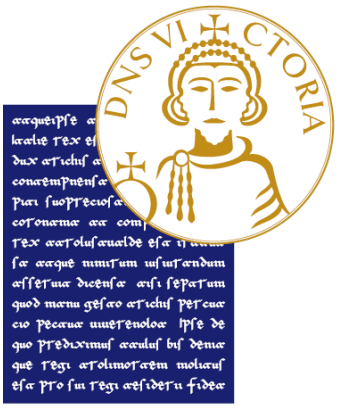
UNIVERSITÀ DEGLI STUDI DEL SANNIO



Department of Engeneering

Three Years Degree in Computer Engeneering

Thesis

in

Cyber Security

STUDY OF POISONING ATTACKS ON CLASSIFIERS FOR MALWARE DETECTION BASED ON MACHINE LEARNING

Supervisors

Prof. Corrado Aaron Visaggio

Student

Roberto Garzone 863000793

Correlatrice

Dott.ssa Sonia Laudanna

Anno Accademico 2020/2021

Indice

Capitolo 1

Introduction

Smartphones, over time, have become an integral part of people's everyday lives, used as the most common means of transmitting sensitive information, such as: photos, contacts or emails.

Such use has transferred the interest of cyber-criminals to the world of telephony, leading them to search for an effective system for obtaining sensitive information.

In fact, in August 2010, Kaspersky[1] recorded the first SMS containing a Trojan, due to which a smartphone sent, without the user's consent, an SMS to premium rate numbers.

In the report generated by McAfee's Global Threat Intelligence[2], dating back to 2018, the presence of 65,000 fake applications was declared within the Google app store, 55,000 of which were uploaded in June of the same year.

Added to these is a significant increase in the presence of Trojans and Backdoors.

Such an increase has generated, within Google, the need to increase security levels, useful for the identification and, consequently, removal of suspicious applications.

Security interventions led, on February 3, 2019, to the elimination of 29 malicious applications, whose activities included the publication of advertisements with pornographic content in full-screen mode every time the phone was unlocked, and the redirection of the user to a page to carry out phishing attacks.

In 2015, the University of Cambridge, analyzing over 20,000 smartphones from various vendors, reported that 87.7% of Android devices are susceptible of at least one critical vulnerability.

The attention of the experimentation was focused on the Android system, as StatCounter[3] has statistically defined that over 75% of smartphones on the market use this operating system.

In this scenario, an artificial intelligence technique that has recently gained traction is known as machine learning[4][5].

It is a technique that allows the computer to learn from experience and, therefore, make decisions based on it, without requiring explicit instructions.

Using machine learning it is therefore possible to identify potential malware that is still unknown[6][7].

This technique is divided into two categories: supervised or unsupervised.The fundamental difference between the two categories is the type of learning: in the first case the learning is carried out with the support of a series of examples, which constitute the dataset, which allow the computer to make a distinction between the elements sent to it as input.

In the unsupervised case, the computer itself will have to identify the characteristics of the individual input data and divide them based on them[7][8]-[9].
Machine learning is widely used in the identification and classification of malware through three different types of analysis: static, dynamic and hybrid[10].
Static analysis is obtained without running the application, therefore without any effect on the device. It is based on the study of specific elements of the source code, such as: permissions, API calls or intent (category or action)[11] and is highly vulnerable to obfuscation-based attacks[12]-[13].
Dynamic analysis, on the other hand, consists of running an application within a sandbox and recording all log events.
Thanks to this analysis it is possible to record all system calls made by the application[14][15] and monitor the behavior of the malware itself.
The last type of analysis is hybrid analysis, which incorporates the methods and advantages of static and dynamic analysis.Hybrid analysis can be performed both locally and on a remote server[16].
Although machine learning solutions are successfully adopted in multiple contexts, the application of these techniques to the cyber security domain is complex and still immature. Among the many open questions affecting machine learning-based security systems, this work examines the robustness of such models against adversarial attacks that aim to influence detection and prediction capabilities.

1.1 Goals

The objective of the experiment is to evaluate the robustness of detectors based on machine learning against poisoning attacks. The classifiers used for the evaluation are based on: Logistic Regression, Random Forest and Support Vector Machine algorithms.
The contributions of this study include:

- Classification of benign and malware-type apks and evaluation of the accuracy of the classifiers;

- Generation of poisoning attacks based on Euclidean distance;

- Generation of poisoning attacks based on Hamming distance;

- Generation of poisoning attacks based on K-Means Clustering;

- Identification of the smallest perturbation that can be added to a sample to make it adversarial;

- Evaluation of the effectiveness of the models considered in the various attack scenarios.

1.2 Context and Motivation

Android happens to be the most popular mobile operating system. The growing popularity of Android has led to an increase in the development of third-party applications, and as a result, the authenticity of these apps is always in question. Third-party apps are those apps created by a vendor other than the device manufacturer. They are intended to satisfy the intentions of the seller, not the user. Third-party applications are vulnerable to different types of attacks. These apps, even if present on official app stores, can be harmful. Several attack scenarios appear to be executed on third-party applications. Attack scenarios, for example, include Android app repackaging, which occurs when the attacker adds malicious features and redistributes them to users who believe they are using a legitimate app or the original app. The act of 'repackaging' the application is also used when trying to reverse engineer the application itself. Repackaging is a widely used practice to especially distribute Trojans[17].

Adversarial machine learning is a field that sits at the intersection of machine learning and cybersecurity. This technique attempts to 'fool' machine learning models via malicious input[18]-[19].

Several works have shown that some adversarial examples generated for one model can also be misclassified by another model. This property, which is not based on the model for which it was created, is called transferability. It can be exploited to perform black-box attacks[20]. There are several methods to create adversarial examples such as Fast Gradient Sign Method (FGSM), Jacobian-Based Saliency Map Approach (JSMA)[21]-[22].

The Poisoning attack is one of the most important techniques for creating adversary attacks.

This attack is carried out in the learning phases of any artificial intelligence algorithm, introducing a series of carefully designed training sets, useful for compromising the entire learning process and creating a backdoor[23].

Poisoning can be of three types:

1. Dataset poisoning: one of the most common attacks, consisting of introducing incorrect data into the training set or modifying the behavior of the algorithm;

2. algorithm poisoning: in this case a specific algorithm is exploited to understand its learning model. There are several ways to poison an algorithm, including:

    - poison transfer learning: attackers train a poisoned algorithm and spread it to other machine learning algorithms
    - data injection and manipulation: attackers inject a series of incorrect data into the dataset
    - logic corruption: the attacker changes the learning method of the algorithm

3. Model poisoning: This is the simplest form of poisoning. The model is replaced with a poisoned one, which remains inside the computer acting as a backdoor for the attacker.

As an example, it is possible to mention one of the most famous experiments on the poisoning attack which saw a group of researchers making pertur-

bations to the images of a panda, loaded into a learning algorithm for the identification of the species, leading the same algorithm to identify it as a panda the image of a gibbon[24][25].

In this work, three poisoning attack scenarios are discussed:

1. Poisoning Attack based on euclidean distance;
2. Poisoning Attack based on Hamming distance;
3. Poisoning Attack based on K-Means Clustering.

All attack scenarios are executed on static *features* of an application, such as permissions. The effectiveness of the created attack is examined using different machine learning classifier models.

1.3 Work Organisation

The thesis consists of 4 chapters plus bibliography.

The first chapter gives a general overview of the motivations that led to the birth of this experimentation and the objectives that we want to pursue. The concept of machine learning, its applications and its implementation is reviewed. The concept of Poisoning Attack and the forms that this can take are analyzed in the same way.

The second part of the thesis involves the analysis of the works present in the literature relevant to the objectives of the experimentation. Works which are then used as a starting point for the construction of the experimentation model itself.

The third chapter includes a practical description of the implementation of machine learning algorithms and the analysis of the results. Regarding the description of the attack scenarios, it is necessary to specify that when we talk about steps, we mean the lines of code of the relevant pseudo-code. Missing steps are steps that indicate the closure of iterative loops and conditional constructs.

The fourth chapter will contain the final considerations, based on the data obtained from the analyzes carried out.

The last chapter is the bibliography, containing the sources of the information included in the thesis.

Capitolo 2

State of art

There are several works in the literature that concern both the generation of adversarial attacks and poisoning attacks in particular.
Xuanzhe Liu conducted a study on the behavioral pattern of Android users[26]. It focuses on analyzing the different usage of applications, based on popularity, task management, network usage and device choice. Knowledge of these patterns helps the research community in discovering new trends that can be followed for the development of new applications and their distribution.
Sen Chen's study, however, illustrates the development of an approach based on adversarial machine learning, divided into two phases. This approach is known as KuafuDet[11]. The first phase consists in the extrapolation of the *features*; which will then be analyzed, in the second phase, with a remote analysis. Adversarial machine learning algorithms are applied for obfuscating malicious applications.
Sen Chen's work acts as a filter for identifying false negatives, proposing a system capable of increasing the accuracy of detection by 15%.

Another study by Sen Chen is called StormDroid[27], a set made up of a series of *features* extracted to perform their classification. Furthermore, a series of API calls are extracted using tree-based metrics. Ít is demonstrated how StormDroid achieves a detection accuracy of 94%.

Weilin Xu proposed a case study based on PDF malware classification and automatic evasion classification methods[28]. In this study, malware variants are generated using genetic programming techniques, called mutations. The evaluations are carried out using two particular classifiers: PDFrate and Hidost. In order to identify malware within the dataset, tools made available by Oracle are applied. After identifying the malicious samples, they are appropriately manipulated, using particular mutation techniques.

Finally, Fei Zhang[29] deduced that *features* can be selected based on generalization capabilities and security regarding evasion attacks. In this regard, it is necessary to implement a selection algorithm based on the wrapper. An example of an evasion attack is then generated, useful for evaluating the robustness of the group of selected *features*. The experiment involved the use of spam filtering and a PDF malware detection system.

Capitolo 3

Experimental design

3.1 Research questions

This work has the main objective, as already anticipated, of evaluating the robustness, in terms of accuracy, of a classification model in the event of a poisoning attack and of determining what and how small the perturbation must be to make a sample of adversarial type.
Two research questions were posed during the experiment:

RQ1: What is the smallest perturbation that can be added to a sample to make it adversarial?

RQ2: How robust are machine learning models considered to be subjected to poisoning attacks?

To answer the first research question it is necessary to perform several tests during which the perturbation undergoes arbitrary and incremental variations. The aim is to define the minimum disruption necessary to contribute

to the alteration of the malware and its subsequent evasion.

Instead, to answer the second research question it is necessary to subject the previously implemented machine learning algorithms to a poisoning attack. The altered malware is collected in a file which is then used as a test set for prediction. On the basis of the predictions obtained using the three classifiers: Logistic Regression, SVM and Random Forest, it will be possible to define which, among the three algorithms tested, will prove to be the most robust, i.e. which algorithm will be most capable of recognizing malware despite the alteration. This robustness is then analyzed using the detection rate of the algorithms, with which it is possible to understand, for each algorithm and for each classifier, in which scenario the lowest number of evaded malware samples is obtained.

3.2 Dataset description

The dataset used includes 2416 samples, downloaded from the AndroZoo repository[30], a dataset of over a million Android applications collected from various stores. We distinguish two types of these samples: benign and malware. Differentiated, in the dataset, by the "CLASS" label, benign samples are indicated with "0", while malware are indicated with "1".

From the samples obtained, the permissions of the individual applications are examined, extracted through a reverse-engineering process using a specific script written in Python: ExtractorAIO.py[31].

This script performs the operations of extrapolation and collection of the permissions of the .apk files. Extrapolation occurs using the jadx module[32]; a Java-Based decompiler, whose function consists in extrapolating the permissions of applications for Android OS. The collection of permissions, correlated to the name of the application, occurs through Unix shell commands, useful for creating and modifying the .csv file, container of the information obtained.

In fig. 3.1 the permission extrapolation process is represented.

From the permission extrapolation process, a .csv file is created like the one shown in fig. 3.2.

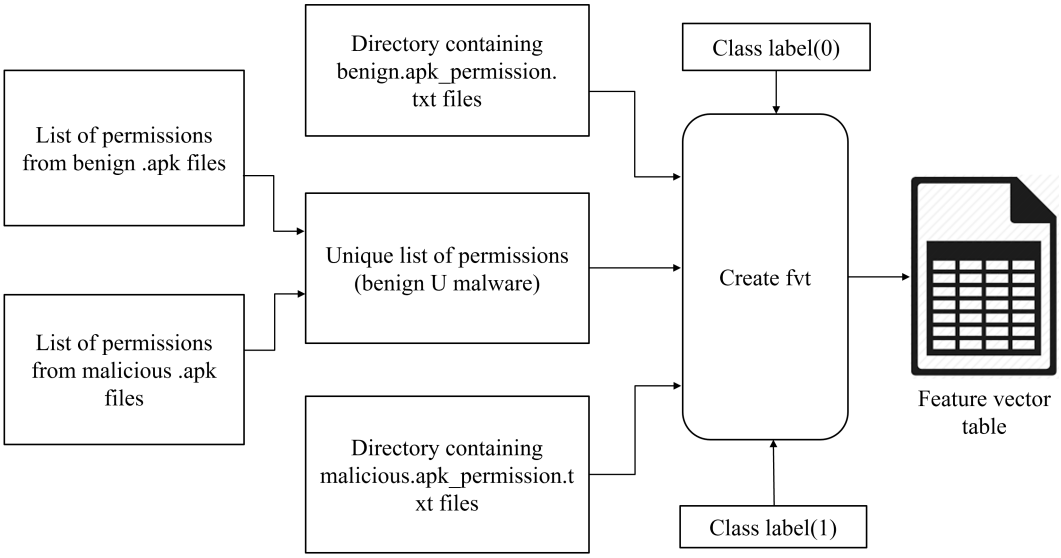


Figura 3.1: Creating the feature vector table

	A	ASH	ASI	ASJ	ASK	ASL	ASM	ASN	ASO
1	NAME	biz.app4m	com.googl	com.noma	com.indiai	com.subsp	br.com.tec	android.ha	CLASS
853	067157FB	0	0	0	0	0	0	0	0
854	06716825	0	0	0	0	0	0	0	0
855	06716981	0	0	0	0	0	0	0	0
856	06716D9A	0	0	0	0	0	0	0	0
857	06716EB0	0	0	0	0	0	0	0	0
858	067170F8	0	0	0	0	0	0	0	0
859	067190EF	0	0	0	0	0	0	0	0
860	0671941C	0	0	0	0	0	0	0	0
861	0671946F	0	0	0	0	0	0	0	0
862	06719C33	0	0	0	0	0	0	0	0
863	06719E0C	0	0	0	0	0	0	0	0
864	0671A331	0	0	0	0	0	0	0	0
865	0671A716	0	0	0	0	0	0	0	0
866	0671A99A	0	0	0	0	0	0	0	0
867	0000003B	0	0	0	0	0	0	0	1
868	0000014A	0	0	0	0	0	0	0	1
869	000002B6	0	0	0	0	0	0	0	1
870	000003D3	0	0	0	0	0	0	0	1
871	00000439	0	0	0	0	0	0	0	1
872	0000049D	0	0	0	0	0	0	0	1
873	0000090C	0	0	0	0	0	0	0	1
874	00000989	0	0	0	0	0	0	0	1
875	00000F8E	0	0	0	0	0	0	0	1
876	00001091	0	0	0	0	0	0	0	1
877	000010F3	0	0	0	0	0	0	0	1
878	00001112	0	0	0	0	0	0	0	1
879	0000120C	0	0	0	0	0	0	0	1
880	0000143E	0	0	0	0	0	0	0	1
881	000014F7	0	0	0	0	0	0	0	1
882	00001802	0	0	0	0	0	0	0	1
883	00001991	0	0	0	0	0	0	0	1

Figura 3.2: Dataset obtained

3.3 Analysis methodology

The poisoning attack is an injection process, which produces perturbations that increase the error rate of the machine learning algorithm.
In order to be able to carry out an evaluation of the robustness of the implemented algorithms, the presence of a classifier, called H, and of a dataset for the classification, which will act as a training set and an appropriately selected test set, is necessary.
The dataset appears as a multidimensional d dimensional array of the type:

$$[D = (X_i, y_i)_{i=1}^{n}] \tag{3.1}$$

dove

$$X_i \in [1, 0]^d \tag{3.2}$$

The classifier H, taking the dataset as input, performs the classification and produces a result y of the type: H(X) = y.
The objective of the poisoning attack is to add small perturbations to the *features* of:

$$X \wedge H(X + \mu) = H(X^*) \mid H(X^*) = y^{'} \rightarrow y^{'} \neq y \tag{3.3}$$

In the scenario proposed in this work, three different poisoning attacks will be performed:

1. Poisoning attack based on the Euclidean distance between the malware and the entire set of benign samples;

2. Poisoning attack based on the Hamming distance between the malware and the entire set of benign samples;

3. Poisoning attack based on K-Means Clustering.

In the experimental phase, a scenario of the type shown in fig.3.3 is analysed, in general form.

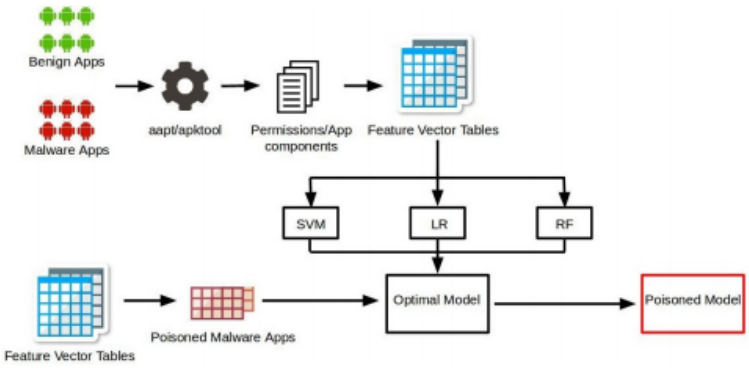


Figura 3.3: Architecture: Find an optimal model and execute a poisoning attack

The attack will be performed on a dataset containing 1120 malicious .apk, provided by AndroZoo[30] and 1296 benign .apk, provided by the Canadian Institute for Cybersecurity[33].

For each .apk file obtained, permissions will be extracted. Subsequently we will verify the effectiveness of the classifiers.

The dataset used for classification will contain both types of apk and the related permissions.

The latter will be binary in nature, but will have to be converted into integers at the time of experimentation.

Malicious samples will be extracted from the dataset, which will act as test sets and will be subjected to the poisoning attack when specific conditions

are met.

The altered malware will finally be sent as input to the optimal model for prediction, i.e. verifying the effectiveness of the algorithm. The latter will be more effective the fewer samples processed.

The evaluation will be based on the analysis of the accuracy, evasion rate, precision and recall of the individual classifiers, calculated before and after the attack.

Weka software was used to perform the classification and prediction. It is software created specifically for such operations. By taking .arff files as input, it is able to extrapolate the *features* of datasets and test sets and then perform the operations of classifying the elements of the dataset and predicting the elements present in the test set, based on the indicated classifier. The result can be saved in different extensions (such as .csv, .html, etc...). For ease of analysis, the .csv extension has been indicated.

3.3.1 Attack scenario based on euclidean distance

For this attack, 10% of the entire test set will be used, obtained by calculating the Euclidean distance between the malware and the benign.

For each benign apk, a permission will be chosen in a completely random manner. The alteration will be carried out only if the permission is present in the benign apk.

The scenario will see the malicious vector of the type:

$$M_n = (m_{n1}, m_{n2}, m_{n3}, ..., m_{nm}) \tag{3.4}$$

while the benign vector will be of the type:

$$B_n = (b_{n1}, b_{n2}, b_{n3}, ..., b_{nm}) \tag{3.5}$$

Once the two vectors have been obtained, the Euclidean distance will be calculated, which can be calculated with the following formula:

$$\sqrt{(m_{n1} - b_{n1})^2 + (m_{n2} - b_{n2})^2 + (m_{n3} - b_{n3})^2 + ... + (m_{nm} - b_{nm})^2} \tag{3.6}$$

where m_{ni} represents the absence or presence of the i-th permission in the M_n malware and b_{ni} represents the presence or absence of the i-th permission in the benign file B_n

A general view of the implementation is provided by the pseudo-code shown in fig. 3.4:

Algorithm 1 Poisoning Euclidean

Input: $Test\_setT, classifierH, benigndatasetB, perturbation\delta$
Output: poisoned sample x_i'

```
1: evasion_count ← 0
2: repeat
3:   count ← 0
4:   repeat
5:     r ← 0, avg ← 0
6:     repeat
7:       sample ← x_i
8:       select random feature r_num
9:       if B[r_num] = 1 and T[r_num] = 0 then
10:        count ← count + 1
11:        sample[r_num] ← 1
12:        r ← r + 1
13:        dist ← 0, sum_dist ← 0
14:        repeat
15:          dist ← distance.euclidean(sample : j[0 : δ])
16:          edist.append(dist)
17:          sum_dist ← sum_dist + dist
18:        until j ≤ |B|
19:      end if
20:      avg ← sum_dist/|B|
21:      edist.sort()
22:      flag ← 1
23:      repeat
24:        if avg < edist[l] then
25:          flag ← 0
26:        end if
27:      until l ≤ |B| * δ
28:      if flag = 0 then
29:        p.append(H.predict_class(sample))
30:      end if
31:      if p = 0 then
32:        evasion_count ← evasion_count + 1
33:      end if
34:    until r ≤ δ
35:  until i ≤ |B|
36: until k ≤ |T|
37: goto 28
```

Figura 3.4: Pseudo-code based on euclidean distance

Specific inputs are provided in this code:

- Test Set (T)
- classifier (H)
- benign dataset (B)
- perturbation (δ)

The scenario is represented in fig. 3.5:

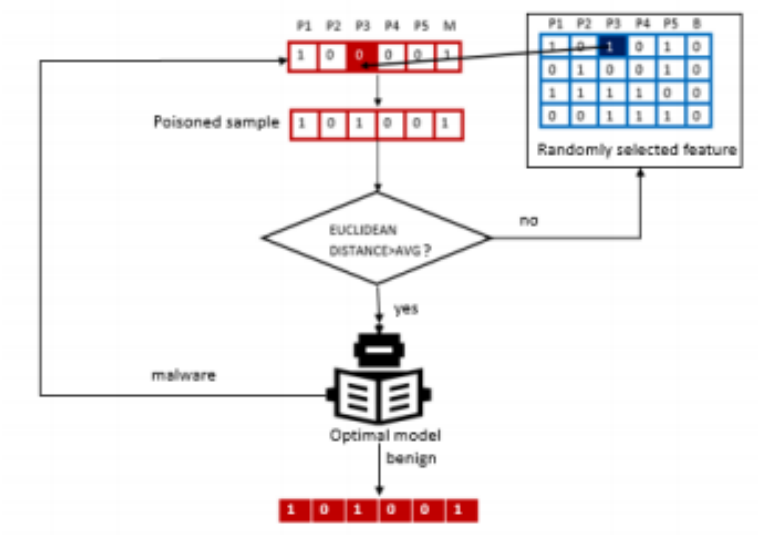


Figura 3.5: Poisoning attack based on euclidean distance

The algorithm can be summarized as follows: first, it is necessary to initialize the *evasion_count* variable (**step 1**) to 0, this variable will count the number of evaded malware. Subsequently, an iterative cycle is started (**step 2**) which, at each iteration, will reset the variable *count* (**step 3**): a counter of the alterations suffered by the same malware. The cycle at step 2, limited by the increment of a k variable, ends when k takes on a value equal to the

size of the previously filled test set.

After resetting *count*, a new cycle opens (**step 4**), limited by the increase of the variable i, which ends when i takes on a value equal to the size of the benign set, previously filled. At **step 5** the variables r and avg are reset to zero, respectively: the perturbation introduced and the average of the distances Euclidean distances between the assumed malware and the benign ones present in the dataset.

A new iterative cycle opens (**step 6**), which ends when the perturbation reaches an arbitrarily imposed limit.

In this cycle we see the loading of the i-th malware into the *sample* variable (**step 7**) and the execution of the alteration (**lines 9 - 11**).

A random value is selected, loaded into the r_num variable (**step 8**), which indicates the position of the *feature* in both the malware and the benign. This is because for the alteration to be possible, a condition must be respected: the *feature* must be present in the benign (1 in the dataset) and absent in the malware (0 in the dataset) (**step 9**).

If the condition is respected we move on to the alteration.

count is incremented by 1 (**step 10**), permission is forced to 1 in *sample* (**step 11**), and perturbation is incremented by 1 (**step 12**).

Once the alteration has been completed, but still in the conditional construct, the variables *dist*, which will contain the subsequently calculated Euclidean distance, and sum_dist, which will calculate the sum of the distances (**step 13**).

A new iterative cycle then opens (**step 14**) which ends when the variable j equals the length of the set of benigns.

In this cycle the Euclidean distance between the malware and the j-th benign is calculated (**step 15**), the calculated distance is saved in a list *edist* (**step 16**) and added to the *sum_ dist* variable (**step 17**).

Once the loop ends, the conditional construct also ends.

Subsequently, the average of the distances is calculated (**step 20**), loaded into *avg*.

The list of distances is sorted in ascending order (**step 21**).

A *flag* is initialized to 1 (**step 22**). This flag, if equal to 0, allows the execution of the prediction on the assumed malware.

A new iterative cycle opens (**step 23**), which ends when l reaches a value equal to the length of the set of benigns multiplied by the limit perturbation given as input to the algorithm, in which it is verified that the average of the distances is less than 70% of the distances contained in *edist* (**step 24**). If the condition is verified *flag* is set to 0 (**step 25**).

At the end of the cycle, a condition is placed on the value of *flag* (**step 28**): if flag is 0 the prediction is executed, otherwise we move on to the next malware (**step 29**). The prediction result is loaded into the variable p.

Subsequently, a condition is placed on p (**step 30**): if p is equal to 0 the *evasion_ count* variable is increased by 1 (**step 32**), otherwise you move on to the next malware. This is because if p is equal to 0, it means that the malware was predicted to be benign.

3.3.2 Attack scenario based on Hamming distance

In the implementation of the poisoning attack based on the Hamming distance, once the dataset is given as input, a random set of malware to be altered is chosen.

Once the choice is made, the Hamming distance of each malware with all benign files will be calculated.

Having binary values, it is possible to calculate the median Hamming distance of the XOR between the two vectors;

$$hd = 1011011001 \oplus 0100110011 = 111111010 \tag{3.7}$$

$$d(1011011001, 0100110011) = 7 \tag{3.8}$$

The benign files will be sorted, in ascending order, based on the distance value. At this point only 0.5% of the benign cases will be taken into consideration for the verification of the alteration conditions.

As in the previous implementation, the malware will only be altered if the randomly extracted *feature* is present only in the benign file.

By satisfying this condition, the malware will be altered, therefore, the *feature* will be added to it.

The algorithm can be thought of as represented in fig. 3.6.

Saranno necessari alcuni input affinchè l'attacco possa essere efficace:

- Dataset (D)
- Test_set, casualmente scelto

- Numero di file .apk benigni (β)
- Limite di perturbazione (δ)

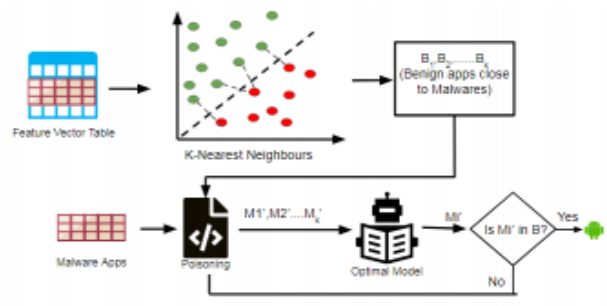


Figura 3.6: Poisoning attack basato sulla distanza di Hamming

The algorithm can be summarized as follows: we start with the initialization of the iteration counter i to 0 (**step 1**), with which we define the term of the entire algorithm.

Subsequently, an iterative cycle is started (**step 2**), which ends when the variable i reaches a value equal to the length of the previously calculated test set.

Once the cycle is started, the i-th malware is assigned to the variable x (**step 3**) and the iteration counter j is initialized to 0 (**step 4**).

A second iterative cycle is then started (**step 5**) which ends when j takes on a value equal to the number of apks present in the dataset.

Inside this cycle, the j-th apk present in the dataset is loaded into the variable b (**step 6**) and it is verified that it is a benign apk (**step 7**), on the basis of the apk class.

If the condition is not met, we move on to the next apk; otherwise, we proceed with the calculation of the Hamming distance between assumed malware and

benign (**step 8**). Distance that is loaded into the variable *h*.

If the distance is not zero (**step 9**), it is added to a two-dimensional array (**step 10**), which includes the benign array in the first column and the relative distance in the second.

Once the cycle is finished, the two-dimensional array is sorted, in ascending order, (**step 14**) and the beningo with the smallest distance is loaded into the list *l*, or more benign if they have the same distance (**step 15**).

Finally, the *j* variable is reset, which will act as the iteration counter for the next cycle (**step 16**).

This cycle ends when *j* takes on a value equal to the length of *l*.

In this cycle, an alteration counter *c* is initialized (**step 18**), the j-th benign present in *l* is loaded into the variable *b* (**step 19**) and the xor is executed, the result of which is loaded into the variable *a*, between the assumed benign and the actual malware (**step 20**).

At this point a bit from the xor result equal to 1 is randomly chosen (**step 21**). The search for the bit equal to 1 continues until: the bit randomly taken is equal to 0 or all the bits of *a* have not been verified.

The position of the bit taken represents the position of the *feature* subject to the alteration condition.

Once the bit has been identified, the condition for the alteration is set (**step 22**).

This condition allows the alteration when the *feature* is present only in the benign otherwise, we move on to the next malware.

If the condition is satisfied we move on to the *feature* alteration phase: the *feature* in the malware is forced to 1 (**step 23**).

The alteration counter is then incremented (**step 24**).

Once the alteration is complete, the class of the altered malware is predicted (**step 26**). If the result of the prediction, loaded into the p variable, is equal to 0 (**step 27**) the malware is added to the list of escaped malware and you can move on to the next malware (**steps 28- 29**); otherwise a new alteration is performed (**step 31**).

The alteration, on the same malware, is performed until the classification algorithm predicts the malware as benign or the alteration counter reaches the limit of perturbations (**step 30**), arbitrarily defined.

A complete view of the algorithm can be seen in fig. 3.7

Algorithm 2 Poisoning Hamming Distance

Input: $DatasetD, Test\_set, H, \beta, \delta$
Output: Evaded samples

$i \leftarrow 0$ {iteration counter}
repeat
  $x \leftarrow Test\_set[i]$
  $j \leftarrow 0$
  repeat
    $b \leftarrow D[j, 1:m]$
    if $b[m] == 0$ **then**
      $h \leftarrow hamming\_distance(x, b)$
      if $h \neq 0$ **then**
        $A[j][2] \leftarrow h$ {A is a 2 dimensional array where, 1st column has benign samples 2^{nd} column has the distance to x}
      end if
    end if
  until $j \leq |D|$
  sort A in ascending order of hamming distance
  $l \leftarrow A[i : \beta]$ {l is the 2-dimensional array of benign samples with the shortest distance to malware x}
  $j \leftarrow 0$
  repeat
    $c \leftarrow 0$
    $b \leftarrow l[j]$
    $a = b$XORx
    select a random number γ s.t $a[\gamma] = 1$
    if $b[\gamma] = 1$ and $x[\gamma] = 0$ **then**
      $x[\gamma] \leftarrow 1$
      $c \leftarrow c + 1$
    end if
    $p \leftarrow H\_predict(x)$ {testing classifier with evaded sample}
    if $p \leftarrow 0$ **then**
      $i \leftarrow i + 1$
      goto 2
    else if $c \leq \delta$ **then**
      goto 21
    end if
  until $j \leq |l|$
until $i \leq |Test\_set|$

Figura 3.7: Pseudo-code based on Hamming distance

Attack scenario based on K-Means Clustering

This algorithm divides the input data, based on their characteristics, into k clusters[34].

The clusters are generated, initially, by assuming k benign vectors as centroids, which will then be iteratively redefined. This step is carried out by calculating the Euclidean distance between the centroids and the remaining benign vectors.

Each vector is then assigned to the group with the least distant centroid.

The search for a new centroid is based on calculating the average of the elements present in the cluster.

The image in fig. 3.8 presents a vision of how this algorithm works:

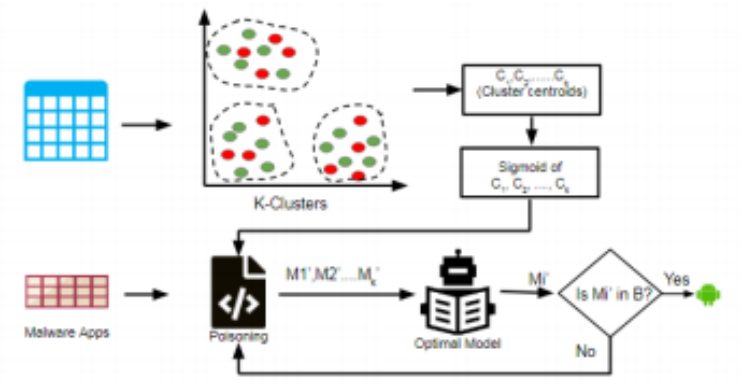


Figura 3.8: Poisoning attack based on K-Means Clustering

As with the previous ones, the algorithm requires specific inputs:

- Dataset (D)
- Test set, randomly chosen
- optimal model (H)

- number of clusters (δ)
- threshold for the sigmoid function (τ)

The algorithm can be summarized as follows: the first element is the definition of a function capable of identifying the correct centroids of the individual clusters and dividing the elements of the benign set into appropriate clusters (**steps 1 - 6**). The function will return the *cluser_means* list, then loaded into *centers*, containing the correct centroids.

An iteration counter i is then initialized (**step 7**), which will limit the execution of the entire algorithm.

Subsequently, an iterative cycle is started (**step 8**), which ends when i takes on a value equal to the limit value of the perturbation, arbitrarily imposed. This loop is used to convert the centroid to a binary value.

First, the i-th centroid is loaded into c (**step 9**) and the iteration counter j is initialized to 0 (**step 10**). An iterative cycle is then started (**step 11**) which ends when j takes on a value equal to the length of the centroid.

Inside the loop a value is calculated, loaded into s using the sigmoid function (**step 12**). If the calculated value is greater than the reference τ value (**step 13**) the j-th position of the centroid is forced to 1 (**step 14**); otherwise it is set to 0 (**step 16**).

Once the operation has been repeated on each position of the contraid, we move on to the next one (**step 18**).

Once the initial cycle is finished, the i iteration counter is reset (**step 20**) and a new cycle is opened. The latter ends when i takes on a value equal to the size of the test set.

The i-th malware is loaded into x (**step 23**) and the j iteration counter is initialized to 0 (**step 25**).

A second cycle is then started which ends when j takes on a value equal to the size of the list of centroids (**step 26**).

In the variable c the j-th centroid is loaded (**step 27**) and, subsequently, an xor is performed between the centroid and the malware considered (**step 28**), the result of which is loaded into the variable a.

At this point, a bit equal to 1 is randomly chosen from the result of the xor (**step 29**). The search for the bit equal to 1 continues until: the bit randomly taken is equal to 0 or all the bits of a have not been verified.

The position of the bit taken represents the position of the centroid bit and malware subjected to the condition for alteration.

Once the bit has been identified, the condition for the alteration is set (**step 30**). This condition allows the alteration when only the centroid bit, in that position, is equal to 1; otherwise, you move on to the next malware.

If the condition is satisfied, we move on to the *feature* alteration phase: the *feature* in the malware is forced to 1 (**step 31**).

The alteration counter is incremented (**step 32**) and the prediction on the altered malware is performed (**step 33**), the result of which is loaded into p.

If the prediction returns bit 0 (**step 34**) the malware is added to the *evaded_samples* list, containing the escaped malware, and the algorithm moves on to the next malware (**steps 35 - 36**); otherwise a new location is searched (**step 38**). The search for the new position has two blocking conditions: when the prediction returns bit 0 and when the alteration counter reaches the perturbation limit (**step 37**).

Once the blocking condition is reached, it is necessary to move on to the next malware (**step 41**).

A clearer view of the implementation is provided by the pseudo-code in fig. 3.9

Algorithm 3 Poisoning K-Means

Input: *Dataset : D, T est set, classifier : H, ♯ clusters : δ, Threshold : τ*

Output: Evaded Samples

```
1: Function K-Means clustering (Dataset D)
2: initially choose p data points from D as centroids
3: (re)assign each vector in D to the cluster to which it is
   more close to based on the mean value of the object in
   the cluster
4: update the cluster means
5: centres ← cluster_means
6: EndFunction
7: i ← 0
8: repeat
9:     c ← centers[i]
10:    j ← 0
11:    repeat
12:        s ← f[c[j]]
13:        if s > τ then
14:            c[j] = 1
15:        else
16:            c[j] = 0
17:        end if
18:        j ← j + 1
19:    until j ≤ |c|
20:        i ← i + 1
21: until i ≤ δ
22: i ← 0
23: repeat
24:    x ← T[i]
25:    j ← 0
26:    repeat
27:        c ← centers[j]
28:        a ← c XORx
29:        count ← 0
30:        select a random number γ s.t a[γ]=1
31:        if c[γ] = 1 and x[γ] = 0 then
32:            x[γ] ← 1
33:            count ← count + 1
34:            p ← H_predict(x)
35:            if p == 0 then
36:                j ← j + 1
37:                goto 26
38:            else if c < δ then
39:                goto 29
40:            end if
41:        end if
42:        j ← j + 1
43:    until j ≤ |centers|
44: until i ≤ |T|
```

Figura 3.9: Pseudo-code based on K-Means Clustering

3.4 Classificators

In this paragraph we will explain the characteristics of the classifiers used for the experimentation and the metrics used to evaluate their performance.
As mentioned previously, the experiment bases its results on the comparison between three different classifiers: Logistic Regression (LR), Support Vector Machine (SVM), Random Forest (RF).
To better understand the tools used for experimentation, it is appropriate to analyze their characteristics.

3.4.1 Logistic Regression

This classifier uses a classification algorithm based on historical data. This means that the greater the number of data analyzed, the greater the accuracy of its predictions.

Logistic Regression is a classification tool belonging to the family of supervised classifiers.

Using statistical methods, it generates a result representing the probability (P) that a given input value belongs to a certain class. Where P is a number between 0 and 1.

The Logistic Regression applied in the experimentation is called binomial, as

the result of the prediction can take on the two logical values 1 (malware) and 0 (benign).

Like all regression analyses, Logistic Regression is a predictive analysis, that is: it measures the relationship between the dependent variable (what you want to predict) and the independent variables (the known characteristics).

The membership probability is calculated using a ***logistic function***.

Finally, the prediction will be an analysis of the result of the function: if the function returns a probability with a value of at most 0.4, class 0 will be defined as the class to which it belongs; otherwise the class to which it belongs will be class 1.

The fig. 3.10 shows the steps needed to obtain the prediction:

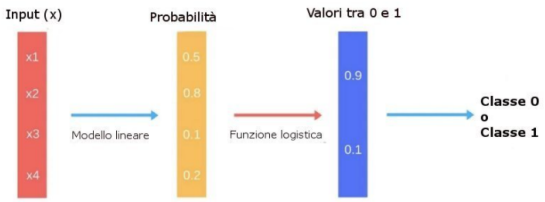


Figura 3.10: Prediction progress Logistic Regression

Logistic Regression, in mathematics, is represented by the following equation:

$$y = \frac{e^{(b_0+b_1*x)}}{1 + e^{(b_0+b_1*x)}} \tag{3.9}$$

Where:

- x is the input value
- b_0 and b_1 are the coefficients of the input values
- y is the output of the prediction

The estimation of the coefficients is carried out using the **maximum likelihood** method, which allows the search for values capable of minimizing the error of the probabilities seen by the model.
Again, logistic regression can be expressed in probabilistic terms, defining that:

$$P(X) = P(Y = 1|X) \tag{3.10}$$

As known, the probability of the input is the result of the logistic regression equation. Consequently it can be stated that:

$$P(X) = \frac{e^{(b_0+b_1*x)}}{1 + e^{(b_0+b_1*x)}} \tag{3.11}$$

Therefore, eliminating the exponential, we obtain that:

$$ln(\frac{p(x)}{1 - p(x)}) = b_0 + b_1 * x \tag{3.12}$$

Dove:

$$ln(\frac{p(x)}{1-p(x)}) \tag{3.13}$$

is called **logit function**, while

$$E(\frac{p(x)}{1-p(x)}) \tag{3.14}$$

in statistics, it is called **odds**

The logit function, representing the inverse of the logistic function, allows you to associate the probability of a prediction with an interval of real numbers. That is, it is the logarithm of the probability that the output belongs to one of the two possible classes.

The odds represents the ratio between the probability of an event and the probability that the event will not happen[35].

3.4.2 Support Vector Machine

The SVM classifier is an automatic supervised classification algorithm, useful for both classification and regression problems, whose performance is maximized in binary classification problems.

The idea behind the SVM is to identify a hyperplane capable of dividing the dataset into two classes.

Therefore, it can be said that at the basis of SVM there are three key concepts:

hyperplane a line, or plane, capable of dividing the entire dataset, as shown in fig. 3.11

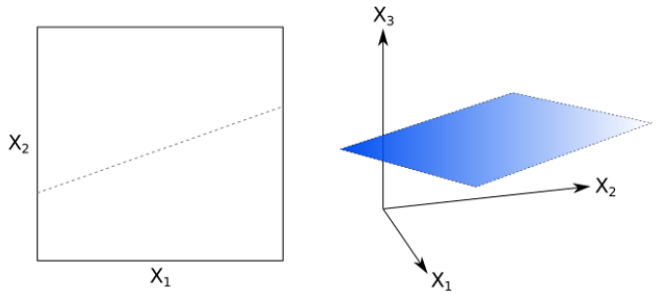


Figura 3.11: Hyperplane

support vectors critical elements of the dataset, are the data closest to the hyperplane, as shown in fig. 3.12. Critical because they are capable of altering the position of the hyperplane itself when they are modified or removed

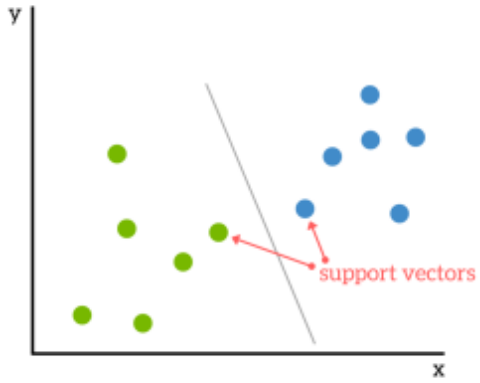


Figura 3.12: Support Vector

margin the distance between the support vectors, of different classes, closest to the hyperplane. The center of the edge intersects with the hyperplane, as shown in fig. 3.13.

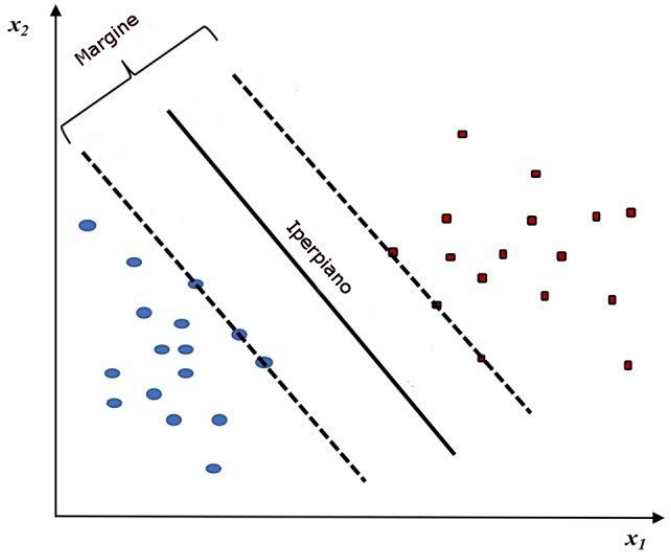


Figura 3.13: Margin

The objective of the SVM is to identify the hyperplane that best divides the data in the dataset. For this purpose two steps are necessary:

1. search for a **linearly separable hyperplane** capable of separating data belonging to different classes. If there are multiple possible hyperplanes, the one that guarantees the greatest number of support vectors is chosen.

2. if the hyperplane is not found, a **non-linear mapping** is applied, with which the training set is placed on a higher dimension (going from two to three dimensions, in the binary case). This ensures division.

The linearly separable hyperplane turns out to be an important concept for SVM, which, consequently, requires clarification.

Lienarily separable Hyperprian

It's a type of hyperplane capable of simplifying the division of the two classes.

It is based on the identification of multiple hyperplanes, theoretically infinite, capable of subdividing the dataset, as shown in fig.3.14.

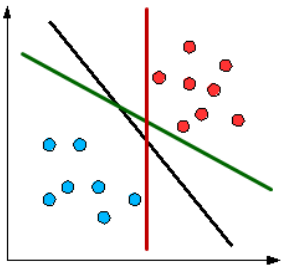


Figura 3.14: Optimal hyperplane search

Once the search is complete, we move on to identifying the optimal hyperplane, i.e. the one with the greatest distance from the data in the dataset, as shown in fig. 3.15.

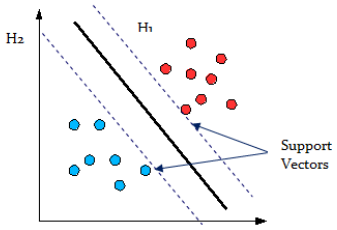


Figura 3.15: Optimal hyperplane

If new data is added, these will be immediately inserted into the group they belong to.

SVM linear model

In mathematics it is possible to define the operations performed by the SVM using a linear model.
In this model we see the hyperplane as the sum between a vector product and a coefficient

$$\overrightarrow{w}\ \overrightarrow{x} + w_0 = 0 \tag{3.15}$$

Where w represents the weight vector, x the feature vector and w_0 the bias. This equation can be written in more detail as

$$w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n = 0 \tag{3.16}$$

Therefore it is possible to state that on n dimensions, the hyperplane is a linear combination of all dimensions equal to zero. In a binary case, points above the hyperplane must meet the condition:

$$w_0 + w_1x_1 + w_2x_2 > 0 \tag{3.17}$$

while the points below the hyperplane must respect the condition:

$$w_0 + w_1x_1 + w_2x_2 < 0 \tag{3.18}$$

By also including margin limits you get two new conditions:

$$w_0 + w_1 x_1 + w_2 x_2 \geq 0 \rightarrow y = 1 \tag{3.19}$$

$$w_0 + w_1 x_1 + w_2 x_2 \leq 0 \rightarrow y = -1 \tag{3.20}$$

Where the class label (y) can only take the values -1 and 1.

The two conditions allow the identification of the margin boundaries.

Support vectors are those data that fall on such boundaries, fig. 3.16.

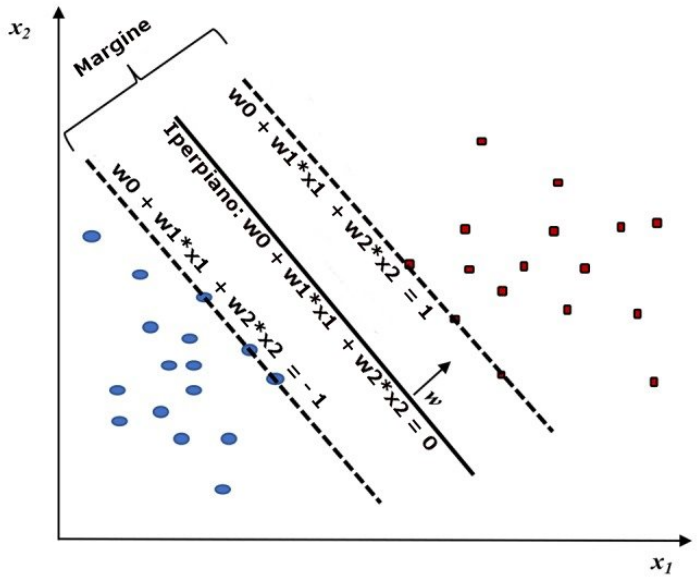


Figura 3.16: Margin coefficients

It must be said that if w represents the weight vector ||w|| represents the length of the vector, whose maximum dimension is:

$$\frac{1}{|w|} + \frac{1}{|w|} = \frac{2}{|w|} \tag{3.21}$$

From this calculation it follows that by minimizing w the optimal hyperplane is obtained[36].

3.4.3 Random Forest

Like other classifiers, Random Forest is a supervised classification algorithm. It is based on the combination of multiple decision trees which, individually, show little accuracy in prediction, accuracy which is increased when the predictions are combined together.

The result of the classifier is therefore nothing other than the average of the numerical result obtained from the various decision trees on which it is based, for a regression problem, or the class returned by the greatest number of decision trees for a classification problem.

The classification model is based on two fundamental concepts, from which the name random derives:

- random sampling of training data points when building trees
- random subsets of features considered when splitting nodes

Randomly sampling training data points while building trees

During the training phase, each decision tree trains on a random sample of data taken from the dataset. Ít is easy for a tree to use the same sample several times.

The idea behind this training is that using random samples has a higher variance than using randomly chosen samples. You also get, overall, a lower

variance than an ordered case, without increasing the distortion of the results.

The final result will then be obtained from the average of the individual decision trees, as shown in fig. 3.17.

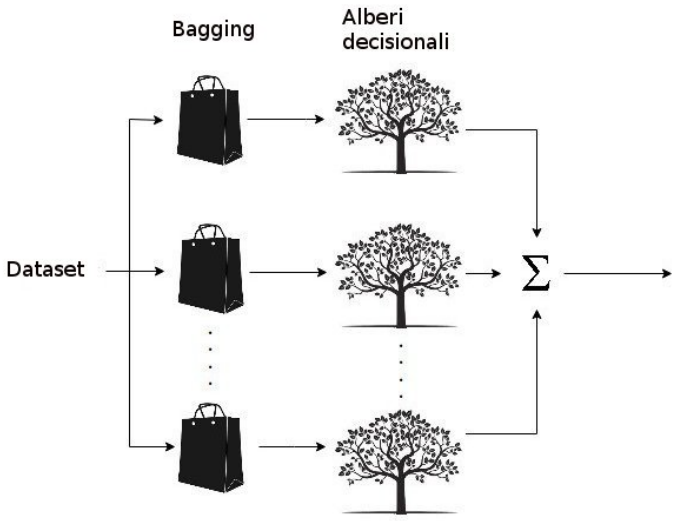


Figura 3.17: Prediction process - Random Forest

Causal subsets of features considered during node splitting

Another fundamental concept, at the basis of Random Forest, is that, in every decision tree, each node must be divided on the basis of randomly assumed characteristics.

Typically, a number of features equal to the square root of the total features is taken into consideration. For example, out of a total of 16 characteristics, only 4 will be taken into consideration, randomly assumed.

It can therefore be said that Random Forest combines a significant number

of decision trees, trained with sets of observations that differ slightly from each other, subdividing the nodes of the individual trees on the basis of a limited number of characteristics[37].

3.5 Classification metrics

So far we have discussed classifiers, their operating processes and their characteristics, but how is the classification and prediction output evaluated? In this regard, we usually apply various performance metrics, among which there are some fundamental ones:

Precision the ratio between the number of correct predictions of a class and the number of times that class is predicted. Calculated as

$$precision = \frac{true\_positive}{true\_positive + false\_positive} \tag{3.22}$$

For an example of precision you can think of an anti-theft device. It turns out to be very precise when it sounds only if it detects a thief. However, attention must be paid to the fact that the precision does not decrease if sometimes it does not detect the thief, the important thing and that it does not detect other living forms. This clarification is due to the fact that this metric is based exclusively on true or false positives, i.e. only when the algorithm predicts the event[38], as shown in fig. 3.18.

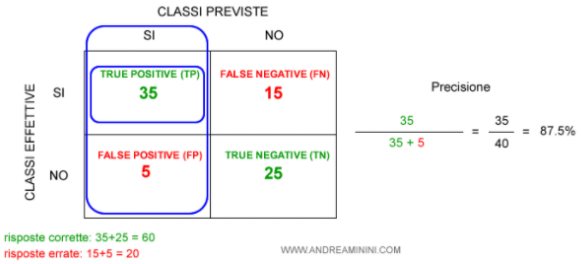


Figura 3.18: Precision

Recall measures the sensitivity of the model, through the ratio between the correct predictions, on a class, and the total number of times the event occurs.

$$recall = \frac{true\_positive}{true\_positive + false\_negative} \tag{3.23}$$

Returning to the example of the burglar alarm, the recall value varies based on the number of times the algorithm was able to identify the thief's entry into the house[38], as shown in fig. 3.19.

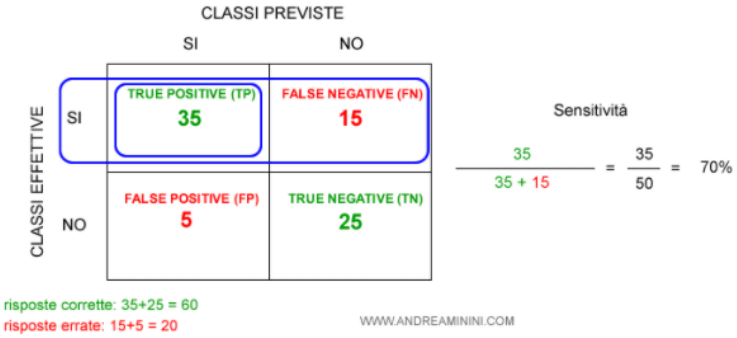


Figura 3.19: Recall

Accuracy represents the percentage of correctly classified instances. The problem with this metric is that it can be misleading when the data subjected to classification are not balanced[39].

Detection Rate indicates the ability of the machine learning algorithm to detect malware in this scenario. It presents the following formula:

$$detection\_rate = \frac{n\_test\_sample - n\_sample\_escaped}{n\_test\_sample} \tag{3.24}$$

The first three metrics are calculated both before and after the attack; the detection rate, however, is calculated exclusively afterwards.

3.6 Results analysis

The following results were obtained by running the algorithms twice.
The first execution allowed the production of the test set, containing the malware that the algorithm managed to alter.
The second execution was useful in identifying the escaped malware, based on the predictions obtained. In particular, this execution was repeated for each classifier.
Through the analyzes carried out it was possible to observe the Precision, Accuracy and Recall values of the three classifiers used: Logistic Regression (LR), Random Forest (RF) and Support Vector Machine (SVM). The analyzes were carried out using Weka software. From the analysis performed, taking the entire dataset as a training set (2416 samples), the results shown in table 3.1 were obtained.

Classifier	Precision	Recall	Accuracy (%)
LR	0.785	0.776	77.649
SVM	0.764	0.733	73.649
RF	0.881	0.875	87.459

Tabella 3.1: Analysis results before the attack

Instead, to build the models relating to the algorithms based on K-Means Clustering and Hamming distance, the entire dataset was used, first trained and subsequently re-evaluated on the test set; while for the algorithm based on the Euclidean distance the prediction was obtained with a split of the dataset into: 90% training set and the remaining 10% test set. Once the prediction was performed, it was possible to evaluate the classifiers in terms of accuracy and detection rate.

Specifically, in the case of a K-Means Clustering type poisoning attack, with a test set including 1120 malware, the results reported in the 3.2 table are obtained.

Classifier	Evaded Sample	Detection Rate
LR	820	0.27
SVM	802	0.28
RF	957	0.14

Tabella 3.2: K-Means Clustering poisoning results

However, in the case of a Hamming distance type poisoning attack, with a test set including 567 malware, it returned the results reported in the 3.3 table.

Classifier	Evaded Sample	Detection Rate
LR	321	0.43
SVM	328	0.42
RF	489	0.14

Tabella 3.3: Hamming distance poisoning results

Finally, in the case of a poisoning attack based on Euclidean distance with a test set including 242 altered malware, it returned the results reported in the table 3.4.

Classifier	Evaded Sample	Detection Rate
LR	139	0.43
SVM	163	0.33
RF	149	0.38

Tabella 3.4: Euclidean distance poisoning results

Capitolo 4

Conclusions

From the analyzes carried out, substantial differences can be seen between the analysis before the attack and that after the attack. It is immediately possible to identify how the analysis after the attack produced different results for each machine learning algorithm.

Since the outputs of the metrics and the evasion count differ for each algorithm and for each classifier, it is best to discuss them separately. At the end of the explanation of the differences it will be possible to define which algorithm proved to be more robust with respect to the attack suffered and with which classifier.

From the analyzes it is also possible to define the most effective poisoning algorithm.

In particular, a poisoning algorithm is defined as more effective if it presents the greatest number of escaped malware; while, a machine learning algorithm, subjected to a poisoning attack, is defined as more robust if it has a higher detection rate.

From the point of view of the robustness of the algorithms, with respect to the attack carried out, it is necessary to have a complete vision, based on the detection rate.

As regards the algorithm based on K-Means Clustering, a detection rate of: 0.27 with the LR classifier, 0.28 with the SVM and 0.14 with the RF is noted. It can therefore be noted that the model built with the RF classifier is the least robust.

As regards the algorithm based on the Hamming distance, we note a detection rate equal to: 0.43 with the LR, 0.42 with the SVM and 0.14 with the RF. In this case the attack is more effective in the case of the model built with the RF classifier.

Finally, regarding the algorithm based on the Euclidean distance, we note a detection rate equal to: 0.43 with the LR, 0.33 with the SVM and 0.38 with the RF. In this case, the least robust machine learning algorithm is the one built with the SVM classifier.

On the basis of all the results obtained, it is possible to define which algorithm proves to be more robust with respect to a poisoning attack.

Although the differences between the outputs of the algorithms for each classifier are substantially different, it is easy to see that the attacks that demonstrated greater effectiveness were those based on Hamming distance and K-Means Clustering, in the analysis using the RF classifier, with a percentage of malware escaped equal to 86%; while the algorithms based on the Euclidean distance and the Hamming distance, both with the LR classifier, proved to be the most robust algorithms, as they had the highest detection rate.

Bibliografia

[1] www.kaspersky.com/blog/insecure-android-devices/10296.

[2] www.mcafee.com/enterprise/en-in/threat-center/global-threatintelligence-technology.html.

[3] https://gs.statcounter.com/os-market-share/mobile/worldwide.

[4] Sai Manoj P D Avesta Sasan Setareh Rafatirad Hossien Sayadi, Nisarg Patel and Houman Homayoun. Ensemble learning for effective run-time hardware-based malware detection: A comprehensive analysis. *ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018.

[5] Monica Kumaran and Wenjia. Lilightweight malware detection based on machine learning algorithms and the android manifest file. *Published in: I (URTC)*, 2016.

[6] Battista Biggio Davide Maiorca Daniel Arp Konrad Rieck Igino Corona Giorgio Giacinto Ambra Demontis, Marco Melis and Fabio Roli. Yes, machine learning can be more secure! a case study on android malware detection. *arXiv:1704.08996v1 [cs.CR]*, 2017.

[7] Justin Sahs and Latifur Khan. A machine learning approach to android malware detection. *European Intelligence and Security Informatics Conference*, 2012.

[8] Nathan S. Eli (Omid) David and Netanyahu. Deepsign: Deep learning for automatic malware signature generation and classification. *International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2015.

[9] Luke Metz Alec Radford and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *ICLR*, 2016.

[10] Vijay Laxmi Vijay Ganmoor Manoj Singh Gaur Parvez Faruki, Ammar Bharmal and Mauro Conti. Android security: A survey of issues, malware penetration, and defenses. *IEEE Communication Surveys & Tutorials, vol. 17, NO. 2*, 2015.

[11] Lingling Fan Shuang Hao Lihua Xu Haojin Zhu Sen Chen, Minhui Xue and Bo Li. Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. *Elsevier Computers Security, Volume 73*, 2018.

[12] Y. Chen V. Rastogi and X. Jiang. Droidchameleon: Evaluating android anti-malware against transformation attacks. *ACM SIGSAC Symposium on Information, Computer and Communications Security, ASIA CCS '13*, pages 329–334, 2013.

[13] Francesco Mercaldo Gerardo Canfora, Andrea Di Sorbo and Corrado Aaron Visaggio. Obfuscation techniques against signature based detection: a case study. *Mobile Systems Technologies Workshop (MST)*, 2016.

[14] Gianluca Dini Andrea Saracino, Daniele Sgandurra and Fabio Martinelli. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, 2016.

[15] Francesco Mercaldo Guangwu HuEmail Xi Xiao, Shaofeng Zhang and Arun Kumar Sangaiah. Android malware detection based on system call sequences and lstm. *Multimedia Tools and Applications, vol. 78, n. 4, Springer*, pages 3979–3999, 2019.

[16] Abdul Wahid Amjad Mehmood Houbing Song Saba Arshad, Munam A. Shah and Hongnian Yu. Samadroid: A novel 3-level hybrid malware detection model for android operating system. *IEEE. Translations, vol. 6*, 2018.

[17] https://ieeexplore.ieee.org/abstract/document/8782574.

[18] https://dl.acm.org/doi/abs/10.1145/2046684.2046692.

[19] ttps://arxiv.org/abs/1611.01236.

[20] http://proceedings.mlr.press/v97/guo19a.html.

[21] https://ieeexplore.ieee.org/abstract/document/1004360/authorsauthors.

[22] https://www.mdpi.com/2504-4990/2/4/30.

[23] https://pralab.diee.unica.it/en/PoisoningAttacks.

[24] https://analyticsindiamag.com/what-is-poisoning-attack-why-it-deserves-immediate-attention/.

[25] https://openai.com/blog/adversarial-example-research/.

[26] Xuan Lu Tao Xie Qiaozhu Mei Feng Feng Xuanzhe Liu, Huoran Li and Hong Mei. Mining behavioral patterns from millions of android users. *IEEE Transactions on Software Engineering*, 2017.

[27] Zhushou Tang Lihua Xu Sen Chen, Minhui Xue and Haojin Zhu. Stormdroid: A streaminglized machine learning-based system for detecting android malware. *ASIA CCS'16, Xi'an, China*, 2016.

[28] Yanjun Qi Weilin Xu and David Evans. Automatically evading classifiers a case study on pdf malware classifiers. *NDSS '16,*, pages 21–24, 2016.

[29] Battista Biggio Daniel S. Yeung Fei Zhang, Patrick P. K. Chan and Fabio Roli. Adversarial feature selection against evasion attacks. *IEEE Transactions on Cybernetics, VOL. 46, NO. 3*, 2016.

[30] https://androzoo.uni.lu/api_doc.

[31] https://github.com/Saket-Upadhyay/Android-Permission-Extraction-and-Dataset-Creation-with-Python.

[32] https://github.com/skylot/jadx.

[33] https://www.unb.ca/cic/datasets/andmal2020.html.

[34] Seth Rogers Kiri Wagstaff, Claire Cardie and Stefan Schroedl. Constrained k-means clustering with background knowledge.

[35] https://www.lorenzogovoni.com/regressione-logistica/.

[36] https://www.lorenzogovoni.com/support-vector-machine/.

[37] https://www.lorenzogovoni.com/random-forest/.

[38] https://www.andreaminini.com/ai/machine-learning/la-differenza-tra-precision-e-recall.

[39] https://docs.microsoft.com/it-it/azure/machine-learning/classic/evaluate-model-performance.