POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

AdvBench: a framework to evaluate adversarial attacks against fraud detection systems

Tesi di Laurea Magistrale in
Computer Science Engineering - Ingegneria Informatica

Author: **Davide Biarese**

Student ID: 970864
Advisor: Prof. Michele Carminati
Co-advisors: Tommaso Paladini
Academic Year: 2022-23

Abstract

In recent years, digital payments and financial services via the Internet have experienced significant growth which has also led to an increase in the number of online frauds. This threat pushed financial institutions to implement automated defence measures called Fraud Detection Systems (FDSs) that use Machine Learning (ML) models to classify transactions. Researchers have shown that ML models are sensitive to Adversarial Machine Learning (AML) attacks, also in the fraud detection domain. AML attacks against FDSs create fraudulent transactions that evade the target ML model detection (evasion attack). Another type of AML attack exists, poisoning attacks. These attacks decrease the ML models' prediction capacity by infecting the training set, hence the ML models will be trained on malicious samples. In the fraud detection domain, if the FDS is periodically retrained, a successful evasion attack infects the training set, performing also a poisoning attack.

In this thesis, we propose a framework to evaluate the effectiveness of adversarial attacks against banking Fraud Detection Systems, using adversarial algorithms adapted to the banking domain to generate adversarial transactions. We adapted five new algorithms to this domain. In particular, we adapted four gradient-based algorithms and one decision-based algorithm. To test our framework, we run poisoning attack simulations against five different models in three scenarios based on the knowledge degree of the attacker. Our results show that gradient-based algorithms, even if they reach the goal of stealing money, are, compared with the other class of attacks, decision-based attacks, way less effective, except in the most favourable scenario, where the attacker knows everything about the FDS. This can be explained because the Oracle used in gradient-based attacks is less precise than the one used in decision-based attacks.

Keywords: Adversarial Machine Learning, Fraud Detection System, Online Banking, Poisoning Attacks, Evasion Attacks

Abstract in lingua italiana

Negli ultimi anni, i pagamenti digitali e i servizi finanziari online hanno subito una crescita significativa che ha portato ad un aumento del numbero di frodi online. La minaccia ha spinto le istituzioni finanziarie a implementare misure di difesa automatizzate chiamate Fraud Detection Systems (FDSs), che utilizzano modelli di Machine Learnings (MLs) per classificare le transazioni. Numerosi ricercatori hanno dimostrato che i modelli di ML sono particolarmente vulnerabili agli attacchi di AMLs, anche nel contesto bancario. Questo tipo di attacchi genera delle transazioni fraudolente che riescono a sfuggire alla rilevazione del modello di ML utilizzato dal FDS (evasion attack). Oltre agli evasion attacks, esistono i poisoning attack. Questo tipo di attacchi cause un deterioramento nelle capacità di predizione dei modelli di ML infettando il training set, infatti i modelli di ML utilizzeranno anche gli adversarial samples durante la fase di training. Nel contesto bancario, se il FDS viene periodicamente allenato con le nuove transazioni in entrata, un evasion attack che ha successo infetterà il training set aggiornato, eseguendo anche un poisoning attack.

In questa tesi proponiamo un framework per valutare l'efficacia degli adversarial attacks contro i Fraud Detection Systems, utilizzando degli adversarial algorithms adattati al dominio bancario per generare le transazioni fraudolente. Abbiamo adattato cinque nuovi algoritmi a questo dominio. In particolare, abbiamo adattato quattro algoritmi basati sul gradient-based e un algoritmo decision-based. Per testare il nostro framework, eseguiamo delle simulazioni di attacchi di avvelenamento contro cinque diversi modelli di ML in tre scenari basati sul grado di conoscenza dell'attaccante. I risultati mostrano che gli algoritmi basati sul gradiente, anche se raggiungono l'obiettivo di rubare denaro, sono, rispetto all'altra classe di attacchi, molto meno efficaci, tranne nello scenario più favorevole dove l'attaccante sa tutto sul FDS. Questo può essere spiegato perché l'Oracle usato in attacchi basati su gradienti è meno preciso di quello usato negli attacchi decision-based.

Parole chiave: Adversarial Machine Learning, Sistemi di Rilevamento Frodi, Online Banking, Poisoning Attacks, Evasion Attacks

Contents

1 | Introduction

In the past years, Internet banking and digital payments have experienced significant growth, especially after the Pandemic crisis. The expansion of the Internet payment channels introduced a new attack surface for fraudsters. Nowadays, Internet banking frauds are constantly evolving, resulting in relevant financial losses[32]. In the EU, 84% credit card fraudulent transactions have taken place over the Internet[6].

The growing threat of these activities brought banks to adopt improved and automatically defensive systems to either prevent or detect fraudulent transactions, Fraud Detection Systems (FDSs) [31, 35] purpose is to detect fraudulent activities when they reach the system. FDSs employ Machine Learning (ML) algorithms and statistical approaches to identify users' spending behaviour. If the incoming transaction does not belong to the user's spending behaviour is classified as fraud. The suspicious transactions reported by the FDS can be examined by an expert who decides whether to block them or not. The fraudsters continuously improve their methods to evade the protection offered by banking defensive systems. Adversarial Machine Learning (AML) attacks showed to be particularly effective against ML models. AML is the study of effective ML techniques against an adversarial opponent. An AML attack may have two goals: make the target ML model misclassify crafted examples at test time (*evasion attacks*) or reduce the ML model's performances by injecting crafted examples to the model's training set that shift the decision boundaries(*poisoning attacks*).

In this thesis, we propose a framework to test the effectiveness of Evasion and Poisoning attacks using various adversarial algorithms against a set of FDSs, we base our framework on the works of Maniscalchi and Monti [37, 39]. Moreover, we select five new algorithms and, starting from the work of Romeo [48], we adapt them to our domain. The new adversarial attack algorithms are: Fast Gradient Sign Method (FGSM)[26], Iterative method [34], Saliency Map [46], LowProFool [5] and ESPA [33]. The first three are commonly used in the image recognition domain, while LowProFool and ESPA were created with tabular data. The algorithms can be divided into two classes: Decision-based and Gradient-based. Decision-based Attacks base their operations on the output of the Oracle, while

Gradient-based Attacks use the computed Gradient. Monti's Attack, ZOO, Boundary, HopSkipJump, and ESPA belong to the first class; FGSM, Iterative, Saliency Map, and LowProFool are Gradient-based attacks.

In the experimental evaluation, we compare the results of the poisoning attacks obtained with new algorithms with the previous attacks introduced in previous works. We tested the algorithms against FDSs that use the following ML models: XGBoost, Random Forests, Neural Networks Logistic Regression, and Support Vector Machines. Each FDS retrains its ML model every two weeks, adding new legitimate transactions to their dataset. We model three scenarios: blackbox, graybox, whitebox. The attacker has complete knowledge of the target FDS in the whitebox, only partial knowledge in the graybox, or no knowledge in the blackbox. Moreover, the attacker can follow three different strategies: *greedy*, *medium*, or *conservative* strategy.

The main contributions of this thesis are:

- The adaptation of AML attack algorithms FGSM, Iterative, Saliency Map, LowProFool, and ESPA, usually used in the image recognition domain or tabular data, to the fraud detection context.
- Investigate how robust are the FDS against gradient-based attacks. We noticed that the gradient-based attacks lead to a far lower amount of money stolen, but the adversarial samples submitted to the target FDS are more evasive. This also shows the importance of Oracle in these adversarial algorithms

This thesis is organized as follows:

- in Chapter 2, we present the problem statement, the theoretical background, the goals of our work, and challenges.
- Chapter 3 formalizes the threat model, in this chapter, we specify the attacker's capabilities, knowledge, strategies, and goals.
- Chapter 4 analyses the datasets that we used in our approach.
- Chapter 5 describes our framework in detail.
- In Chapter 6 we describe the changes introduced to adapt each algorithm.
- In Chapter 7 we describe what are Fraud Detection Systems (FDSs), the concept drift, and the feature engineering process.
- Chapter 8 describes the algorithm implementation and software package used.

- Chapter 9 is the experimental validation chapter. In this chapter, we present the results of our experiments.
- Chapter 10 highlights the limitations of our approach and proposes possible future work directions.
- In Chapter 11 we draw conclusions from our results.

2 | Background and Motivation

2.1. Motivation

In the last few years, especially during the pandemic crisis, online banking, and digital payments have encountered significant expansion [24, 43]. This growth happened because, with these tools, a user can manage their banking account and check the operations done or their balance from the desk and make payments or buy things with the e-commerce platforms and wait for them at home. With these types of services that can move a significant amount of money, the interest of fraudsters arises. In the EU, the vast majority of credit card frauds, approximately 84%, are led via the Internet [6]. The impact of frauds, despite their number being small compared with the total of transactions and even if their number and volume are decreasing [6], is relevant because we also have to consider that 25% of the frauds are not recovered by bank [32].

2.1.1. Internet Banking Fraud

An Internet Banking fraud is a fraud committed using online technologies. The fraudster objective is to gain access to a user's online bank account and transfers funds on behalf of him . The current and most challenging fraud schemes are information stealing and transaction hijacking [15]. In information stealing, the fraudster's aim is to steal sensitive data such as login credentials or one-time password (OTP) and then submit a transaction towards his account. In this scheme, the fraudster commits a transaction from their connection. In transaction hijacking, the fraudster redirects the victim's legitimate transactions towards his bank account. This scheme is more difficult to detect because the connection comes from the victim's computer and IP address. Nowadays, on the Internet, there are various threats to cyber security. One of the most common is *phising*[28]. In phishing, an attacker creates fake web pages, emails, or messages similar to legitimate ones. The victims, thinking that the emails or the webpages are real, interact with them and share personal information, such as credit card numbers or login credentials, or download malware. Fraudster exploits this information to steal the victim's identity

and submit fraudulent transactions on behalf of them. Another way to steal victims' informations is *Banking Trojans*. This type of malware is usually spread via emails or downloads on the Web, cracked software, and corrupted devices. *Zeus*, also called *Zbot*, is the most common banking malware and was first discovered in 2007. In 2010, with the introduction of two-factor authentication schemes by financial institutions, the Zeus kit was expanded with Zeus-in-the-Mobile (ZitMo) that targets mobile devices to intercept mobile transaction authentication numbers[27]. Following Zeus, several dangerous banking trojans were developed.
Another common malware class is *Ransomware* [44]. This malware encrypts sensitive data, or even the entire victim's, with an encryption key known only by the attacker. To give the encryption key to the victim to retrieve his data, they ask for a ransom.
The last threat is not a computer technique, but is based on human trust, and is called *Social Engineering*[49]. With *Social Engineering*, the attacker gains the victim's trust and convinces them to share sensitive data. This data will be exploited to conduct the fraud.

2.2. Fraud Detection System

To deal with the fraud problem, banks utilize tools called Fraud Detection Systems FDSs. FDSs are automated tools that highlight suspicious transactions that will be checked by human operators. Before classifying the incoming transactions, FDSs may aggregate it with the user's past transactions to capture its spending behaviour and be more capable of recognizing outliers[53]. FDSs also needs to work with a common scale of values, hence the transactions are standardized. We can divide FDSs into two categories: *user-centric* and *system-centric*: the first approach uses one model for each user, so the transaction will be anomalous or not according to the specific user, the latter uses a single model for all the system and the transaction will be classified as fraud if it will be anomalous with respect of all the dataset. The design of FDSs presents several challenges [1]. First of all, the *imbalanced dataset*: datasets used to train the models are heavily imbalanced due to the fact that the frauds are a small portion of the total transactions. Another relevant challenge is the *concept drift*, which is the mutation of the underlying model. In our scenario, the user can change their spending behaviour during their life. Several FDSs use machine learning models to classify transactions.

2.2.1. Machine Learning

Machine Learning is a computer science field that tries to construct programs that improve through time by learning from the data during the training and learning patterns

[38]. We can divide machine learning into three categories: *Supervised learning*, *Unsupervised learning* and *Reinforcement learning*. FDSs mostly uses *Supervised learning* and *Unsupervised learning* [31, 35, 42]. *Supervised learning* uses labeled datasets to train a model that will classify an entry of the test set or produce an outcome. We call the input variables *features*, and the output *labels*. We can distinguish three different types of *Supervised learning*:

- we will speak about *classification* if the label is categorical.
- if the output label is continuous we will speak about *regression*
- if the output is a probability we are in the *probability estimation* case

In *Unsupervised learning*, the training set is without labels. The aim of *Unsuperived learning* is to find patterns in the data. Examples of unsupervised learning techniques are:

- *Clustering* that try to group data based on their similarities or differences
- *Principal Component Analysis* that reduce the dimensionality of the features space

2.2.2. Adversarial Machine Learning

According to Huang et al. [29], Adversarial Machine Learning (AML) is *"..the design of machine learning algorithms that can resist these sophisticated attacks, and the study of the capabilities and limitations of attackers"*. We can model the struggle between the attacker and the defender as a game. The attacker can alter the *training set* or the *evaluation set* or both of them. In the same work, they propose a taxonomy of Adversarial Attacks (AAs) based on three properties:*Influence*, *security violation* and *specificity*.
Influence describes the capability of the attackers. According to this property, we can have *causative attacks* and *exploratory attacks*. *Causative* attacks alter the dataset used in the training phase, *exploratory* attacks aim to gain information about the training process or the data used.
Security Violation describes which violation the attacker causes: *integrity* attack result in a classification error of the introduced data. Hence, malicious data is classified as benign(false negative). *Availability* attacks cause great number of misclassifications, both false negative and false positive, that the model under attack becomes unusable. With *privacy* attacks, the adversary obtains information from the learner, compromising the secrecy or privacy of the system's users.
Specificity describe the focus of the attack, that can *targeted* or *indiscriminate*. *Targeted* attack s aim to cause the misclassification of a small set of target points, equivalently, a

specific group of system's users, *indiscriminate* attacks instead, does not have a specific focus, but involve a very general class of points or, equivalently, a group of system's users selected randomly.
An additional way to distinguish between AAs is *Evasion Attack* vs. *Poisoning Attack*.

Evasion Attack

An *evasion attack* is an AA that wants the target to misclassify the Adversarial Sample (AS) during the evaluation time. According to the taxonomy previously exposed, it can be categorized as an *exploratory integrity* attack. It can be either *targeted* or *indiscriminate*. In [9] authors present a general framework for evasion attacks. Considering a classification problem where:

- $\mathcal{X}$ is the feature space;
- $\mathcal{Y} = \{-1, +1\}$ is the label space (-1 represents the malicious class and $+1$ the legitimate one);
- f is the function that maps $\mathcal{X} \to \mathcal{Y}$;
- $\mathcal{D}_{tr} = \{x_i, y_i\}_{i=1}^{n}$ is the training set, with n independent and identically distributed samples drawn from the underlying probability distribution $p(\mathcal{X}, \mathcal{Y})$;

$y^c = f(\mathbf{x})$ is usually obtained by thresholding a continuous discriminant function $g : \mathcal{X} \to \mathbb{R}$, $f(\mathbf{x}) = -1$ if $g(\mathbf{x}) < 0$, $f(\mathbf{x}) = +1$ otherwise.
The optimal attack strategy for evasion strictly depends on the assumptions of the attacker's knowledge. The attacker can have full or partial knowledge about the training set, the feature space used by the classifier, the type of a learning algorithm and its decision function, the trained classifier model *i.e.*, the weight of a linear classifier or the feedback from the classifier, the labels returned by the classifier for samples chosen by the adversary. An attacker with partial knowledge of the classifier f or its training data $\mathcal{D}$ can use a surrogate classifier $\hat{f}$ trained with a dataset $\hat{\mathcal{D}}$ drawn from the same distribution $p(\mathcal{X}, \mathcal{Y})$, $y^c = f(\mathbf{x})$ from which $\mathcal{D}$ was drawn, which uses $\hat{g}(\mathbf{x})$ as discriminant function to mimic the behaviour of the target classifier. Under these assumptions, for any attacker's desired sample $\mathbf{x}^0$, an optimal attack strategyis to find a sample $\mathbf{x}^*$ minimizing $g(\cdot)$ or its estimate $\hat{g}(\cdot)$, subject to a bound on its distance from $\mathbf{x}^0$ as in Equation 2.1.

$$\begin{aligned} \mathbf{x}^* = \arg\min_{x} \quad & \hat{g}(\mathbf{x}) \\ s.t. \quad & d(\mathbf{x}, \mathbf{x}^0) \leq d_{max} \end{aligned} \tag{2.1}$$

Adversarial algorithms can be divided into two distinct classes: decision-based and gradient-based attacks. Decision-based attacks base their procedure on the label returned by the Oracle. Gradient-based attacks compute the gradient from the Oracle and use it in their procedure.
In the literature exist several examples of algorithms designed for crafting adversarial samples. One of the domains where the academic community was and is more prolific is computer vision. In their work, Szegedy et al. [51] show that a NN used as an object recognize images, can be fooled by adding an imperceptible perturbation to the image itself. The goal is to find a perturbation $\mathbf{r}$ as small as possible such that $f(\mathbf{x} + \mathbf{r}) = l$ starting from $f(\mathbf{x}) \neq l$. Finding the optimal solution to the previous problem is difficult, so they approximate it using a Limited Memory BFGS (L-BFGS) and solve the Equation 2.2.

$$minimize\ c|r| + loss_f(\mathbf{x}+\mathbf{r}, l)\ subject\ to\ x + r \in [0, 1]^m \tag{2.2}$$

They also demonstrated that a perturbation effective against one Neural Network (NN) can fool another NN even if it was trained with a different set of examples.

In the following part of this section about *Evasion Attacks*, we present the literature review about Adversarial Attacks (AAs) used to craft adversarial samples to perform an evasion attack. To conduct the literature review about adversarial algorithms, we followed two paths: the first searches works about adversarial algorithms in the financial-banking domain. The second path searched for the most relevant works in the image domain. To do this, we searched for the most significant survey searching, *Threat of Adversarial Attacks on Deep Learning in Computer Vision: A Survey*[2]. Then we select another survey *A survey on adversarial attacks and defences*[18].

Fast Gradient Sign Method

An efficient solution to the problem of finding the best perturbation to fool a classifier was proposed by Goodfellow et al.[26]. Their method, called Fast Gradient Sign Method (FGSM), computes the perturbation in a single step, using the gradient of the loss function of the input data to maximize the loss.

$$\rho = \epsilon \cdot sign(\nabla \mathcal{J}(\theta, x, y)) \tag{2.3}$$

Equation 2.3 computes the perturbation, $\mathcal{J}$ is the loss function of the classifier, and ∇ is the gradient of the loss function with respect to the input x and label y. The adversarial

sample is defined by Equation 2.4

$$x^{adv} = x + \rho \tag{2.4}$$

. Kurakin et al.[34] proposed the targeted version of the algorithm, defined with the Equation 2.5. This variant maximises the probability of a specific target y_{target}.

$$x^{adv} = x - \epsilon \cdot sign(\nabla \mathcal{J}(\theta, x, y_{target})) \tag{2.5}$$

With the Equation 2.5, the perturbation is applied in the direction that decreases the loss function with respect to y_{target}.
In th same work, Kurakin et al. proposed versions, $FastGrad.L_2$, Equation 2.6, which instead of using the sign of the gradient as the original FGSM uses the gradient itself normalized with its L_2 norm.

$$x^{adv} = x + \epsilon \cdot \frac{\nabla \mathcal{J}(\theta, x, y)}{||\nabla \mathcal{J}(\theta, x,)||_2} \tag{2.6}$$

Basic Iterative Method/PGD

An improvement to fast gradient was proposed by Kurakin et al.[34]. While the Fast Gradient Sign Method (FGSM) perturbs the images by taking a single step in the direction that increases the loss function, Basic Iterative Method (BIM) iteratively perturbs the image with a small step, and at each iteration, it adjusts the direction.

$$x^{adv}_{N+1} = Clip_{x,\epsilon}\{x^{adv}_N \ + \alpha \ sign(\nabla_x \mathcal{J}(x^{adv}_N, y_{true}))\} \tag{2.7}$$

The Equation 2.7 computes the adversarial sample at each iteration with BIM. α that determines the step size and the number of iterations is computed by the formula $\lfloor \min(\epsilon + 4, 1.25\epsilon) \rfloor$.
A variant of BIM is PGD proposed by Madry et al. [36]. Essentially, the difference between BIM and Project Gradient Descent (PGD) is that PGD initializes the example to a random point in the ball of interest (decided by the L_∞ norm) and does random restarts while BIM initializes to the original point.

Jacobian Saliency Map

Papernot et al. [46] created an adversarial attack that wants to achieve its goal by perturbing only a few pixels, i.e., features. To select the best pixels to perturb, they introduced the *adversarial saliency map*. This map indicates which input features should

be perturbed to increase the probability that the classifier outputs the desired target label. More precisely, the *adversarial saliency map* is defined as follows:

$$\mathbf{S}(\mathbf{X}, t)[i] = \begin{cases} 0 \; if \frac{\partial \mathbf{F}_t}{\partial \mathbf{X}_i} < 0 \; or \sum_{j \neq t} \frac{\partial \mathbf{F}_t}{\partial \mathbf{X}_i} > 0 \\ \left(\frac{\partial \mathbf{F}_t}{\partial \mathbf{X}_i}\right) \left| \sum_{j \neq t} \frac{\partial \mathbf{F}_t}{\partial \mathbf{X}_i} \right| \; otherwise \end{cases} \tag{2.8}$$

In Equation 2.8, $\mathbf{F}$ is the function learned by the classifier, and t is the target label. $\frac{\partial \mathbf{F}_t}{\partial \mathbf{X}_i}$ should be positive in order to increase $\mathbf{F}_t$ when feature $\mathbf{X}_i$ increase. For the same reason $\sum_{j \neq t} \frac{\partial \mathbf{F}_t}{\partial \mathbf{X}_i}$ need to be negative to decrease or stay constant when $\mathbf{X}_i$ increases. If instead of increasing a feature, the feature is decreased, the saliency map is computed the same as before but with some little differences(Equation 2.9):

$$\mathbf{S}(\mathbf{X}, t)[i] = \begin{cases} 0 \; if \frac{\partial \mathbf{F}_t}{\partial \mathbf{X}_i} > 0 \; or \sum_{j \neq t} \frac{\partial \mathbf{F}_t}{\partial \mathbf{X}_i} < 0 \\ \left| \frac{\partial \mathbf{F}_t}{\partial \mathbf{X}_i} \right| \left(\sum_{j \neq t} \frac{\partial \mathbf{F}_t}{\partial \mathbf{X}_i} \right) \; otherwise \end{cases} \tag{2.9}$$

The authors propose the Algorithm 2.1 for crafting adversarial samples to fool a Deep Neural Network. The algorithm iteratively perturbs two input features selected by SALIENCY_MAP. It halts when the sample is classified as the target class, when the maximum number of iterations is reached, or when the feature search domain is empty.

Algorithm 2.2 corresponds to the subroutine *saliency_map* in Algorithm2.1.

Algorithm 2.2 **Increasing pixels intensity saliency map**
$\nabla \mathbf{F}(\mathbf{X})$ is the forward derivative, Γ the features still in the search space, and t the target class

Input: $\nabla \mathbf{F}(\mathbf{X})$, Γ, t

for each pair $(p, q) \in \Gamma$ **do**
  $\alpha = \sum_{i=p,q} \frac{\partial \mathbf{F}_t(\mathbf{X})}{\partial \mathbf{X}_i}$
  $\beta = \sum_{i=p,q} \sum_{j \neq t} \frac{\partial \mathbf{F}_j(\mathbf{X})}{\partial \mathbf{X}_i}$
  if $\alpha > 0$ and $\beta < 0$ and $-\alpha \times \beta >$ max **then**
    $p_1 \, p_2 \leftarrow p, q$
    $max \leftarrow -\alpha \times \beta$
  end if
end for

Algorithm 2.1 **Crafting adversarial samples for LeNet-5**
$\mathbf{X}$ is the benign image, $\mathbf{Y}^*$ is the target Network output, $\mathbf{F}$ is the function learned by the network during training, Υ is the maximum distortion and θ is the change made to the pixels

Input: $\mathbf{X}$, Y^*, $\mathbf{F}$, Υ, θ

$\mathbf{X}^* \leftarrow \mathbf{X}$
$\Gamma = \{1...|\mathbf{X}|\}$ ▷ search domain is all pixels
`max_iter` $= \lfloor \frac{784 \cdot \Upsilon}{2 \cdot 100} \rfloor$
$s = \arg\max_j \mathbf{F}(\mathbf{X}^*)_j$ ▷ source class
$t = \arg\max_j \mathbf{Y}^*_j$ ▷ target class
while $s \neq t$ & `iter` $<$ `max_iter` & $\Gamma \neq 0$ **do**
    Compute Forward derivative $\nabla(\mathbf{F}(\mathbf{X}^*)$
    p_1 p_2 = `saliency_map`$(\nabla\mathbf{F}(\mathbf{X}^*), \Gamma, \mathbf{Y}^*)$
    Modify p_1 and p_2 in $\mathbf{X}^*$ by θ
    Remove p_1 from Γ if $p_1 == 0$ or $p_1 == 1$
    Remove p_2 from Γ if $p_2 == 0$ or $p_2 == 1$
    $s = \arg\max_j \mathbf{F}(\mathbf{X}^*)_j$
    `iter++`
end while
return $\mathbf{X}^*$

Algorithm 2.2, for each pair of input features $i = p, q$, computes the sum of forward derivatives $\frac{\partial \mathbf{F}_t(\mathbf{X})}{\partial \mathbf{X}_i}$, and the sum forward derivatives $\frac{\partial \mathbf{F}_j(\mathbf{X})}{\partial \mathbf{X}_i}$. It selects the pair that maximizes the increase of probability of the target and decreases the probabilities of the other output labels. In the case where the algorithm decreases pixel intensity, the algorithm works as the increasing case except for the *saliency map*. When pixel intensity is decreased, the constraints are different, the left operand of the multiplication operation is now constrained to be negative, and the right operand to be positive

ZOO

The previous algorithms presented are all white-box attacks, **i.e.** they need access to the classifier that they target. We can use these attacks in a black-box scenario, **i.e.** attacker doesn't have access to the target classifier, leveraging the transferability property using a substitute model. This property allows an adversarial sample crafted against a classifier to be efficient against other classifiers. Chen et al. [20] propose a black-box attack that does not rely on a substitute model. They modify Carlini&Wagner attack [14] with two different approaches:

1. Loss function $f(x,t)$, based on the targeted classifier F, defined in Equation 2.10.

$$f(x,t) = max\{max_{i \neq t} log[F(x)]_i - log[F(x)]_t, -k\} \tag{2.10}$$

where $k \geq 0$ and log0 is defined as $-\infty$. For untargeted attacks the loss became:

$$f(x) = max\{log[F(x)]_{t_0} - max_{i \neq t_0} log[F(x)]_i, -k\} \tag{2.11}$$

where t_0 represent the original class label for x and $max_{i \neq t_0} log[F(x)]_i$ in Equation 2.11

2. The second approach is computing an *approximate* gradient using a finite difference method. Denoting the objective function of the model as f and the input vector as x, the gradient $\frac{\partial f(x)}{\partial x_i} = \hat{g}$, is computed with the Equation 2.12 where h is a small constant and e_i is a standard basis vector where only the i-th is 1.

$$\hat{g}_i := \frac{\partial f(x)}{\partial x_i} \approx \frac{f(x + he_i) - f(x - he_i)}{2h} \tag{2.12}$$

With only one more objective function the coordinate-wise Hessian can be estimated with Equation 2.13.

$$\hat{h}_i := \frac{\partial^2 f(x)}{\partial^2 x_i i} \approx \frac{f(x + he_i) - 2f(x) + f(x - he_i)}{h^2} \tag{2.13}$$

With the gradients, the stochastic coordinate descent algorithm can be applied. Note that usually, the gradient estimate is enough for a successful attack.

Boundary

Brendel et al. [13] a novel type of attack, *decision-based attack* which relies only on the final output of the targeted classifier, while attacks like Zoo need the scores produced by the target. They call this attack Boundary Attack. The algorithm initialization finds a random point with the target label. After that, the procedure continues and performs a walk along the boundary between the adversarial and the non-adversarial region to minimize the distance from the original image while staying adversarial. Denoting x_0 as the original image, $\tilde{x}^k$ as the adversarial image at the k-th step, in the k-th step we draw a perturbation η^k from a distribution subject to the following constraints:

1. the perturbed sample is still inside the input domain, Equation 2.14.

$$\tilde{x}_i^{k-1} + \eta_i^k \; in \; [0, 255] \tag{2.14}$$

2. The perturbation has a relative size of δ, Equation 2.15.

$$||\eta^k||_2 \; = \; \delta \cdot d(x, \tilde{x}^{k-1}) \tag{2.15}$$

3. The perturbation reduces the distance of the perturbed image towards the original input by a relative amount ϵ, Equation 2.16.

$$d(x, \tilde{x}^{k-1}) \; - \; d(x, \tilde{x}^{k-1} + \eta^k) \; = \; \epsilon \cdot d(x, \tilde{x}^{k-1}) \tag{2.16}$$

Practically, first, they sample from an iid Gaussian distribution, $\eta^k \sim \mathcal{N}(0, 1)$, then rescale and clip the sample to hold the constraints 1) and 2). As second step, the project η^k onto a sphere around the original image that $d(x, \tilde{x}^{k-1} + \eta^k) \; = \; d(x, \tilde{x}^{k-1})$ and the first constraint holds. This step is called *orthogonal perturbation*. The last step is a small movement towards the original image such that 1) and 3) hold.

Hop Skip Jump

Starting from the work of [13], Jordan et at. propose a new decision-based attack, Hop Skip Jump Attack [19]. The principal aspect of this attack, is that it works with a gradient direction estimate based only on the output labels of the target classifier. The algorithm is composed by the following steps, first, the current adversarial sample from the last iteration is pushed towards the boundary via a binary search, then the gradient direction is estimated. Thirdly, the step size along the direction estimated is initialized, and is decreased via geometric progression until perturbation becomes successful. The step size is initialized as in Equation 2.17

$$\|x_t \; - \; x^*\|_2 / \sqrt{t} \tag{2.17}$$

where x_t is the adversarial sample at the t-th iteration and x^* is the target image.

Low Pro Fool

Ballet et al.[5] propose a gradient-based method introducing the concept of imperceptibility. This concept can be formalized as follows: considering a set of examples $\mathcal{X}$, where

each example $x^{(i)}$with $i \in [1..N]$ is associated with a label $y^{(i)}$. The set $\mathcal{X}$ is defined by the set of features $j \in \mathcal{J}$, $x_j, j \in \mathcal{J} = [1..D]$. Also we have a classifier f that map $\mathcal{X} : \mathcal{R}^D \to \{0, 1\}$ and $d : \mathcal{R}^D \to [0, 1]$ that map the perturbation $r \in \mathcal{R}^D$ to its perceptibility value. Finally, $A \subseteq \mathcal{R}^D$ is the set of valid samples. Given a sample $x \in \mathcal{R}^D$, its original label $s = f(x)$, the target label $t \neq s$, the optimal perturbation vector $r^* \in \mathcal{R}^D$ is defined in Equation 2.18.

$$\begin{aligned} r^* \;&=\; argmin_r \quad d(r) \quad for \quad r \in \mathcal{R}^D \\ s.t. \quad f(x) = s \neq f(x + r^*) = t \quad &and \quad x + r^* \in A \end{aligned} \tag{2.18}$$

Given $\upsilon = [\upsilon_1, ..., \upsilon_j, ..., \upsilon_D]$ where $\upsilon_j > 0, \forall j \in \mathcal{J}$ is the feature importance vector, we define d including υ, Equation 2.19.

$$d_\upsilon(r) = \|r \odot \upsilon\|_p^2 \quad where \odot \; is \; the \; Hadamard \; product \tag{2.19}$$

The objective function in Equation 2.20 is an aggregation of the class change constraint and the minimization of d_υ:

$$g(r) \;=\; \mathcal{L}(x + r\;, t) \;+\; \lambda \|\upsilon \odot r\|_p \tag{2.20}$$

$\mathcal{L}(x, t)$ is the value of the loss of the model f for sample x and target t, $\lambda > 0$. $\|\upsilon \odot r\|_p$ allows to minimize the perturbation r with respect to its perceptibilty, minimizing $\mathcal{L}(x + r, t)$ ensures that the perturbation leads towards the target class.

Algorithm 2.3 LowProFool Algorithm

Input: Classifier f, sample $\mathbf{x}$, target label t, loss function $\mathcal{L}$, trade-off factor λ, feature importance vector υ, maximum number of iterations N, scaling factor α.

Output:adversarial sample x'

$r \leftarrow [0, 0, ..., 0]$
$x_0 \leftarrow \; x$
for i *in* $0...N - 1$ **do**
    $r_i \leftarrow \nabla_r(\mathcal{L}(x_i, t) \; + \lambda \|\upsilon \odot r\|_p)$
    $r \leftarrow r + \alpha r_i$
    $x_{i+1} \leftarrow clip(x + r)$
end for
$x' \leftarrow \arg\min_{x_i} d_\upsilon \; \forall i \in [0...N - 1]$ s.t. $f(x_i) \neq f(x_0)$
return x'

In Algorithm 2.3, we iterative refine the perturbation r, using the gradient of computed with Equation 2.20.

Espa

Starting from Gen Attack [3], Kumar et al.[33] propose the Evolutionary based Specialized Perturbation Attack (ESPA). This algorithm generates a population of encoders. It will use the best encoder to produce the perturbation that, added to the original sample, produces the adversarial sample.

Algorithm 2.4 Evolution based Specialized Perturbation Attack (ESPA)

Input: a set X of fraud examples, population size P , generation size G, elite members e, batch size n, classifier f , feature importance vector υ, mutation rate ρ

Output:an encoder from the last generation, with the highest fitness.

Initialize:A population P of encoders ϕ_i, with randomly initialized weights θ_i.

for *generation* g=1 to G **do**
  for *individual* $i = 1,..P$ **do**
    $fitness_i = 0$
    for $sample = 1, ..nn$ **do**
      $x_f \sim U(X)$
      $x_f^* \leftarrow x_f + \upsilon \cdot tanh(\phi_i(x_f))$
      $fitness_i \leftarrow fitness_i + \log(f(x_f^*))$
    end for
    return $fitness_i/n$
  end for
  Create new population from top e elite members.
  Mutate the population according to rate ρ
  return the new population
end for
return *final_encoder*

The Algorithm 2.4 starts with a population of P random encoders. It computes fitness ratings for each encoder, averaging the results over n samples. A fraud sample x_f is drawn from set X using a uniform distribution. The perturbation produced using the encoder added to x_f generates the perturbed sample, which is passed through the classifier to calculate the fitness. The next step is to create and mutate (i.e., change the parameters at random) a new population from the best e encoders. This process iterates G times. The best encoder from the previous generation will generate the perturbation needed

to create adversarial samples. t reduces the number of queries required for a successful attack.

Poisoning attacks

With *poisoning attacks*, an attacker can change the training set. According to the previous taxonomy, *poisoning attacks* are *causative* because they alter the training set. The crafting of poisoning samples requires solving a bilevel optimization problem where the outer optimization wants to maximize the classification error on the untainted validation set, and the inner optimization corresponds to training the classifier on the poisoned data [40]. Given a classification task where:

- $\mathcal{X}$ is the features space;
- $\mathcal{Y}$ is the label space;
- $\{$ is the latent function that map $\mathcal{X} \rightarrow \mathcal{Y}$;
- $\mathcal{D}_t r = \{x_i, y_i\}_{i=1}^n$ is training set with n iid samples;

we can estimate f with a model $\mathcal{M}$ trained by minimizing the loss function $L(\mathcal{D}, \mathbf{w})$ with respect to its hyperparameters $\mathbf{w}$. Like with *evasion attacks*, the attacker can have different levels of knowledge of the target, leading to different scenarios. Denoting Θ as the encoding space, $\theta \in \Theta$, $\theta = (\mathcal{D}, \mathcal{X}, \mathcal{M}, w)$. In the perfect knowledge scenario, the attacker has full access to the tuple. If the access to Θ is partial, we are in the limited-knowledge scenario.

The bilevel optimization problem can be expressed as Equation 2.21.

$$\begin{aligned} \mathcal{D}_c^* \in \underset{\mathcal{D}_c' \in \Phi(\mathcal{D}_c)}{\arg\max} \quad & \mathcal{A}(\mathcal{D}_c', \theta) = L(\hat{\mathcal{D}_{val}}, \hat{\mathbf{w}}), \\ s.t. \ \hat{\mathbf{w}} \ & \in \underset{w' \in \mathcal{W}}{\arg\min} \mathcal{L}(\hat{\mathcal{D}_{tr}} \cup \mathcal{D}_c', \mathbf{w}') \end{aligned} \tag{2.21}$$

$\mathcal{D}_c' \in \Phi(\mathcal{D})$ where $\mathcal{D}_c'$, is a set of manipulated samples, $\Phi(\mathcal{D})$ is the space of possible modifications. $\hat{\mathcal{D}}$ is the surrogate dataset available to the attacker. It can be split in $\hat{\mathcal{D}_{tr}}$ and $\hat{\mathcal{D}_{val}}$. The former with $\mathcal{D}_c'$ is used to train the surrogate model, while the latter evaluates the impact of the poisoning on clean data through $mathcal A(\mathcal{D}_c', \theta)$. In literature, there are several works of poisoning attacks. Biggio et al. [10] propose a poisoning attack against Support Vector Machine (SVM). The goal of the attacker is to a point (x_c, y_c) that injected in the training set $\mathcal{D}_{tr}$ maximize the decreasing of the accuracy. To reach this goal, the attacker maximizes the *hinge loss* on the validation set $\mathcal{D}_{val}$ on the SVM

trained on $\mathcal{D}_{tr} \cup (x_c, y_c)$ through a *gradient ascent* technique. This algorithm assumes that the attacker knows the learning algorithm and training data.

AML in fraud detection system

Banks in the last years have progressively used ML models to detect fraudolent transactions [4, 52] and avoid fraudsters to be successful. Moreover, due to the nature of the domain and the continuous domain shift, they periodically retrain the models with the new transactions. With the evolution of defensive mechanisms, also AML attacks evolved. [15] shows that FDS systems are vulnerable to *mimicry attacks*. In this type of attack, the attacker tries to replicate the spending behaviour of the victim to avoid being detected. To reach this goal, they first recover the last transactions of the victim, using, for example, a *trojan*, then study and exploit them to craft adversarial transactions that mimic the good ones. Carminati et al. [16] study the effect of *evasion* attacks against FDS. First, they assume different degrees of the attacker's knowledge, including training data, past victim transactions, the model features, and the model parameters/hyperparameters. The attacker, exploiting the past victim transactions knowledge, crafts fraudolent transactions trying to mimic the spending behaviour, selects the maximum amount and the best timestamp balancing the financial gain and the risk to be detected. Then, the attacker tests the crafted adversarial samples against an oracle. If the surrogate model classifies the sample as legitimate, it is submitted to the target FDS. Otherwise, the amount is decreased to reduce the suspiciousness until either the oracle classifies it as legitimate or the maximum iterations are reached. Monti [39] extend the work of Carminati et al. [16] by studying the effect of *poisoning* attacks against FDS. Since the attacker can not access the training set of the bank, it exploits the periodic update of the training set and the periodical retrain. The attack works in two phases. The first is the *evasion* phase, where the attacker crafts adversarial samples using the [16] technique, and then in the *poisoning* phase, all the misclassified adversarial transactions are included in the training set, so when the FDS will be retrained the decision boundary slowly shift in favor of the attacker. The *evasion* and the *poisoning* phase are repeated until FDS detects all fraudolent transactions. Cartella et al. [17] adapted three algorithms, Zoo, Boundary, and Hop Skip Jump, to suit fraud detection systems domain. The adaptation process needs to deal with several challenges. The principals are the imbalance between fraudolent and legitimate transactions, the different value range of the features, and the field editability. To deal with the dataset imbalance, they introduced the decision threshold. With a classification probability higher than the threshold, the surrogate model classifies the sample as fraudolent. Otherwise, it classifies it as legitimate. Differently from the

image domain, where pixel values have a limited range and one data type, transactions can be very heterogeneous. Finally, in the fraud detection domain, an attacker can edit only a limited number of features, i.e., amount and timestamp. To deal with this, they add specific constraints to avoid the editing of non-editable features. Romeo, in his thesis [48], uses the adapted algorithms to generate the adversarial transactions, showing that with this method, an attacker can steal a relevant amount of money.

2.2.3. Goal and Challenges

Goal

1. Build a framework to evaluate the effectiveness of adversarial attacks in the fraud detection domain
2. Adapt new adversarial algorithms to the fraud detection domain.
3. Bypassing the FDSs with fraudolent transactions generated through the adapted algorithms

Challenges

1. Adapting state-of-the-art adversarial attack algorithms of the image recognition domain to the banking fraud detection context.
2. Reconstruct the adversarial transactions in the original input space starting from the modified features. As stated in Section 2.2, FDSs aggregate transactions to extract user spending behaviour. After crafting the adversarial transaction, the aggregation procedure has to be reversed, to retrieve the adversarial transactions in the original input space.

3 | Threat Model

To define the threat model, we use the framework already used in [8, 11, 16, 23]. Following this framework, we define the *Adversary's goal*, *Adversary's knowledge*, *Adversary's capability* and *Attacker's strategies*.

3.1. Adversary' goal

Following the previous taxonomy of adversarial attacks (see Section 2.2.2), the *Adversary's goal* can be defined in terms of *security violation*, *attack specificity* and *error specificity*. The fraudster wants to carry out an *Integrity violation* since they want the FDS to classify the fraudolent transactions as legitimate. From the point of view of *attack specificity*, the attacker wants to perform an attack that can be *targeted* if the attack aims at specific victims. An attack can be *indiscriminate* if the victims are randomly selected. Since the attacker wants to force the target FDS to classify a fraudolent transaction as legitimate, and the transaction can be either fraudolent or legitimate, the problem is a binary classification.

3.2. Adversary's knowledge

We can model the adversary's knowledge of the FDS target as a tuple $\theta = (D, X, f, w, U, P)$:

- D is the model training data;
- X is the feature set used by the FDS
- f is the learning algorithm and the loss function of the model
- w are the model parameters/hyperparameters
- U represents the past transactions of the victims
- P is the update policy of the FDS

We denote x as full knowledge, $\tilde{x}$ as partial knowledge, and $\hat{x}$ as zero knowledge. Depend-

ing on the knowledge of the attacker, we describe three scenarios:

- *Whitebox*: The attacker has complete knowledge of the FDS that they target. This scenario is unrealistic, but it is meaningful as a worst-case scenario. According to the definition above, this scenario is defined as $\theta = (D, X, f, w, U, P)$
- *Graybox*: The attacker has a limited knowledge of the FDS. They know only the feature set used by the FDS, some past transactions of the victims, and the update policy. $\theta = (\hat{D}, X, \hat{f}, \hat{w}, \tilde{U}, P)$
- *Blackbox*: The black box scenario is the most realistic one and is the scenario where the attacker knows less. The attacker knows partially past victims' transactions. $\theta = (\hat{D}, \hat{X}, \hat{f}, \hat{w}, \tilde{U}, \hat{P})$

In every scenario, the attacker uses past victims' transactions to compute the aggregate features(see Section 7.2).

3.3. Adversary's capability

The adversary's capability describes how much the attacker can manipulate data and the domain-specific data manipulation constraints. In the banking domain, the attacker can directly manipulate only the test set, performing an *evasion* or *exploratory* attack. Indirectly, they also perform a *poisoning* attack, since the adversarial transactions classified as legitimate are included in the training set that the FDS will use during the update, causing the poisoning. The attacker has another strong constraint. They can manipulate a small set of features. They use the victims' past transactions, recovered using a trojan, for example, to compute aggregate features and test the adversarial sample against an oracle. In particular, they can modify only the Amount and the Timestamp.

3.4. Attacker strategy

The attacker can craft adversarial samples following three different strategies: *conservative*, *medium*, *greedy*. According to the selected strategy, the attacker decides how many transactions submits per victim, the minimum and the maximum amount, and the minimum and maximum amount increase with respect to the last successful fraud. Conservative strategy tries to minimize the risk of being detected during the attack, while in the greedy strategy, the risk of being detected is higher because the attacker tries to maximize the profit even at the risk of being more detectable. The medium strategy is a trade-off between the financial profit and the risk of being detected. In Table 3.1 are

listed the parameters for each strategy.

Table 3.1: Parameters for the three attack strategies

	Conservative	**Medium**	**Greedy**
count_increase(%)	40	33,33	25
Amount_increase(%)	75	125	175
min_increase(€)	25	40	50
max_increase(€)	2500	4000	5000
std_max_increase(€)	0,75	1	1,5
min_increase_from_previous_week(%)	20	30	40

4 | Dataset and Preprocessing

In this work, we use two real banking datasets from an important Italian bank. The two datasets are named *2012_13* and *2014_15*. These datasets are the same used in the works of Carminati et al.[16], Monti [39], Romeo [48], Paladini [45] and Maniscalchi [37]. In the previous works, there were three datasets: *2012_13*, *2014_15*, *segnalaz_2014_15*. *2012_13* and *2014_15* were datasets inspected and cleared by an expert, hence without fraud, while *segnalaz_2014_15* contained just confirmed fraudolent transactions for the period of *2014_15*. In this work, *2014_15* is obtained by merging the original *2014_15* and *segnalaz_2014_15*. For privacy reasons, all the personal data are replaced with hashed values. The datasets are composed of the following features:

- TransactionID: Unique identifier of the transaction (String)
- IP: IP address of the connection from which the transaction is executed. This value is hashed (String)
- IDSessione:Unique identifier of the session. This value is assigned from the banking platform. (String)
- Timestamp:Timestamp(date and time) when the transaction is executed.(Timestamp)
- Amount: the amount of the transaction in Euro.(float)
- UserID: Unique identifier that identifies the user.(String)
- IBAN: The hashed value of the IBAN of the beneficiary of the transaction. (String)
- Confirm_SMS: boolean value that indicates if a confirmation code is needed to complete the transaction. (bool)
- IBAN_CC:Country Code(CC) of the iban of the transaction beneficiary.(String)
- CC_ASN: Tuple composed by the Country Code(CC) and the Autonomous System Number(ASN) of the connection of the device from which the transaction is executed. (String, Int)

- FRAUD: a boolean flag that says whether the transaction is legitimate or fraudolent.(bool)

The *2012_13* dataset contains 571712 transactions from 53764 different users. *2014_2015* contains 471787 transactions from 58508 different users. The first dataset covers from December 2012 to September 2013, and the second covers transactions from October 2014 to February 2015.

Table 4.1: Number of transactions, users, and time interval of each dataset.

Datasets	Transactions	Users	Time Interval
2012_13	571712	53764	12/2012 - 09/2013
2014_15	471787	58508	10/2014 - 02/2015

4.1. Data Augmentation

In *2014_15*, the fraudolent transactions are only 0,124% of the total. This disproportion is a known characteristic real banking datasets. In real banking datasets, the share of fraudolent transactions is between 0,1-1% [1]. This characteristic leads to a problem: ML models for supervised learning that are used to classify the transaction assume to have a balanced distribution between the classes. The imbalance of the datasets complicates the classification and affects the performance. We perform a data augmentation to achieve 1% of fraudolent transactions in our datasets. In the procedure of data augmentation, we craft "synthetic frauds" that replicate the attack schemes that fraudsters usually conduct against victims: information stealing and transaction hijacking (Section 2.1.1). The process works as follows: in the first step, we divide the users into three sets depending on the number of transactions and the average amount per transaction. The sets are *high*, *medium*, *low*. From each of these sets, we randomly draw a fixed number of users. Then, for each victim, we craft one or more frauds.

Table 4.2: Number of transactions and average amount per transaction of each victim's profiles. min_count and max_count are respectively the minimum and maximum transaction count. min_mean and max_mean are respectively the minimum and maximum average amount of each transaction

	low_profile	medium_profile	high_profile
min_count	5	15	35
max_count	14	34	1100
min_mean	0	1500	3000
max_mean	1499	2999	50000

The datasets obtained by merging the original datasets with the synthetic frauds are called *augmented_2012_13* and *augmented_2014_15*

4.2. Data Exploration

We analyze the fraudolent transactions separately from the legitimate ones. For the amount distribution, we group the amounts using custom bins. In both *augmented_2012_13* and *augmented_2014_15*, the AMOUNT distribution is highly skewed. Datasets have respectively an average amount of 1812,44€ and 1810,04€ while the medians are 617,10€ and 596,51€. In Figure 4.1, we can see the amount distributions of legitimate and fraudolent transactions for both datasets. If we inspect only the legitimate transactions, the means are 1787.22€ and 1755.06€, and the medians are 613.49€ and 589€, very similar to the entire datasets, because the legitimate transactions are the vast majority. The distributions of fraudolent transactions are still skewed but more towards the higher values. The means are 5874.59€ and 7740.35€ while the medians are 1598.02€ and 1889.16€. In the first dataset, on average, each user execute 10.69 transactions, and in the second, 8.12. As we can see in Figure 4.2, with regards to legitimate transactions, in the first dataset, 80% of users execute up to 12 transactions, and 50% of users execute no more than 5 transactions, and the mean is 10.62. In the second dataset 80% of the users execute up to 9 transactions, the 50% of the users execute from 1 to 4 transactions, and the mean is 8.05. In the fraudolent class, in both datasets, the median is 1 transaction per user, and 80% of users execute up to 6 transactions. The means are respectively 4.59 and 4.68 transactions per user. In Figures 4.3, 4.4 we can see the fraudolent transaction count per IBAN_CC. The IT(Italy) Country Code, is respectively the 33,4% and 39,3% of the transactions in *augmented_2012_13* and *augmented_2014_15*. If we look at the Country Codes non-italian, the most frequent IBAN_CCs are DE(Germany), GB(United King-

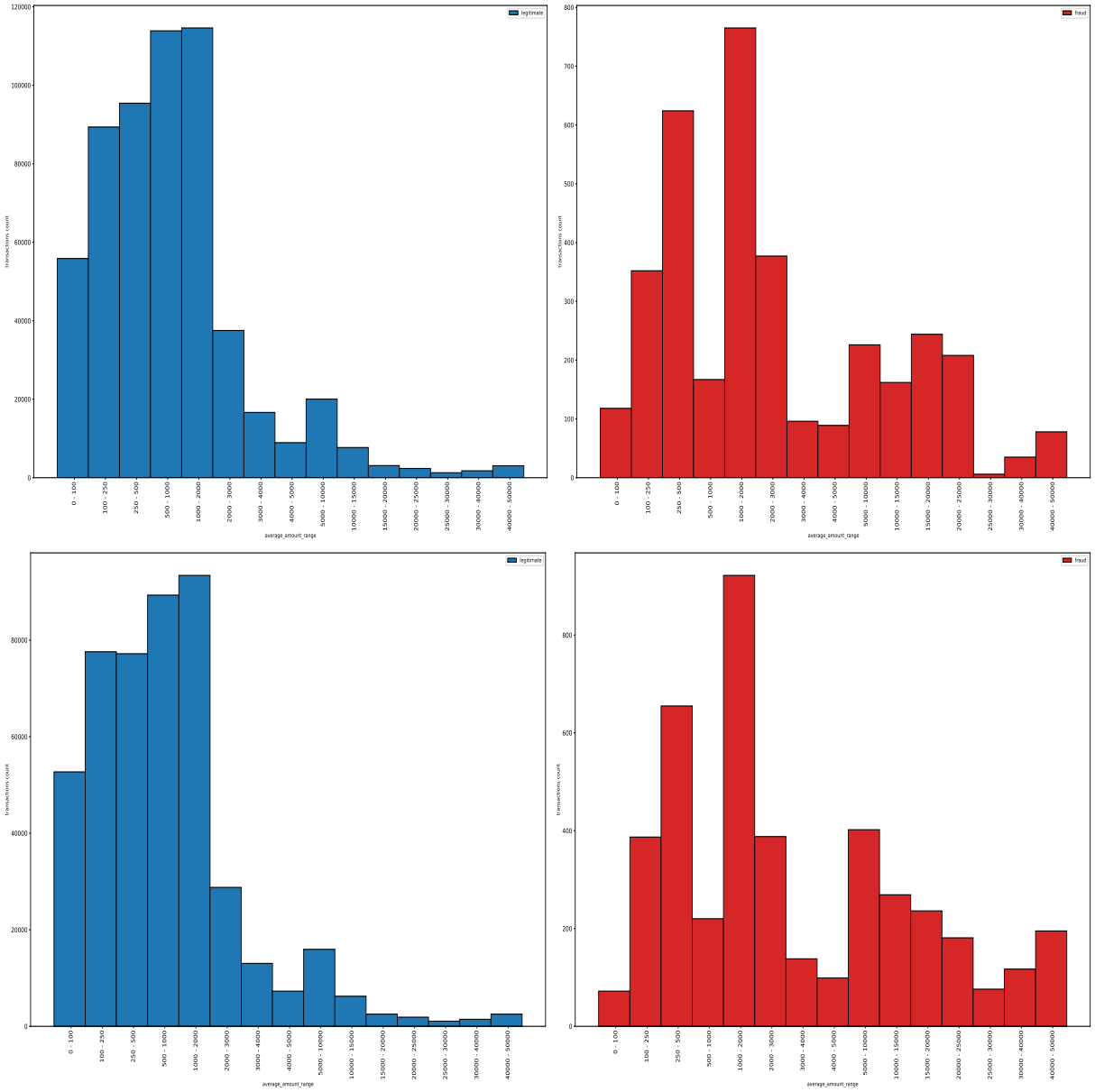


Figure 4.1: Amount distributions of legitimate and fraudolent transactions for *augmented_2012_13* and *augmented_2014_15*

dom), FR(France), ES(Spain), RO(Romania), SI(Slovenia) for *augmented_2012_13* and DE(Germany), GB(United Kingdom), FR(France), ES(Spain), RO(Romania), SK(South Korea) for *augmented_2014_15*.

In Figures 4.5, 4.6 we can see the same thing for legitimate transactions. In both datasets, 98% of the legitimate transactions have an IT IBAN_CC. The most frequent non-italian IBAN_CCs are DE(Germany), GB(United Kingdom), FR(France), ES(Spain), RO(Romania), NL(Netherlands), AT(Austria) and CH(Switzerland). South Korea and Slovenia are be-

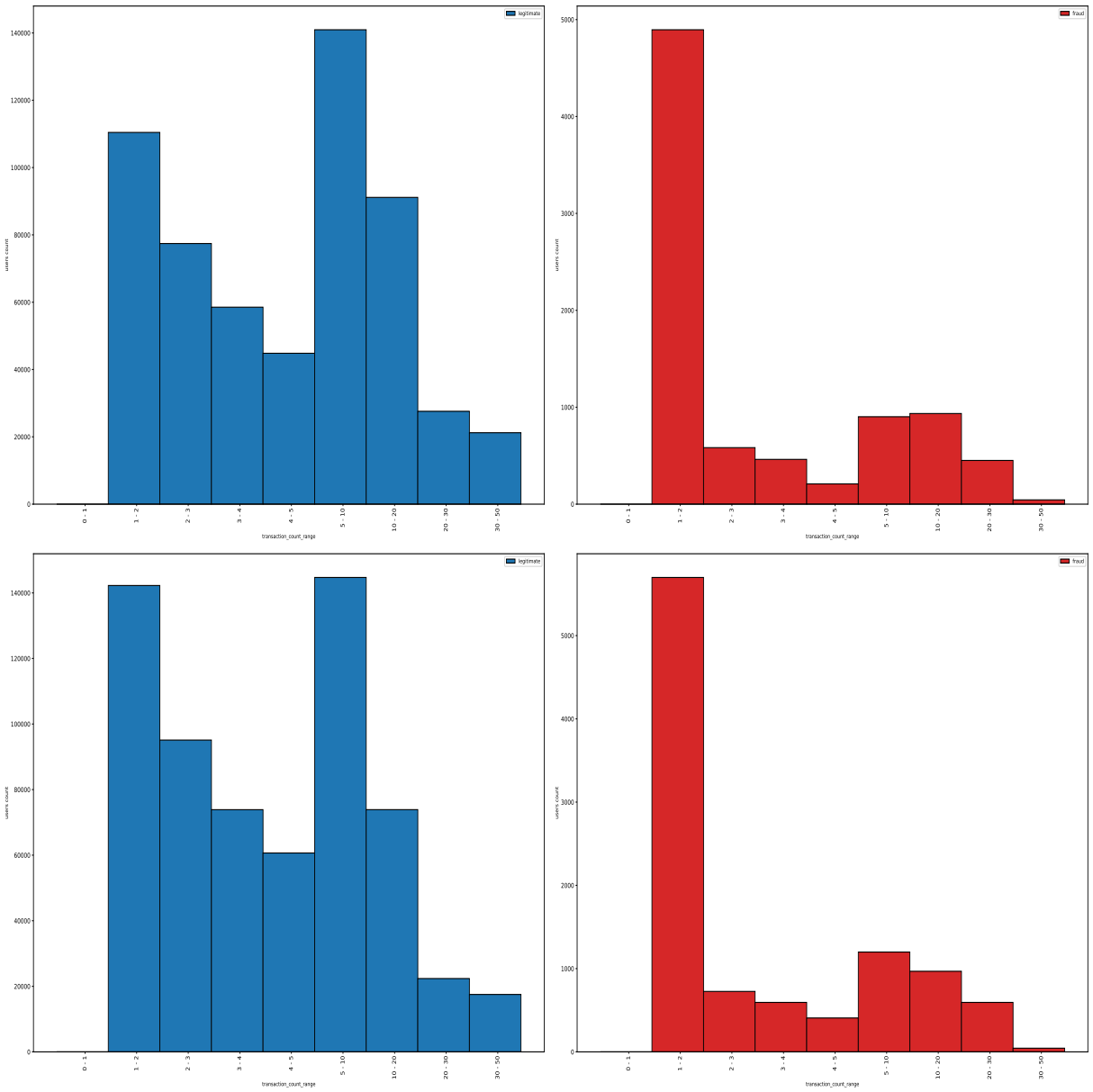


Figure 4.2: Transactions count distribution per user, for *augmented_2012_13* and *augmented_2014_15*

low 3%.

The most significant differences between legitimate and fraudolent transactions are in the Amount and Timestamp. In Figure 4.1, we can see that the most common spending range in legitimate transactions is 250-2000€ for both datasets, while for the fraudolent transactions, the most frequent range is 1000-2000€. The legitimate transactions are executed during the day, with a very small fraction executed during night time, while the fraudolent transactions have more even distribution as shown in Figures 4.7, 4.8.

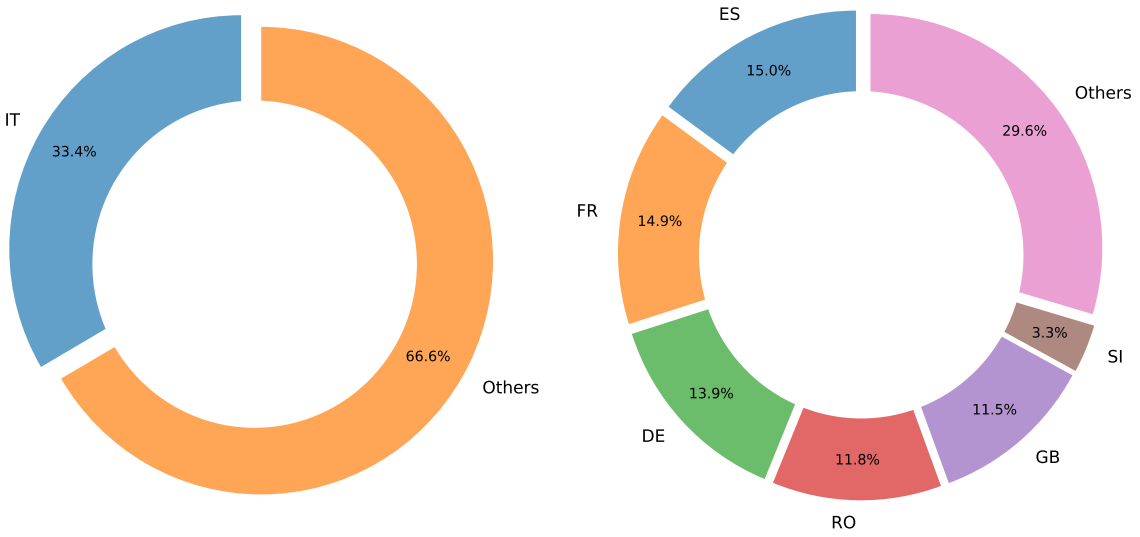


Figure 4.3: Fraudolent transaction count per IBAN CC for *augmented_2012_13*

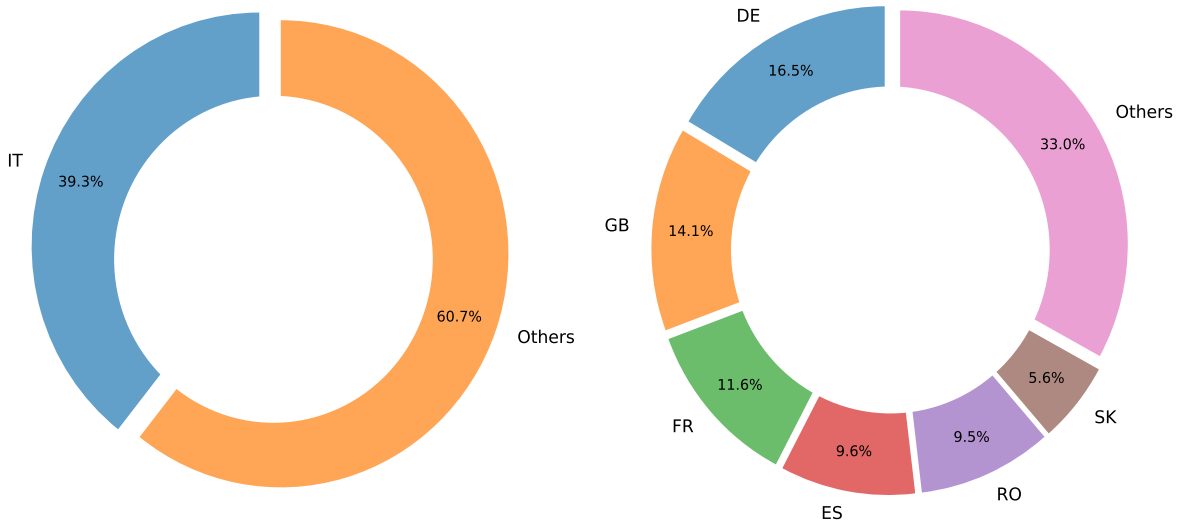


Figure 4.4: Fraudolent transaction count per IBAN CC for *augmented_2014_15*

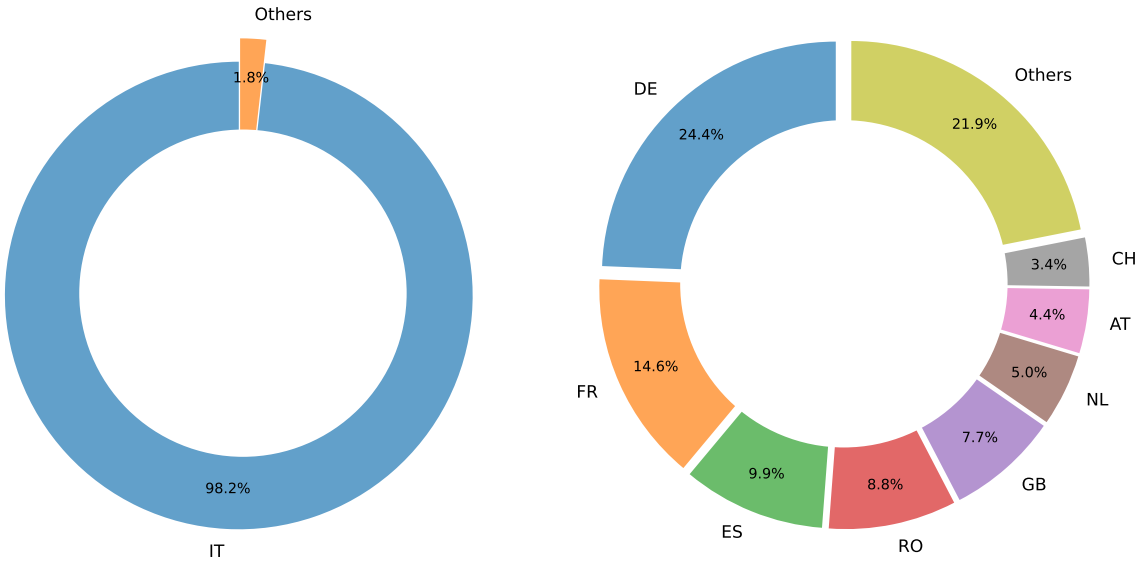


Figure 4.5: Legitimate transaction count per IBAN CC for *augmented_2012_13*

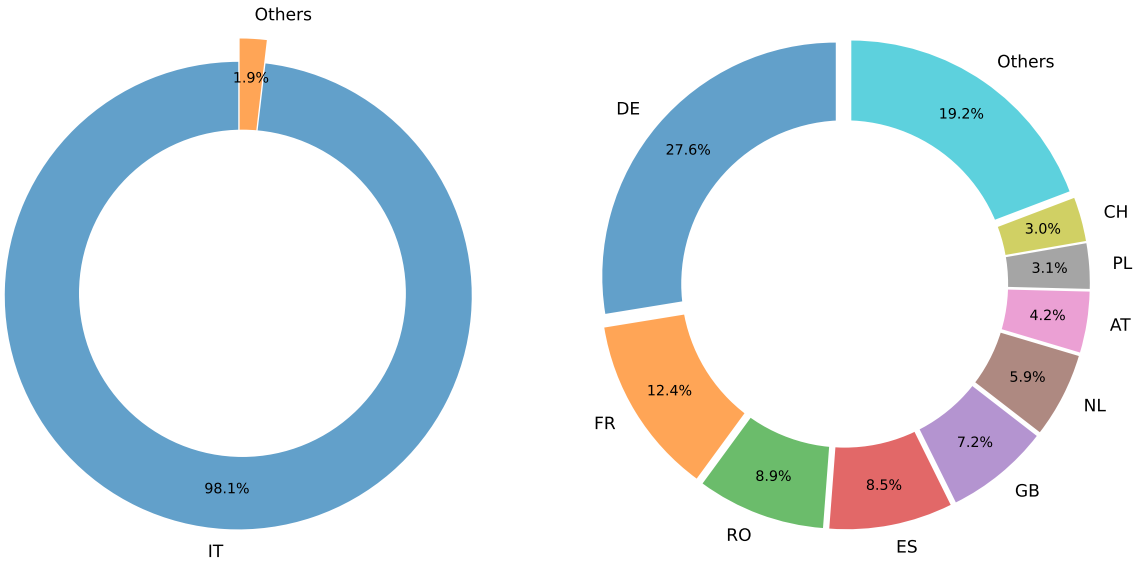


Figure 4.6: Legitimate transaction count per IBAN CC for *augmented_2014_15*

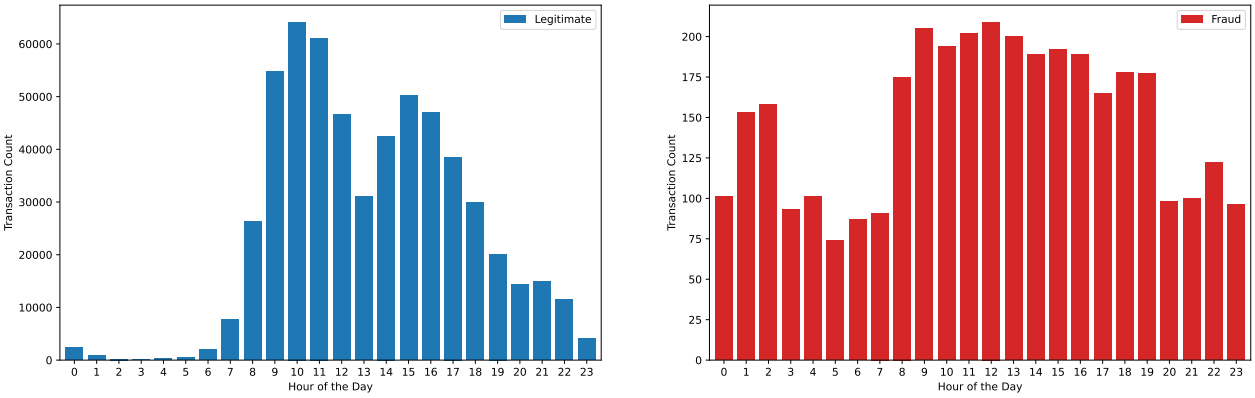


Figure 4.7: Legitimate and fraudolent transactions count for *augmented_2012_13*

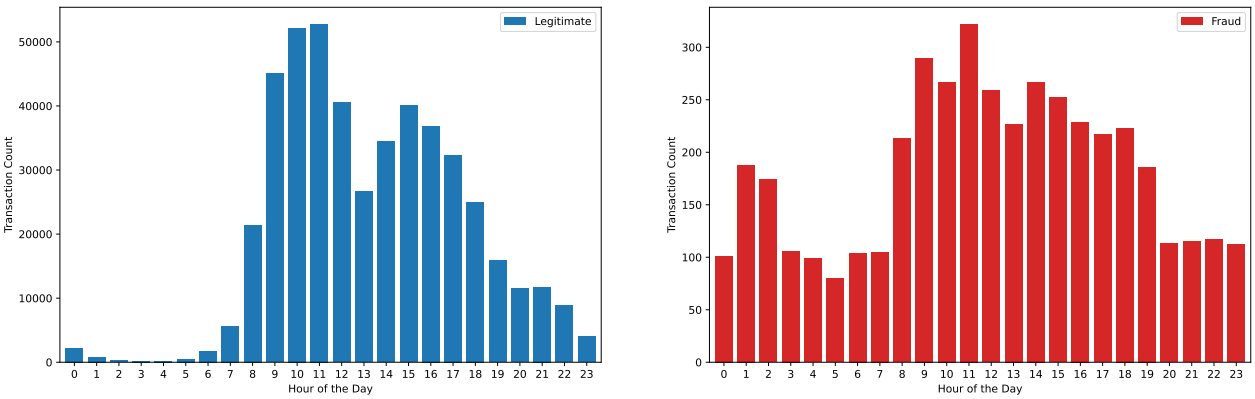


Figure 4.8: Legitimate and fraudolent transactions count for *augmented_2014_15*

5 | Approach

In this chapter, we first describe the approach that we followed to build the framework, providing a high-level view of the proposed architecture. Secondly, we describe the approach of the attacks and the assumptions we rely on.

5.1. Framework Overview

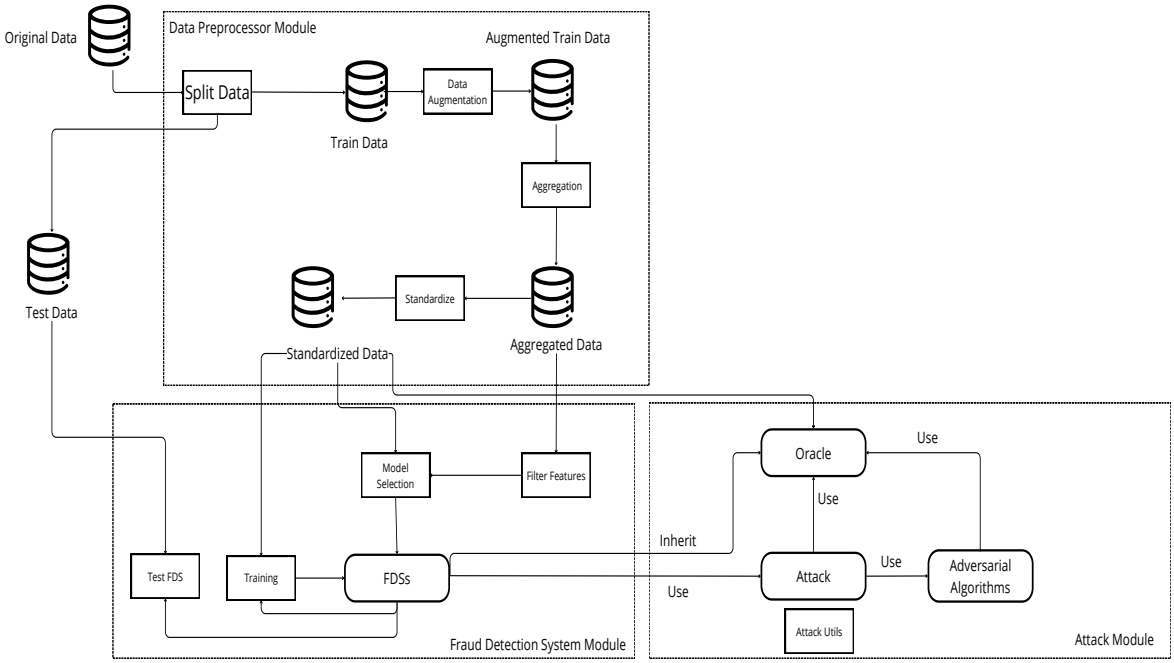


Figure 5.1: Framework overview

Our framework is composed of three main modules as we can see in Figure 5.1. The first one is the **Dataset Preprocessor Module**, which is in charge of performing all the preprocessing steps. It contains components that handle tasks like splitting trans-

actions into training and test sets, aggregating features, rescaling, and performing Data augmentation. The second module is **Fraud Detection System Module**. This module is in charge of simulating a FDS. Its subcomponents deal with the model creation, the hyperparameters tuning, and the feature selection. The third is **Adversarial Attack Module**. With this module, we define the adversarial attacks, instantiate them, and perform simulations of Poisoning and Evasion attacks. In the following sections, the subcomponents are presented in detail.

5.2. Dataset Preprocessor Module

5.2.1. Dataset Augmentation

This component is in charge of performing dataset augmentation. As described in Chapter 4, the datasets we use are heavily imbalanced. To deal with this problem, we perform Dataset Augmentation. To do this task, synthetic frauds that replicate real fraudolent transactions are injected in the original datasets.

Victim Selection

To generate synthetic frauds, first, we have to select the victims. We first discard users with insufficient data, considering only those who have performed more than three transactions eligible for further processing. Second, victims are categorized according to the average amount and number of transactions performed, in three categories(see Table 4.2) :

- High Profile: Users with this profile have a high average amount or perform a high number of transactions.
- Medium Profile: Users with this profile have a medium average amount or perform a medium number of transactions.
- Low Profile: In this profile, we find users who have a small average spending amount or carry out a small number of transactions.

Synthetic Fraud Generation

Once the victims are selected, we craft the synthetic frauds following two fraud schemes: *Information Stealing* and *Transaction Hijacking*. In *Information Stealing*, the attacker obtains access to the victim's bank account and performs the transactions on their behalf. In *Transaction Hijacking*, a legitimate transaction is intercepted and redirected

towards an attacker's account. In both schemes, the attacker can only edit Amount and Timestamp. In the *information_stealing*, the IP, IDSessione and IBAN are randomly generated hashed strings since, in our datasets, these features are hashed to preserve privacy. IBAN_CC, CC_ASN and Confirm_SMS are assigned according to their distribution in the real datasets. In *transaction_hijacking*, the IP, IDSessione, and CC_ASN, are the same as the hijacked transaction. The Amount selection process is the same in both schemes. Regarding the Amount of fraudulent transactions, we provide the generator with a range of values within which it will pick the transaction amount. Timestamp in *information_stealing*, is randomly selected between the first and the last Timestamp of the dataset, while in *transaction_hijacking* the Timestamp is the same of the hijacked transactions.

5.2.2. Transactions Aggregation

With this module, we perform the transaction aggregation to obtain datasets with aggregated features. This step is necessary because, from the raw features, we can extract a limited amount of knowledge, while with the aggregated features, we can model the spending behaviour more in detail. The detailed aggregation procedure is described in Section 7.2.

5.3. Fraud Detection System Module

In our framework, we implement 5 ML models utilised for fraud detection: XGBoost (XGBoost), Random Forest (RF), Neural Network (NN), Logistic Regression (LR), Support Vector Machine (SVM). See Appendix A for more details about each model. This module is in charge of creating the ML models, fitting them, and performing the model selection.

5.3.1. Features Filtering

The features filtering component performs a first skimming of the features starting from the aggregated features set. The skimming is done via correlation filters [54]. The correlation filters select only the features more correlated with the desired target and uncorrelated each other.

5.3.2. Model Selection

The model selection component selects the best combination of hyperparameters and features set for each FDS. The component selects the features set and the hyperparameters simultaneously. The procedure is based on forward selection. The Algorithm 5.1 starts from an empty model and progressively includes new features. Forward model selection divides the starting feature set(the aggregated feature set) into chunks of features. At each step, the chunk of features that brings the best improvement to the overall performance, when added to the final feature set, is included in the final feature set. To assess the performance of each chunk, we perform *k-fold cross validation* with a set of m models. Hyperparameters of each model are selected via Random Search approach [7]. After each iteration, the new final feature set with the hyperparameters of the best model is saved. This procedure continues until adding new chunks of features to the final set adds no more improvement. The chosen metric is F1-Score. We chose this metric because it suits well the imbalanced datasets. If we use Accuracy, even a model that classifies every transaction as legitimate achieves a high Accuracy. In Appendix B, there is a brief explanation of the metrics that can be used to evaluate the models. Along with Forward Model Selection, we can also follow a grid search approach to find the hyperparameters.

5.4. Adversarial Attack Module

Inside this module, there are the components in charge of instatiating the adversarial attacks and performing the simulation of Evasion or Poisoning Attacks.

5.4.1. Oracle

This component instantiates the Oracle that will be used in the attack. The Oracle is a substitute model used during the attacks to filter the adversarial transactions and select only the more stealthy ones.

5.4.2. Attack

This component carries on the simulation of the attack. It can perform both an Evasion Attack and a Poisoning Attack. The difference between them is that the Evasion Attack stops before the first update of the FDS. It loads the configuration files, instantiates the FDS and the Oracle, and carries out the simulation.

5.4.3. Attack Utils

This component contains all the utilility functions needed to carry out the attack simulation.

5.4.4. Adversarial Algorithms

This component contains the adversarial algorithms adapted to the fraud detection domain. In Chapter 6 we present how we adapt the algorithms.

5.5. Assumptions

To conduct the Attack simulation, our approach relies on the following assumptions:

(A1) The Fraud Detection System ML model is periodically updated according to its update policy. During each update, the model is retrained with the new legitimate transactions added to the training set;

(A2) The attacker has obtained a banking dataset with real bank transactions that are used to train the Oracle;

(A3) The attacker has retrieved past victim transactions and is able to observe the transactions made by the victim during the attack;

(A4) The attacker is capable of submitting transactions on behalf of the victim, and there are enough funds available on the victim's account;

(A5) The attack is detected as soon as one of the fraudulent transactions submitted by the attacker is classified by the FDS as a fraud;

(A6) the Oracle's ML model has *Amount* and *time_from_previous_trans_global* in its aggregated features set.

We share all the assumptions with the attacks described in Monti [39] and in Carminati et al. [16]. The assumption (A1) is needed only for the Poisoning Attack, the Evasion Attack does not need it.

5.6. Evasion Attack and Poisoning Attack

The Evasion Attack can be divided into three phases:

- First phase: the attacker retrieves past victim's transactions. They will use these

transactions to extract the victim's spending behaviour, compute the aggregate features, and reconstruct the adversarial transaction in the original input space.

- Second phase is the crafting phase: the first adversarial crafting is done following *domain-specific-decision-based* rules that try to replicate the victim's spending behaviour. After crafting the first transaction, the transaction is tested against the Oracle. If the Oracle classifies the transaction as legitimate, it is submitted to the target otherwise, the attacker exploits an adversarial algorithm to improve the chance of the fraudolent transaction to evade the FDS.
- The third phase is the Evasion Phase: during this phase, all the transactions classified as legitimate in the second phase are submitted to the FDS, which classify the incoming transactions.

The Poisoning Attack shares all the phases of the Evasion Attack, but it adds a fourth phase, which is the Poisoning phase. In this last step, all the new transactions classified as legitimate are included in the training set, which is periodically updated, causing a deterioration in the capacity of detecting fraudolent transactions.

Algorithm 5.1 Forward Model Selection

Input: *dataset_name*: the name of the dataset to load
model_name: the name of model to use
metric: the name of metric to optimize
Procedure:

```
available_features = getFeaturesFromConfigFile(dataset_name)
tolerance = getParamFromConfigFile("tolerance")
max_attempts = getParamFromConfigFile("max_attempts")
random_models_num = getParamFromConfigFile("random_models_num")
folds_num = getParamFromConfigFile("folds_num")
df = loadStandardizedDataset(dataset_name)
X_train, y_train = splitTrainTest(df)
overall_best_score = 0
attempts = 0
current_feature_set = set()
while (attempts < max_attempts) ∧ (available_features  current_feature_set ≠ 0) do
    features_to_process = available_features \ current_feature_set
    for feature in features_to_process do
        temp_feature_set = current_feature_set ∪ feature
        models_params = generateModelsParams(model_name, random_models_num)
        features_scores[feature] = getFeatureBestScore(cross_val_results, metric)
    end for
    best_feature, best_feature_score = getBestFeature(features_scores)
    current_feature_set = current_feature_set ∪best_feature
    current_score = best_feature_score
    saveResultsToFile()
    if (current_score - overall_best_score) < tolerance then
        attempts = attempts + 1
    else
        attempts = 0
    end if
    if attempts == 0  then
        overall_best_score = current_score
    end if
end while
```

6 | Algorithms Adaptation

In Section 2.2.2, we presented the original version of these algorithms, and now we describe how we adapt them. In all the algorithms, we introduce a decision threshold. Oracle uses this threshold to decide if a transaction is fraudolent. Boundary, HopSkipJump, and ZOO maintain the modification introduced by Romeo[48].

6.1. ZOO

First, we employ the novel loss function proposed by Cartella et al. [17]:

$$f(x,t) = max[Z(x)]_t - \tau, -k \tag{6.1}$$

In eq. (6.1), t is the target fraud class, τ is the model decision threshold, and k is the standard variable for attack transferability. This loss function ensures that minimum values are obtained only for adversarial samples X^* such that $[Z(x^*)]_t \leq \tau$. The decision threshold *tau* is the decision threshold that we mentioned above, it depends on the Attacker's knowledge. To estimate the gradient, we use different values h according to the different editable features. We introduce this difference with the original algorithm because, while in the image domain, each pixel can have a value in [0...255], in our domain, the editable features don't have the same range.

6.2. Boundary

The first change we make to Boundary attack is how the adversarial sample is initialized. In the original algorithm, it is used a $\mathcal{U}(0, 255)$ distribution. In this context, the features have different ranges of values, hence, we draw the initial adversarial sample still from a uniform distribution, but the range goes from minimum(x_{min}) to maximum(x_{max}) that each editable feature can assume. The range of values a feature can assume in the initialization phase is decreased. This is necessary because the algorithm procedure starts only when an initial adversarial sample belonging to the target class(Legitimate) is found. By

decreasing the range of values, we enhance the probability of finding a legitimate initial sample. Moreover, the Orthogonal perturbation is not the same for every editable feature, but it is scaled according to which feature is perturbing.

6.3. HopSkipJump

HopSkipJump shares the same modification introduced in Boundary attack. The initialization is done by drawing an initial adversarial sample from the distribution $\mathcal{U}(x_{min}, x_{max})$ instead of $\mathcal{U}(0, 255)$, where x_{min}, x_{max} denote respectively the min and the max value that a feature can assume. Also, in this case, we restrict the ranges to improve the probability of finding a suitable initial sample. Furthermore, we deal with the perturbation produced by HopSkipJump in the same way we dealt with the Orthogonal perturbation in Boundary.

6.4. FGSM

Fast Gradient Sign Method is one of the most simple adversarial attacks. The only change that we introduce is that the noise added to the transaction is scaled depending on the feature perturbed. This is necessary because the features have a different range of values.

6.5. Basic Iterative Method

BIM is the iterative version of FGSM. Like this one, the noise applied to the editable features is scaled according to the different ranges of values. Moreover, since the gradient direction,i.e., the sign, is computed at each iteration, we need to restore the consistency among the features in the transaction. To do this, we reconstruct the raw transaction with the perturbed features and then compute the transaction in the aggregate feature space after each perturbation.

6.6. Saliency Map

The original version of Saliency Map selects the two features that lead more towards the target class. Since we can edit only two features, the modified version of Saliency Map takes, at each iteration, the feature among *Amount* and *time_from_previous_trans_global* that leads more towards the target class. Moreover, if the algorithm doesn't find a legitimate sample, it repeats the procedure with the same perturbation amount but with a different sign. As already done with BIM, the perturbation is scaled according to the

feature, and after each iteration, the consistency among features is restored.

6.7. LowProFool

Like the other algorithms, the noise is scaled according to the feature that is perturbed. Then, since at each iteration, the gradient is computed from the original sample, and the last computed perturbation, at each iteration, the consistency among the features of $perturbed\_sample = original\_sample + last\_perturbation$ is restored.

6.8. ESPA

This algorithm trains an encoder that produces the perturbation. This perturbation is multiplied by an importance vector and then applied to the transaction. The changes we introduce affect the training phase, where the algorithm finds the final encoder. First of all, during the training phase, the perturbation produced by each encoder is scaled according to the feature, like the other algorithms. Now, for each transaction perturbed, we restore the consistency among features. With the consistency restored, we can score correctly each encoder.

7 | Fraud Detection System

The Fraud Detection Systems (FDSs) are implemented with a *system-centric* approach instead of a *user-centric* approach. In this approach, there is a single machine learning model that represents the entire system, while in the *user-centric*, there is one ML model for each user in the system.

7.1. Update policy and concept drift

With a *user-centric* approach, the ML model tries to learn the spending behaviour of the single user. With a *system-centric* approach, the scope is the entire system. During its lifetime, the spending behaviour of a person can change due to different reasons, such as personal needs (like buying a new car) or other causes(like a financial crisis). This change in spending behaviour is called *concept drift*. The FDS deal with this phenomenon by assigning a discount weight to each transaction in the dataset. The weight is computed to give more relevance to the latest transactions.

$$weight_{tran} = -exp(\frac{t}{k}) \tag{7.1}$$

Equation 7.1, computes the weight per transaction. t is the difference in hours between the timestamp FDS training and the timestamp of the transaction and k in a constant equal to $4380h = 0.5year$. Moreover, to include the new transactions that happen during the time, the FDS is updated according to an *update policy*, which can be *weekly* or *biweekly*.

7.2. Feature Engineering

As said before, the FDS doesn't classify the incoming transactions directly in the original feature space, but starting from the raw transactions, it computes the *aggregate features* by aggregating the new transactions with the past ones. The FDS uses two types of features, the *direct features* and *aggregate features*. *Direct features* are the features that

can be extracted directly from the raw transactions. In this work, we use the same direct and aggregated features used by Monti, Paladini, Carminati et al., and Romeo in their works [16, 39, 45, 48]. In the *direct features* there are:

- *Amount*: the amount in euro transferred in the transactions
- *Time_x*, *Time_y*: These features are cyclic encoding of the time when the transaction occurred. The time is derived from the *timestamp*. The encoding is needed because, without that, the distance between hours can't be computed properly. For example, without encoding, the distance between 15:00 and 18:00 is 3 hours, but between 23:00 and 02:00 is 21. After converting time in seconds with Equation 7.2, time is encoded in two dimensions, using sine and cosine functions as shown in Equations 7.3, 7.4.

$$t = ts_h * 3600 + ts_m * 60 + ts_s \tag{7.2}$$

$$time_x = cos\frac{t * 2\pi}{86400} \tag{7.3}$$

$$time_y = sin\frac{t * 2\pi}{86400} \tag{7.4}$$

- *is_national_iban*: this boolean value specifies if the transaction beneficiary's IBAN is or is not Italian, IT as Country Code (CC).
- *is_international*: boolean value that specifies if the transaction beneficiary's IBAN has the same nationality as the customer, i.e., they have the same Country Code.
- *confirm_sms*: this feature is a boolean value that says if the transaction needs a confirmation SMS or not

Before continuing with the aggregated features, we define three sets: *group*, *function*, *time*, that will be used to compute the aggregated features.

- *group*: is the set of original raw features: *IP, IBAN, IBAN_CC, CC_ASN, SessionID*
- *function*: This set contains the operations that are used to compute the aggregate

features: *sum, count, mean, std*

- *sum*: is the function that sums the amount of the transactions.
- *count*: is the function that counts the transactions.
- *mean*: is the function that computes the mean of the amount of the transactions.
- *std*: is the function that computes the standard deviation of the amount of the transactions.

- *time*: in this set there are the possible time spans: 1h, 1d, 7d, 30d, 8670h. These time windows represent one hour, one day, seven days, thirty days, and one year.

Considering these three sets, *group, function and time*, the aggregated features are constructed in the following manner:

- *time_from_previous_trans_global*: time elapsed in hours from the last transaction made by the same user.
- *is_new_group*: this boolean value specifies if the user executes a transaction towards a new group. For example, *is_new_cc_asn*.
- *time_since_same_group*: this is the time elapsed in hours since the last transaction towards the same group, made by the user. For example, *time_since_same_iban_cc* is the time elapsed in hours between the last transaction made by the user towards the same *IBAN_CC*.
- *group_function_time*: this kind of feature aggregates the past user's transactions according to the value of *group*, in a time window specified in *time*, applying *function*. For example, *iban_mean_7d* is the average amount of the transactions towards the same *IBAN* in the last seven days.
- *difference_from_group_meantime*: this feature is the difference of amount between the new transaction and the average amount of the transactions towards the same group in the considered time window. For example, *difference_from_iban_1d* is the difference between the amount of the new transaction and the past transactions in the last day towards the same *IBAN*.

If the *group* value is not specified, the past transactions are grouped only by the users, and in the place of *group* there is *amount*. *amount_sum_7d* is the sum of the amount of all the user's transactions executed in the last seven days. Features grouped only by user,

i.e., *group* value equal to none, and with a time window of one year(8670h) are renamed from *amount_sum_8670h* in *global_amount_sum_8670h*.

7.3. FDS and Oracle Model Selection

To select the model's features and hyperparameters, first, we filter all the aggregated features obtained after the aggregation process, described in Section 7.2. We use a correlation filter method to filter the features [54]. Starting from the complete set of the aggregate features, we compute the correlation matrix first, then analyze two features at a time, and if these two are highly correlated($\geq 95\%$), the feature less correlated to the target is removed. The next step to select the features and the hyperparameters of the FDSs' models is to compute the forward model selection. This procedure selects, at the same time, the features set and the hyperparameters set. The procedure aims to find the best combination of hyperparameters and features. In the forward model selection procedure, we use the *augmented_2014_15* dataset up to 05/01/2015, which is the attack starting date. This dataset is split into two subsets, *training set* and *validation set*. The first is used to train the models, and the second is used to assess the performance of the models. When the forward selection is ended, the models are tested on the portion of *augmented_2014_15* after 05/01/2015, which is the test set. The procedure is described in Section 5.3.2. In Table 7.1 there are the FDS models' hyperparameters and in Table 7.2 there are FDS models' features.

For what concerns the Oracle's features and hyperparameters selection, we use *augmented_2012_13* dataset. We select the features for the blackbox scenario using the correlation filter methods explained above. In the graybox scenario, the Oracle's features are the same as used by the FDS, hence, we don't have to select the Oracle's features set for the graybox scenario. The hyperparameter tuning is done using a grid search method from a predefined parameters grid. We repeat this procedure for blackbox and graybox scenarios. We split the dataset into the *training set* and *test set*, using 10/08/2013 as a split date. The first dataset is used to train the models and perform *k-fold cross-validation*. We select the parameters configuration with the highest F1-Score. In the Table 7.3, there are the features used by the Oracle in the blackbox scenario, and in Tables 7.4, 7.5, there are the parameters for the Oracle's models in blackbox and graybox scenario. We use XGBoost as Oracle's model for the decision-based attacks, because it is the one with the best performance, and Neural Network as Oracle's model in gradient-based attacks because it has the best performance among the models that allow us to compute the gradients.

In Table 7.6 there are the scores of each FDS model and in Table 7.7 there are the

Table 7.1: Selected hyperparameters per each FDS model.

Model	Hyperparameter	Value
RF	random_state	721077
	n_estimators	455
	max_depth	95
	max_features	"sqrt"
	class_weight	"balanced"
	criterion	"entropy"
	min_samples_split	3
	min_samples_leaf	3
LR	random_state	721077
	penalty	"l2"
	C	21.812987353186795
	class_weight	"balanced"
	tol	0.9185493914343154
	solver	"newton-cholesky"
	l1_ratio	"None"
	max_iter	4000
SVM	random_state	721077
	penalty	"l1"
	loss	"squared_hinge"
	dual	"False"
	tol	0.8143378370585579
	class_weight	"balanced"
	C	26.271913849254833
	max_iter	10000
XGB	random_state	721077
	max_depth	42
	learning_rate	0.14578773732253003
	booster	"gbtree"
	gamma	0.004965417762352055
	n_estimators	301
	scale_pos_weight	115.41233766233766
NN	epochs	40
	batch_size	2048
	fl_neurons	151
	sl_neurons	55
	activation_funct	"relu"
	dropout_rate	0.002730031098755359

Table 7.2: Selected features per each FDS model.

Model	Features
RF	Amount, time_from_previous_trans_global,time_since_same_cc_asn, Time_x, is_national_iban, ip_std30d, iban_std30d, iban_cc_mean14d, difference_from_amount_mean30d, iban_sum1d, amount_sum7d, ip_mean30d, amount_std7d, iban_sum30d, time_since_same_iban, Time_y
LR	Amount, time_from_previous_trans_global, ip_count30d, iban_cc_count1h, iban_count7d, ip_count14d, difference_from_iban_mean1d, session_count30d
SVM	Amount, time_from_previous_trans_global, time_since_same_cc_asn, iban_std1h, amount_count1d, is_new_iban_cc, time_since_same_session, time_since_same_iban, amount_std1d, is_national_iban
XGB	Time_x, Amount, time_from_previous_trans_global, time_since_same_cc_asn, difference_from_iban_mean1h, is_national_iban, amount_std7d, iban_cc_mean30d, iban_sum30d, ip_std30d, is_new_cc_asn, iban_std30d, iban_count14d, iban_count7d, is_new_iban, amount_std30d, iban_cc_mean1d, amount_count7d, amount_count1d, difference_from_ip_mean7d, difference_from_amount_mean30d, iban_cc_mean7d, session_std1h, session_count30d, Time_y, session_sum1h, difference_from_amount_mean1h,iban_cc_mean14, ip_mean30d, time_since_same_iban_cc, iban_mean30d
NN	time_from_previous_trans_global, amount_sum30d, is_new_iban, iban_cc_mean14d, ip_count30d, time_since_same_iban, iban_sum7d, time_since_same_session, iban_count30d, is_international, iban_std30d, time_since_same_iban_cc, ip_count14d, ip_sum30d, ip_mean30d, Amount, difference_from_iban_mean7d, difference_from_session_mean1h, iban_count1d, amount_sum14d, is_national_iban, difference_from_amount_mean1h, ip_count7d, amount_sum7d, amount_std30d, iban_cc_mean1d, difference_from_iban_mean1h, iban_sum30d, iban_std1h, iban_count7d, amount_std1d, iban_sum14d, Time_x, iban_count1h, difference_from_amount_mean14d, session_std1h

performance of the Oracle in graybox scenario, and in Table 7.8 the performance in blackbox scenario.

Table 7.3: Selected features for the Oracle in blackbox scenario.

	Features
Oracle	Amount, Time_x, Time_y, amount_count14d, amount_count1d, amount_count30d, amount_count7d, amount_std30d, amount_sum1d, amount_sum1h, amount_sum30d, amount_sum7d, cc_asn_sum14d, cc_asn_sum30d, cc_asn_sum7d, difference_from_amount_mean14d, difference_from_amount_mean1d, difference_from_amount_mean1h, difference_from_amount_mean30d, difference_from_amount_mean7d, difference_from_iban_mean14d, difference_from_iban_mean1d, difference_from_iban_mean1h, difference_from_iban_mean30d, difference_from_iban_mean7d, difference_from_ip_mean14d, difference_from_ip_mean30d, difference_from_ip_mean7d, difference_from_session_mean1h, iban_cc_count1h, iban_cc_mean14d, iban_cc_mean1d, iban_cc_mean30d, iban_cc_mean7d, iban_cc_std14d, iban_cc_std1d, iban_cc_sum14d, iban_cc_sum30d, iban_cc_sum7d, iban_count14d, iban_count1d, iban_count1h, iban_count30d, iban_count7d, iban_mean30d, iban_std7d, iban_sum1d, iban_sum1h, iban_sum7d, ip_count30d, ip_mean30d, ip_std14d, ip_std30d, ip_std7d, ip_sum14d, ip_sum30d, ip_sum7d, is_international, is_national_iban,is_new_cc_asn, is_new_iban, is_new_iban_cc, is_new_ip, session_count7d, session_std1h, time_since_same_cc_asn, time_since_same_iban, time_since_same_iban_cc, time_since_same_session, time_from_previous_trans_global

Table 7.4: Selected hyperparameters per each Oracle's model in blackbox scenario.

Model	Hyperparameter	Value
XGB	max_depth	6
	learning_rate	0.5
	booster	"gbtree"
	gamma	0.1
	subsample	1
	n_estimators	512
	scale_pos_weight	161.03296703296704
NN	epochs	60
	batch_size	512
	fl_neurons	64
	sl_neurons	16
	activation_funct	"relu"
	dropout_rate	0.1

Table 7.5: Selected hyperparameters per each Oracle's model in graybox scenario.

Model	Hyperparameter	Value
XGB	max_depth	6
	learning_rate	0.5
	booster	"gbtree"
	gamma	0.1
	subsample	1
	n_estimators	512
	scale_pos_weight	161.03296703296704
NN	epochs	80
	batch_size	1024
	fl_neurons	32
	sl_neurons	64
	activation_funct	"relu"
	dropout_rate	0.1

Table 7.6: Final performance evaluation of the FDS Machine Learning (ML) models.

Model	Precision	Recall	F1-Score	FPR
RF	86.06%	67.51%	75.66%	0.11%
LR	15.77%	58.46%	24.83%	3.18%
SVM	15.39%	63.35%	24.77%	3.55%
XGB	82.94%	69.77%	75.78%	0.15%
NN	34.65%	79.70%	48.30%	1.53%
Model	**AUC-ROC**	**AUC-PRC**	**W-MCC**	**Cost-Accuracy**
RF	96.28%	79.53%	71.23%	83.70%
LR	89.75%	16.37%	59.86%	77.64%
SVM			63.38%	79.90%
XGB	96.73%	80.03%	73%	84.81%
NN	94.08%	66.48%	79.59%	89.09%

Table 7.7: Final performance evaluation of the Oracle Machine Learning (ML) models in graybox scenario.

Model	Precision	Recall	F1-Score	FPR
XGB	86.42%	76.38%	81.09%	0.11%
NN	31.69%	90.67%	46.97%	1.75%
Model	**AUC-ROC**	**AUC-PRC**	**W-MCC**	**Cost-Accuracy**
XGB	99.19%	87.91%	78.47%	88.14%
NN	99.06%	81.96%	89.17%	94.46%

Table 7.8: Final performance evaluation of the Oracle Machine Learning (ML) models in blackbox scenario.

Model	Precision	Recall	F1-Score	FPR
XGB	86.42%	80.0%	89.09%	0.13%
NN	31.33%	90.48%	46.54%	1.78%
Model	**AUC-ROC**	**AUC-PRC**	**W-MCC**	**Cost-Accuracy**
XGB	99.16%	89.12%	81.51%	89.94%
NN	98.93%	82.08%	88.96%	94.43%

Algorithm 8.1 Adversarial Transaction Generation

Input: *past_t*: list of known past user's transactions
o: Oracle
max_iters: maximum number of iterations
Output: raw adversarial transaction

$t \leftarrow Transaction()$
$t.iban, t.iban\_cc, t.asn\_cc, t.sessionID, t.confirm\_sms \leftarrow RandomInfo()$
$t.amount \leftarrow SelectAmount(past\_t)$
$t.timestamp \leftarrow SelectTimestamp(past\_t)$
$aggr\_t \leftarrow AggregateTransaction(past\_t, t)$
$std\_aggr\_t \leftarrow StandardizeTransaction(aggr\_t)$
while $iter < max\_iters$ **do**
  $t\_prob \leftarrow PredictTransaction(o, std\_aggr\_t)$
  if $t\_prob < o.threshold$ **then** ▷ transaction classIfied as legitimate by the Oracle
    return t
  end if
  $new\_std\_aggr\_t \leftarrow AdversarialAttackAlgorithm(std\_aggr\_t, o)$
  $new\_amount \leftarrow InverseStandardize(new\_std\_aggr\_t.amount)$
  $tmp\_var \leftarrow new\_std\_aggr\_t.time\_from\_last\_trans\_global$
  $new\_time\_from\_last\_trans\_global \leftarrow InverseStandardize(tmp\_var)$
  $new\_timestamp \leftarrow GetTimestampLastTran(past\_t) + new\_time\_from\_last\_trans\_global$
  $t.amount \leftarrow new\_amount$
  $t.timestamp \leftarrow new\_timestamp$
  $aggr\_t \leftarrow AggregateTransaction(past\_t, t)$
  $std\_aggr\_t \leftarrow StandardizeTransaction(aggr\_t)$
  $iter \leftarrow iter + 1$
end while
return $NULL$ ▷ no adversarial transaction found

8 | Implementation Details

In this chapter, we describe in detail the implementation of our framework.

8.1. Adversarial Transaction Generation

The adversarial transaction generation Algorithm 8.1 works as follow:

1. Figure out which features from the aggregated features set need to be used to reconstruct the raw *Amount* and *Timestamp*, which are the only editable features. The needed features are *Amount* and *time_from_previous_trans_global*. We assume that these features are in Oracle's aggregated features set(see Section 5.5)

2. Craft the first transaction following the decision-based rules as did in Monti[39]

3. test the fraudolent transaction with the Oracle. If it is classified as legitimate, it will be submitted to the FDS, otherwise, we continue with the following steps

4. Exploit one of the adversarial algorithms to modify the fraudolent transaction
5. Since only *Amount* and *Timestamp* can be modified, we first revert the standardization, and then we use the past victim's transactions and *Amount* and *time_from_previous_trans_global* to recover the modIfied *Amount* and *Timestamp*
6. With the new values computed in the previous step, the new raw fraudolent transaction is generated
7. the steps from 3 to 6 are repeated until the fraudolent transaction is classified as legitimate, or a maximum number of iterations is reached

8.2. Adversarial Algorithms Implementation

For each new algorithm that we introduce, we provide a pseudocode and the hyperparameters.

8.2.1. ESPA

To speed up the training phase, Algorithm 8.2 selects the best encoder and the procedure in Algorithm 8.3 is parallelized for each encoder. The encoders are neural networks with a single neuron and a linear activation function. The algorithm generates the initial population randomly and loops G times. In this loop, innerLoop(), an Algorithm 8.3, is executed in parallel. This procedure computes the fitness score of each encoder. The perturbation produced by the encoder is scaled with scalePerturbation(), and the adversarial sample is reconstructed with updateTransaction(). After that, the classifier assigns a fitness score to the encoder based on the adversarial transaction. This procedure is done n times, and the scores are averaged. When the parallel procedure is completed, Algorithm 8.2 selects the best e encoders and generates a new population with generateNewPopulation; mutatePopulation mutates the population with a mutation rate ρ.

Algorithm 8.2 Evolution based Specialized Perturbation Attack (ESPA) encoder selection

Input: a set X of fraud examples, population size P , generation size G, elite members e, batch size n, classifier f(i.e. the Oracle) , feature importance vector υ, mutation rate ρ
Output: an encoder from the last generation, with the highest fitness.

encoders = generateFirstPopulation(P) algorithmiccommentgenerate the first encoders array with size P. Encoders parameters are randomly chosen
for *generation* g=1 to G **do**
    fitness = array(P, 0) ▷ initialize fitnesses array as an array with size P and value 0 foreach element
    fitness = parallell(innerLoop(), encoders, n, f, υ)
    eliteMembers = selectElitePopulation(e, encoders, fitness). ▷ select Elite Encoders according to their fitness score.
    encoders = generateNewPopulation(eliteMembers)
    encoders = mutatePopulation(encoders, ρ)
end for
finalEncoder = selectBestEncoder(encoders, fitness)
return finalEncoder

Algorithm 8.3 Evolution based Specialized Perturbation Attack (ESPA) inner loop that computes the fitness for each encoder

Input: a set X of fraud examples, batch size n, classifier f , feature importance vector υ

for *batch_size* i=0 to n-1 **do**

    x_f = drawFromDistribution($simU(X)$) ▷ draw a fraud
    perturbation = $\upsilon \cdot$ $tanh(f(x_f))$
    perturbation = scalePerturbation(perturbation) ▷ scale the perturbation according to the features ranges.
    perturbedSample = x_f + perturbation
    perturbedSample = updateTransaction(perturbedSample) ▷ restore the consistency among the features
    fitness[i] += np.log(f(perturbedSample))
end for
fitness[i] /= n
return fitness[i]

Parameters

- g: is the number of generations
- p: is the population of encoders for each generation.
- *batch_size*: is the number of different samples we test for each encoder to give a score to the encoder.
- ρ: this is the mutation rate
- e: This number tells how many of the best encoders of the current generation are used to generate the next generation.
- *importance*: this parameter tells what type of importance vector is used, If it is set to "Pearson" it computes the Pearson correlation, If it is set to "custom" it uses a custom importance vector.
- *amount_importance*, *time_importance*, *normalize*: these three parameters are used only If the importance is set to "custom". Since we can only edit two features, we set their importance with *amount_importance* and *time_importance* and with *normalize* we tell If the importance vector needs to be normalized or not.

8.2.2. FGSM

Algorithm 8.4 FGSM algorithm

Input: x, which is the initial transaction, y is the target label

```
gradient = computeGradient(x, y)(1 - 2 * int(targeted))
perturbation = applyNorm(gradient, norm)
perturbation = eps*perturbation
perturbation = scalePerturbation(perturbation)▷ scale the perturbation according to the features ranges.
advTransaction = x + perturbation
advTransaction = updateTransaction(advTransaction)        ▷ restore the consistency among the features
return advTransaction
```

Algorithm 8.4 depicts the implementation of FGSM. First, it computes the gradient of the loss function. If *targeted* is set to true, the gradient changes in sign because the purpose is to minimize the loss function for the intended target y. If *targeted* is false, the gradient maximizes the loss of y. After computing the gradient, it applies the norm. The perturbation is then scaled according to the different value ranges of the features. Finally, the adversarial transaction is reconstructed using updateTransaction.

Parameters

- *targeted*: this boolean parameter, specifies If the attack is targeted or not
- *norm*: with this parameter, which can be "inf", 1 or 2, set the norm that will be applied to the gradient. With "inf" the algorithm is the original version, where the perturbation is multiplied by the sign of the gradient, with 1 or 2 the gradient is normalized by the L1 norm (Manhattan distance) or L2 norm(Euclidean distance).
- *eps*: the amount of the perturbation step

The parameter *targeted* will be set to true, to target the chosen class, *norm* is set to "inf", to test the original FGSM version, and *eps* are empirically set to find the amount of perturbation that achieve best results.

8.2.3. Iterative Method/Project Gradient Descent

When *random_eps* is set to True, Algorithm 8.5 creates a random perturbation and computes *random_eps_step* based on the ratio of *eps* and *eps_step.* For a number equal to *num_random_init*, FGSM runs for *max_iter* iterations. During the first iteration, the adversarial sample is randomly initialized. To avoid unnecessary iterations, after each application of the perturbation step, the changed transaction is reassembled with UPDATETRANSACTION and then evaluated by the Oracle. When the transaction is classified as legitimate, or the maximum number of iterations is achieved, the algorithm terminates.

Parameters

- *targeted*: this parameter says If the attack targets a specific class or not
- *eps*: this is the maximum amount of perturbation applied to the sample
- *eps_step*: *eps_step* is the amount of perturbation added at each iteration
- *random_eps*: with this flag set as True, instead of using *eps* as a max perturbation, the perturbation is drawn from a uniform distribution, and the *eps_step* is changed to maintain the ratio between *eps* and *eps_step*
- *max_iter*: number of max iterations to conduct the attack.
- *num_random_init*: with this parameter, we set how many times the random initialization is done. With $num\_random\_init > 0$, the iterative method is a PGD.

As said above, the targeted parameter is set to True, since we target a specific class(the

Algorithm 8.5 Basic Iterative Method algorithm

Input: x, which is the initial transaction, y is the target label

```
if random_eps then
    eps_step = computeRandomEps(eps_step, eps)
end if
for i in range num_random_init do
    for j in range (max_iter) do
        if j == 0 then
            x_adv = random_init(x)
        end if
        x_adv = FGSM(x_avd, y, eps_step, eps)
        x_adv = updateTransaction(x_adv)        ▷ restore the consistency among the features
        prob = preditctProb(x_adv)              ▷ predict the probability using the Oracle
        if prob < decisionThreshold then
            break
        end if
    end for
end for
return x_adv
```

legitimate one). *eps* and *eps_step* need to be empirically searched, to find the best combination within this domain, the same for *num_random_init* and *random_eps*. *max_iter* must be set to find the right trade-off between time and results.

8.2.4. Saliency Map

The initial phase of the Algorithm 8.6 is to limit the search space to editable features. The first loop computes the saliency map similarly to Algorithm 2.2, but returns only one feature instead of two. The perturbation is scaled based on the feature that is altered, and after recreating the transaction using updateTransaction, the search space is updated, removing the feature that achieved the maximum bound if *theta*>0 or the minimum bound if *theta* < 0. Next, the Oracle evaluates the hostile sample. If it is determined to be legitimate, the algorithm returns the adversarial transaction; otherwise, the loop continues until the search space is empty or the maximum number of iterations is reached. If no satisfactory adversarial transaction is found after the initial loop, the procedure is repeated with the sign of *theta* inverted.

Algorithm 8.6 Saliency Map algorithm

Input: x, which is the initial transaction, y is the target label

searchSpace = computeSearchSpace() ▷ restrict the search space only to the editable features. Search space is an array with the same shape as x, where each element can be 1 or 0. if it is 1 it means that the feature with that index can be used in the procedure
x_adv = x
while (currentPred != y or searchSpace.sum() > 1) and i < max_iter **do**
    feat_ind = computeSaliencyMap(x_adv, y, searchSpace) ▷ compute the saliency map and return the index of the feature that contributes most in increasing the probability of getting y
    x_adv[feat_ind] += scalePerturbation(theta) ▷ perturb the adversarial sample with theta, scaled according to the feature that is perturbed
    x_adv = updateTransaction(x_adv)
    searchSpace = updateSearchSpace(searchSpace) ▷ update the search space, if a feature reach maximum(when theta is >0) or minimum(when theta is <0) value allowed, that feature is removed from search space
    currentPred = preditctProb(x_adv) ▷ predict the probability using the Oracle
end while
if currentPred != y **then**
    searchSpace = computeSearchSpace() ▷ restrict the search space only to the editable features. Search space is an array with the same shape as x, where each element can be 1 or 0. if it is 1 it means that the feature with that index can be used in the procedure
    theta = -theta
    while (currentPred != y or searchSpace.sum() > 1) and i < max_iter **do**
        feat_ind = computeSaliencyMap(x_adv, y, searchSpace) ▷ compute the saliency map and return the index of the feature that contributes most in increasing the probability of getting y
        x_adv[feat_ind] += scalePerturbation(theta) ▷ perturb the adversarial sample with theta, scaled according to the feature that is perturbed
        x_adv = updateTransaction(x_adv)
        searchSpace = updateSearchSpace(searchSpace) ▷ update the search space, if a feature reach maximum(when theta is >0) or minimum(when theta is <0) value allowed, that feature is removed from search space
        currentPred = preditctProb(x_adv)▷ predict the probability using the Oracle
    end while
end if
return x_adv

Parameters

- *theta*: this is the amount of perturbation applied at each iteration, to the selected feature. It can be either negative or positive.
- *max_iter*: the maximum number of iterations to find a sample.

The amount of *theta* must be empirically determined, taking into account *max_iter*. If *theta* is <0, the algorithm attempts to locate the adversarial sample by lowering the selected feature at each iteration. If it does not locate a valid sample, it searches for one by increasing the selected feature with each iteration. *max_iter* should be set to discover a trade-off between outcomes and time spent. This parameter determines the choice of *theta*.

8.2.5. Low Pro Fool

The LowProFool Algorithm 8.7 consists of an initialization phase followed by a main loop. The gradient is computed within the loop for each iteration. To create the adversarial transaction, scale this value by the factor *eta*. The perturbation is then adjusted based on the range of values of the features. If the Oracle classifies the transaction as legitimate, the perturbation is saved. After each iteration, *eta* is lowered. When the loop is completed, the optimal perturbation is used to create the adversarial transaction.

Algorithm 8.7 LowProFool algorithm

Input: x, which is the initial transaction, y is the target label

```
pertubations = [0...0]                          ▷ initialize pertubation array
bestNormLoss = initializeLoss()
bestPertubations = perturbations
sample = x
for i in range n_steps do
    grad = getGradients(sample, perturbation, y)
    perturbations -= eta*grad
    perturbations = scalePerturbations(perturbations)
    x_adv = x + perturbations
    x_adv = updateTransaction(x_adv)
    eta = eta*eta_decay
    prob = preditctProb(x_adv)          ▷ predict the probability using the Oracle
    if prob < decisionThreshold then
        normLoss = computeNormLosses(importance_vec, perturbations)
        if normLoss < bestNormLoss then
            bestNormLoss = normLoss
            bestPertubations = perturbations
        end if
    end if
end for
x_adv = sample + bestPerturbations
x_adv = updateTransaction(x_adv)
return x_adv
```

Parameter

- *n_steps*: *n_steps* is the number of iterations to compute the optimal perturbation to create the adversarial sample
- *importance*: this parameter tells what type of importance vector is used, If it is set to "Pearson" it computes the Pearson correlation, If it is set to "custom" it uses a custom importance vector.
- *amount_importance*, *time_importance*, *normalize*: these three parameters are used only If the importance is set to "custom". Since we can only edit two features, we set their importance with textbfamount_importance and *time_importance* and

with *normalize* we tell If the importance vector needs to be normalized or not.

- *lambd*: this parameter weights the importance vector in the process of computing the perturbation vector.
- *eta*: this parameter sets the rate of updating the perturbation vector.
- *eta_decay*: this is the step by step eta decrease

The number of steps needs to be set to obtain a good result in a reasonable amount of time required. The other parameters need to be set empirically, to best suit the problem in this domain.

8.3. External Libraries

For the implementation, we use the Python[1] programming language, version 3.9.16. The main Python external libraries that we use are:

- numpy[2] (version 1.23.5), for fast operations on complex structures and its mathematical functions;
- pandas[3] (version 1.5.3), for data analysis and manipulation such as loading a CSV file in memory or computing merge operation between two datasets;
- scikit-learn[4] (version 1.1.3), for the ML algorithms implementation;
- matplotlib[5] (version 3.7.0), for visualizing and plotting data;
- xgboost[6] (version 1.7.3), for the XGBoost algorithm implementation;
- tensorflow[7] (version 2.10.0), for the implementation of Neural Networks;
- keras[8] (version 2.10.0), for its simple Application Program Interfaces (APIs) to implement Neural Networks;
- Adversarial Robustness Toolbox[41] (version 1.13.1), library for Machine Learning Security. We adapt the Adversarial Robustness Toolbox (ART) implementation of

[1]https://docs.python.org/release/3.9.16/
[2]https://numpy.org/doc/1.23/
[3]https://pandas.pydata.org/pandas-docs/version/1.5.3/
[4]https://devdocs.io/scikit_learn/
[5]https://matplotlib.org/3.7.0/contents.html
[6]https://xgboost.readthedocs.io/en/stable/
[7]https://www.tensorflow.org/versions/r2.10/api_docs
[8]https://keras.io/api/

the adversarial algorithms attacks FGSM, Iterative, Saliency Map, and LowProFool to the fraud detection context.

9 | Experimental Validation

In this chapter, we present the results of the experiment of this work. In the first part we discuss the metrics we will use to evaluate the results. Then, we discuss the goals of the experiments. After that, we present the experimental setup. Finally, we discuss the results. We conduct Poisoning Attack simulations instead of Evasion Attack because the Poisoning Attack already includes the Evasion. With our framework, we can also run experiments to test the effectiveness of Evasion Attacks by ending the run as soon as we reach the first update time. With reference to the Section 5.6, we stop at the *evasion_phase*.

9.1. Metrics

First of all, we introduce some terms used in the definition of our evaluation metrics:

- $F_{generated}$: set of all the adversarial transactions generated by the attacker;
- $F_{filtered}$: set of the adversarial transactions that are classified as legitimate by the attacker's Oracle $F_{filtered} \subseteq F_{generated}$;
- $F_{accepted}$: set of the filtered adversarial transactions that are misclassified as legitimate by the targeted FDS $F_{accepted} \subseteq F_{filtered} \subseteq F_{generated}$;
- $F_{rejected}$: set of the filtered adversarial transactions that are correctly classified as fraudulent by the targeted FDS $F_{rejected} \subseteq F_{filtered} \subseteq F_{generated}$;
- D: set of all the transactions, where $F_{accepted} \cup F_{rejected} \subseteq F_{generated}$, respectively with the wrong (i.e., legitimate) and correct (i.e., fraud) labels;
- V: set of victims of the poisoning attack;
- W: set of weeks of the attack simulation, where $W = \{0...7\}$;
- A_f: amount of fraud $f \in F_{generated}$;
- $A_{v,w}$: amount of money stolen from victim $v \in V$ in week $w \in W$;

- ΔT_f: time difference between the time of execution of fraud $f \in F_{generated}$ and the start of the attack.

The metrics that we use to evaluate our work are the following:

- **Total Money Stolen**: the total amount of money stolen by the attacker from all victims during the attack:

$$\text{Total Money Stolen} = \sum_{f \in F_{accepted}} A_f \tag{9.1}$$

- **Injection Rate**: the ratio between the number of crafted frauds classified as legitimate by the attacker's Oracle and the number of all the generated frauds:

$$\text{Injection Rate} = \frac{|F_{filtered}|}{|F_{generated}|} \tag{9.2}$$

- **Evasion Rate**: ratio between the number of filtered frauds that evade the FDS and the total number of injected frauds:

$$\text{Evasion Rate} = \frac{|F_{accepted}|}{|F_{filtered}|} \tag{9.3}$$

- **Time Before Detection**: metric that represents the average time in days before an adversarial transaction is detected by the FDS:

$$\text{Time Before Detection} = \frac{\sum_{f \in F_{rejected}} \Delta T_f}{|F_{rejected}|} \tag{9.4}$$

- **Attack Detection Rate**: ratio between the number of victims protected by the FDS and the total number of victims:

$$\text{Attack Detection Rate} : \frac{|F_{rejected}|}{|V|} \tag{9.5}$$

- **Poisoning Rate**: the ratio between the number of filtered frauds injected in the banking dataset and the total number of transactions:

$$\text{Poisoning Rate} = \frac{|F_{accepted}|}{|D|} \tag{9.6}$$

9.2. Goals

The goals of our experiments are:

(G1) Study the efficacy of the adapted adversarial attack algorithms (FGSM, BIM, ESPA, Saliency Map and Low Pro Fool) in bypassing FDSs with fraudolent transactions and steal money;

(G2) Analyse the performance of our approach with respect to Monti's approach [39] and Romeo's adapted algorithms [48]against FDSs;

9.3. Experimental Setup

We simulate a poisoning attack against different FDSs that employs different models (XGBoost, Random Forest, Logistic Regression, Neural Network, Support Vector Machine). The FDSs use *augmented_2014_15* dataset while in the blackbox and graybox scenarios the Oracle uses *augmented_2012_13* with up to one month and half past transactions of the victims according to the threat model(Chapter 3), in whitebox scenario, the Oracle's dataset is the same of the FDS. The simulation starts on 05/01/2015, and the update policy is *bi-weekly*. We run a simulation for every combination of algorithm, model, strategy and scenario. In the whitebox scenario, not all the algorithms can be run against every FDS. For gradient-based attacks(i.e., FGSM, BIM, Saliency Map, and Low Pro Fool), the Oracle needs to be a gradient-based model, so we exclude simulations against Random Forest and XGBoost. In detail, we run 45 simulations each for Monti's approach, Boundary, Hop Skip Jump, Zoo, and ESPA. For each gradient-based attack, we run 156 simulations, 39 for each one. We also run simulations against Active Learning models, for more details, see Appendix C. Each simulation is repeated three times to average the results.

9.4. Algorithms Hyperparameters

In Table 9.1 are listed the hyperparameters of the adversarial algorithms. These parameters are obtained empirically. The selection can be done also using a random approach.

Table 9.1: Tested hyperparameters per each adversarial attack algorithm.

Adversarial Algorithm	Hyperparameter	Value
Boundary	init_size	25
	targeted	True
	sample_size	5
	delta	1
	epsilon	0.1
	max_iter	10
HopSkipJump	init_size	25
	targeted	True
	sample_size	5
	delta	1
	epsilon	0.1
	max_iter	10
ZOO	confidence	0.0
	targeted	True
	learning_rate	0.01
	variable_h	0.1
	binary_search_steps	1
	abort_early	True
	max_iter	25
ESPA	g	3
	p	100
	ρ	0.1
	batch_size	5
	e	10
	time_importance	0.1
	amount_importance	1
	normalize	False
FGSM	targeted	True
	num_random_init	0
	eps	0.65
Iterative	targeted	True
	random_eps	False
	num_random_init	0
	max_iter	50
	eps_step	0.01
	eps	1
Saliency Map	theta	-0.025
	max_iter	50
LowProFool	n_steps	25
	lambd	1.35
	eta	0.5
	eta_decay	0.95
	importance	custom
	time_importance	1
	amount_importance	1
	normalize	False

9.5. Blackbox Experiment

In the blackbox scenario, as mentioned in Chapter 3, the attacker knows prior victims' transactions only up to 45 days before the attack start date. In this case, the attacker sets the *decision_threshold* = 0.35. Table 9.2 shows that FGSM and Iterative are the worst algorithms against all FDS models in each strategy, except for Neural Network with medium strategy, where LowProFool is the worst. The best algorithms (according to the attacker) are always decision-based. In Figure 9.1, we can see the poisoning rates

in blackbox scenario expressed in $percentage * 10^4$. Poisoning rates against XGBoost, Random Forest, and Logistic Regression are consistent across all three strategies, with a split between decision-based and gradient-based attacks. The values of these rates decrease when the attacker takes a more aggressive strategy. This is especially noticeable with decision-based attacks. The trend of poisoning rates in SVM looks similar to prior models, with decreasing values as we go towards a greedy strategy. The difference is more visible because, when is used a greedy strategy, the rates of decision-based and gradient-based attacks become closer. Decision-based attacks continue to yield the best results when used against Neural Networks, although the difference between the two classes is not as great as it was with previous models. Moreover, the curves, after initial growth, reach a plateau around the fourth-fifth week. Notice that NN is the model with the smallest poisoning rates. In contrast to previous models, there is no decline in rates when switching strategies. We can explain this behaviour because the Oracle used by gradient-based attacks is a Neural Network. This model is more restrictive with respect to XGBoost used by the other class of attacks. If we look at the FPR(False Positive Rate) in Table 7.8, we can see that NN has a FPR more than 13 times higher than XGB. This characteristic, combined with the low decision threshold, has a relevant impact on the number of transactions filtered and executed against FDS.

If we look at the detection metrics in Figure 9.2, we can see that the gradient-based attacks have better performances with the highest evasion rates, the lowest detection rates, and the highest Times before detection(NOTE: 56 days is the span of the attack simulation, where there are 56 days in time before detection means that the attack, at least in one of the three tests, was not detected). Against each FDS except Neural Network, the Time before Detection decreases as we go from conservative to medium strategy and from medium to greedy. Now we discuss in detail the results for every FDS.

9.5.1. XGBoost

In terms of money stolen, ESPA gets the best outcomes with a conservative and medium strategy. With the first strategy, 7% more money was stolen with respect to the Boundary attack, which is the second-best attack. FGSM achieves the poorest outcome, stealing 49551€. The Saliency Map outperforms the other gradient-based attacks, with a performance of 835137€. The difference between ESPA and ZOO, the second-best algorithm, is now 25% when using the medium strategy. Every decision-based attack operates effectively. Even though it is just 85% of the worst attack of the other class, HopSkipJump, LowProFool also steals a significant amount of money, 1396386€. The Iterative method yields the worst results. The Monti algorithm yields the greatest results when using a

Table 9.2: Total Money Stolen in € in a blackbox scenario. For each ML model we highlight in red (green) the worst (best) result from the defender's standpoint.

Average Total Stolen						
Blackbox						
	Attack Method	**XGB**	**RF**	**NN**	**LR**	**SVM**
Conservative	**Monti**	2081206.48	3012053.81	139323.77	731440.35	587283.3
	Boundary	2140471.04	2702642.02	127883.4	558454.7	918581.95
	HopSkipJump	1848274.23	2168036.56	120504.03	569256.91	733593.46
	ZOO	748958.56	2987258.67	118462.25	570567.94	870621.23
	ESPA	2288830.38	2286030.16	210142.38	629341.42	840682.12
	FGSM	49551.98	226382.22	29154.31	12200.88	35436.73
	Iterative	125916.88	69655.73	19031.73	33677.53	198713.87
	Saliency Map	835137.71	1384647.68	102718.29	95973.09	260074.27
	LowProFool	290426.12	169899.2	51288.23	141332.38	234994.72
Medium	**Monti**	2184995.7	3003238.9	217167.25	676707.23	837752.28
	Boundary	2161524.47	3607307.9	148946.48	602613.24	672033.4
	HopSkipJump	1629964.77	1953876.52	148327.78	657983.05	538771.34
	ZOO	2265230.21	2598275.78	233433.87	674801.43	543000.36
	ESPA	2826074.72	2521943.15	118087.49	490820.58	532550.4
	FGSM	150289.34	276918.3	50388.49	63311.98	38086.15
	Iterative	52567.88	142751.13	48863.4	23622.38	25777.03
	Saliency Map	274194.72	459062.69	131453.28	100444.61	234854.85
	LowProFool	1396386.87	467262.75	45895.15	141286.88	189081.17
Greedy	**Monti**	3812069.08	3767059.13	158958.74	660054.22	410477.67
	Boundary	2445220.84	2864780.31	157182.39	645165.54	680410.01
	HopSkipJump	2499656.21	2959326.94	95223.66	739739.93	701274.9
	ZOO	2414880.77	3300384.2	199854.88	593738.02	722905.36
	ESPA	2898026.8	2548912.23	92386.75	426754.53	626193.92
	FGSM	342555.86	234414.43	22050.88	46086.99	17124
	Iterative	157130.94	620651.97	15739.34	88867.71	40451.42
	Saliency Map	1071231.43	818623.85	130539.99	141882.64	307997.81
	LowProFool	846282.99	521703.41	59620.4	179233.04	265625.08

greedy strategy, and ESPA comes in second. FGSM, Iterative, Saliency Map, and LowProFool yield the best results when examining the detection metrics. They are practically never detected, and their evasion rate is always greater than 0.9. With all the strategies, ESPA had the best injection rate.

9.5.2. Random Forest

The results of Random Forest are very similar to those of XGBoost. Every algorithm does better than against XGBoost except LowProFool, which, with every strategy, steals

less money, and Iterative, with a conservative strategy. The best algorithms are Monti and Boundary. Monti excels with a conservative and greedy strategy, while Boundary excels with a medium strategy. Iterative achieves the worst results with a conservative and medium strategy, while FGSM is the worst with a greedy strategy. The better results, with respect to XGBoost, are confirmed by the detection metrics. Gradient-based attacks have an Evasion Rate of 1, hence they are never detected. Also, decision-based attacks have a higher detection rate. ESPA confirms the best Injection Rate.

9.5.3. Neural Network

The Neural Network results differ significantly from the previous ones; the quantity of money stolen is much smaller. In fact, the most money stolen is 210142€ by ESPA with a conservative approach, 233433€ by ZOO with a medium strategy, and 199854€ by ZOO with a conservative strategy. These quantities are more than ten times lower than those obtained from prior FDSs. FGSM, LowProFool, and Iterative produce the worst results with their respective strategies. The interesting point is that the difference between algorithms that perform well and those that do not perform well is substantially less. On average, the best algorithm steals 8.3 times more than the worst algorithm; for XGBoost and Random Forest, the ratios are 41.4 and 28.19, respectively. The average amount stolen from all attacks reduces when we move from a conservative to a greedy strategy, from 102056.48€ to 126951.46€ and 103506.33€, respectively. The Saliency Map has the best Evasion Rate considering all the strategies.

9.5.4. Logistic Regression

Against Logistic Regression, the best and worst algorithms are Monti and FGSM, Monti and Iterative, and HopSkipJump and FGSM, with conservative, medium, and greedy strategies. On average, across the three strategies, the best result is 45 times the worst. LR suffers more than Neural Network, however in comparison to Random Forest and XGBoost, the quantities stolen are 3 to 5 times smaller. Evasion rates confirm the trend observed with prior FDSs, with gradient-based attacks achieving superior performance.

9.5.5. SVM

With a conservative strategy, Boundary outperforms SVM with 918581€ stolen. The worst algorithm is FGSM, which steals 35436€. Monti has the highest outcomes with a medium strategy, earning 837752€, while Iterative has the poorest with 25777€. Finally, with a greedy strategy, ZOO wins with 722905€ stolen, while FGSM only steals 17124€.

The average ratio between the best and worst results across the three strategies is 33.5. As with Neural Networks, the average amount stolen from all attacks reduces as we progress from a conservative strategy (519997.96€) to a medium and greedy strategy (401322.99€ and 419162€). The evasion rates are significantly more uniform.

9.6. Graybox Experiment

In the Graybox scenario, the attacker has access to the feature set of the target model and sets the *decision_threshold* = 0.425. Looking at the Oracle performances in this scenario in Table 7.7, we can see that they are similar to the performances in the black-box scenario. Table 9.3 shows the total amount of stolen money. We can see that the difference between the performances of decision-based attacks and gradient-based attacks is smaller, particularly against LR, where the algorithms perform similarly, and against NN, where gradient-based attacks outperform decision-based attacks. Figure 9.3 illustrates the poisoning rates. We can see a trend similar to the blackbox example, with a few changes. Against XGBoost and Random Forest, decision-based attacks outperform gradient-based attacks, but the latter improve their performances with poisoning rates that reach 400%(e4), reducing the gap. The same behaviour occurs against SVM, with decision-based attacks that partially reduce their outcomes, and with the tiny gradient-based attacks improvement. Against NN and LR, there are the most significant differences compared with the blackbox scenario. Poisoning rates against NN keep the same shape seen in the blackbox scenario, although no algorithms have a poisoning rate of less than 100%(e4). FGSM, Saliency Map, and LowProFool have surpassed decision-based algorithms with conservative and medium strategies. With LR, there is no longer a gap between the two types of algorithms. This is because, although decision-based algorithms produced similar results, gradient-based algorithms showed a significant improvement. In the blackbox scenario, the poisoning rates of FGSM, Iterative, Saliency Map, and LowProFool are always in the range of 0-200%(e4). In the current situation, they are over 400%(e4), except for LowProFool and Iterative, which achieve a poisoning rate of roughly 350%(e4).

The detection metrics in Figure 9.4 confirm the trend seen in the previous scenario, with gradient-based attacks that are the most evasive, with the highest evasion rates and the highest time-before-detection. Now we discuss in detail the results for every FDS.

Table 9.3: Total Money Stolen in € in a graybox scenario. For each ML model, we highlight in red (green) the worst (best) result from the defender's standpoint.

Average Total Stolen						
Graybox						
	Attack Method	**XGB**	**RF**	**NN**	**LR**	**SVM**
Conservative	**Monti**	2052867.03	2496579.91	81171.23	589362.28	506775.03
	Boundary	3146751.71	1949687.01	119875.11	612047.26	731164.59
	HopSkipJump	1939163.46	3247700.34	157125.67	533466.02	502163.84
	ZOO	1232656.08	2493057.53	118172.53	537455.24	461878.82
	ESPA	1753581.95	1931721.27	166525.98	543705.25	692817.96
	FGSM	654003.94	233940.74	258101.33	415827.59	292206.88
	Iterative	238036.74	163099.56	123621.86	367852.72	122187.06
	Saliency Map	402400.67	444488.58	338797.3	428644.66	190524.41
	LowProFool	321746.86	737098.77	203127.19	657005.06	391665.83
Medium	**Monti**	2295163.95	3049471.51	86019.08	551380.46	663207.03
	Boundary	5331152.91	2726002.47	120934.5	542154.62	795372.75
	HopSkipJump	3340155.41	2765772.98	109676.94	578338.83	799269.5
	ZOO	1867823.31	3438436.49	156663.94	653542.69	714953.55
	ESPA	2388347.76	2650221.71	156632.34	593042.19	663684.26
	FGSM	232889.73	486918.32	317593.06	424768.5	313858.94
	Iterative	369176.26	92018.28	234984.23	361589.11	80510.67
	Saliency Map	950036.77	829993.39	341770.12	465374.16	224439.51
	LowProFool	820434.99	463237.39	248510.52	506401.9	561218.11
Greedy	**Monti**	2772912.52	3504860.5	271916.76	631618.46	633208.65
	Boundary	2380964.43	2789257.85	178513.71	522402.54	746302.36
	HopSkipJump	2313522.29	3475682.77	186303.03	561052.97	375502.87
	ZOO	2661879.91	4422676.19	145296.46	560964.92	443951.02
	ESPA	2181738.93	3000601.71	90625.23	579438.82	589493.02
	FGSM	1104226.14	314553.7	193289.53	456234.42	386634.74
	Iterative	541561.8	364985.7	438107.1	395362.96	100390.57
	Saliency Map	1255217.27	625678.49	415924.11	339410.69	245692.96
	LowProFool	1357629.1	592464.7	199673.98	513763.43	134320.35

9.6.1. XGBoost

Boundary Attack achieves the best results with conservative and medium strategies, whereas Monti excels with greedy strategy. Iterative yields the worst results with a conservative and greedy strategy, whereas FGSM yields the worst with a medium strategy. The average best-to-worst result ratio is now 13.74, which is three times smaller than it was in the blackbox scenario. The relative standard deviation of stolen money is 63.7%. This means that the amount distribution is less dispersed than in the blackbox scenario. The improvement in results is especially significant for Iterative and FGSM, which were

the poorest algorithms in the previous situation. Evasion rates are often higher than in the blackbox scenario, with gradient-based attacks performing particularly well.

9.6.2. Random Forest

The best results against this model are obtained by HopSkipJump with a conservative strategy (3247700€ stolen) and ZOO with a medium and greedy strategy (3438436€, 4422676€). Iterative obtains the worst outcomes with the first two strategies, while FGSM with the third. Iterative with medium approach steals the smallest quantity of money (92018€). In the blackbox scenario, the ratio between the average best and worst result was 28.19, whereas it is 23.78 now. This outcome is driven by the poor performance of the Iterative with Medium approach, which has a ratio of 37.3. The relative standard deviation is the same as in the black box situation. Poisoning rates follow the same pattern as in the black box scenario. The detection metrics have improved, and gradient-based attacks with an evasion rate of 1 are not detected.

9.6.3. Neural Network

This model remains the least vulnerable to the attacks. Gradient-based algorithms produce the best outcomes in this situation, Saliency Map with the first two tactics and Iterative with the last. Monti and ESPA had the worst outcomes. The ratio between the best and worst results is now 4.32, which is half that in the blackbox scenario. The Evasion rates for decision-based attacks are similar to the blackbox scenario, with gradient-based attacks showing the greatest improvement.

9.6.4. Logistic Regression

Logistic Regression is the model against which the results are more similar among the algorithms. The relative standard deviation is 17%, meaning that the results are concentrated around the mean. The ratio between the average best and worst results is only 1.81. The difference from the prior scenario is explained by the significant improvement in the performance of gradient-based methods. Decision-based attacks steal the same amount of money, even though ZOO and Monti obtain the highest success with medium and greedy strategies. Like with Neural Network, evasion rates for decision-based algorithms remain similar, whereas gradient-based attacks show the most significant improvement.

9.6.5. SVM

For what concerns SVM, the Boundary attack steals the largest amount of money with conservative and greedy strategies, and HopSkipJump excels with the medium strategy. Iterative is always the worst algorithm. In the blackbox scenario, the average stolen from all algorithms is smaller with the conservative and greedy strategies, and higher with the medium. The growth in gradient-based attacks, which all improve their results, is offset by a drop in decision-based performance. Gradient-based algorithms enhance evasion rates in Neural Network and Logistic Regression.

9.7. Whitebox Experiment

The whitebox scenario is the most favourable for the attacker (and the worst for the defender). In this case, the attacker knows everything about the FDS target, including the model, feature set, hyperparameters, and the whole dataset used to train the FDS. Even if it is unrealistic, simulating an attack under these conditions is important since it demonstrates what would happen in the worst-case situation. With full knowledge, the attacker's Oracle is the same FDS that they target; thus, there are no simulations with gradient-based attacks against XGBoost and Random Forest because the gradient cannot be computed with these two models. Table 9.4 shows the total amount of money stolen. Each algorithm produces a good outcome. Against NN, LR, and SVM, where we have all the attacks, we can see that the differences are far less significant than in the previous scenarios. Regarding the poisoning rates, shown in Figure 9.5, all algorithms enhance their outcomes. This is especially noticeable for NN, where the range of poisoning rates passes from a lower end of 100%(e4) and no algorithms beyond 250%(e4) in graybox scenario, to a lower end of 300%(e4).

Now we discuss the results for each model.

9.7.1. XGBoost

Against XGBoost, we have only decision-based algorithms,i.e., Monti, Boundary, HopSkipJump, ESPA. Boundary obtains the best performances with conservative and greedy strategies, Monti with medium strategy. ESPA is always the worst algorithm here. The ratio between best and worst results is 1.85, 1.44, and 1.85 for the respective strategy.

Table 9.4: Total Money Stolen in € in a whitebox scenario. For each ML model, we highlight in red (green) the worst (best) result from the defender's standpoint.

Average Total Stolen						
Whitebox						
	Attack Method	**XGB**	**RF**	**NN**	**LR**	**SVM**
Conservative	**Monti**	3803685	3822391.03	632395.35	2102740.6	1304522.23
	Boundary	4976029.02	3602903.47	2647572.21	3285230.12	1021521.1
	HopSkipJump	3363850.48	3380687.96	1681895.12	2490673.27	702081.67
	ZOO	3380634.46	3309209.68	1318637.09	3177600.25	890822.53
	ESPA	2679514.32	2795766.92	623205.94	1913103.77	895512.86
	FGSM	NaN	NaN	2180335.53	2875326.12	918850.68
	Iterative	NaN	NaN	1137096.53	2803023.52	1223952.68
	Saliency Map	NaN	NaN	1161834.21	2594363.88	1354501.17
	LowProFool	NaN	NaN	639573.54	2565094.89	757756.11
Medium	**Monti**	4131103.2	2532505.07	682702.5	1998310.53	1321727.46
	Boundary	4053858.13	5387053.68	2841459.93	2979620.83	593870.46
	HopSkipJump	4025189.58	3852592.01	1337075.92	1911122.74	826849.89
	ZOO	3791039.03	3518560.14	1208144.4	2363381.7	1050754.94
	ESPA	2850838.16	3157158.4	961510.72	1934535.21	967496.68
	FGSM	NaN	NaN	596909.1	2391879.93	1029678.86
	Iterative	NaN	NaN	864106.28	2252110.85	627759.4
	Saliency Map	NaN	NaN	822234.51	1944812.43	904148.41
	LowProFool	NaN	NaN	1159773.94	2073511.98	871177.07
Greedy	**Monti**	3939759.25	3522945.99	1354161.6	1899675.57	1406408.44
	Boundary	6256906.55	4486587.36	3540390.8	2734219.3	645797.44
	HopSkipJump	4432491.63	4822413.19	1917646.98	2259522.18	895729.31
	ZOO	5429493.12	3998836.93	1398593	2825262.42	728652.5
	ESPA	3372949.18	4333485.85	2206899.2	1638749.67	721528.09
	FGSM	NaN	NaN	1128600.34	2393514.43	914140.96
	Iterative	NaN	NaN	820699.87	2253159.51	1264997.29
	Saliency Map	NaN	NaN	860519.14	1954623.9	971314.42
	LowProFool	NaN	NaN	706842.71	2167825.41	1290206.63

9.7.2. Random Forest

Random Forest shares the same considerations as XGBoost. We can test only decision-based algorithms. Here, with a conservative strategy, Monti is the best algorithm and ESPA the worst. With medium and greedy, Monti becomes the worst and Boundary becomes the best.

9.7.3. Neural Network

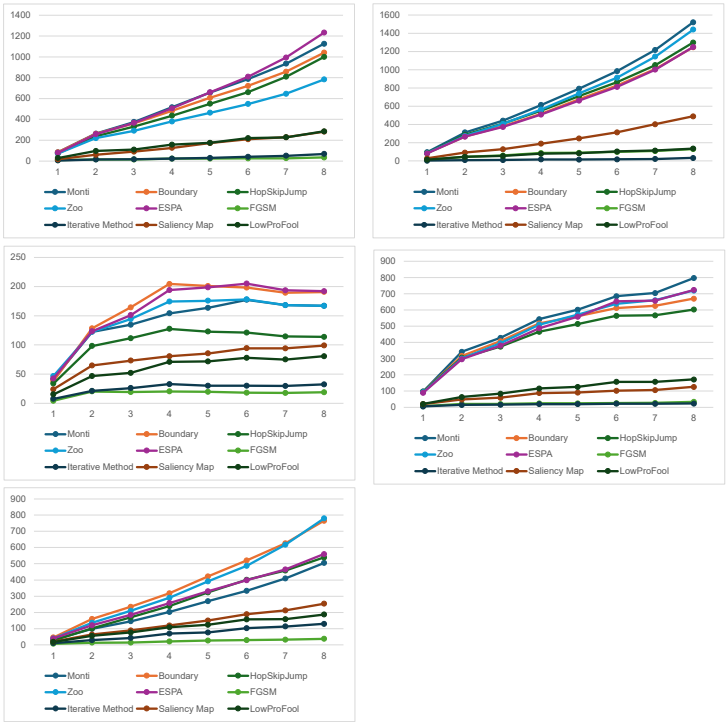
Against Neural Networks, the Boundary Attack always results in the most stolen money. Monti, FGSM, and LowProFool produced the worst outcomes. The most significant result is that the quantities stolen are up to ten times more than those stolen in prior scenarios, with the Boundary attack having an average amount stolen across all strategies that is 21 times greater than the average amount in the graybox scenario.

9.7.4. Logistic Regression

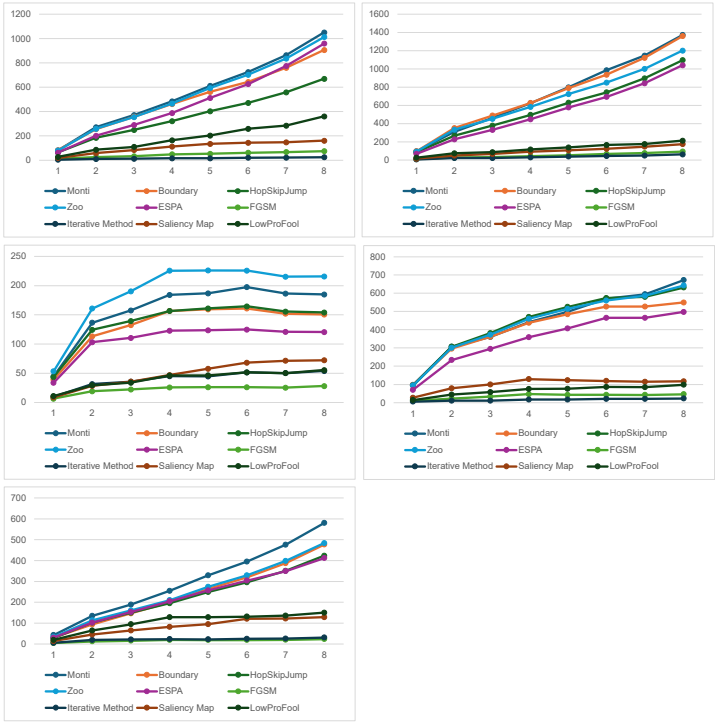
Against Logistic Regression, the performances improve for every algorithm. The total amount stolen distribution continues to be one of the most uniform, with a ratio between the average best and the average worst result of 1.66 and a relative standard deviation of 15.6%. Boundary obtains the best results with conservative and medium strategies, ZOO steals the largest amount with a greedy strategy. The worst results are obtained by ESPA with a conservative and greedy strategy and HopSkipJump with a medium strategy.

9.7.5. SVM

Finally, we analyze the algorithms' performances against SVM. With a relative standard deviation of 23.88%, excluding XGBoost and Random Forest, where we have only decision-based algorithms, SVM has the second most uniform stolen amount distribution. The ratio between the average best and worst result is 2.11. With medium and greedy strategies, Monti steals the largest amount of money and Boundary the smallest. With a Conservative strategy, the best result is obtained by the Saliency Map and the worst by HopSkipJump.

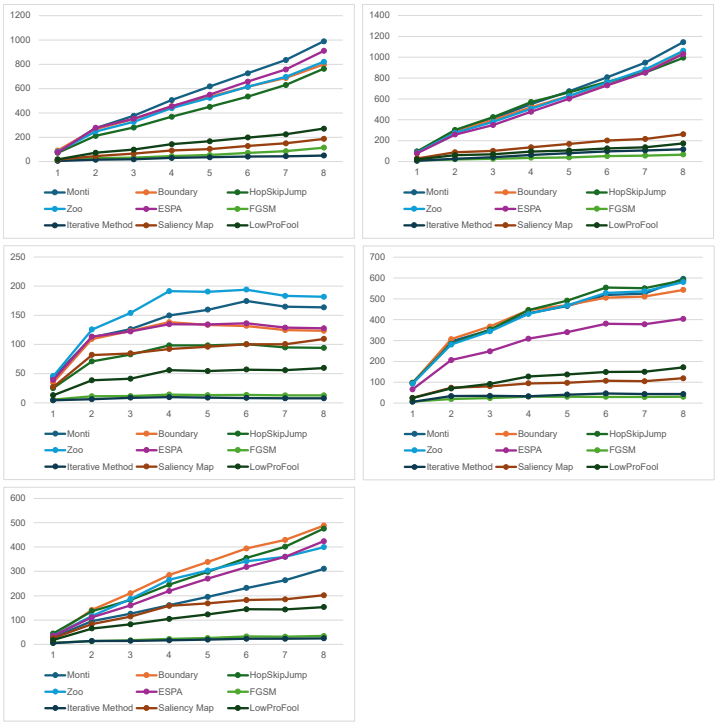


(a) Poisoning rate after each week in blackbox scenario and with a conservative strategy.



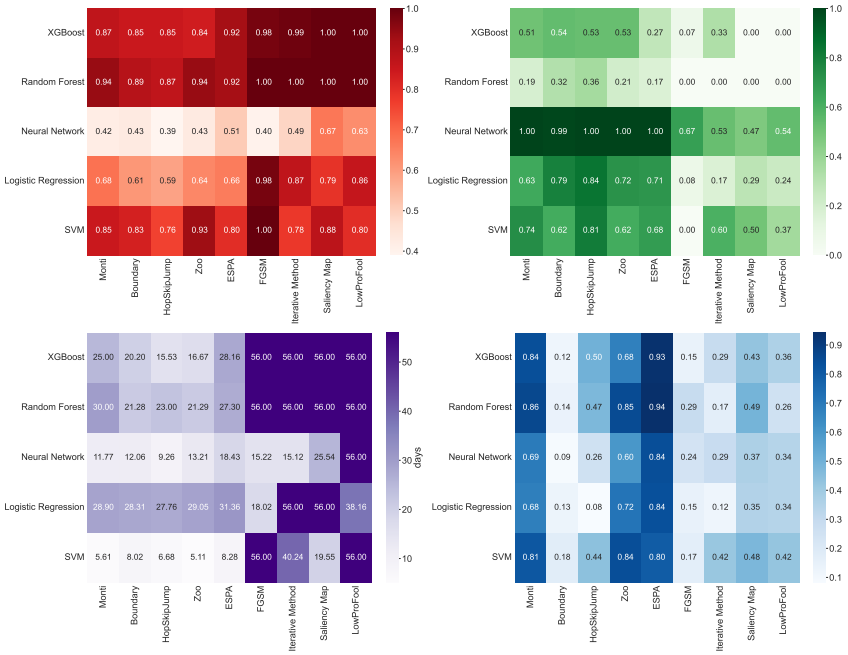
(b) Poisoning rate after each week in blackbox scenario and with a medium strategy.

Figure 9.1: Poisoning rates in blackbox scenario for each strategy and each model. From left to right: XGB, RF, NN, LR, SVM.

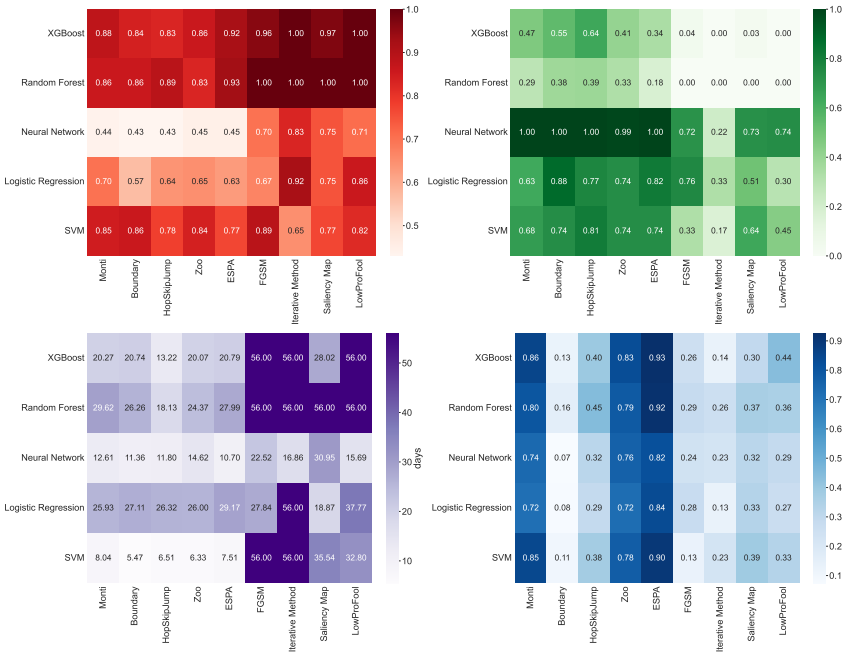


(c) Poisoning rate after each week in blackbox scenario and with a greedy strategy.

Figure 9.1: Poisoning rates in blackbox scenario for each strategy and each model. From left to right: XGB, RF, NN, LR, SVM.

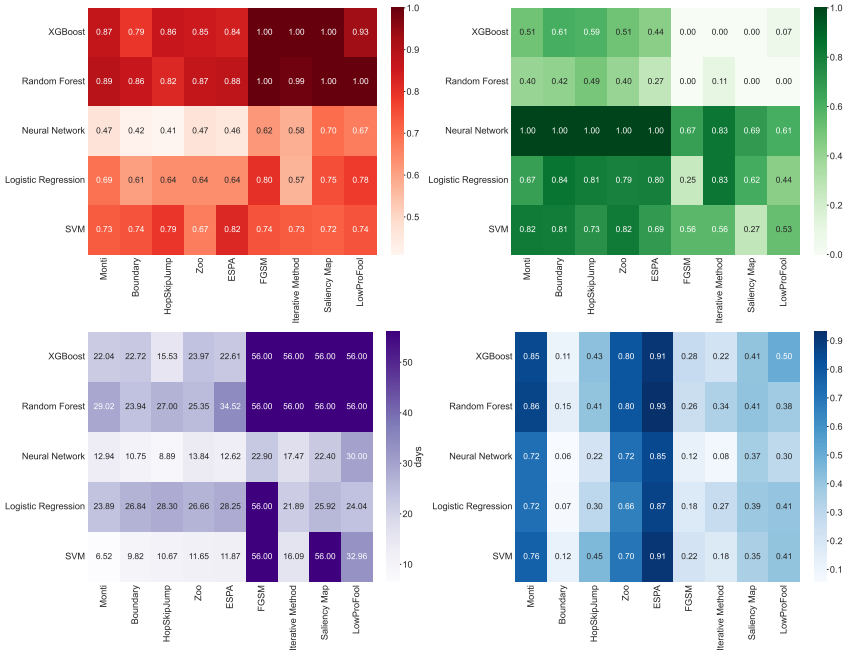


(a) Detection metrics in blackbox scenario and with a conservative strategy.



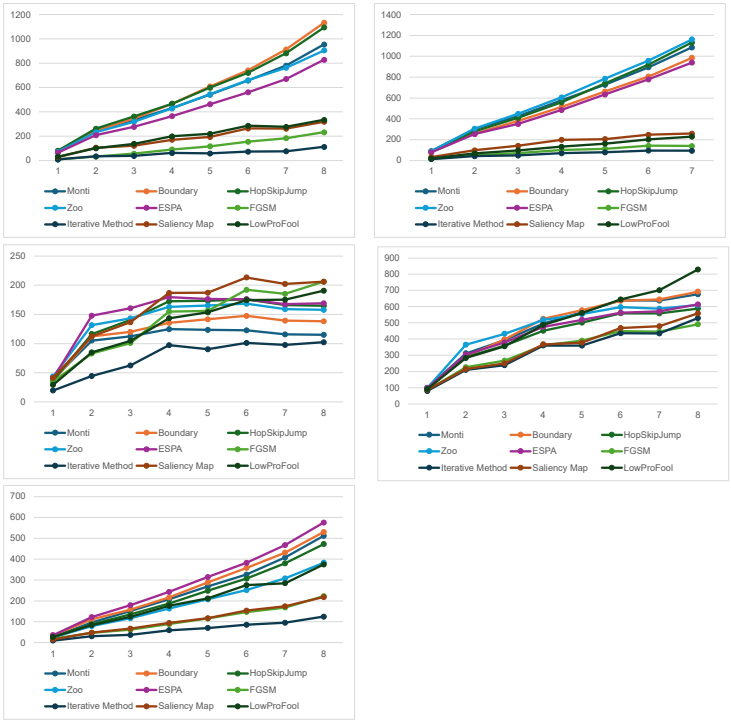
(b) Detection metrics in blackbox scenario and with a medium strategy.

Figure 9.2: Detection metrics in blackbox scenario. From left to right: Evasion Rate, Detection Rate, Time Before Detection, Injection Rate.

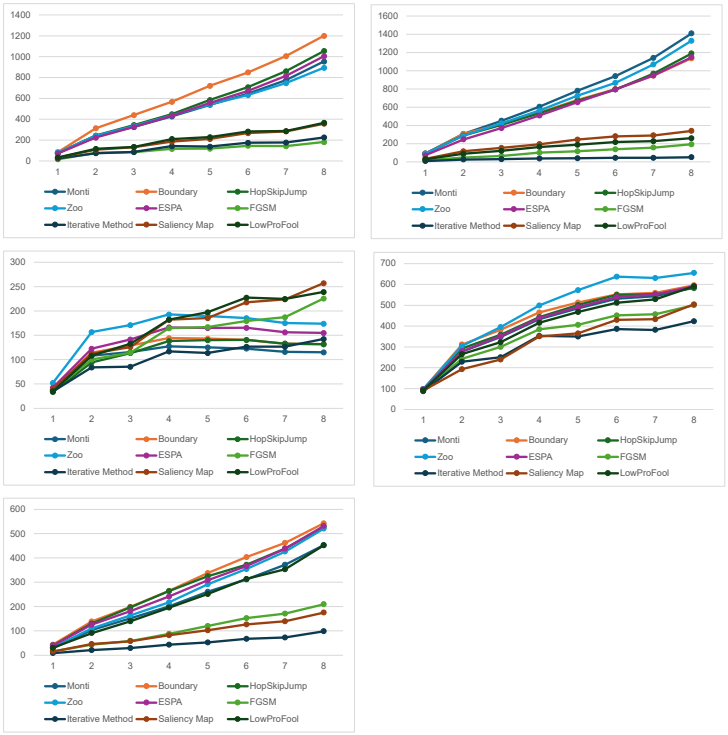


(c) Detection metrics in blackbox scenario and with a greedy strategy.

Figure 9.2: Detection metrics in blackbox scenario. From left to right: Evasion Rate, Detection Rate, Time Before Detection, Injection Rate.

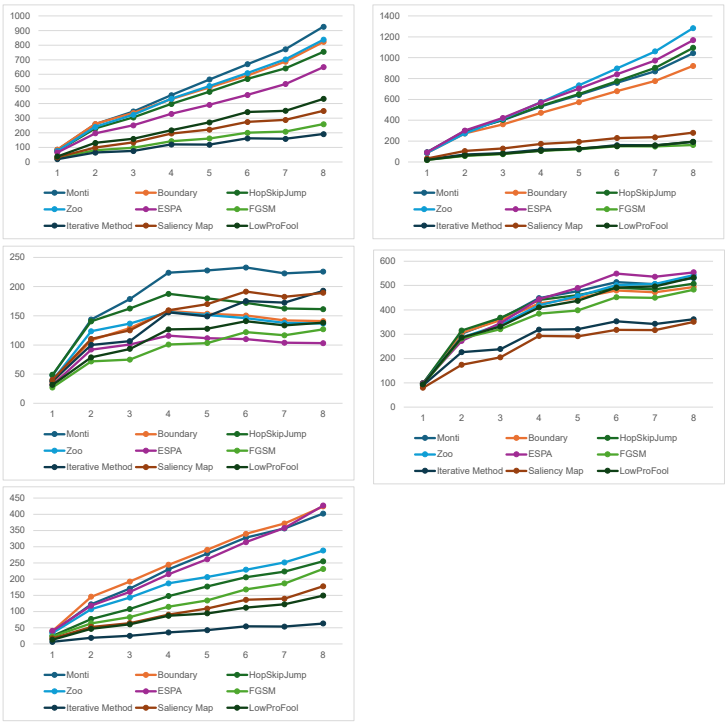


(a) Poisoning rate after each week in graybox scenario and with a conservative strategy.



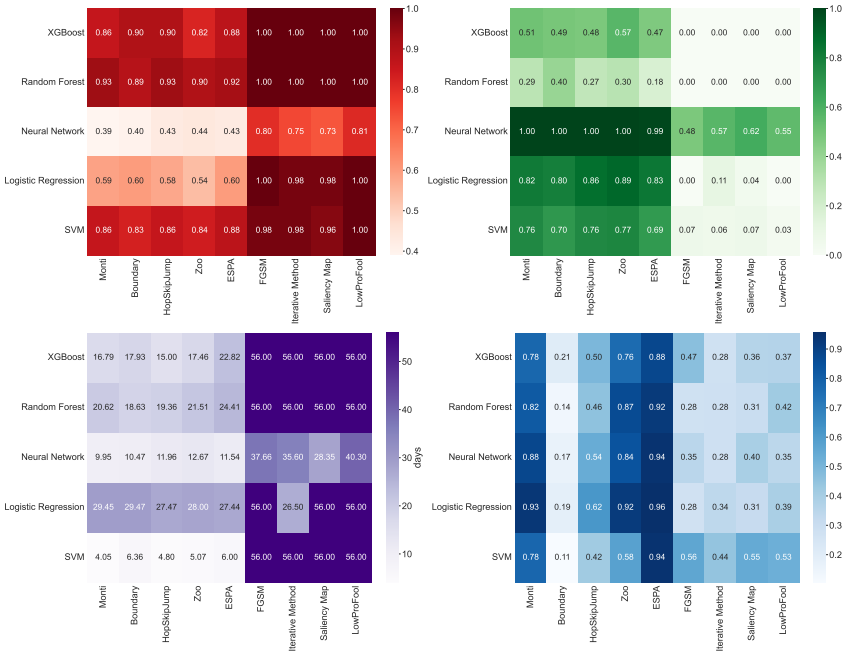
(b) Poisoning rate after each week in graybox scenario and with a medium strategy.

Figure 9.3: Poisoning rates in graybox scenario for each strategy and each model. From left to right: XGB, RF, NN, LR, SVM.

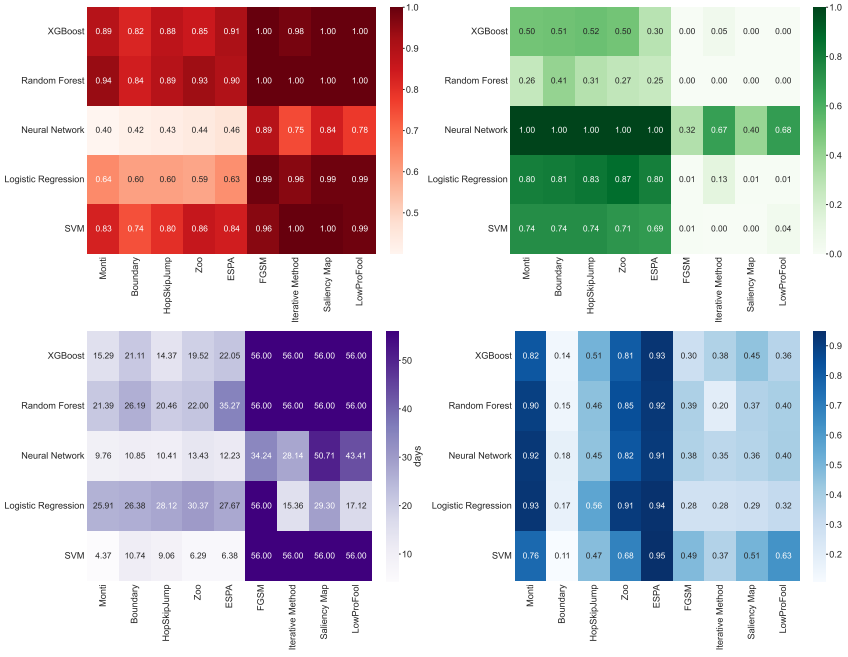


(c) Poisoning rate after each week in graybox scenario and with a greedy strategy.

Figure 9.3: Poisoning rates in graybox scenario for each strategy and each model. From left to right: XGB, RF, NN, LR, SVM.

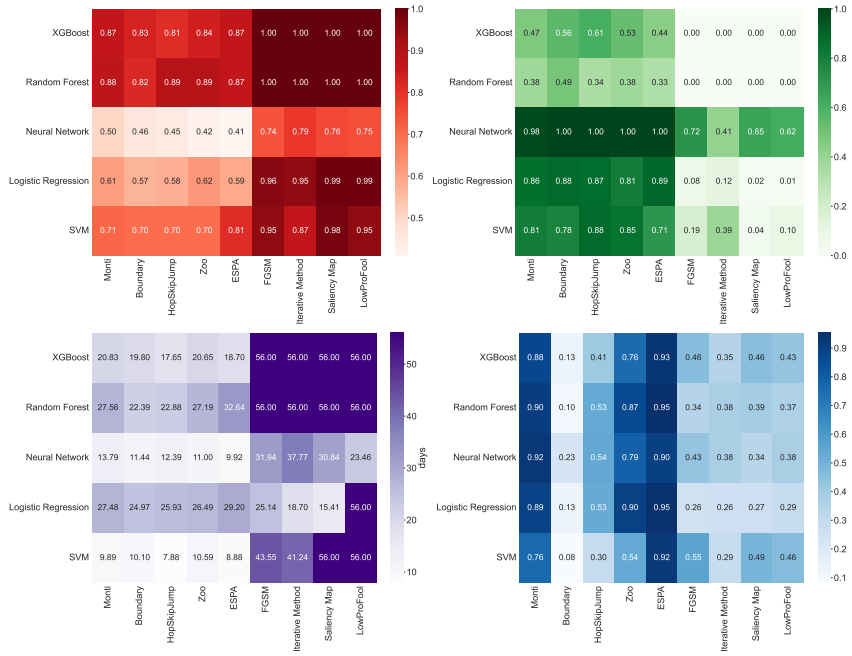


(a) Detection metrics in graybox scenario and with a conservative strategy.



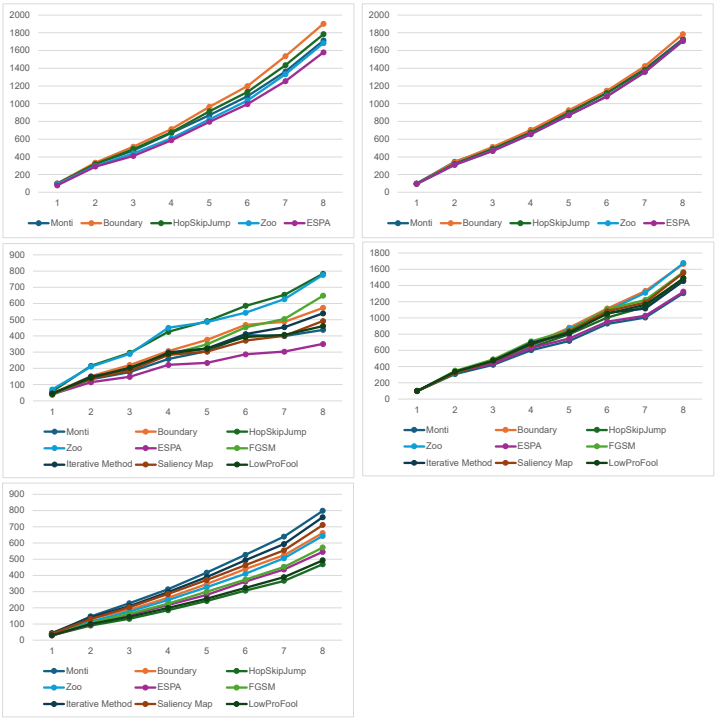
(b) Detection metrics in graybox scenario and with a medium strategy.

Figure 9.4: Detection metrics in graykbox scenario. From left to right: Evasion Rate, Detection Rate, Time Before Detection, Injection Rate.

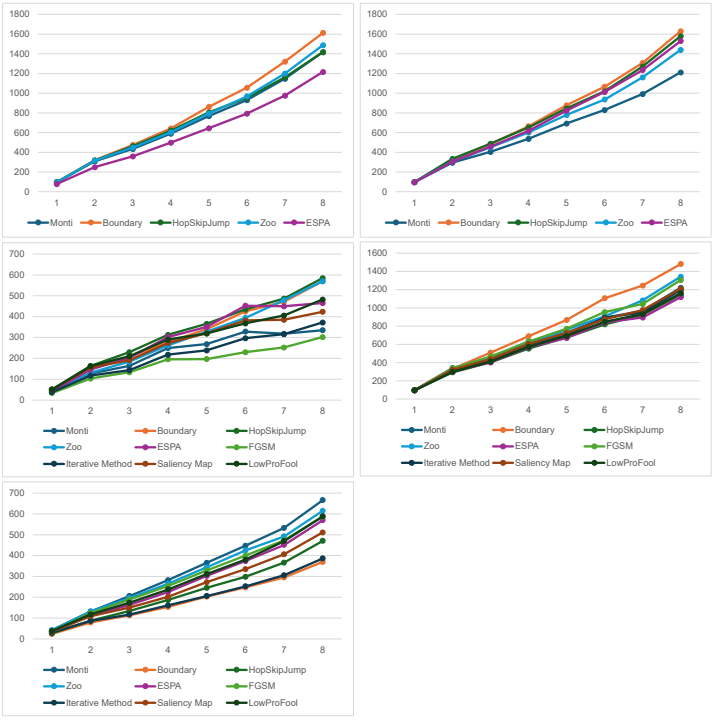


(c) Detection metrics in graybox scenario and with a greedy strategy.

Figure 9.4: Detection metrics in graykbox scenario. From left to right: Evasion Rate, Detection Rate, Time Before Detection, Injection Rate.

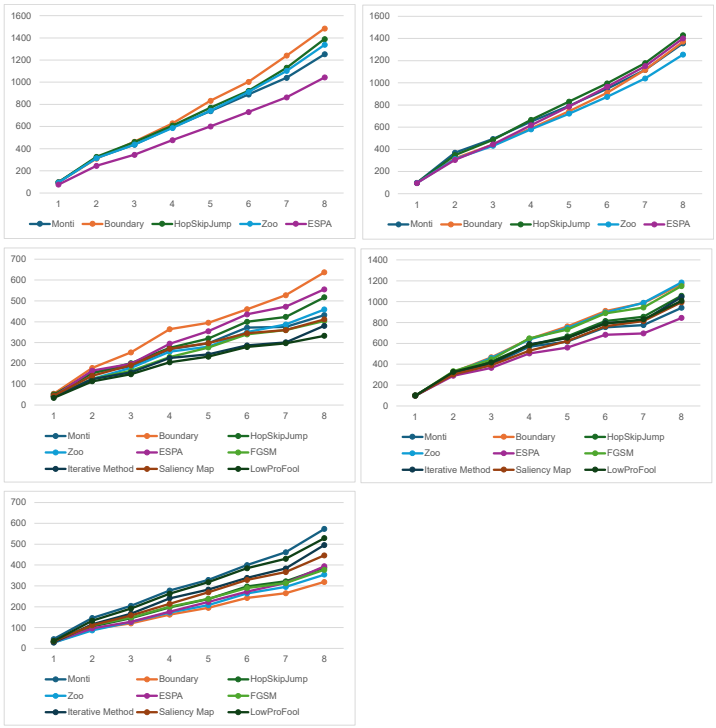


(a) Poisoning rate after each week in whitebox scenario and with a conservative strategy.



(b) Poisoning rate after each week in whitebox scenario and with a medium strategy.

Figure 9.5: Poisoning rates in whitebox scenario for each strategy and each model. From left to right: XGB, RF, NN, LR, SVM.



(c) Poisoning rate after each week in whitebox scenario and with a greedy strategy.

Figure 9.5: Poisoning rates in whitebox scenario for each strategy and each model. From left to right: XGB, RF, NN, LR, SVM.

10 | Limitations and Future Works

10.1. Limitations

Our approach has the following limitations:

- The two datasets that we used belong to the same institution. In blackbox and graybox scenarios, this can influence the results of the experiments.
- Gradient-based attacks can't be used against every model in whitebox scenario.
- We assume that each ML model has *Amount* and *time_from_previous_trans_global* in the feature set to extract the raw features *Amount* and *Timestamp*.

10.2. Future Works

Related to this work, we propose the following direction for future research:

- Improve the gradient-based attacks, by using a double Oracle: the first should be used to classify the crafted transaction and the second to compute the gradient. This could improve the stolen amount and allow us to carry out a simulation against every model in the whitebox scenario.
- Generate the adversarial transactions using adversarial algorithms based on GANs (Generative Adversarial Networks) and LLM(Large Language Modeling).

11 | Conclusion

With this work, we present a framework to test the effectiveness of adversarial algorithms against different FDSs. The framework can be expanded with new algorithms and FDS. In our experiments, we conducted a poisoning attack simulation over 8 weeks, with every combination algorithm, model, strategy, and scenario. The framework permits the simulation of an evasion attack alone, excluding the poisoning phase. In conclusion, we can say that with the newly adapted algorithms, an attacker can steal a relevant amount of money. More in detail, decision-based attacks introduced in previous works [39][48] and the one that we adapted, ESPA, are the most effective. Gradient-based attacks introduced are less effective from the amount stolen perspective, but if we look at the detection metrics are better. We can explain this duality with the fact that the Oracle used with gradient-based attacks has a False Positive Rate more than 10 times higher than the Oracle used with the other attack class. Hence, on one side, the Neural Network filters a smaller number of transactions than the XGBoost, which leads to a smaller amount of money stolen. On the other side, the transactions that are filtered are more effective in evading the FDS target. In the whitebox scenario, where every algorithm uses the same Oracle, the differences between decision-based and gradient-based attacks are less significant. In addition to what said above about the Oracle, we can think that the bottleneck is the Oracle in gradient-based attacks.

Bibliography

[1] A. Abdallah, M. A. Maarof, and A. Zainal. Fraud detection system: A survey. *Journal of Network and Computer Applications*, 68:90–113, 2016. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2016.04.007. URL https://www.sciencedirect.com/science/article/pii/S1084804516300571.

[2] N. Akhtar and A. Mian. Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access*, 6:14410–14430, 2018. doi: 10.1109/ACCESS.2018.2807385.

[3] M. Alzantot, Y. Sharma, S. Chakraborty, H. Zhang, C.-J. Hsieh, and M. Srivastava. Genattack: Practical black-box attacks with gradient-free optimization, 2019.

[4] J. O. Awoyemi, A. O. Adetunmbi, and S. A. Oluwadare. Credit card fraud detection using machine learning techniques: A comparative analysis. In *2017 International Conference on Computing Networking and Informatics (ICCNI)*, pages 1–9, 2017. doi: 10.1109/ICCNI.2017.8123782.

[5] V. Ballet, X. Renard, J. Aigrain, T. Laugel, P. Frossard, and M. Detyniecki. Imperceptible adversarial attacks on tabular data, Dec. 2019. URL http://arxiv.org/abs/1911.03274.

[6] E. C. Bank. Report on card fraud in 2020 and 2021, 2023. URL https://www.ecb.europa.eu/pub/cardfraud/html/ecb.cardfraudreport202305~5d832d6515.en.html#:~:text=In%20relative%20terms%2C%20card%20fraud,in%20both%202020%20and%202021.

[7] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13:281–305, 03 2012.

[8] B. Biggio and F. Roli. Wild patterns: Ten years after the rise of adversarial machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2154–2156, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3264418. URL https://doi.org/10.1145/3243734.3264418.

[9] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part III 13*, pages 387–402. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-40994-3.

[10] B. Biggio, B. Nelson, and P. Laskov. Poisoning attacks against support vector machines, Mar. 2013. URL `http://arxiv.org/abs/1206.6389`.

[11] B. Biggio, G. Fumera, and F. Roli. Security evaluation of pattern classifiers under attack. *IEEE Transactions on Knowledge and Data Engineering*, 26(4):984–996, 2014. doi: 10.1109/TKDE.2013.57.

[12] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. doi: 10.1023/a:1010933404324. URL `https://doi.org/10.1023/a:1010933404324`.

[13] W. Brendel, J. Rauber, and M. Bethge. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models, Feb. 2018. URL `http://arxiv.org/abs/1712.04248`.

[14] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57, 2017. doi: 10.1109/SP.2017.49.

[15] M. Carminati, M. Polino, A. Continella, A. Lanzi, F. Maggi, and S. Zanero. Security evaluation of a banking fraud analysis system. *ACM Transactions on Privacy and Security (TOPS)*, 21(3):1–31, 2018.

[16] M. Carminati, L. Santini, S. Zanero, and M. Polino. Evasion attacks against banking fraud detection systems. *3rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 285–300, 2020. URL `https://www.usenix.org/conference/raid2020/presentation/carminati`.

[17] F. Cartella, O. Anunciacao, Y. Funabiki, D. Yamaguchi, T. Akishita, and O. Elshocht. Adversarial attacks for tabular data: Application to fraud detection and imbalanced data, Jan. 2021. URL `http://arxiv.org/abs/2101.08030`. arXiv:2101.08030 [cs].

[18] A. Chakraborty, M. Alam, V. Dey, A. Chattopadhyay, and D. Mukhopadhyay. A survey on adversarial attacks and defences. *CAAI Transactions on Intelligence Technology*, 6(1):25–45, 2021. doi: https://doi.org/10.1049/cit2.12028. URL `https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/cit2.12028`.

[19] J. Chen, M. I. Jordan, and M. J. Wainwright. Hopskipjumpattack: A query-efficient decision-based attack, Apr. 2020. URL http://arxiv.org/abs/1904.02144.

[20] P.-Y. Chen, H. Zhang, Y. Sharma, J. Yi, and C.-J. Hsieh. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 15–26, Nov. 2017. doi: 10.1145/3128572.3140448. URL http://arxiv.org/abs/1708.03999.

[21] T. Chen and C. Guestrin. Xgboost. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016. doi: 10.1145/2939672.2939785.

[22] N. Cristianini and E. Ricci. *Support Vector Machines*, pages 928–932. Springer US, Boston, MA, 2008. ISBN 978-0-387-30162-4. doi: 10.1007/978-0-387-30162-4_415. URL https://doi.org/10.1007/978-0-387-30162-4_415.

[23] A. Demontis, M. Melis, M. Pintor, J. Matthew, B. Biggio, O. Alina, N.-R. Cristina, F. Roli, et al. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *28th USENIX security symposium*, pages 321–338. USENIX Association, 2019.

[24] B. di Italia. sistema dei pagamenti, 2022. URL https://www.bancaditalia.it/pubblicazioni/sistema-pagamenti/2022-sistema-pagamenti/statistiche_SDP_20220523.pdf.

[25] A. Dignani. Fraudsdigger: an active learning tool for online banking fraud detection. Master's thesis, Politecnico di Milano, 2018.

[26] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples, 2014. URL https://arxiv.org/abs/1412.6572v3.

[27] K. P. Grammatikakis, I. Koufos, N. Kolokotronis, C. Vassilakis, and S. Shiaeles. Understanding and mitigating banking trojans: From zeus to emotet. In *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 121–128, 2021. doi: 10.1109/CSR51186.2021.9527960.

[28] S. Gupta, A. Singhal, and A. Kapoor. A literature survey on social engineering attacks: Phishing attack. In *2016 International Conference on Computing, Communication and Automation (ICCCA)*, pages 537–540, 2016. doi: 10.1109/CCAA.2016.7813778.

[29] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar. Adversarial

machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, AISec '11, pages 43–58, New York, NY, USA, Oct. 2011. Association for Computing Machinery. ISBN 978-1-4503-1003-1. doi: 10.1145/2046684.2046692. URL https://dl.acm.org/doi/10.1145/2046684.2046692.

[30] A. Jain, J. Mao, and K. Mohiuddin. Artificial neural networks: a tutorial. *Computer*, 29(3):31–44, 1996. doi: 10.1109/2.485891.

[31] S. Khatri, A. Arora, and A. P. Agrawal. Supervised machine learning algorithms for credit card fraud detection: A comparison. In *2020 10th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, pages 680–683. IEEE, 2020. doi: 10.1109/Confluence47617.2020.9057851.

[32] KPMG. Global banking fraud survey, 2019. URL https://assets.kpmg/content/dam/kpmg/xx/pdf/2019/05/global-banking-fraud-survey.pdf.

[33] N. Kumar, S. Vimal, K. Kayathwal, and G. Dhama. Evolutionary adversarial attacks on payment systems. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 813–818, Dec. 2021. doi: 10.1109/ICMLA52953.2021.00134.

[34] A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial machine learning at scale, Feb. 2017. URL http://arxiv.org/abs/1611.01236.

[35] M. R. Lepoivre, C. O. Avanzini, G. Bignon, L. Legendre, and A. K. Piwele. Credit card fraud detection with unsupervised algorithms. *Journal of advances in information technology*, 7(1), 2016.

[36] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks, Sept. 2019. URL http://arxiv.org/abs/1706.06083.

[37] L. Maniscalchi. Fraudbench: A benchmarking software for fraud detection systems. Master's thesis, Politecnico di Milano, 2023.

[38] T. M. Mitchell. *Machine learning*. MacGraw-Hill, 1997.

[39] F. Monti. Poisoning attacks against banking fraud detection systems. Master's thesis, Politecnico di Milano, 2020.

[40] L. Muñoz-González, B. Biggio, A. Demontis, A. Paudice, V. Wongrassamee, E. C. Lupu, and F. Roli. Towards poisoning of deep learning algorithms with back-gradient optimization. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence*

and Security, AISec '17, pages 27–38. Association for Computing Machinery, Nov. 2017. ISBN 978-1-4503-5202-4. doi: 10.1145/3128572.3140451. URL https://dl.acm.org/doi/10.1145/3128572.3140451.

[41] M.-I. Nicolae, M. Sinn, M. N. Tran, B. Buesser, A. Rawat, M. Wistuba, V. Zantedeschi, N. Baracaldo, B. Chen, H. Ludwig, I. Molloy, and B. Edwards. Adversarial robustness toolbox v1.17.0. *CoRR*, 1807.01069, 2018. URL https://arxiv.org/pdf/1807.01069.

[42] X. Niu, L. Wang, and X. Yang. A comparison study of credit card fraud detection: Supervised versus unsupervised. *arXiv preprint arXiv:1904.10604*, 2019.

[43] F. Norrestad. Number of active online banking users worldwide in 2020 with forecasts from 2021 to 2024, by region, 2021. URL https://www.statista.com/statistics/1228757/online-banking-users-worldwide/.

[44] P. O'Kane, S. Sezer, and D. Carlin. Evolution of ransomware. *IET Networks*, 7 (5):321–327, 2018. doi: https://doi.org/10.1049/iet-net.2017.0207. URL https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-net.2017.0207.

[45] T. Paladini. Rad-x : an adversarial training approach for fraud detection systems. Master's thesis, Politecnico di Milano, 2022.

[46] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings, Nov. 2015. URL http://arxiv.org/abs/1511.07528.

[47] J. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Adv. Large Margin Classif.*, 10, 06 2000.

[48] G. Romeo. Improving poisoning attacks against banking fraud detection systems. Master's thesis, Politecnico di Milano, 2023.

[49] F. Salahdine and N. Kaabouch. Social engineering attacks: A survey. *Future Internet*, 11(4), 2019. ISSN 1999-5903. doi: 10.3390/fi11040089. URL https://www.mdpi.com/1999-5903/11/4/89.

[50] C. Sammut and G. I. Webb, editors. *Logistic Regression*, pages 631–631. Springer US, Boston, MA, 2010. ISBN 978-0-387-30164-8. doi: 10.1007/978-0-387-30164-8_493. URL https://doi.org/10.1007/978-0-387-30164-8_493.

[51] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fer-

gus. Intriguing properties of neural networks, 2014. URL http://arxiv.org/abs/1312.6199.

[52] D. Varmedja, M. Karanovic, S. Sladojevic, M. Arsenovic, and A. Anderla. Credit card fraud detection - machine learning methods. In *2019 18th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pages 1–5, 2019. doi: 10.1109/INFOTEH.2019.8717766.

[53] C. Whitrow, D. J. Hand, P. Juszczak, D. Weston, and N. M. Adams. Transaction aggregation as a strategy for credit card fraud detection. *Data mining and knowledge discovery*, 18:30–55, 2009.

[54] L. Yu and H. Liu. Feature selection for high-dimensional data: A fast correlation-based filter solution. In *Proceedings, Twentieth International Conference on Machine Learning*, volume 2, pages 856–863, 01 2003.

A | Appendix A

A.1. Supervised Machine Learning Models

In this section, we describe the supervised Machine Learning models that we use for our experiments.

A.1.1. Feed Forward Neural Network

A Feed Forward Neural Network is an Artificial Neural Network [30] where each layer of *neurons* (nodes) is connected only with the following layer, differently from other models such as Recurrent Neural Networks in which the connections can form a cycle. In a Feed Forward Neural Network there are three types of layers:

- Input Layer: this layer contains the neurons that receive the input data. The number of neurons is equal to the number of features in the input data. The data is passed to the Hidden Layers;
- Hidden Layers: in thes layers the neurons have different *activation functions* such as *sigmoid* (input values mapped between 0 and 1), *tanh* (input values mapped between -1 and 1) or *Rectifier Linear Unit* ($f(x) = 0 if x \leq 0; f(x) = x if x > 0$). After all the hidden layers, data are passed to the output layer;
- Output Layer: it is the last layer, which compute the network's final result. When the classification task is a binary classification, the *activation function* should be *sigmoid*

Neurons get connected by a *weight*, which measures their strength. The range of a weight's value is between 0 and 1.

A.1.2. Random Forest

Random Forest [12] models are used in regression and classification problems. Is an ensemble model that combines decision trees where at each split randomly select a subset

of the features. The predictions by majority voting among the decision trees, this reduce the risk of problems such as bias or overfitting. There are three main hyperparameters: *node size*, *number of trees* and *number of sampled features*.

A.1.3. Extreme Gradient Boosting

XGBoost[21] is an efficient and scalable implementation of gradient tree boosting. XGBoost models combines a sequence of predictors(such as decision trees) and minimize the error of the previous predictor through gradient descent algorithm.

Logistic Regression

Logistic regression [50] models are used for classification problems. Denoting the i-th class as C_i and the input vector as $\mathbf{x}$, Logistic Regression models learn the parameters from the input data and output the conditioned probability of $\mathbf{x}$ belonging to C_i (value between 0 and 1). In a binary classification problem, the conditioned probability is defined as:

$$p(C_1|x) = \frac{1}{1 + e^{-\mathbf{w}^T\mathbf{x}}} = \sigma(\mathbf{w}^T\mathbf{x}) \tag{A.1}$$

$\mathbf{w}$ is the vector of weights, C_1 is the positive class and $\sigma(a) = 1/(1 + e^{-}a)$ is the sigmoid function. The conditioned probability of $\mathbf{x}$ belonging to the negative class C_0 can be computed as: $p(C_0|\mathbf{x}) = 1 - p(C_1|\mathbf{x})$.

Support Vector Machine

Support Vector Machine (SVM) [22] models are primarily used to solve classification problems, but can also be employed for regression ones. In a binary classification task, the objective is to find a hyperplane in an N-dimensional space (N is the number of features), maximizing the *margin*, which is the minimum distance between data points of both classes. All the data points belonging to the same side of the hyperplane are assigned to the same class. The dimension of a hyperplane depends on the number of features N. Support Vector Machine are kernel-based algorithms that evaluates the kernel function only for a subset of the points in the dataset, the *Support Vector*. SVM do not provide the class probabilities as output, but directly the output class. To obtain the probabilities we use Platt scaling[47].

B | Appendix B

B.1. Machine Learning Evaluation Metrics

In this section, we describe the metrics that we use to evaluate our FDS Machine Learning models. Frauds belong to the positive class (class 1) and legitimate transactions belong to the negative class (class 0). Thus, True Positive (TP) represent frauds that are classified as fraud, True Negative (TN) represent legitimate transactions that are classified as legitimate, False Negative (FN) represent frauds that are wrongly classified as legitimate, and False Positive (FP) represent legitimate transactions that are wrongly classified as frauds. We define the following evaluation metrics:

- **Recall**: also known as True Positive Rate, it measures how many positive samples are correctly classified as positive over all the positive samples:

$$Recall = \frac{TP}{TP + FN} \tag{B.1}$$

- **Precision**: it is the ratio between the number of correctly classified positive samples and the total number of positive samples predicted by the classifier:

$$Precision = \frac{TP}{TP + FP} \tag{B.2}$$

- **F1-Score**: F1-Score combines Precision and Recall by computing the harmonic mean between them:

$$F1 - Score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{B.3}$$

- **False Positive Rate**: it represents how many negative samples are wrongly considered as positive, with respect to all negative samples:

$$FPR = \frac{FP}{FP + TN} \tag{B.4}$$

- **Matthews Correlation Coefficient**: it measures the correlation between predicted values and the ground truth:

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{B.5}$$

We actually use a weighted version of the MCC(Matthews Correlation Coefficient), assigning weights to the two classes depending on their incidence in the training set:

$$w_0 = \frac{TP + FN}{TP + FN + FP + TN}, \quad w_1 = \frac{TN + FP}{TP + FN + FP + TN} \tag{B.6}$$

We define the weighted version of MCC as:

$$W - MCC = \frac{w_1 TP \cdot w_o TN - w_0 FP \cdot w_1 FN}{\sqrt{(w_1 TP + w_o FP) w_1 (TP + FN) w_0 (TN + FP)(w_0 TN + w_1 FN)}} \tag{B.7}$$

- **Area Under Curve of Receiver Operating Characteristic**: ROC is a probability curve and AUC represents the degree or measure of separability. It tells how much the model is capable of distinguishing between classes. The ROC curve is plotted with TPR (Recall) against the FPR where FPR is on the x-axis and TPR is on the y-axis:

$$ROC = \int_{x=0}^{1} Recall * (FPR^{-1}(x))dx \tag{B.8}$$

- **Area Under Curve of Precision Recall Curve**: The Precision Recall Curve is obtained by plotting the Precision against the Recall at various threshold settings:

$$PRC = \int_{x=0}^{1} Precision * (Recall^{-1}(x))dx \tag{B.9}$$

- **Cost-Accuracy**: this is a custom Accuracy metric[39], that deals with the problem of an imbalanced dataset. Is defined as follows:

$$\text{Cost-Accuracy} = 1 - \frac{FP + k \cdot FN}{FP + TN + k \cdot (TP + FN)} \tag{B.10}$$

where $k = \frac{TN+FP}{TP+FN}$.

C | Appendix C

In this Appendix, we present the results of the experiments against Active Learning. We did not include these results in the thesis results because the Active Learning model does not work properly. Our Active Learning model is a method that combines unsupervised and supervised learning[25]. The unsupervised model computes the anomaly score on the unlabeled data, while the supervised learning model gives a probability score to the labeled data. These two scores are then combined to compute the final prediction. The unsupervised learning module is an Autoencoder, while the supervised module is a Random Forest. The features and hyperparameters selection process is the same as described in 7.3. In Table C.1 there are the selected features and in Table C.2 the selected hyperparameters.

Table C.1: Selected features per Active Learning model.

Model	Features
Active Learning	Amount, time_from_previous_trans_global, difference_from_amount_mean1h, is_new_ip, iban_count30d, iban_std30d, difference_from_iban_mean7d, iban_count1d, is_national_iban, ip_count30d, is_new_iban_cc, difference_from_amount_mean1d, difference_from_iban_mean30d, is_new_iban, difference_from_session_mean1h, iban_count14d, iban_count7d, difference_from_iban_mean14d, amount_std1d, amount_count1d, iban_count1h, iban_cc_count1h, time_since_same_iban

The performances of the model are poor, as we can see in table C.3. If we look at the metrics, we can see that more than 12% of negative samples are wrongly classified as positive. Only 6.87% of positive samples are correctly classified. The good performances obtained by the FDS are explained with these metrics. The FDS fails to distinguish legitimate transactions from fraudolent transactions, hence, the adversarial samples, which try to mimic the legitimate transactions, are more frequently classified as fraudolent, like real legitimate transactions, are correctly classified as fraudolent. In Table C.4 are listed the results of the simulations against Active Learning.

Table C.2: Selected hyperparameters per Active Learning

Model	Hyperparameter	Value
Active Learning	ae_epochs	80
	ae_batch_size	1024
	ae_encoding_size	188
	ae_bottleneck_size	34
	ae_dropout_rate	0.5018747099715827
	ae_output_activation	"sigmoid"
	ae_lambda_reg	0.00036125514803957474
	ae_activation	"relu"
	ae_weight	0.2
	rf_random_state	721077
	rf_n_estimators	485
	rf_max_depth	45
	rf_criterion	"gini"
	rf_class_weight	"balanced"
	rf_min_samples_split	25
	rf_weight	0.8

Table C.3: Performance evaluation of Active Learning model.

Model	Precision	Recall	F1-Score	FPR
Active Learning	6.87%	89.22%	12.75%	12.34%
	AUC-ROC	**AUC-PRC**	**W-MCC**	**Cost-Accuracy**
	94.68%	74.42%	76.89%	88.44%

Table C.4: Total Money Stolen against Active Learning model. We highlight in red (green) the worst (best) result from the defender's standpoint.

Total Money Stolen (€)					
weekly update policy					
		Attack Method	Conservative	Medium	Greedy
Black-box	Active Learning	Monti	91924.31	228521.98	186644.07
		Boundary	186818.18	155806.07	177160.62
		HopSkipJump	133925.08	170612.05	154172.91
		ZOO	189490.59	209624.62	175376.63
		ESPA	194825.44	140595.74	210357.49
		FGSM	17475.53	33954.6	66301.34
		Iterative	7894.72	18936.86	5769.78
		Saliency Map	117198.67	86374.88	32569.4
		LowProFool	165585.58	59081.93	55019.4
Grey-box	Active Learning	Monti	179420.61	200089.44	176397.26
		Boundary	177128.01	135164.45	209247.96
		HopSkipJump	207014.07	144692.57	224254.34
		ZOO	280144.21	186036.5	181215.88
		ESPA	173608.17	133548.67	140303.17
		FGSM	321260.04	100543.12	171313.39
		Iterative	160352.37	177949.78	134338.3
		Saliency Map	84614.18	141469.13	142956.41
		LowProFool	223267.74	233837.92	220457.49
White-box	Active Learning	Monti	2616796.56	2737284.29	1887710.69
		Boundary	2926212.48	3292894.14	3037108.80
		HopSkipJump	2976141.95	2670737.25	2558438.36
		ZOO	2710916.92	2829249.91	2491264.98
		ESPA	2525984.78	2700680.71	2382620.9
		FGSM	NaN	NaN	NaN
		Iterative	Nan	NaN	NaN
		Saliency Map	NaN	NaN	NaN
		LowProFool	NaN	NaN	NaN

List of Figures

List of Tables

List of Acronyms