," in *Proceedings of the 2021 ACM Workshop on Information Hiding and Multimedia Security*, 2021.

[241] N. Mehrabi, M. Naveed, F. Morstatter, and A. Galstyan, "Exacerbating algorithmic bias through fairness attacks," 2020.

[242] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[243] S. Caton and C. Haas, "Fairness in machine learning: A survey," *arXiv preprint arXiv:2010.04053*, 2020.