



POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

**Interpretable Machine Learning for
malware characterization and
identification**

Supervisor

prof. Antonio Lioy
prof. Andrea Atzeni

Candidate

Filippo GIOVAGNINI

DECEMBER 2023

Summary

Malware remains a pervasive and evolving threat to cybersecurity. The rapid proliferation of new malware variants requires innovative solutions for timely identification and classification. This thesis presents a comprehensive study focused on the development of a machine learning model to address this challenge. The primary objective of this research is to create a machine learning model for malware identification and classification that prioritizes interpretability. The model aims to provide clear insights into the decision-making process, allowing security analysts to understand the features and characteristics that drive their classifications. This approach is essential for building confidence in automated cybersecurity systems. Interpretable machine learning techniques improve transparency and trust in classification models, facilitate model validation and help users to identify and address potential biases or errors. Interpretable models enable collaboration between domain experts and data scientists, as they can collectively analyze and refine the models based on the interpretable insights. Moreover, interpretability provides valuable explanations for end-users, allowing them to comprehend and act upon the model's predictions with confidence. The literature on this topic is still evolving, and there is some confusion about how to assess the level of interpretability of a model and how to classify the various interpretable techniques. Therefore, I first conducted extensive research on the state-of-the-art of current interpretable machine learning techniques in the wild in order to create a coherent taxonomy for defining and classifying these techniques. I then delved into the interpretable techniques applied to malware identification and categorization. I always pay more attention to the interpretability aspects than to the performance aspects. I analyzed the first studies on this topic in detail and then moved on to the most recent and significant ones. The majority of the studies were conducted avoiding neural networks due to their computational cost, preferring traditional ML algorithms such as Random Forest, Gaussian Naive Bayes, Decision Tree, K Nearest Neighbour and Support Vector Machine. Nevertheless, I wanted to harness the power of neural networks and found a very promising project. The authors used an algorithm that had never been used before in this field, the Grad-CAM algorithm. The Grad-CAM is a relaxed generalization of the CAM approach, which produces Class Activation Map by using a global average pooling before the softmax layer. It does not require any modification to the model architecture, thus this approach is not limited to a specific model but can be generalized and applied to any other convolutional neural network. At the end, it produces a heatmap that highlights the decisive area of the input image utilized for the prediction. In fact, they first transformed an Android application into images that were used to train a convolutional neural network, and then applied the Grad-CAM algorithm to these images for interpretability purposes. The real advantage would be to apply a reverse engineering phase at the end of these project steps. This phase should automate the process of converting the image back into memory and thus into code, so that the security analyst could have immediate access to the suspicious code. This is what I did in my thesis, briefly following the works done by these authors. The process began with the collection of a large and diverse dataset of Android malware samples, along with a pool of benign applications. It was apparent, however, that a

significant portion of the datasets available online primarily provided descriptive reports or meta-data about these applications, rather than granting access to the raw .apk files themselves. This dichotomy in the data offerings necessitated a more intricate and exhaustive search for a dataset that aligns with the specific requirements of my research methodology. At the end I composed my dataset taking benign samples from the *CIC-AndMal2017* dataset and Android malware from the *Malradar* dataset. I first tried to use this dataset with the deep neural network developed in that project, but the results were not very satisfying in terms of precision and recall, so I built another model exploiting the InceptionV3 neural network developed and pre-trained by Google, adding some layers to the base model to perform classification. InceptionV3 is characterized by its extensive use of inception modules. These modules are composed of a series of convolutional and pooling layers of different sizes that work in parallel and only at the end of the module are merged together, which allows the network to capture features at multiple scales and resolutions simultaneously. Inception modules are designed to improve the network's ability to recognize complex patterns in the input data. With this new model, the performance of the network reached very promising results with an overall recall of 97% and I decided to use it in my project. The reverse engineering phase was carried out by creating a legend file while generating the application images. Before doing it the all dataset applications had to be decompiled to extract smali code, that is more resistant to obfuscation than Java source code. This format allows extracting the opcodes for each class and method of the malware. Each opcode was mapped to an ASCII character, and then the characters were stored in a text file, preserving the order in which they appear in the code and keeping track of where classes begin and end, storing this information in the legend text file, which will make it possible to reverse the process from an image area to the corresponding code. We will always be able to identify a specific pixel on the image with its corresponding opcode and line of smali code, using the legend text file. Comparing this file with the Grad-CAM heatmaps we can trace back to the exact opcodes and classes of the source code that contribute the most for the predictions. The results are very significant, as we now have immediate access to the precise classes to which the suspect code belongs and can perform various studies on it, from manual revision to more automated analysis. An example of a partially automated analysis I did is to take the source code of the most relevant classes and pass it to ChatGPT, asking it to tell me the reason for that choice. The answers were particularly shrewd and helpful in immediately understanding the code and identifying the most suspicious passages. They also illustrated what were the possible threats that could be carried out with that code. In the future, it is possible to improve this work by adapting the project to receive applications from operating systems other than Android, and by developing a fully automated analysis tool that takes the relevant classes of the tested applications and passes them to ChatGPT, without human interaction in this process.

Acknowledgements

Contents

1	Introduction	8
1.1	Motivation	8
1.2	Problem statement and research questions	10
2	Background	12
2.1	Overview of ML	12
2.1.1	ML applied in the Cybersecurity field	13
2.2	Types of ML algorithms	14
2.2.1	Supervised learning	14
2.2.2	Unsupervised learning	19
2.2.3	Semi Supervised learning	21
2.2.4	Reinforcement learning	22
2.3	Overview of Malware	23
2.4	Types of malware	24
2.5	Malware detection techniques	26
2.5.1	Signature based	27
2.5.2	Behavioral based	28
2.5.3	Heuristic based	29
2.6	Malware analysis	30
2.6.1	Static analysis	31
2.6.2	Dynamic analysis	31
2.6.3	Disassemblers	32
2.7	Anti-detection techniques	32
2.7.1	Anti-disassembly	33
2.7.2	Code obfuscation	33
2.7.3	Obfuscated malware	34
3	Related work	35
3.1	Definition of interpretability in ML	35
3.1.1	Properties of explanations	35
3.2	Interpretable machine learning classification and techniques	37

3.2.1	Based on the scope	37
3.2.2	Model-specific methods	38
3.2.3	Model-agnostic methods	39
3.2.4	Intrinsic methods	41
3.2.5	Post hoc methods	42
3.3	Projects	42
3.3.1	NODENS	42
3.3.2	PAIRED	46
3.3.3	TAMI	49
3.4	Summary	53
4	Dataset	54
4.1	Overview of common datasets	54
4.2	Data characteristics and statistics	60
5	Experimental Design and Methodology	63
5.1	From <i>.apk</i> to <i>.png</i>	64
5.2	CNN model evaluation	66
5.3	Grad-CAM application and reverse engineering of the results	69
6	Results and Analysis	71
6.1	CNN performances	71
6.1.1	Basic CNN	71
6.1.2	InceptionV3 CNN	73
6.2	Results analysis	75
7	Conclusions	81
7.1	Limitations and Future works	82
A	User manual	83
A.1	Instructions	84
B	Developer manual	86
	Bibliography	91

Chapter 1

Introduction

1.1 Motivation

Cybersecurity involves safeguarding computer networks, devices, and data from unauthorized access or misuse. It also encompasses the preservation of information confidentiality, integrity, and availability. This can be achieved through various means, such as implementing security measures at the application, network, host, and data levels. As more and more people rely on the internet, the number of internet-connected systems has grown. While this has led to increased usage of the internet, it has also made it a target for cyber criminals who use increasingly sophisticated methods to attack. In 2021, the number of internet users worldwide rose by 0.3 billion [1] and global cyber attacks increased by 42%, according to the 2022 Cyber Trends Report [2] and more than 87.6 million new malwares have been reported [3]. As the COVID-19 pandemic has created new social and economic norms, the need for a safe and secure cyberspace has become even more critical. This highlights the importance of establishing a stable cyber security system to maintain the privacy, accessibility, and integrity of information transmitted over the internet [4]. However, traditional signature-based and rule-based cybersecurity measures are facing challenges in keeping up with the growing amount of information on the internet [5]. At the same time, cybercriminals are constantly finding new ways to outsmart law enforcement, using technological advances like Artificial Intelligence to make their attacks more sophisticated and effective [6]. As a result, researchers are turning to AI-based approaches, particularly ML and DL, to enhance cybersecurity performance. Although AI techniques have shown promising results in areas such as intrusion detection, spam email filtering, botnet detection, and fraud detection, they also have limitations. Malware detectors based on machine learning (ML) have been becoming more popular in malware detection because malware and its variants are being increasingly threatening. With the increasing complexity of cyber threats and the vast amount of data generated by modern networks, traditional machine learning techniques have become increasingly difficult to interpret and understand. Most of the existing ML models are black-box, which makes their prediction results undependable, and therefore need further interpretation in order to be effectively deployed in the wild. A black box machine learning model is a mathematical representation that is either excessively intricate for humans to comprehend or intentionally kept confidential from the proprietary of the algorithm, making it impossible to grasp its internal mechanisms [7]. AI models can be complex and lack interpretability, which has been a concern for the European Union's General Data Protection Regulation [4].

For these reasons, in the last few years, there has been a growing interest in the field of interpretable machine learning (IML) and its potential applications in cybersecurity. This has led to a growing need for more transparent and explainable ML algorithms that can help security professionals better understand the underlying mechanisms of cyber attacks and make more informed decisions. From the mid 2018 the interest around this topic, judging by the frequency of the search terms *Interpretable Machine Learning* and *Explainable AI* on Google, has really took off (figure 1.1)

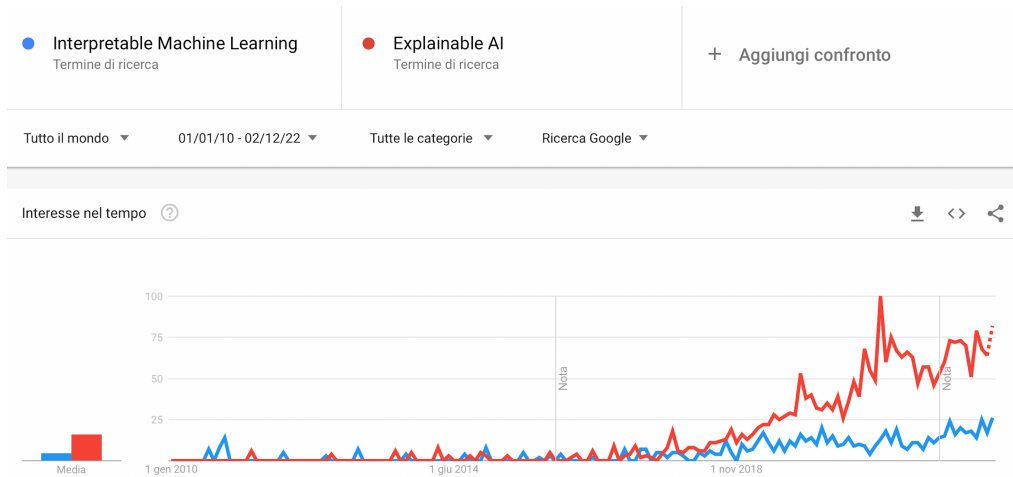


Figure 1.1. Google search trends for *Interpretable Machine Learning* and *Explainable AI* (accessed December 02, 2022).

Artificial intelligence (AI) techniques, such as machine learning and deep learning (DL) algorithms, have emerged as important tools for delivering advanced and intelligent cybersecurity services and management. The use of artificial intelligence (AI) in the field of cybersecurity encounters various challenges due to the limitations of AI-based approaches. These challenges include difficulties in accessing cybersecurity data, vulnerability to adversarial attacks targeting AI models, and concerns regarding ethics and privacy. One specific limitation that requires greater attention is exactly the black-box nature of AI models when they are integrated into the cybersecurity domain. The black-box nature of AI models in cybersecurity poses challenges as the decisions they generate lack transparency and justification. This lack of understanding makes it difficult for people to comprehend how these decisions are reached and leaves cyber defense mechanisms vulnerable to information breaches and AI-based cyber threats. In response to these drawbacks, IML has emerged as a solution to address the growing black-box issue in AI. IML enables users and experts to understand the logical explanations and underlying data evidence behind the results produced by AI-based statistical models by providing interoperability and transparency. In conclusion, the motivations for applying IML to the field of cyber security can be summarized as follows:

- **Establishing trust:** integrating IML is crucial for building trust, as it enables transparency and enhances understanding of decision models in cyber security. This transparency helps stakeholders trust the outcomes and reasoning behind cybersecurity-related decisions.
- **Regulatory compliance:** another motivation for adopting IML in cyber security is to meet the requirements of new regulations and laws such as the General Data Protection Regulation (GDPR). These regulations emphasize the need to provide explanations to the broader society, including in the realm of cyber security.
- **Justice, social responsibility, and risk mitigation:** applying IML in cyber security is driven by the recognition that protecting cyber security often involves addressing significant social problems and potential risks to human lives. It extends beyond mere cost-benefit calculations and emphasizes the importance of justice, social responsibility, and risk mitigation.
- **Addressing biases and effectiveness misconceptions:** biases within cyber security systems and misunderstandings regarding their effectiveness are key factors motivating the use of IML. For example, biased training data can undermine the credibility of model outputs, particularly when working with neural networks that learn patterns from the training data.
- **Providing justifications:** IML offers the ability to provide compelling justifications for decisions made by the cyber security system. This capability ensures that the created defensive

mechanisms are not only fair and socially responsible in their decision-making but also capable of defending their results through well-founded justifications.

1.2 Problem statement and research questions

If a machine learning model performs well enough and has an acceptable predictive performance, why do we not just trust the model and disregard why it made a certain decision? Doshi-Velez and Kim answer this question by stating that *the problem is that a single metric, such as classification accuracy, is an incomplete description of most real-world tasks* [8]. Nevertheless, it is worth taking into account that, although the relevance and importance of interpretability has clearly been explained, not all ML systems require interpretability, as there are situations where being able to provide high predictive performance is enough [8], with no need for explaining decisions. However, in the majority of cases, it is also very important to know why a certain prediction was made because a correct prediction only partially solves the original problem, for example for everything that concerns high-stake decision, which also includes malware detection and classification [9].

Interpretable machine learning (ML) refers to the ability to understand and explain the inner workings and decisions of ML models in a transparent and understandable manner [10]. The motivations for interpretability have been widely discussed and can be summarized into three key reasons: audit, validation, and discovery [11], as it is also illustrated in figure 1.2.

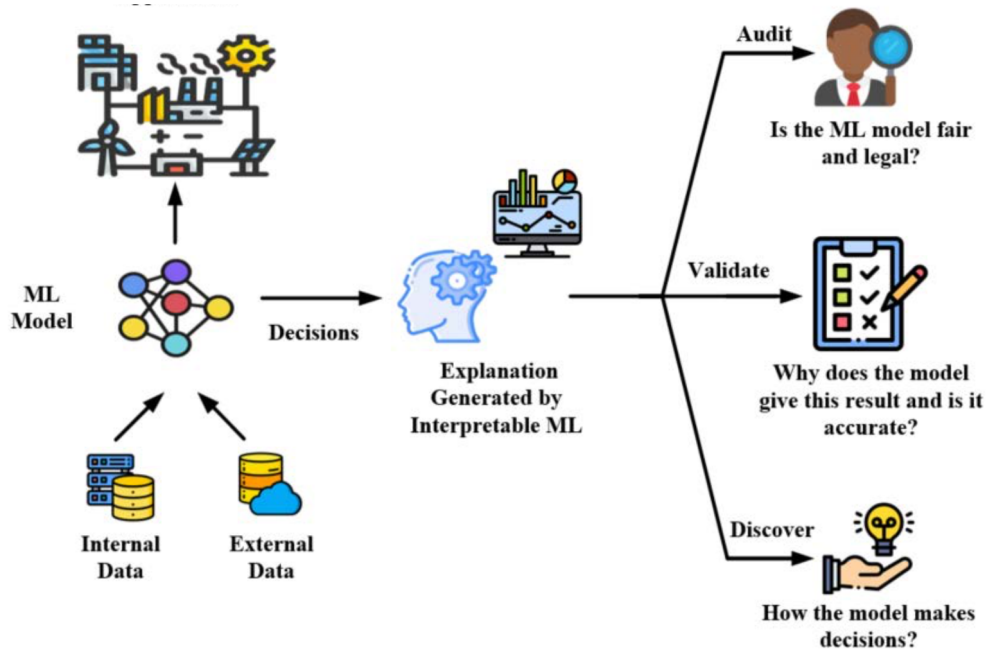


Figure 1.2. Illustration of an Interpretable Machine Learning scheme (source: [10]).

1. **Audit:** Interpretable ML methods are important for auditing the fairness and ethics of ML models, particularly in human-oriented applications like credit risk assessment and medical screening. ML models can inherit biases from the training data, leading to automated injustices. Interpretable ML helps quantify and reduce these biases by providing explanations. The need for interpretability is recognized in laws and regulations, such as the European Union’s General Data Protection Regulation (GDPR), which requires interpretability when ML models make decisions about individuals.
2. **Validation:** Interpretable ML plays a crucial role in verifying the safety and reliability of ML algorithms. ML models used in critical systems require rigorous verification to gain

trust. Most ML models are black boxes, making it challenging to ensure they function as expected. Overfitting, where a model performs well on training data but fails in practice, is a common concern. Interpretability aids in validating ML models by providing insights into potential overfitting. Researchers can assess if the model has focused on non-informative features in the raw data, which are typically difficult to understand.

3. Discovery: Interpretable ML models facilitate understanding the reasons behind model outputs and discovering correlations between different factors. This is valuable as it provides meaningful knowledge and can even contribute to the development of new theories. By interpreting the model, researchers can gain insights into the relationships and influences among variables, helping to uncover patterns and insights that might not be apparent otherwise.

This thesis presents an exhaustive research on the topic of IML applied in the cybersecurity field to find out the best performing model. For evaluate a model I will attribute a strong importance to the interpretability factor, rather than to the ordinary metrics usually used to evaluate a ML algorithm. This is because already exist a lot of tools that performs automatic malware detection and malware classification [12] [13] and they are also very powerful in terms of performance, but they work black-boxes. This thesis presents a tool mainly built upon two previous works on ML-based Android malware classification that provide also a thorough explanation of their results. The first project, presented in paper [14], aim's is to provide an explicable method. Specifically, the aim is to provide interpretability for the model's decision-making process in the context of malware detection and family identification in Android. To achieve this goal, the proposed approach involves the utilization of class activation maps, typically employed for debugging deep neural networks, while this time it will be adapted to visualize and shed light on the reasoning behind the model's decisions. For this task, the Gradient-weighted Class Activation Mapping (Grad-CAM) algorithm [15] is exploited. The Grad-CAM algorithm was never explored before in the field of cybersecurity and this could be the first of many studies that follow this promising thread. The second project, called **SCIBA** and presented in paper [16], conducted by the same authors of the [14], develops the former work slightly modifying the Deep Neural Network (DNN) and main of all by adding a reverse engineering phase at the end. This part is crucial for my interpretability purposes, because it takes the results of the Grad-CAM function and, from that, goes back to the original source code, permitting a security analyst ora generic end user to debug directly the suspicious code. This thesis tested this approach with a modified version of the Mal-Radar [17] dataset, where some malware families were deleted and a set of Benign samples were added.

Chapter 2

Background

2.1 Overview of ML

ML is a subfield of artificial intelligence that focuses on the development of algorithms and models that enable computers to learn from data and make predictions or take actions without being explicitly programmed [18]. It is concerned with building systems that can automatically extract patterns and relationships from data, learn from examples or experiences, and generalize that knowledge to new or unseen instances. ML aims to address complex problems that are difficult to solve using traditional programming approaches. Unlike conventional programming, where rules and instructions are explicitly defined by humans, ML algorithms learn from data to discover hidden patterns and make informed decisions. The primary goal of ML is to develop computational models that can generalize and adapt to new situations or data, allowing computers to perform tasks more accurately and efficiently. At the heart of ML is the concept of learning from data. This process involves the utilization of statistical techniques and computational methods to automatically detect patterns and relationships within a given dataset. Through iterative processes, the algorithms adjust their internal parameters based on the patterns observed in the data, enabling them to improve their performance over time. By learning from the data, the algorithms can make predictions, classify new instances, or take actions based on the learned knowledge. Automated pattern extraction and generalization are fundamental aspects of ML that distinguish it from traditional programming. The ability to automatically identify patterns and generalize from data allows ML models to handle complex, real-world problems that may not have explicit rules or solutions [19]. This flexibility and adaptability are key advantages of ML over conventional programming approaches. These aspects enable ML models to identify hidden trends, correlations, and dependencies within the data. For example, in image recognition, a model can learn to recognize certain objects or features by extracting visual patterns from a large dataset of labeled images. This capability has been leveraged in various applications such as face recognition, object detection, and autonomous driving. Generalization is another essential aspect of ML, allowing models to apply their learned knowledge to unseen data or new instances. By capturing the underlying patterns and relationships in the training data, ML models can make accurate predictions or take appropriate actions on previously unseen examples. For instance, a spam email filter trained on a large dataset of labeled emails can generalize its knowledge to classify new incoming emails as spam or legitimate. Automated pattern extraction and generalization have significant implications across various domains. In healthcare, ML models can analyze medical records, genetic data, and clinical images to assist in disease diagnosis, personalized treatment recommendations, and drug discovery [20]. In finance [21], ML algorithms can analyze market data, identify trading patterns, and make predictions on stock prices or market trends. In natural language processing, ML techniques enable machines to understand and generate human language, facilitating tasks such as language translation, sentiment analysis, and chatbot interactions.

2.1.1 ML applied in the Cybersecurity field

ML has emerged as a crucial tool in the field of cybersecurity, addressing the ever-evolving threat landscape and providing effective defenses against malicious activities. Its ability to analyze vast amounts of data, identify patterns, and detect anomalies has made it an invaluable asset in detecting and mitigating cyber threats. One of the primary applications of ML in cybersecurity is in threat detection [22]. By analyzing network traffic, the algorithms can learn normal patterns of behavior and identify anomalies that may indicate a cyber attack. These algorithms can detect network intrusions, unauthorized access attempts, and abnormal user behaviors in real-time, enabling prompt response and mitigation. These techniques are also extensively used in intrusion detection and prevention systems (IDS/IPS) [23]. By analyzing system logs, network packets, and other relevant data, ML algorithms can identify known attack patterns and automatically block or alert security teams about potential threats. This helps in fortifying network defenses and preventing unauthorized access. Vulnerability assessment is another critical area where ML plays a significant role. It helps to analyze system configurations, application codes, and patch history to identify vulnerabilities that could potentially be exploited by attackers. By automating the identification of weaknesses and prioritizing their remediation, ML enhances the efficiency of vulnerability management processes and reduces the organization's attack surface [24]. However, the field of cybersecurity presents unique challenges that can limit the effectiveness of machine learning approaches [25]. Here, I briefly explore some of these challenges and limitations:

- **Imbalanced Data and Class Imbalance:** one challenge in cybersecurity is dealing with imbalanced datasets, where the number of instances in different classes is significantly skewed. For instance, in malware detection, the number of benign files usually far exceeds the number of malware samples. Class imbalance can lead to biased models, where the classifier tends to favor the majority class. It can also result in low sensitivity in detecting the minority class, which is typically the class of interest (e.g., malware). Addressing class imbalance requires careful preprocessing techniques such as oversampling, undersampling, or the use of specialized algorithms designed for imbalanced data.
- **Evolving Threat Landscape:** the cybersecurity landscape is constantly evolving, with new types of threats emerging regularly. Traditional machine learning models often struggle to adapt to these new threats as they rely on historical patterns. Zero-day attacks and polymorphic malware, for example, can bypass traditional signature-based detection systems. Machine learning models trained on older data may struggle to generalize to new and previously unseen attack vectors. Continuous monitoring, updating of training datasets, and the use of adaptive machine learning algorithms are some approaches to tackle the challenge of the evolving threat landscape.
- **Adversarial Attacks:** adversarial attacks pose a significant challenge to machine learning models [26]. Attackers can intentionally manipulate or perturb input data to deceive the models and evade detection. This kind of attacks can be crafted to introduce subtle modifications that are imperceptible to humans but can mislead machine learning models. Adversarial training and the use of robust machine learning techniques that can withstand such attacks are important areas of research to enhance the resilience of cybersecurity systems.
- **Data Quality and Representativeness:** the quality and representativeness of training data are paramount in building effective cybersecurity models. Machine learning models heavily rely on labeled datasets for training. However, in the field of cybersecurity, acquiring high-quality and diverse datasets can be challenging. Anomalies in the data, such as label noise or missing data, can affect the performance of the models. Moreover, the representativeness of the training data is crucial to ensure that the models learn from a wide range of attack scenarios. Efforts should be made to collect comprehensive and representative datasets to improve the robustness and generalizability of machine learning models in cybersecurity.
- **Scalability and Real-Time Processing:** cybersecurity systems often require real-time processing capabilities to detect and respond to threats promptly. However, some machine learning algorithms can be computationally expensive and may not scale well to handle large volumes of network traffic or log data in real-time. Efficient feature extraction techniques,

model optimization, and the use of distributed computing frameworks can help address the scalability challenge, enabling the application of machine learning in real-time cybersecurity systems.

- **Interpretability:** the interpretability of machine learning models is a critical concern in cybersecurity. Black-box models, such as deep neural networks, may achieve high accuracy but lack interpretability. Understanding the decision-making process of a model is crucial in cybersecurity to identify vulnerabilities, assess model trustworthiness, and explain the reasons behind certain predictions [27]. Interpretable machine learning techniques will be widely studied later in this thesis.

2.2 Types of ML algorithms

ML algorithms can be classified into different categories based on their learning approach and the nature of the task they aim to solve. Understanding these different types is essential for selecting the appropriate algorithm for a given problem and this section presents several types of ML algorithms, including supervised learning algorithms, unsupervised learning algorithms, reinforcement learning algorithms, semi-supervised learning algorithms, and transfer learning algorithms. Machine learning algorithms play a crucial role in enabling computers to learn from data and make predictions or decisions without explicit programming. These algorithms are designed to automatically discover patterns, relationships, and insights within datasets, allowing them to generalize from existing examples to new, unseen instances. By understanding the characteristics and capabilities of different machine learning algorithm types, we can better comprehend their applicability and select the most suitable algorithms for the specific research problem.

2.2.1 Supervised learning

Supervised learning algorithms form a fundamental category within machine learning. They learn from labeled data, where the desired output or target variable is provided for each input example. By generalizing from the labeled examples, these algorithms aim to make predictions or classify new, unseen instances accurately [28]. Supervised learning algorithms are widely used in various applications, ranging from image recognition to sentiment analysis.

Decision Tree

Decision trees are popular and intuitive models used for both classification and regression tasks in machine learning. They represent a tree-like structure where each internal node represents a decision based on a feature, and each leaf node corresponds to a predicted class or value. Decision trees are particularly useful for capturing non-linear relationships between features and the target variable, and they offer interpretability and ease of understanding [29]. The decision-making process in a decision tree involves traversing the tree based on the feature values of the input instance until a leaf node is reached. At each internal node, a decision or test is made based on the value of a specific feature. The decision splits the data into separate branches, leading to different child nodes corresponding to different outcomes or decisions. This process continues recursively until a leaf node is reached, where a predicted class or value is assigned. To construct a decision tree, various algorithms can be used, such as the ID3 (Iterative Dichotomiser 3), C4.5, or CART (Classification and Regression Trees) algorithms [29]. These algorithms employ different criteria to determine the optimal features and split points at each node. One commonly used criterion is information gain, which measures the reduction in uncertainty achieved by splitting the data based on a particular feature. Information gain is based on the concept of entropy, which represents the impurity or uncertainty in a set of examples. The goal of a decision tree algorithm is to maximize information gain at each step, resulting in more informative and accurate splits. Another criterion is the Gini impurity, which measures the probability of misclassifying a randomly chosen element in a set. Similar to information gain, the decision tree algorithm aims to minimize the Gini impurity by selecting the features and split points that lead to the greatest

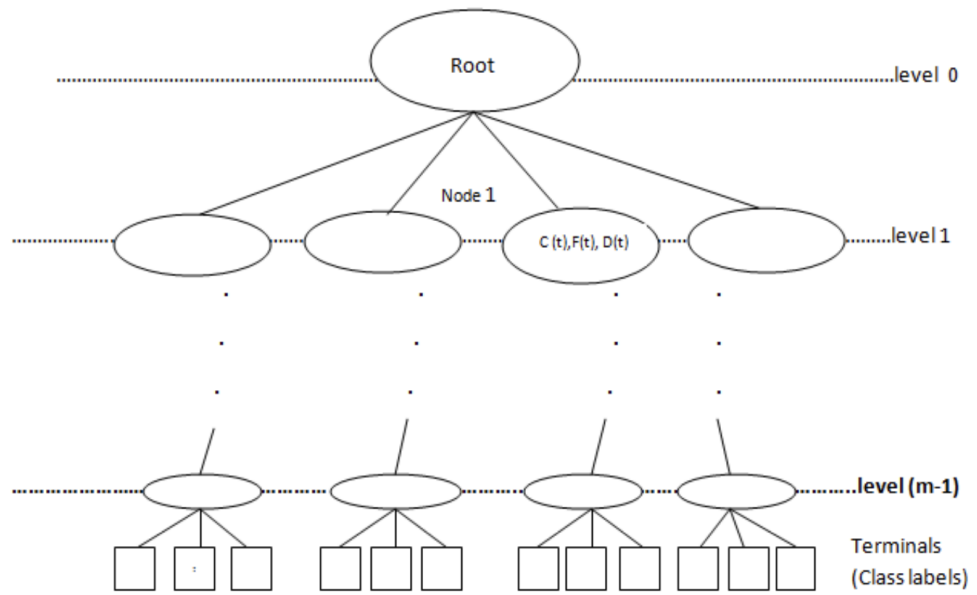


Figure 2.1. Example of general balanced (complete) decision tree from [29].

Notes:

$C(t)$ = subset of classes reachable from node (t)

$F(t)$ = feature subset provided at node (t)

$D(t)$ = decision rule provided at node (t)

reduction in impurity. The decision tree algorithm recursively applies the selected criterion to determine the best feature and split point for each internal node. This process continues until certain termination conditions are met, such as reaching a maximum depth, reaching a minimum number of samples at a node, or when further splits do not significantly improve the predictive performance. Decision trees have several advantages that make them popular in machine learning:

- **Interpretability:** Decision trees provide a clear and understandable decision-making process. The tree structure allows users to trace the path from the root to a leaf node, understanding how the input features influence the final prediction. This interpretability is valuable in domains where explainability is crucial, such as healthcare or finance.
- **Handling both categorical and continuous features:** Decision trees can handle a mix of categorical and continuous features without requiring additional preprocessing. The splitting process considers different types of features, ensuring efficient utilization of the available information.
- **Non-linear relationships:** Decision trees can capture non-linear relationships between features and the target variable. By recursively partitioning the feature space, decision trees can model complex decision boundaries, making them suitable for datasets with intricate patterns.
- **Robustness against outliers and missing values:** Decision trees are robust against outliers and can handle missing values in the data. They do not rely on statistical assumptions and can handle data with irregularities without significantly impacting their performance.

Despite their advantages, decision trees also have some limitations:

- **Overfitting:** Decision trees are prone to overfitting, especially when the tree becomes overly complex. A highly complex tree can capture noise or irrelevant patterns in the training data, resulting in poor generalization to unseen data. Techniques such as pruning or setting a maximum depth can help mitigate overfitting.

- Sensitivity to small variations in data: Decision trees can be sensitive to small variations in the training data. A slight change in the training set may result in a different decision tree structure. Techniques such as randomizing the order of training examples or applying ensemble methods can address this issue, for example Random Forests can help improve performance by combining multiple decision trees while maintaining their interpretability and ease of understanding.
- Difficulty in capturing certain concepts: Decision trees struggle to capture concepts that require multiple levels of abstraction or interactions between features. They may not be as effective as other models, such as neural networks, in handling complex tasks that demand high-level feature representations.

Support Vector Machine

Support Vector Machines (SVMs) are powerful supervised learning models used for both classification and regression tasks. They are particularly effective in solving complex problems with non-linear decision boundaries. SVMs have gained significant popularity in machine learning due to their ability to handle high-dimensional data and provide robust generalization performance [30]. The key idea behind SVMs is to find the optimal hyperplane that separates different classes in the feature space while maximizing the margin, which is the distance between the hyperplane and the nearest data points of each class. The bigger the margin, the better will be the generalization to unseen data. To achieve this, SVMs transform the original input space into a higher-dimensional feature space using a kernel function. The kernel function allows SVMs to implicitly map the input data into a higher-dimensional space, where it may become easier to separate the classes linearly. This is known as the kernel trick and eliminates the need to explicitly compute the transformed feature space, making SVMs computationally efficient. The SVM training process involves finding the hyperplane that separates the classes while satisfying the margin maximization objective. This is formulated as an optimization problem, aiming to minimize the classification error and maximize the margin simultaneously. The optimization is typically solved using convex optimization techniques, ensuring that the solution is both globally optimal and computationally efficient. One of the distinguishing features of SVMs is their ability to handle both linearly separable and non-linearly separable data. In cases where the data is not linearly separable, SVMs introduce a soft-margin approach, allowing some misclassifications while still maintaining a good margin. This is achieved by introducing slack variables that represent the margin violations. The optimization problem is then modified to balance the trade-off between maximizing the margin and minimizing the misclassifications. The choice of the kernel function is critical in SVMs as it determines the mapping of the data to the higher-dimensional space. Different kernel functions, such as linear, polynomial, radial basis function, or sigmoid, can be used based on the problem at hand. The selection of the kernel function depends on the characteristics of the data and the desired decision boundary shape. Radial basis function kernels are commonly used as they are capable of capturing complex non-linear relationships. SVMs offer several advantages that contribute to their popularity in machine learning:

- Effective in high-dimensional spaces: SVMs can handle datasets with a large number of features, making them suitable for high-dimensional data such as text or image data. The kernel trick allows SVMs to implicitly perform calculations in the transformed feature space without explicitly dealing with the high dimensionality.
- Robust to overfitting: SVMs tend to generalize well and are less prone to overfitting, especially when the margin is maximized. By maximizing the margin, SVMs focus on the most informative samples, leading to better generalization to unseen data.
- Flexibility in handling non-linear data: Through the use of different kernel functions, SVMs can effectively model complex decision boundaries and capture non-linear relationships in the data. This flexibility enables SVMs to solve a wide range of classification and regression tasks.

- Sparse solution representation: SVMs often have sparse solution representations, meaning that only a subset of the training examples, known as support vectors, is used in the decision-making process. This property makes SVMs memory-efficient and allows for faster prediction on new instances.

However, SVMs also have some limitations:

- Sensitivity to parameter tuning: SVMs have hyperparameters that need to be carefully tuned, such as the choice of the kernel function, the regularization parameter, and the kernel-specific parameters. Selecting appropriate hyperparameters is crucial for obtaining optimal performance, and improper tuning can result in suboptimal models.
- Computational complexity: SVMs can be computationally expensive, especially when dealing with large datasets. The training time complexity is generally between $O(n^2)$ and $O(n^3)$, where n is the number of training examples. This makes SVMs less suitable for extremely large datasets.
- Lack of probabilistic outputs: SVMs do not directly provide probabilistic outputs. Instead, they assign new instances to classes based on their position relative to the decision boundary. Techniques such as Platt scaling or utilizing the decision function values can be used to approximate probabilities, but they are not inherent to SVMs.

In conclusion, Support Vector Machines are versatile and powerful machine learning models that excel in handling high-dimensional data, nonlinear relationships, and achieving robust generalization performance. By finding an optimal hyperplane that maximizes the margin, SVMs provide effective solutions to a wide range of classification and regression problems. However, proper parameter tuning and computational complexity should be taken into account when applying SVMs in practical applications.

Naive Bayes

Naive Bayes is a popular probabilistic classifier based on Bayes' theorem [31]. It assumes that the features are conditionally independent given the class label, hence the name *naive*. Despite this simplifying assumption, Naive Bayes classifiers often perform well in practice and are computationally efficient. The foundation of Naive Bayes is Bayes' theorem, which provides a way to calculate the probability of an event based on prior knowledge. In the context of classification, Naive Bayes calculates the probability of a given instance belonging to a particular class. The class with the highest probability is chosen as the predicted class for that instance. Naive Bayes classifiers have several advantages. They are computationally efficient and can handle high-dimensional feature spaces. They require relatively small amounts of training data and are particularly well-suited for text classification tasks, such as spam filtering or sentiment analysis. Naive Bayes classifiers also provide interpretable results, as the probabilities can be used to assess the importance of different features in the classification decision. However, Naive Bayes classifiers have limitations. The assumption of feature independence may not hold in some cases, leading to suboptimal results. Additionally, if a particular feature value has not been observed in the training data for a given class, the conditional probability estimation may become zero, causing the classifier to assign zero probability to that class. This problem can be addressed by applying smoothing techniques, such as Laplace smoothing, which add a small value to all observed feature counts. In conclusion, Naive Bayes is a simple yet effective probabilistic classifier based on Bayes' theorem. Despite its naive assumption of feature independence, Naive Bayes classifiers perform well in practice, especially in situations where the assumption holds or when dealing with large amounts of data. By estimating conditional probabilities and making use of prior knowledge, Naive Bayes classifiers provide a powerful tool for classification tasks in various domains.

Random Forest

Random Forest is an ensemble learning method that combines the predictions of multiple decision trees to make more accurate and robust predictions. It is widely used for both classification and

regression tasks. Random Forests derive their strength from the principle of *wisdom of crowds*, where the collective decision of multiple models tends to outperform the decision of any individual model [32]. Random Forests are constructed by creating an ensemble of decision trees, each trained on a random subset of the training data. This process is known as bagging (bootstrap aggregating). Bagging involves sampling the training data with replacement to create different subsets, ensuring that each tree is trained on a slightly different dataset. Additionally, at each split in a decision tree, only a random subset of features is considered for determining the best split. This technique is known as random subspace method or feature bagging. By incorporating randomness in both data sampling and feature selection, Random Forests reduce overfitting and increase the diversity among individual trees. During the prediction phase, each decision tree in the Random Forest independently produces a prediction. For classification tasks, the final prediction is determined by majority voting, where the class with the most votes among the trees is selected as the predicted class. In regression tasks, the final prediction is typically the mean or median of the predictions from individual trees. The aggregation of predictions from multiple trees helps to smooth out noise and make more accurate predictions, as errors made by some trees can be compensated by the correct predictions of others. Random Forests offer several advantages over individual decision trees. First, they provide robustness against overfitting, as the averaging of predictions reduces the impact of individual noisy or biased trees [33]. Second, Random Forests can handle high-dimensional data with a large number of features, as the random subspace method ensures that different trees consider different subsets of features, capturing diverse aspects of the data. This reduces the risk of selecting irrelevant or redundant features. Additionally, Random Forests can estimate the importance of each feature based on how much they contribute to the accuracy of the model, allowing for feature selection or identifying important variables in the data. Random Forests also have some limitations to consider. As an ensemble method, they tend to be more computationally expensive and require more memory compared to individual decision trees. Training a Random Forest involves constructing multiple decision trees and aggregating their predictions, which increases the computational complexity. However, parallelization techniques can be employed to speed up the training process. Another consideration is the interpretability of Random Forests, as the collective decision of multiple trees may be challenging to interpret compared to a single decision tree. However, feature importance measures can provide insights into the relative importance of different features in making predictions. In summary, Random Forests are powerful and versatile machine learning models that leverage the collective wisdom of multiple decision trees to make accurate and robust predictions. By combining bagging and random subspace methods, Random Forests reduce overfitting, handle high-dimensional data, and provide estimates of feature importance. With their wide applicability and superior performance in various domains, Random Forests have become a popular choice for both researchers and practitioners in the field of machine learning.

Neural Network

Neural networks, inspired by the structure and function of the human brain, are a powerful class of models within the field of machine learning. They are capable of learning complex patterns and have achieved remarkable success in various domains, including computer vision, natural language processing, and speech recognition. A neural network consists of interconnected nodes, called neurons, organized in layers. The three main types of layers are the input layer, one or more hidden layers, and the output layer. Each neuron in a layer is connected to neurons in the adjacent layers through weighted connections. The input layer receives the features or inputs of the model, which are then passed through the network, layer by layer, until the final output is generated. The fundamental operation in a neuron is a weighted sum of its inputs, followed by the application of an activation function. Each input is multiplied by a weight, which determines its importance in the computation. The neuron then calculates the sum of the weighted inputs and adds a bias term. The purpose of the bias term is to introduce flexibility and shift the activation function's threshold. The weighted sum is then passed through the activation function, which introduces non-linearity into the neuron's output. Common activation functions include the sigmoid function [34] and the rectified linear unit (ReLU) function [35]. During the training process, neural networks learn by adjusting the weights and biases of the connections to minimize the difference between the predicted output and the true output. This

is achieved through a process called backpropagation, which combines the chain rule of calculus with gradient descent optimization. Backpropagation involves computing the gradient of the loss function with respect to the network's weights and biases and updating them accordingly. The number and size of hidden layers in a neural network depend on the complexity of the problem at hand. Deep neural networks, characterized by multiple hidden layers, have gained significant attention due to their ability to learn hierarchical representations of data. By learning multiple layers of abstraction, deep neural networks can capture intricate patterns and dependencies in the data. Neural networks are typically trained using large labeled datasets through a process called supervised learning. The training data consists of input examples and their corresponding target outputs. During training, the network adjusts its weights and biases iteratively to minimize a loss function, which quantifies the discrepancy between the predicted output and the true output. Common loss functions include mean squared error for regression tasks and categorical cross-entropy for classification tasks. One of the challenges associated with neural networks is the risk of overfitting, where the model performs well on the training data but fails to generalize to unseen data. Techniques such as regularization, dropout, and early stopping are often employed to mitigate overfitting and improve the network's generalization ability. Furthermore, training deep neural networks can be computationally expensive, especially for large-scale datasets. The interpretability of deep neural networks remains a challenge, as understanding the decision-making process of complex models with numerous parameters is not always straightforward.

2.2.2 Unsupervised learning

Unsupervised learning algorithms explore and discover hidden patterns or structures within unlabeled data. Unlike supervised learning, where labeled examples are provided, unsupervised learning operates on unannotated data, making it particularly useful when the task at hand lacks labeled training data or when the goal is to gain insights into the underlying structure of the data [28].

K-means clustering

K-means clustering is a widely used unsupervised learning algorithm that aims to partition a dataset into K clusters based on similarity. It is an iterative algorithm that seeks to minimize the within-cluster sum of squares, also known as inertia or distortion and it is particularly useful for discovering distinct groups or patterns within data [36].

The K-means algorithm begins by randomly initializing K centroids, which act as the initial cluster centers. Each data point in the dataset is then assigned to the nearest centroid based on a distance metric, commonly the Euclidean distance. This assignment step ensures that each data point is assigned to the cluster with the closest centroid. After the initial assignment, the algorithm proceeds to update the centroids based on the current assignments. The centroid of each cluster is recalculated as the mean of all data points assigned to that cluster. This step ensures that the centroids represent the center of their respective clusters, as they are adjusted to minimize the distance to the data points within the cluster. The assignment and centroid update steps are iteratively performed until convergence, typically defined by a predetermined number of iterations or when the assignments and centroids no longer change significantly. It is important to note that K-means clustering can converge to a local minimum, meaning that the final clustering solution obtained may vary depending on the initial centroid locations. K-means clustering is computationally efficient and can handle large datasets with a relatively low time complexity. Additionally, K-means is conceptually straightforward and easy to implement. However, one limitation of K-means clustering is its sensitivity to the initial centroid locations. Different initializations can result in different clustering outcomes. To mitigate this, K-means++ initialization is commonly used. It initializes the centroids in a way that promotes better coverage of the data space and leads to more stable and representative clustering results. Determining the optimal value of K , the number of clusters, is another important consideration in K-means clustering. The choice of K is problem-dependent and often involves a trade-off between capturing fine-grained cluster structures and avoiding overfitting or underfitting. It is important to note that K-means clustering assumes that clusters are spherical and have similar variances. Thus, it

may not perform well on datasets with complex or non-spherical cluster shapes. K-means clustering finds applications in various domains, including customer segmentation, image compression, anomaly detection, and document clustering. By grouping similar data points together, K-means clustering allows for the identification of distinct patterns or subgroups within the data. In customer segmentation, for instance, it can be used to identify different customer segments based on their purchasing behavior [37], enabling targeted marketing strategies. In summary, K-means clustering is a popular unsupervised learning algorithm that partitions a dataset into K clusters based on similarity. Despite its simplicity, it is a powerful tool for exploring data and identifying underlying patterns or groups.

Principal Component Analysis

Principal Component Analysis (PCA) is a dimensionality reduction technique widely used in unsupervised learning. It aims to transform high-dimensional data into a lower-dimensional representation while preserving most of the original data's variance. PCA accomplishes this by identifying the principal components, which are orthogonal axes that capture the maximum amount of variation in the data [38]. The PCA algorithm works in several steps. First, it centers the data by subtracting the mean of each feature, ensuring that the transformed data has zero mean. This step is important because it removes any bias or offset in the data and focuses on the relative variations. Then, PCA computes the covariance matrix of the centered data or performs the singular value decomposition (SVD) of the data matrix. The covariance matrix represents the relationships between different features, while the SVD provides a factorization of the data matrix into singular values and corresponding orthogonal eigenvectors. The eigenvectors obtained from the covariance matrix or the singular vectors from the SVD are the principal components of the data. These components represent the directions in the data space along which the data varies the most. The first principal component captures the maximum amount of variance, the second principal component captures the maximum remaining variance orthogonal to the first, and so on. By selecting a subset of the principal components that capture a significant amount of the variance, the dimensionality of the data can be reduced. The reduced-dimension representation obtained through PCA is achieved by projecting the original data onto the selected principal components. This projection can be performed by matrix multiplication, where the data matrix is multiplied by the matrix of selected principal components. The resulting projected data lies in a lower-dimensional space spanned by the chosen principal components. PCA has numerous applications across various domains. In data visualization, PCA can be used to project high-dimensional data onto a lower-dimensional space, such as a 2D or 3D plot. By visualizing the data in a reduced dimension, patterns and relationships that might be obscured in the original high-dimensional space can become more apparent. This aids in exploratory data analysis [39], allowing for better understanding and interpretation of complex datasets. In addition to data visualization, PCA is widely used for feature extraction [40]. By selecting a subset of the principal components that capture most of the variance in the data, irrelevant or redundant features can be eliminated. This process helps to reduce the dimensionality of the data while retaining the most important information. The reduced set of features can then be used as input for subsequent machine learning algorithms, improving efficiency and mitigating the curse of dimensionality. Furthermore, PCA can be employed as a preprocessing step to remove noise or redundant information from the data. By discarding the components that account for relatively small variances, which are often associated with noise or measurement errors, PCA helps to enhance the signal-to-noise ratio in the data. This denoising effect can lead to improved performance in subsequent analysis tasks, such as classification or clustering.

Association rule learning

Association rule learning, also known as market basket analysis, is a valuable unsupervised learning technique that reveals interesting relationships or associations among items in a dataset [41]. It is commonly used in various domains, particularly in retail and e-commerce settings, to uncover purchasing patterns and identify frequently co-occurring items in customers' transactions. One of the well-known algorithms for association rule learning is the Apriori algorithm [41]. It works by

identifying frequent itemsets and generating association rules based on those itemsets, it follows a bottom-up approach and employs an iterative process to progressively discover associations. The algorithm starts by scanning the dataset to identify all frequent individual items, that is frequently called $1 - \text{itemset}$. The frequency of an item is determined by a user-defined minimum support threshold. Once the frequent itemsets are identified, the Apriori algorithm proceeds to generate frequent $k - \text{itemsets}$, where $k > 1$, by joining the frequent $(k - 1) - \text{itemsets}$. It then prunes the generated candidate itemsets that do not meet the minimum support threshold. After obtaining all frequent itemsets, association rules are generated by examining the subsets of each itemset. The algorithm calculates various measures, including support, confidence, and lift, to determine the strength and significance of the association rules. Support measures the frequency of an itemset in the dataset, confidence quantifies the conditional probability of the consequent given the antecedent, and lift indicates the strength of the association between the antecedent and the consequent beyond what would be expected by chance. Association rule learning provides valuable insights into the co-occurrence and dependencies among items. For example, in a retail setting, it can reveal that customers who purchase diapers are also likely to buy baby formula. This information can guide marketing strategies, such as placing related items near each other in stores or making personalized recommendations to customers based on their previous purchases. Furthermore, association rules can be enhanced with additional measures and constraints to ensure their relevance and usefulness. For instance, the notion of lift can be combined with a minimum confidence threshold to identify strong rules that are not just coincidental associations. Adding constraints such as minimum and maximum itemset lengths or specifying the presence or absence of certain items can further refine the discovered rules and make them more actionable. However, it's important to note that association rule learning has certain limitations. The combinatorial explosion of itemsets can become computationally expensive, especially with large datasets. Consequently, optimizations, such as pruning techniques and efficient data structures, are employed to improve the efficiency of the Apriori algorithm. Additionally, association rule learning may suffer from the curse of dimensionality when dealing with high-dimensional datasets, as the number of possible itemsets exponentially increases with the number of dimensions.

2.2.3 Semi Supervised learning

Semi-supervised learning algorithms are designed to leverage both labeled and unlabeled data in the learning process. Unlike supervised learning, which relies solely on labeled data, and unsupervised learning, which focuses on unlabeled data, semi-supervised learning algorithms take advantage of the additional information provided by the unlabeled examples to improve the model's performance [42]. In many real-world scenarios, obtaining labeled data can be expensive, time-consuming, or even impractical. For instance, in medical diagnostics, annotating large amounts of data with expert labels may require specialized expertise and significant effort and semi-supervised learning offers an attractive solution by effectively utilizing a combination of limited labeled data and readily available unlabeled data. The key idea behind semi-supervised learning is that the unlabeled data contains valuable information about the underlying structure of the data distribution. By exploiting this additional information, the algorithms aim to enhance the model's generalization capabilities and improve its performance. One common approach in semi-supervised learning is to use the unlabeled data to induce a smoothness assumption or regularization constraint on the learned decision boundary. This assumption assumes that nearby points in the input space should have similar outputs and so similar examples should produce consistent predictions. One popular semi-supervised learning algorithm is the self-training method [43]. This approach starts with an initial model trained on the limited labeled data, then the model is used to generate pseudo-labels for the unlabeled data, assigning predicted labels to these examples. The model is then retrained using the combined labeled and pseudo-labeled data. This iterative process is repeated until convergence or a predefined stopping criterion is met. The self-training method assumes that the pseudo-labels generated for the unlabeled data are reliable and accurate representations of their true labels. Another approach in semi-supervised learning is co-training [43], which leverages multiple views or multiple sets of features to improve learning from unlabeled data. Co-training algorithms assume that different views or feature sets provide complementary information about the underlying data. The algorithm trains multiple models, each using a different subset of features, and the models iteratively exchange and update their predictions on

the unlabeled data. This process aims to refine the models' predictions by incorporating diverse perspectives and reducing uncertainty. It's important to note that the success of semi-supervised learning algorithms heavily depends on the quality and representativeness of the unlabeled data. If the unlabeled data is noisy or contains significant outliers, the performance of the semi-supervised learning algorithm may degrade. Careful preprocessing and filtering of the unlabeled data may be necessary to ensure its reliability and usefulness in the learning process. Semi-supervised learning algorithms also face challenges in balancing the impact of labeled and unlabeled data. The algorithm needs to appropriately weigh the contributions of labeled and unlabeled examples to avoid overfitting or underutilizing the available information. This balance is often achieved through heuristics, hyperparameter tuning, or additional constraints during the training process. The application of semi-supervised learning extends to various domains and tasks. Taking into account the cybersecurity field, semi-supervised learning has been used for, among others, unknown malware detection [44], network anomaly detection [45] and for intrusion detection system [46]. In conclusion, semi-supervised learning algorithms offer a valuable approach to leveraging both labeled and unlabeled data for improved learning and prediction performance. By incorporating the additional information present in unlabeled data, these algorithms can effectively learn from limited labeled data and make use of readily available but unlabeled examples. The success of semi-supervised learning depends on appropriate regularization techniques, careful consideration of the quality of the unlabeled data and balancing the contributions of labeled and unlabeled examples.

2.2.4 Reinforcement learning

Reinforcement learning (RL) is a type of machine learning that involves learning through interactions with an environment. In particular, RL algorithms learn by receiving feedback in the form of rewards or punishments, enabling them to make sequential decisions that aim to maximize a cumulative reward signal. This paradigm is inspired by how humans and animals learn to perform tasks through trial and error, adapting their behavior based on the outcomes they experience [47]. Reinforcement learning algorithms consist of three key components: the agent, the environment, and the reward signal. The agent is the learner or decision-maker that interacts with the environment. It perceives the current state of the environment and selects actions based on its current policy. The environment represents the external system or context in which the agent operates. It provides the agent with feedback and updates its state based on the actions taken. The reward signal is a numerical signal that provides feedback to the agent, indicating the desirability or quality of the agent's actions. The goal of the agent is to learn a policy that maximizes the expected cumulative reward over time. The RL process involves the agent taking actions in the environment, receiving feedback in the form of rewards, and updating its policy or strategy to improve future decision-making. The agent's policy is a mapping from states to actions, defining the behavior or action selection strategy. The objective of RL is to find an optimal policy that maximizes the cumulative reward over time. One fundamental RL algorithm is called Q-learning [48]. Q-learning is a model-free algorithm that learns the value of state-action pairs. It estimates the expected cumulative reward, known as the Q-value, for each state-action pair. Q-learning starts with an initial estimate of the Q-values and iteratively updates them based on the observed rewards and the agent's interactions with the environment. The Q-values are updated using the Bellman equation, which combines the immediate reward and the estimated future rewards to update the value of a state-action pair. Through this iterative process, the agent gradually learns the optimal Q-values and can make informed decisions based on them. Another popular RL algorithm is called the policy gradient method. Policy gradient algorithms directly optimize the policy itself, rather than estimating value functions. These algorithms use gradient ascent to iteratively improve the policy based on the observed rewards. The policy is represented by a parameterized function, such as a neural network [49], and the algorithm adjusts the parameters to maximize the expected cumulative reward. Policy gradient methods have been especially used in tasks with continuous action spaces, as they can learn complex policies directly. One of the key challenges in reinforcement learning is the exploration-exploitation trade-off. The agent must balance between exploring new actions and exploiting its current knowledge to maximize the expected reward. If the agent only focuses on exploiting known actions, it may miss out on discovering better actions.

Conversely, if the agent explores too much, it may waste time and resources on suboptimal actions. Various exploration strategies, such as epsilon-greedy, softmax, or Thompson sampling, are employed to address this trade-off. Another important aspect of reinforcement learning is the use of function approximators, such as neural networks, to approximate value functions or policies. These approximators enable RL algorithms to handle large state and action spaces and generalize from limited experiences. However, the use of function approximators introduces challenges, such as overfitting, stability, and generalization issues, which require careful design and training techniques. Reinforcement learning algorithms have been successfully applied in a wide range of domains. One notable application is in game playing. RL algorithms have achieved impressive results by learning to play games such as chess, Go, and video games. This aspect has been exploited also in cybersecurity [50], where game theory simulations have been applied as defense strategies to countermeasure cyberattacks.⁶ Reinforcement learning has several advantages and unique characteristics compared to other types of machine learning. RL algorithms excel in scenarios with sparse or delayed feedback, where the reward signal is only received after a series of actions. RL is particularly well-suited for tasks with sequential decision-making and learning from interactions. It has the potential to tackle complex problems that involve long-term planning and exploration in dynamic environments. In conclusion, reinforcement learning is a powerful paradigm in machine learning that enables agents to learn optimal policies through interactions with an environment. By leveraging rewards and punishments, RL algorithms can make sequential decisions to maximize cumulative rewards. Q-learning and policy gradient methods are popular algorithms in RL, each with its own advantages and applications. Reinforcement learning has shown remarkable success in game playing and robotic control, and it continues to be an active area of research with the potential for solving complex real-world problems.

2.3 Overview of Malware

Malware, which is short for *malicious software*, encompasses a broad range of programs intended to perform harmful or undesirable actions on a computer system. It includes various types of code, scripts, and software designed to disrupt operations, gain unauthorized access, compromise privacy, or engage in abusive behavior [51]. It is important to distinguish malware from defective software, which refers to software with legitimate purposes but contains harmful bugs or programming errors. The intended target of the malware is the computer system whose security is being breached. The individual or group responsible for launching the malware with the intention of attacking one or more targets is commonly known as the initiator. It is worth noting that the initiator may or may not have precise knowledge of the specific set of targets, depending on the type of malware involved. The identification of software as malware is typically based on its intended use, rather than the specific techniques or technologies used to create it. In most cases, cybercriminals utilize malware to gain access to sensitive data, extort ransoms, or inflict significant damage on the affected systems.

Malware serves various purposes, and some of its common uses include:

- **Financial Gain:** Malware is frequently employed to generate financial profits. For instance, attackers may infect devices to steal credit card information or cryptocurrency. Alternatively, they may sell their malware to other cybercriminals or offer it as a service known as *malware-as-a-service*.
- **Revenge or Personal Agenda:** In certain cases, individuals may deploy malware as a form of revenge or to further their personal agenda. For example, in 2016, Brian Krebs of Krebs on Security experienced a significant Distributed Denial of Service (DDoS) attack after discussing a DDoS attacker on his blog.
- **Political or Social Motivations:** Some malware is designed to advance political or social causes. Nation-state actors and hacker groups, such as state-run hacker groups in China and North Korea or entities like Anonymous, may engage in cyber attacks for ideological purposes.

- **Entertainment:** Shockingly, some cybercriminals carry out attacks purely for amusement, finding pleasure in victimizing others.

It is worth noting that non-malicious actors may also create or deploy certain types of malware for legitimate reasons. For example, malware can be used to test the security of a system or as part of a legal act or police operation to gather intelligence.

2.4 Types of malware

The categorization of malwares can be done based on how they spread and what actions they perform after infecting a victim's computer [52]. Understanding these distinctions can provide insights into the behavior and potential risks associated with different types of malicious software. One common way to categorize malware is by its **spreading mechanism**. While terms like trojan, virus, and worm are often used interchangeably, they actually represent distinct methods by which malicious software infects target computers.

- **Trojan Horse.** A Trojan disguises itself as legitimate software and tricks users into downloading and executing it. It conceals a malicious payload that can have various effects, such as deleting user files, installing additional malware, or introducing Potentially Unwanted Apps (PUAs). Trojans can be hidden in games, apps, or software patches, or they may rely on social engineering techniques, such as email attachments in phishing campaigns. Unlike viruses or worms, Trojans cannot self-replicate and depend on user interaction or system operators to activate them [53].
- **Virus.** Computer viruses are self-replicating programs that spread by infecting other files or programs. They are typically transmitted through infected websites, file sharing networks, or email attachments. Viruses remain dormant until the infected host file or program is activated, at which point they embed copies of themselves into other executables or boot sectors. Viruses rely on the transfer of infected files to new systems to propagate. The means of transmission, such as files, media files, or network files, are commonly referred to as the virus vector. Some viruses possess complex code that allows them to modify their copies during replication [51]. In addition to replication, viruses can perform various harmful actions, such as creating backdoors, damaging files, stealing information, creating botnets, displaying advertisements, or even causing physical damage to equipment.
- **Worm.** Worms are self-replicating and actively spreading malicious programs that exploit vulnerabilities in a target's operating system or installed software. They rely on these vulnerabilities to propagate across networks. Worms consume significant bandwidth and processing resources due to continuous scanning for potential targets. Their presence can render hosts unstable and cause system crashes. In addition to replication, computer worms often contain payloads, which are pieces of code designed to carry out malicious actions on infected hosts. These actions may include data theft, file deletion, or the creation of bots, which can result in infected systems becoming part of a botnet.

Understanding the distinctions between Trojans, viruses, and worms provides valuable insights into the strategies and potential impacts of different types of malware. By comprehending their spreading mechanisms and associated behaviors, users and security professionals can better protect computer systems and respond effectively to potential threats.

Another way to categorize malwares is looking at **what they do**.

- **Adware.** Adware is any software package which automatically plays, displays, or downloads advertisements to a computer. These can be in the form of pop-up ads, ad banners in websites, or advertisements displayed by software, that lure the user into making a purchase. The goal of Adware is, in fact, to generate revenue for its author. Often times adware comes even bundled with free, or discounted, versions of non-malicious software and/or applications since it is usually seen by developers as a way to recover development costs [54]. Adware, by

itself, is annoying but somewhat harmless, since it is solely designed to deliver ads; however, it often comes bundled with spyware (such as keyloggers), and/or other privacy-invasive software that is capable of tracking user activity and stealing information [55].

- **Backdoor.** A backdoor, also called Remote Access Trojan (RAT), is a vulnerability deliberately buried into software's code allowing to bypass typical protection mechanisms, like credentials-based login authentication. Once a system has been compromised (by others types of malware or other methods), one or more backdoors may be installed. This is done with the purpose of allowing the attacker easier access in the future without alerting the user or the system's security programs. Moreover, backdoors may also be installed before other malicious software, to allow attackers entry [54]. On the other hand, it is not uncommon to see many (non-malicious) device or software manufacturers ship their products with intentionally hidden backdoors to allow company personnel or law enforcement to access to the system when needed [56]. Alternatively, backdoors are sometimes hidden in programs also by intelligence services.
- **Browser Hijacker.** A Browser Hijacker, also called hijackware, is a type of malicious program which considerably modifies the behaviour of the victim's web browser. It can be used to make money off unwanted ads, to steal information from users, or to infect the systems with other malware by redirecting users to malicious websites [56].
- **Bots/Botnet.** In general, bots (short for robots) are software programs designed to automatically perform specific operations. Some bots are used for legitimate and harmless purposes such as video programming, video gaming, internet auctions and online contest, among other functions. It is however becoming increasingly common to see bots being used maliciously. Malicious bots can be (and usually are) used to form botnets. A botnet is defined as a network of host computers (zombies/bots) that is controlled by an attacker, the bot-master [53]. Botnets are frequently used for DDoS (Distributed Denial of Service) attacks, but there are other ways that botnets can be useful to cybercriminals: for example they can be employed to carry out brute force attacks on websites or to distribute malware.
- **Crypto-miner.** Crypto-miners are a relatively new malware type. Cybercriminals employ this type of malicious tools to mine Bitcoin and/or other similar digital currencies on the target machine by exploiting the computing power of the victim's system, without the owner realising it. Obviously, the mined coins end up in the attackers' digital crypto wallets. Recently, a more modern method of crypto-mining that works within browsers (also called crypto-jacking), has become quite popular [57].
- **File-less malware.** As the term suggests, File-less malware is a type of malware that operates from a victim's computer memory, not from files on the hard drive, taking advantage of legitimate code and tools already existing within the system. File-less malware leaves no malware files to scan and no malicious processes to detect, and is therefore harder to detect and remove than traditional malware: it is, in fact, up to ten times more successful. Furthermore, it also renders forensic analysis more difficult because it disappears from the victim's computer upon rebooting.
- **Keylogger.** Keyloggers, often considered as being a sub-category of spyware, are malicious programs which secretly track keystrokes on a keyboard, without the system owner consent, action that is usually referred to as keylogging or Keystroke logging. The collected data is stored and sent to the attacker who uses it to figure out highly sensitive information such as passwords, usernames and payment details. Keylogging can be performed in various ways, ranging from hardware and software-based approaches to the more sophisticated electromagnetic and acoustic analysis [54].
- **RAM scraper.** RAM scraper malware, also known as Point-of-Sale (POS) malware, targets POS systems like cash registers or vendor portals, harvesting data temporarily stored in RAM (Random Access Memory). Doing so the attacker can easily access unencrypted credit card numbers [56].
- **Ransomware.** Ransomware, also known as encryption or crypto Trojan, is a malicious program that, after having infected a host or network, holds the system captive and displays a

message requesting a ransom to the host/network users. In particular, it encrypts data on the infected system (or anyway it locks down the system preventing user access) and only unblocks it when the correct password, decryption key, is entered. Without the latter, is practically impossible to regain access to the system. Digital currencies such as Bitcoin and Ether are the most common means of payment, making it difficult to track cybercriminals. Moreover, paying the ransom does not guarantee the user to receive the necessary decryption key. Additionally, some forms of ransomware threaten victims to publicize sensitive information within the encrypted data in order to convince them to pay the ransom.

- **Rogue Security Software.** Rogue Security Software can be considered as a form of scareware. This type of malware programs presents itself as a security tool to remove risks from the user's system. In reality, this fake security software installs more malware onto the system [56].
- **Rootkit.** A rootkit is generally thought as a type of malicious software, or a collection of software tools, designed to remotely access or control a computer without being detected by users or security programs. An attacker who has installed a rootkit on a system is able to perform a wide range of malicious activities such as remotely executing files, logging user activities, installing hidden malware, etc. Since a rootkit operates stealthily and continually hides its presence, its prevention, detection and removal can be difficult and often relies on manual methods [55]. More recently, the term rootkit has often been used to refer also to concealment routines in a malicious program. These highly advanced and complex routines are written to hide malware within legitimate processes on the infected computer. In fact, once a malicious program has been installed on a system, it is essential that it remains hidden, to avoid detection and disinfection [54].
- **Scareware.** Scareware is a generic term for malware that uses social engineering to frighten and manipulate users, inducing them into thinking their system is vulnerable or has been compromised. However, in reality no danger has actually been detected: it is a scam. The attack succeeds when the user purchases unwanted, and potentially dangerous, software in an attempt to eliminate the *threat*. Generally, the suggested software is additional malware or allegedly protective software with no value whatsoever [57]. Some versions of scareware act as a sort of shadow version of ransomware; they claim to have taken control of the victim's system and demand a ransom. However they are actually just trying to fool the victim.
- **Spyware.** Spyware, another name for privacy-invasive software, is a type of malicious software that spies on user activity on the infected system. Specifically it can collect various types of personal information about users, such as Internet browsing habits, credit card details and passwords, without their knowledge. The information gathered is then sent back to the responsible cybercriminal(s). However, the functions of spyware often go far beyond simple activity monitoring and information gathering. In fact, they may also interfere with the user's control of the computer in other ways, such as installing additional software, redirecting web browser activity and changing computer settings [54]. Law enforcement, government agencies and information security organizations often use spyware to monitor communications in a sensitive environment or during an investigation.

At the end, it's important to note that generally a malware is categorized both by its spread mechanism and its behavioural category. Furthermore, many instances of malware fit into multiple categories: for example *Stuxnet* is both a worm, a virus and a rootkit.

2.5 Malware detection techniques

Malware detection methods are fundamentally categorized in different categories from different points of view. This thesis assumes three categories as it's illustrated in figure 2.2.

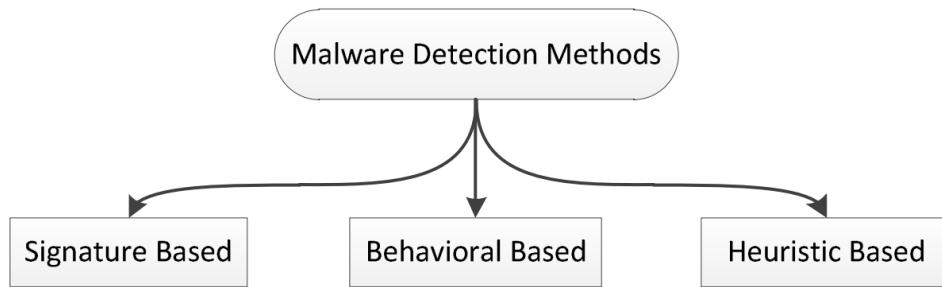


Figure 2.2. Illustration of malware detection methods (source: [58])

2.5.1 Signature based

Signature-based detection is a widely used technique for detecting and identifying known malware. It relies on the use of signatures, which are unique patterns or characteristics associated with specific malware strains [58]. These signatures can be derived from various attributes of malware, such as file hashes, byte patterns, or behavioral indicators. Although various formats exist for creating malware signatures, *YARA* stands out as a widely adopted and popular format in the field of malware analysis. With *YARA*, analysts have the capability to generate signatures using both textual and binary patterns, making it a versatile tool for detecting malicious code [59].

```

1 rule RuleName
2 {
3     meta:
4     description = "description of rule"
5     author = "name"
6     date = "dd/mm/yyyy"
7     reference = "url"
8
9     strings:
10    $text_string1 = "text1 you wish to find in malware"
11    $text_string2 = "text2 you wish to find in malware"
12
13    $hex_string1 = {hex1 you wish to find in malware}
14    $hex_string2 = {hex2 you wish to find in malware}
15
16    $reg_exp_string1 = /regular expression1 you wish to find in malware/
17    $reg_exp_string2 = /regular expression2 you wish to find in malware/
18
19    condition:
20    $text_string1 or $text_string2 or
21    $hex_string1 or $hex_string2 or $reg_exp_string1 or $reg_exp_string2
22 }
  
```

Listing 2.1. Example of a simple YARA rule

An example of *YARA* rule is illustrated in listing 2.1 (source: [52]), where as you can see the Yara rules must start with the keyword *rule*, followed by the *RuleName*, which serves as the identifier for the rule. The *RuleName* follows to the same naming conventions as the *C* programming language. It is case-sensitive, limited to a maximum of 128 characters, and can consist only of alphanumeric characters (including underscores). However, the first character cannot be a digit. Additionally, there is a set of reserved keywords in *YARA* that cannot be used as identifiers [60]. The main body of *YARA* rules comprises three sections: *meta*, *strings*, and *condition*.

- **Meta section.** In the meta section, located at the beginning of the rule, the rule author can provide supplementary information about the rule in the form of attribute-value pairs, commonly referred to as metadata. The values can be strings, integers, or boolean values. It is important to note that metadata cannot be utilized within the condition section [61]. Commonly used meta tags include *author* and *description*, which provide details about the rule's author and purpose. Malware analysts often include tags containing hashes of malicious files used to create the rule or references to relevant blog posts [62].
- **Strings section.** This section contains the strings, patterns, or signatures that a file must contain in order to trigger the rule. The strings section is optional and can be omitted if not needed. *YARA* supports three types of string searches: hexadecimal strings, text (ASCII) strings, and regular expressions.
- **Condition section.** The condition section, which is the only truly required section in *YARA* rules, contains the conditions that determine when the rule is triggered. These conditions are Boolean expressions similar to those used in programming languages [61]. By employing logical and relational operators, conditions can be made as complex as necessary to accommodate the specific requirements of the rule author [62].

Signature-based detection works by comparing the signatures of files or system elements against a database of known malware signatures. When a file or system element matches a signature in the database, it is flagged as malicious. This technique is effective in identifying previously encountered malware strains for which signatures are available. One of the primary advantages of signature-based detection is its efficiency [63]. The comparison process is relatively fast, making it suitable for real-time scanning of files and system elements. Additionally, signature databases can be regularly updated with new signatures, providing users with the latest protection against known threats. Despite its advantages, signature-based detection has several limitations [63]. The technique heavily relies on the availability and accuracy of malware signatures. To maintain an effective detection capability, signature databases must be continuously updated. However, this approach may struggle to keep pace with the rapidly evolving landscape of malware, as new malware variants are constantly being developed. Furthermore, it is limited to detecting only known malware and may not be effective against new or modified malware variants, such as zero-day exploits and polymorphic malware, which modify their signatures with each infection. This limitation creates a window of vulnerability during which new malware can go undetected until a signature is developed and distributed. In recent years, machine learning techniques have been increasingly employed in signature-based detection [64] to improve accuracy and overcome some of its limitations. Machine learning models can analyze large amounts of data and learn to identify complex patterns that may not be apparent to traditional signature-based methods. These models can be trained on features extracted from malware samples, such as file header information, opcode sequences, or static and dynamic behavioral attributes. Supervised learning algorithms, such as decision trees, support vector machines (SVM), or neural networks, can be trained using labeled datasets that include both benign and malicious samples. The trained models can then classify new files or system elements as either benign or malicious based on the learned patterns. Machine learning-based signature detection can help identify unknown or modified malware variants that may not match traditional signatures. However, machine learning-based signature detection also faces challenges. It requires carefully curated and representative training datasets to ensure accurate and reliable detection. Furthermore, adversarial attacks can be launched to manipulate or evade machine learning models. Malware authors can employ various techniques, such as obfuscation, code injection, or adversarial samples, to bypass signature-based machine learning detection systems. This thesis will deal with detection evasion in the next section.

2.5.2 Behavioral based

Behavioral analysis is an important approach in malware detection that focuses on studying the behavior of programs or processes to identify potential malicious activities. Unlike signature-based detection, behavioral analysis aims to detect previously unseen or zero-day malware by examining their actions and deviations from normal behavior. Behavioral analysis involves monitoring various aspects of a program's execution, such as system calls, network traffic, file system

activity, and registry modifications. By observing these activities, analysts can identify suspicious behaviors that indicate the presence of malware. The main purpose of this approach is to detect new and unknown malware variants, as it does not rely on pre-existing signatures. However, it can also result in a higher false-positive rate due to the potential for legitimate programs to exhibit behavior that may appear malicious [65]. There's a multitude of behaviors that point to potential danger. Some examples include any attempt to discover a sandbox environment, disabling security controls, installing rootkits, and registering for autostart [66]. Behavioral analysis is performed through dynamic analysis, i.e. the malware has to be executed (in a controlled environment, such as a virtual machine or sandbox) to study its behavior during runtime. This approach allows analysts to observe the malware's interactions with the system, including its system calls, network communications, and file operations. Dynamic analysis provides valuable insights into the malware's actual behavior and helps in identifying malicious activities. This malware detection solution employs comprehensive techniques to thoroughly assess the behavior of malicious code. These solutions meticulously analyze and evaluate every line of code executed by the malware within its relevant context. They carefully analyze all requests made by the malware to access particular files, processes, connections, or services. This process extends to each instruction carried out at the operating system level or by any invoked programs, even encompassing low-level code that may be concealed by rootkits [66]. Some used techniques are:

- **API monitoring.** API monitoring involves the monitoring and analysis of the Application Programming Interfaces (APIs) called by a program during its execution. APIs are sets of functions and procedures that allow software applications to interact with the operating system, hardware devices, and other software components. In the context of malware detection, API monitoring focuses on tracking the sequence of API calls made by a program and analyzing their parameters and behavior. By monitoring these API interactions, analysts can gain insights into the program's behavior and identify potential malicious activities or deviations from normal behavior [67].
- **Network traffic analysis.** Malware often relies on network communications to carry out its malicious activities, such as establishing connections with command-and-control (C&C) servers, downloading additional payloads, or exfiltrating sensitive data. Network traffic analysis is a crucial component of behavioral analysis that focuses on monitoring and examining the network communications generated by malware to detect and identify malicious behavior. Network traffic analysis involves capturing, inspecting, and analyzing the packets that are transmitted over a network. This analysis can be performed at different levels, including the network layer, transport layer, and application layer, depending on the information needed and the specific characteristics of the malware under investigation [68].
- **Memory analysis.** Memory analysis involves examining the memory space of a process or system to identify any suspicious behavior or the presence of malicious code. By analyzing the contents of memory, analysts can uncover hidden malware activities and identify indicators of compromise. Memory analysis techniques encompass various methods, such as examining process memory dumps, analyzing operating system data structures, and investigating runtime artifacts. Through these approaches, analysts can identify code injection, detect rootkit presence, uncover hidden processes, or identify malicious payloads that may have been loaded into memory. One of the key advantages of memory analysis is its ability to reveal malware that actively hides itself or evades traditional detection methods [69].

2.5.3 Heuristic based

Heuristic malware detection methods are proposed to overcome the disadvantages of signature-based and behavioral-based detection. Heuristic malware detection methods use data mining and machine learning techniques to learn the behavior of an executable file [58]. The feature that are used as input in these ML models are various, but the most common are:

- **OpCode.** An OpCode, also known as Operational Code, is a component of a machine language instruction that specifies the operation to be performed. Specifically, a program

is defined as a sequence of organized assembly instructions. Each instruction consists of an operational code and one or more operands [58]. Extensive research on OpCodes has been conducted by Bilar [70]. He investigated the potential of individual OpCodes as a feature for detecting malware. Bilar statistically analyzed the effectiveness of individual OpCodes and presented clear evidence of their reliability in determining whether an executable file is malicious. These findings demonstrate that OpCodes can serve as an efficient representation for executable files.

- ***N-Grams.*** N-Grams are all substrings of a word with a length of N. For example, the string *MALWARE*, can be segmented in various 5-grams: *MALWA*, *ALWAR*, *LWARE* and so on [58]. So, N-grams are generated by sliding a window of size n over a given sequence of bytes or instructions. This window captures the contiguous subsequences, creating the n-grams, used then as features for analyzing and characterizing executable files. By extracting n-grams from malware samples and legitimate files, statistical patterns and structural differences can be identified, aiding in the classification process.
- ***Control flow graph.*** A control flow graph (CFG) is a graphical representation of the flow of execution within a program or code snippet. It illustrates, with a directed graph, the sequence of instructions and the control transfers between them (i.e. what happens after what). CFGs are commonly used in program analysis and software engineering to understand program behavior and detect various properties and vulnerabilities. For example Zhao [71] introduced a detection technique that CFG features for identifying PE files. Initially, CFGs were generated for each executable file and subsequently, the extracted features from the CFGs were employed as training data. These features encompassed details related to nodes, edges, and subgraphs within the CFG. Following the selection of relevant features, data mining algorithms, such as Decision Tree, were applied for classification based on these features.

2.6 Malware analysis

Malware analysis is the process of extracting information from malicious software samples in order to understand their purpose and functionality. This information can be used to develop effective detection techniques, contain damage, reverse the effects of malware on a system, develop removal tools, and design methods to guard systems against future infections. In the past, malware analysis was done manually, which was a time-consuming and complex process that required high expertise. However, the number of malware samples that need to be analyzed on a daily basis has grown to the point where manual analysis is no longer feasible. As a result, a number of analysis tools have been developed to automate the process. The process of taking some man-made things (object, idea, software...) apart to extract any kind of knowledge is called *Reverse engineering* [72]. In the context of malware analysis, reverse engineering is used to understand how malicious software works, aiming to gain insights into its inner workings, detect its capabilities and develop effective countermeasures to mitigate the threat it poses. For example, this can be done by disassembling the malware, which means converting the machine code into a human-readable form. Once the malware has been disassembled, analysts can look for patterns and techniques that are commonly used by malware authors. A brief description of disassembler is provided later in this section. There are two main types of malware analysis: static and dynamic. Static analysis involves examining the malware's code without running it. This can be done by disassembling the code and analyzing the instructions. Dynamic analysis involves running the malware in a controlled environment and monitoring its behavior. This can be done in a sandbox or on a virtual machine. Static and dynamic analysis can be used together to provide a more comprehensive understanding of malware. For example, static analysis can be used to identify the malware's code and functions, while dynamic analysis can be used to see how the malware behaves when it is executed. Malware analysis is an important part of computer security. By understanding how malware works, security teams can develop effective defenses against it.

2.6.1 Static analysis

Static analysis is a type of malware analysis that examines the code of an executable file without actually running it. This can be done by disassembling the code and analyzing the instructions. Static analysis can be used to identify the malware's code and functions, as well as its strings, byte-sequence n-grams, library or API calls, opcode frequency distributions, and peculiar attributes found in the executable header. Static analysis is not always effective against zero-day and evolutionary malware. This is because zero-day malware has not been seen before and does not have a signature that can be used to identify it. Evolutionary malware is constantly changing, making it difficult to identify with static analysis. Despite its limitations, static analysis can be a valuable tool for malware analysis. It can be used to identify the malware's code and functions, which can then be used to develop effective detection techniques. Static analysis can also be used to understand how the malware works, which can help to develop effective removal tools. The main advantages of static analysis is that it is faster and safer than dynamic analysis and that it takes into account the entire program code and not just sub-parts of it. Despite that, there are also some disadvantages of using static analysis. The information collected during this type of analysis is often simple and not always sufficient for a conclusive decision on the malicious intent of a file and, most of all, malicious code can be purposefully designed to be hard to analyze statically. Overall, static analysis is a valuable tool for malware analysis. It is not always effective, but it can be used to identify the malware's code and functions, understand how the malware works, and develop effective detection and removal techniques.

2.6.2 Dynamic analysis

Dynamic analysis is a type of malware analysis that examines the code of an executable file while it is running. This can be done in a controlled environment, such as a sandbox or virtual machine. Dynamic analysis can be used to identify the malware's code and functions, as well as its behavior. Dynamic analysis is more effective than static analysis at detecting malware that has been designed to evade static analysis. This is because dynamic analysis can see how the malware behaves when it is actually running, while static analysis can only see the code. However, dynamic analysis is also more time-consuming and computationally expensive than static analysis. This is because dynamic analysis requires running the malware, which can take a long time and can use a lot of resources. There are two main types of dynamic analysis: basic and advanced. Basic dynamic analysis involves observing the sample under analysis interacting with the system. Advanced dynamic analysis, on the other hand, consists of directly examining the executed malware internal state while it is being run. When using dynamic techniques, malware analysts don't simply run malware executables on their own systems, which most probably are connected to the Internet, as they could easily escape the analysis environment and infect other hosts/networks. It is, in fact, advised to deploy dynamic techniques on safe and controlled (isolated) environments such as dedicated stand-alone hosts, virtual machines or emulators. The use of clean dedicated hosts, reinstalled after each dynamic analysis run, is however not the most efficient solution due to the environment re-installation process overheads. On the other hand, using virtual machines (for example *VMware*) to perform dynamic analysis is more efficient. In fact, in this case, since the malware only affects the virtual machine environment, it is enough, after a dynamic analysis run, to simply discard the infected hard disk image and replace it with a clean one. Unfortunately, a significant drawback is that the malware being analysed may determine it is running in a virtualized environment and, as a result, modify its behaviour. To counter this last problem one could make use of emulators, which are theoretically undetectable by analysed malware. These tools, however, run the code under analysis significantly slower and are therefore sometimes detectable using specially crafted time-related code. Overall, dynamic analysis is a valuable tool for malware analysis. It is more effective than static analysis at detecting malware that has been designed to evade static analysis. However, dynamic analysis is also more time-consuming and computationally expensive than static analysis.

2.6.3 Disassemblers

In the field of computer science and reverse engineering, disassemblers play a pivotal role in analyzing and understanding the inner workings of executable files. A disassembler is a software tool designed to convert machine code, which is the binary representation of a program, back into human-readable assembly code, making it easier to understand. By doing so, disassemblers enable developers, security researchers, and hackers alike to gain insights into the functionality, vulnerabilities and overall structure of compiled software. There are two types of disassembler algorithms: linear and flow-oriented (recursive). The linear variant is relatively simpler to implement but is also more basic and prone to errors compared to the flow-oriented approach.

- **Linear Disassembler.** The linear disassembly strategy is founded on the fundamental assumption that a program's instructions are arranged in a linear sequence, one after another. This type of disassembler follows a straightforward approach by iteratively disassembling each instruction in a code block without deviation. It disassembles instructions sequentially, relying on the size of the currently processed instruction to determine which bytes to disassemble next, disregarding control-flow instructions [72].

Linear disassemblers are relatively simple to implement and generally work well when handling small code segments. However, they can introduce occasional errors even with benign binaries. One of the main drawbacks of this technique is that it blindly disassembles code until the end of the section, assuming that the encountered bytes consist solely of packed instructions, without distinguishing between code, data, and pointers.

For instance, in a PE-formatted executable file, the executable code is typically located within a designated *.text* section. However, in most binaries, this code section may also contain data, including pointer values. The linear disassembler blindly disassembles and interprets these pointers as instructions. Malware authors can exploit this behavior by, for example, embedding data bytes in the code section that form the opcodes of multi-byte instructions.

- **Flow-Oriented Disassembler.** The flow-oriented disassembly strategy, which is widely utilized by commercial disassemblers such as *IDA Pro* [72], is a more advanced approach compared to the previous method.

Unlike the linear strategy, the flow-oriented strategy thoroughly examines each instruction, constructs a list of locations to be disassembled (those reached by the code), and keeps track of the code flow. This means that, while disassembling a code section, this type of disassembler does not blindly parse the bytes immediately following JMP instructions. Instead, it disassembles the bytes located at the addresses specified by the jump destinations.

This behavior enhances resilience and generally yields better results compared to the linear approach. However, it also introduces greater complexity. Unlike a linear disassembler, which disassembles instructions in a straightforward sequence, flow-oriented disassemblers must make choices and assumptions, especially when encountering conditional branches and call instructions. Specifically, when dealing with conditional branches, the disassembler needs to follow both the false branch (usually processed first by most flow-oriented disassemblers) and the true branch. In most cases, compiler-generated code would produce identical output regardless of which branch is processed first. However, in handwritten assembly code and anti-disassembly techniques, processing one branch before the other can result in different disassembly listings for the same block of code, which can lead to analysis issues.

2.7 Anti-detection techniques

Since the inception of malware in the 1960s, initially created for amusement by hackers, a fierce competition has emerged between attackers and defenders. Anti-malware groups have been diligently developing sophisticated and ingenious techniques to safeguard against malware attacks. Simultaneously, malware developers have been devising and implementing new strategies and

methods to elude detection by anti-malware systems. Early anti-malware tools primarily operated on the assumption that the structure of malware remains relatively unchanged over time. Initially, the machine code of malware was entirely unprotected, allowing analysts to leverage opcode sequences to identify specific malware families. However, significant progress has been made, giving rise to the advent of second-generation malware [73]. This new breed of malware employs various obfuscation techniques and has the ability to generate multiple variants of itself, specifically designed to evade detection based on opcode signatures. This has posed a formidable challenge for anti-malware developers. The first instance of malware demonstrating detection avoidance behavior was the Brain virus [74], which emerged in 1986. This malware concealed the infected section of a disk, tricking the computer into displaying clean data instead of the compromised portion whenever a user attempted to access it. Since then, the growing popularity of detection evasion techniques among malware creators has underscored the paramount importance of malware survival within a victim's system. The longer malware remains undetected, the greater the harm it can inflict and the more profitable it becomes for its author. Another aspect to take into consideration are the effects that anti-detection techniques produce to the interpretability of malware detection methods. Security analysts normally have to produce much effort on extracting knowledge from malware, especially when malware goes through a lot of obfuscation and becomes intricately difficult to understand.

2.7.1 Anti-disassembly

Anti-disassembly techniques involve the deliberate use of specially designed code or data within a program to mislead disassembly analysis tools, resulting in an inaccurate program listing [72]. When attackers employ these techniques, it forces malware analysts to invest significant time in analysis, making it difficult to retrieve or reconstruct the source code within a reasonable timeframe. Additionally, anti-disassembly techniques can impede various automated analysis tools and heuristic-based engines that rely on disassembly analysis to detect or classify malware. These techniques exploit the inherent weaknesses and assumptions of disassemblers when analyzing code, providing an opportunity for malware authors to deceive the analysis process. For instance, during the disassembly of a program, each sequence of executable code instructions can have multiple representations, some of which may be invalid and obscure the true intentions of the program. Consequently, malware authors can incorporate code sequences that mislead the disassembler, resulting in an output of instructions that differ from those executed during runtime [72]. A simple example of anti-disassembly construct can be created by inserting in the code two conditional *jump* instructions (one the opposite of the other, such as *jz* and *jnz*) that have the same target. This implies that the target location will always be reached by the code and so the combination of the two jumps acts like an unconditional jump instruction. Although this example may appear simple, it can deceive a linear disassembler. However, a flow-oriented disassembler would overcome this malware author's attempt. To fool this type of disassembler a control-flow obscuring technique, such as *Return Pointer Abuse* [72], must be used.

2.7.2 Code obfuscation

Code obfuscation is the process of intentionally modifying source code to make it more difficult to understand, analyze, or reverse engineer, both for humans and automated tools, while maintaining its original functionality. Consequently, the new version of the program is considered computationally equivalent to the original [75]. Initially, this technology was developed to safeguard the intellectual property of software developers. However, it has also been extensively exploited by malware creators to evade detection. In particular, malware employs obfuscation techniques to alter its code and create new iterations [75], which can be even more resistant to disassembly and analysis. It involves applying various techniques and transformations to the code, such as renaming identifiers, altering control flow, encrypting or encoding sections of code, inserting irrelevant or redundant code snippets, and employing anti-debugging mechanisms. All these processes badly impact the interpretability of malware detection, making the code harder to understand and analyze, hindering the interpretation of its behavior. Obfuscation techniques can be broadly categorized into two main types:

- **Data-based obfuscation.** Data-driven obfuscation methods center around altering the values of data and non-control computations. This type of obfuscation usually includes various simple but effective techniques that make the analysis slower. Some of these techniques are *constant folding, dead code insertion and arithmetic substitution via identities* [75].
- **Control-based obfuscation.** Control-based obfuscation, on the other hand, modifies the code's control flow structures in unconventional manners to deliberately complicate both static and dynamic code analysis [75]. Quite all static analysis tools, for example, take for granted that function calls result in a *return* or that the *call* instruction is exclusively used for invoking functions and so on. When one or more of these assumptions is violated, the disassembler and other static tools are tricked and may produce a meaningless source code or an unintelligible control flow graph.

Nevertheless, malware authors frequently combine these two types of obfuscation techniques in intricate and inventive ways to enhance the resulting protection [75].

2.7.3 Obfuscated malware

In the recent era advanced malware make their appearance [52]. Their main characteristic is that they dynamically modify their internal structure across different variations while they maintain their malevolent behavior, aiming to evade detection [75]. This results in variations that make it difficult to recognize and categorize based on traditional detection methods, reducing drastically interpretability of detection techniques. Two main types of these advanced malware are *packed malware* and *polymorphic malware*.

Packed malware

Packers are employed to compress original executable files into smaller sizes. Additionally, encryption is sometimes applied to the compressed files to make the unpacking process more challenging. In recent times, there has been a growing trend among malware creators to exploit packers (or even multiple packers simultaneously) in order to create multiple variations of the same original malware code. According to Perdisci et al. [76], over 80% of newly discovered malware are actually packed versions of pre-existing malware.

Polymorphic malware

Polymorphic malware consists of two parts: an encrypted main body and a decryptor. When the malware is executed, the decryptor runs and allows the original malware code to be executed by decrypting the encrypted body. During replication, the malware encrypts its code using a different key and generates a new decryptor, which is then enclosed within the new variant of the malware. However, *polymorphic malware* employs a powerful toolkit called the *Mutation Engine* (MtE) to generate countless distinct decryptors. The *Mutation Engine* uses various obfuscation methods, such as *dead-code insertion, register reassignment, subroutine reordering and instruction substitution* [75], to rearrange the decryptor code and prevent detection based on signatures. As a result, the appearance of the malware changes with each infection, posing even more challenges for interpretability. Malware authors employ various techniques to armor the malware and evade detection through emulation. However, most antivirus scanners are now capable of effectively addressing these techniques and defeating polymorphic malware.

Chapter 3

Related work

3.1 Definition of interpretability in ML

IML is a branch of ML that focuses on creating algorithms that are easily comprehensible and interpretable by humans. The field lacks a definitive explanation of interpretability, and there is currently no mathematical method to measure the interpretability of a model or assess the validity of an explanation [77]. Evaluating interpretability can be done through two main methods: objective evaluations, which are quantifiable metrics, and human-centered evaluations, which involve input from domain experts or non-experts. The former, also known as algorithm-centered evaluations, typically involve quantitative analysis and report on the performance of the model's properties. Supervised ML techniques can be evaluated using clear metrics such as accuracy and F-score, while the evaluation of unsupervised learning methods is more complex, with metrics typically divided into internal and external validation.

On the other hand, a comprehensive framework for human-center evaluations is currently lacking. Human-center evaluations can be both qualitative and quantitative, as these two aspects are independent of each other. Ideally, Human-center evaluations should take into account both the cognitive and emotional aspects involved in human-machine interactions, such as co-adaption and co-creation, going beyond just assessing usability and user experience. [78].

3.1.1 Properties of explanations

The goal is to offer an explanation for the predictions made by a machine learning model. This explanation typically establishes a distinct correlation between the feature values of a given instance and the model's prediction. Different forms of explanations can be derived from a group of data instances, such as the k-nearest neighbor model or, for example, we can utilize a support vector machine to predict malware presence and then use the local surrogate approach, which generates decision trees as explanations, to elucidate the predictions. Alternatively, we can replace the support vector machine with a linear regression model, which inherently includes an explanation method (interpretation of the weights). Looking closer to these explanations and explanation methods we can derive some properties that can be used to judge how good they are and to classify them better [77]. These are not standardized properties, so for many of them there is not a clear method to measure them correctly.

Properties of explanation methods

- **Expressive power** refers to the ability of the method to produce various types of explanations, such as IF-THEN rules, decision trees, weighted sums, natural language, or others.
- **Translucency** describes to the degree to which the explanation method relies on inspecting the inner workings of the machine learning model, including its parameters. For instance,

methods based on inherently interpretable models like linear regression are high translucent, while those solely manipulating inputs and observing predictions demonstrate zero translucency. The desired level of translucency depends on the specific context. High translucency allows the method to utilize more information for generating explanations, whereas low translucency enhances the explanation method’s portability.

- **Portability** represents the extent to which the explanation method can be applied to different machine learning models. Methods with low translucency achieve higher portability as they treat the machine learning model as a black box. Surrogate models might be the most portable explanation method. Conversely, methods designed solely for specific models, such as recurrent neural networks, have limited portability.
- **Algorithmic complexity** characterizes the computational complexity of the explanation generation method. This property becomes crucial when computation time becomes a bottleneck in producing explanations.

Properties of explanations

- **Accuracy** is the ability of an explanation to effectively predict unseen data. If the explanation is used for predictions instead of the machine learning model, high accuracy is crucial. However, when both the model and explanation have low accuracy, the focus may be on explaining the black box model’s behavior, where only fidelity becomes significant.
- **Fidelity** describes how well the explanation approximates the predictions of the black box model. High fidelity is essential for an explanation to be valuable in understanding the machine learning model. Accuracy and fidelity are closely related; when the black box model has high accuracy and the explanation has high fidelity, the explanation also exhibits high accuracy. Some explanations offer only local fidelity, where they approximate well for specific subsets of data or individual instances, such as with local surrogate models or Shapley Values.
- **Consistency** is the degree to which explanations differ between models that were trained for the same task and produce similar predictions. High consistency indicates that explanations remain similar across different models, implying reliability. However, it becomes tricky when two models with similar predictions use different features (*Rashomon Effect*), where high consistency may not be desirable as the explanations should differ significantly.
- **Stability** shows how similar the explanations are for similar instances within the same model. High stability means slight variations in instance features do not substantially change the explanation, except when these variations strongly affect the prediction. A lack of stability can result from high variance in the explanation method or non-deterministic components, like data sampling in local surrogate methods. High stability is always desirable.
- **Comprehensibility** is the extent to which humans can understand the explanations. While challenging to define and measure, it is crucial for effective interpretability. Comprehensibility can vary depending on the audience and can be measured by factors like explanation size (number of features with non-zero weight in a linear model) or testing how well people can predict the model’s behavior from the explanations. Complex feature transformations may reduce comprehensibility.
- **Certainty** expresses whether the explanation reflects the certainty of the machine learning model. Some models provide predictions without indicating confidence levels. An explanation that includes the model’s certainty is highly useful.
- **Degree of importance** describes how well the explanation represents the significance of features or parts of the explanation. For instance, if a decision rule is generated as an explanation, it should clearly highlight the most important conditions of the rule.

- **Novelty** is the extent to which the explanation reflects whether a data instance belongs to a *new* region, far from the training data distribution. High novelty may indicate low model accuracy and render the explanation less useful.
- **Representativeness** shows the coverage of instances by an explanation. Explanations can span the entire model (e.g., interpreting weights in a linear regression model) or focus on individual predictions (e.g., *Shapley Values*).

3.2 Interpretable machine learning classification and techniques

Interpretable machine learning techniques offer numerous benefits. They improve transparency and trust in classification models, facilitate model validation and help users to identify and address potential biases or errors. Interpretable models enable collaboration between domain experts and data scientists, as they can collectively analyze and refine the models based on the interpretable insights. Moreover, interpretability provides valuable explanations for end-users, allowing them to comprehend and act upon the model’s predictions with confidence. These techniques can be categorized through various taxonomies based on different aspects of evaluation as it is illustrated in diagram 3.1.

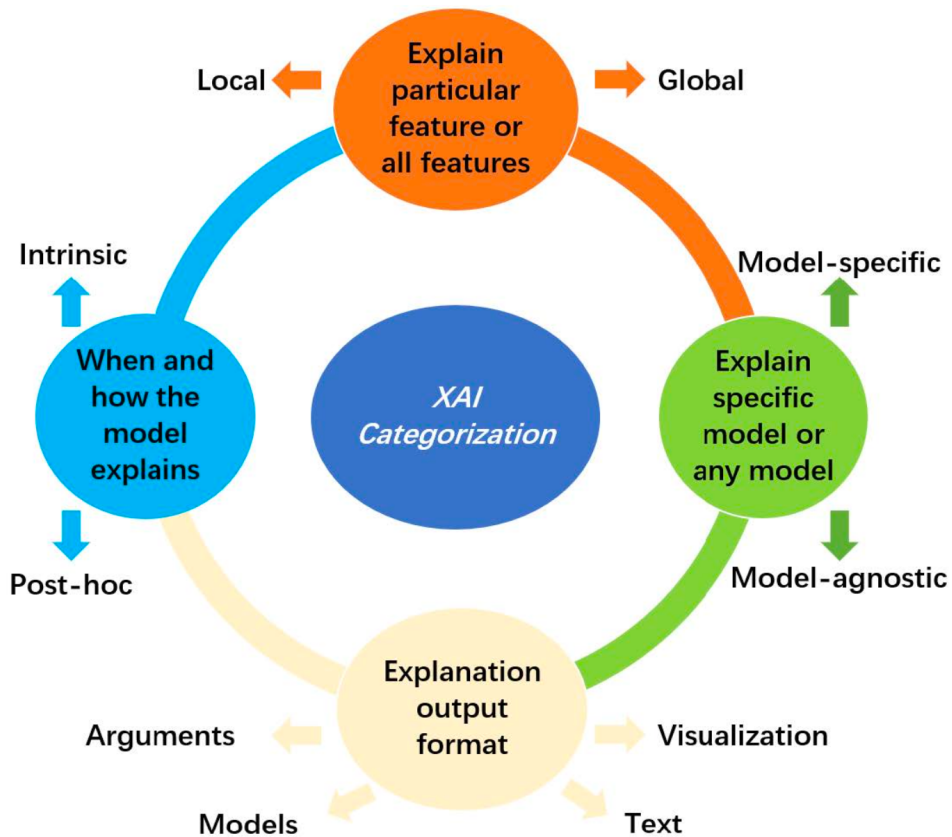


Figure 3.1. Overview diagram of the XAI categorization (source: [4])

3.2.1 Based on the scope

- **Local interpretability:** it refers to the ability to understand and explain the predictions of an AI model on an individual instance or data point. It involves generating explanations

concurrently with the model's predictions by using data that the model produces during the prediction-making process. Local interpretability techniques aim to explain individual predictions or decisions made by the model (figure 3.2). They provide insights into why a specific prediction was made by identifying the important features or factors contributing to that particular prediction. For example, Decision Trees and Sparse Linear models are considered to have local interpretability because they inherently provide self-explanatory insights into their decision-making.

- **Global interpretability:** it focuses on understanding the overall methodology and behavior of an AI model after it has been trained and decisions have already been made. It involves applying interpretation methods to analyze the model retrospectively. Techniques in this category aim to provide an overall understanding of how the model works and the factors influencing its predictions. These techniques focus on explaining the model's behavior as a whole, rather than individual predictions (figure 3.2). One common post-hoc explanation method is Local Interpretable Model-agnostic Explanations (LIME), which works independently as an external interpretable model. LIME aims to explain the predictions of any black-box model by generating locally faithful explanations. Another post-hoc method is Permutation Importance, which measures the importance of features by permuting their values and evaluating the impact on the model's performance.

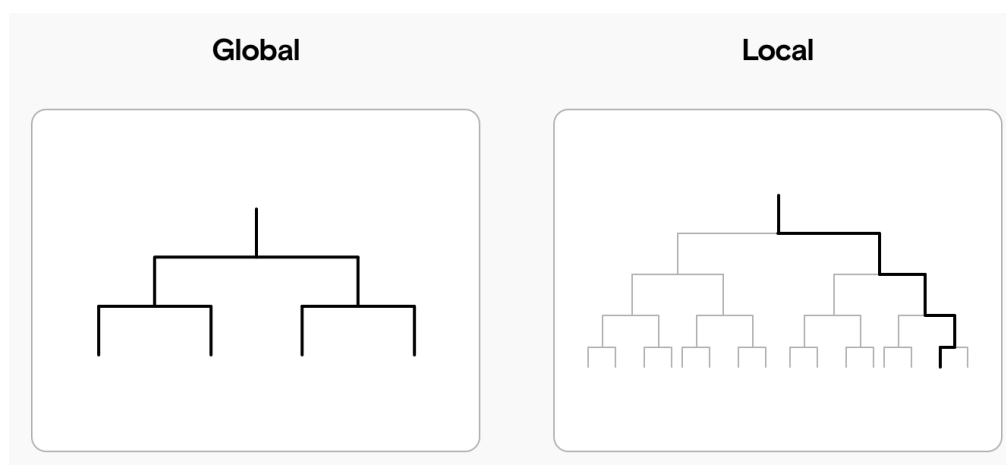


Figure 3.2. Illustration of Global and Local Interpretability aspect (source: <https://www.datarevenue.com/en-blog/interpretable-machine-learning>)

3.2.2 Model-specific methods

Model-specific techniques are tailored to a specific type of machine learning model or algorithm. These techniques leverage the inherent structure or characteristics of the model to provide interpretability. Some of the major interpretable models are:

- **Linear model.** Linear models, such as logistic regression or linear SVM, have a simple and transparent structure, making them inherently interpretable. The coefficients assigned to each feature in the linear model can be directly interpreted as the importance or contribution of that feature to the predictions [79]. Positive coefficients indicate a positive association with the predicted class, while negative coefficients indicate a negative association.

In addition to the coefficients, model-specific methods for linear models may include:

- *Coefficient Magnitude and Significance:* by examining the magnitude and statistical significance of the coefficients, one can determine which features have the most impact on the predictions.

- *Feature Scaling*: *scaling* the features in a linear model can help in interpreting the importance of each feature more accurately. The magnitude of the coefficients can be directly compared to assess the relative importance of different features.

Nevertheless, there are constraints to consider. Outliers have huge effects on regression and boundaries, thereby impacting the performance of the models. Moreover, these models assume attribute independence, which limits their ability to accurately represent feature relationships.

- **Decision Trees and Random Forest.** Decision trees and ensemble methods like random forests have an inherent structure that can be visualized and interpreted [80]. Interpretable methods specific to decision trees and random forests include:
 - *Tree Visualization*: decision trees can be visualized as hierarchical structures, with each node representing a decision based on a feature, and each leaf node representing a class prediction. By examining the decision paths and the splits at each node, one can understand the decision-making process of the model.
 - *Feature Importance Measures*: random forests provide measures of feature importance based on how much each feature decreases the impurity or error in the ensemble. These measures, such as *Gini* importance or *mean decrease impurity*, rank the features by their importance, allowing for interpretation of feature contributions. This work will also analyze the *Recursive Feature Elimination* later, as a feature importance measure that recursively delete the most useless feature.

Nevertheless, decision trees struggle to handle linear relationships and lack smoothness. Moreover, trees are inherently unstable. As the depth increases, the number of leaf nodes grows rapidly, posing challenges for interpretation.

- **Decision Rule.** Decision rule models typically produce rules in the form of *if-then* statements, where specific conditions on input features determine the predicted outcome [81]. These rules are often straightforward and can be expressed in a human-readable format, making them highly interpretable. By examining the generated rules, analysts can identify potential biases, errors, or inconsistencies in the model. They can also evaluate whether the rules align with domain knowledge and expectations, further enhancing the model’s reliability. However, it is important to note that decision rule models also have limitations in terms of interpretability. In complex domains with high-dimensional feature spaces, decision rules may become numerous, lengthy and intricate, making interpretation more challenging.

3.2.3 Model-agnostic methods

A model-agnostic method refers to an approach that separates the interpretation process from the specific machine learning models used [79]. This method offers the advantage of providing greater flexibility in interpretation compared to model-specific methods. By being independent of the underlying model architecture, model-agnostic methods can be applied to a wide range of models without being restricted to a particular type. Their aim is to provide explanations for the predictions of any black-box model, regardless of its underlying algorithm. These methods operate independently of the specific model architecture and focus on understanding the model’s behavior through perturbations and approximations. Some major methods are:

- **Partial Dependence Plot (PDP).** A partial dependence plot is a visualization technique used to understand the relationship between a target feature and the predicted outcome while holding all other features constant. It allows to examine the impact of a specific feature on the model’s predictions, controlling for the effects of other variables. Interpreting a partial dependence plot involves analyzing the shape and trend of the curve. A positive slope indicates a positive correlation, suggesting that higher values of the feature are associated with higher predictions. A negative slope suggests a negative correlation, meaning that higher feature values lead to lower predictions. A flat or near-flat curve indicates that the selected feature has little to no influence on the model’s predictions. Its computation is

intuitive and it works very well if the features are independent [79]. When the assumption of independence is not true, this method cannot bring valuable results.

- **Individual Conditional Expectation (ICE).** ICE is a technique used to understand and visualize the relationship between a specific input feature and the corresponding output predictions of a machine learning model. It provides a detailed understanding of how the model's predictions change as the value of a particular feature varies. The ICE plot focuses on a single instance or observation and examines how the predicted outcome varies by manipulating a specific feature while holding all other features constant. By plotting the feature values on the x-axis and the corresponding predicted outcomes on the y-axis, an ICE plot visualizes the individual responses of the model for the chosen instance. This plot consists of a set of curves, each representing the changing predictions for different values of the selected feature. Each curve on the plot represents the predicted outcome for a specific feature value while holding other features constant. ICE plots are particularly useful in cases where there is a need to explain the impact of a single feature on the model's predictions while considering the influence of other variables. They offer interpretability by allowing users to observe how the model's predictions change for different feature values, aiding in the detection of potential biases. The main limitation is that these plots are designed to display the relationship between a single feature and the predicted outcomes, meaning they can effectively represent only one feature at a time. Additionally, ICE plots assume independence among input features, which may not hold true in all cases.
- **Accumulated Local Effect (ALE).** ALE plot is a technique used to visualize the relationship between a specific input feature and the model's predictions. It provides insights into how the feature's values influence the model's output while controlling for the effects of other variables. The ALE plot calculates the average effect of varying a single feature on the predicted outcome by considering its interactions with the other features in the dataset. It does so by dividing the feature's range into equally spaced intervals and calculating the difference in predictions between each interval while keeping the other features fixed. The differences are then accumulated to show the overall effect of the feature across its entire range. Analysts can gain a better understanding of how changes in the selected feature's values affect the model's predictions. They can identify non-linear relationships, interactions with other features, and potential thresholds or ranges where the feature has a significant impact on the outcome. The ALE plot enhances interpretability by providing a concise and intuitive representation of the feature's influence on the model's behavior.
- **Local Interpretable Model-agnostic Model (LIME).** LIME aims to provide local explanations for individual predictions made by complex models. LIME works by generating local surrogate models to approximate the behavior of the black-box model around a specific instance of interest. The process begins by selecting an instance for which an explanation is desired. Perturbed instances are then generated by introducing small changes to the feature values of the selected instance while keeping the ground truth label fixed. These perturbed instances are passed through the black-box model and their predictions are recorded. A subset of the perturbed instances, along with their predictions, is chosen to form a neighborhood. The selection can be based on criteria such as diversity or similarity to the original instance. The neighborhood is then used to train a surrogate model that is interpretable, such as a linear model or a decision tree. To generate explanations, the perturbed instances in the neighborhood are weighted based on their similarity to the original instance. Instances that are more similar receive higher weights, indicating their importance in the explanation. The surrogate model is fitted using the weighted instances and their corresponding predictions. LIME provides local explanations that help in understanding how the black-box model arrived at its predictions for specific instances. It does not offer a comprehensive understanding of the model's behavior but focuses on explaining individual predictions. There are limitations associated with using LIME for tabular data, which include challenges in determining the appropriate kernel setting and defining the neighborhood accurately. These issues remain unresolved and it can be difficult to fully trust the outcomes provided by LIME.
- **Shapley Additive Explanations (SHAP).** SHAP aims to explain the predictions made

by complex models by providing a unified framework based on cooperative game theory to allocate the contribution of each feature in the prediction process. SHAP works by considering different sets of features and evaluating their impact on the prediction. To calculate SHAP values, the technique uses a unified framework that combines the idea of local explanations with a global perspective. It creates a set of weighted samples from the training data, where each sample represents a specific combination of features. For each sample, it computes the model’s prediction and assigns weights based on the feature importance. Next, SHAP values are computed by iteratively attributing the differences in model predictions to the features. It evaluates the contribution of each feature by considering its inclusion or exclusion in every possible subset of features and calculating the change in prediction. This process accounts for interactions and dependencies between features, providing a more accurate explanation of feature importance. The SHAP values are then aggregated to ensure consistency and global interpretability. This step produces a set of values that represent the relative importance of each feature in the prediction. The final SHAP values can be used to interpret the model’s predictions by understanding the impact of each feature on a particular instance. Positive SHAP values indicate a positive influence on the prediction, while negative values suggest a negative influence. The Shapley value offers contrasting explanations and stands out as the only explanation method grounded in a robust theoretical foundation. It is remarkable to depict a prediction as a game where the feature values play a role. However, a drawback is that computing Shapley values can be computationally intensive and time-consuming. It’s also important to note that Shapley values can be prone to misinterpretation and that explanations generated using this method always incorporate all the features. Additionally, unlike LIME, it cannot be used without a prediction model. Another requirement is access to the data, which can be problematic due to the inclusion of unrealistic data instances.

3.2.4 Intrinsic methods

Intrinsic interpretability methods refer to techniques used to understand and interpret the decision-making processes of machine learning models directly within the model itself by design, rather than relying on external methods or post-hoc explanations. These methods aim to provide insights into how a model arrives at its predictions by examining its internal mechanisms, such as *feature importance*, *attention mechanism* or *decision rules*.

- **Attention mechanism.** Bahdanau et al [82] introduced the attention mechanism to address the bottleneck that arises when using a fixed-length encoding vector, where the decoder’s access to the input information becomes limited. This is expected to pose a particular challenge for long and/or intricate sequences, as their representation’s dimensionality would be constrained to match that of shorter or simpler sequences. At its core, an attention mechanism assigns weights or scores to different elements of the input sequence based on their importance or relevance. These weights indicate the degree of attention or emphasis that the model should place on each element when processing the sequence. By allocating higher weights to more important elements, the model effectively learns to selectively attend to the most informative parts of the sequence. The process of attention typically involves three main components: the query, the key and the value [83]. The query represents the current position or element of interest in the sequence, while the key-value pairs represent the other elements in the sequence. The attention mechanism computes similarity scores between the query and each key, and these scores are used to determine the weights for the corresponding values. The values are then combined, weighted by their respective scores, to form the attended representation of the input sequence.
- **Layer-wise Relevance Propagation (LRP).** LRP is a method used to attribute the relevance or importance of each input feature to the final prediction made by a deep neural network. By backpropagating relevance scores from the output layer to the input layer, LRP highlights the contribution of each feature along the network’s layers [84]. To perform LRP, the relevance is initially assigned to the output neuron corresponding to the predicted class. From there, relevance is redistributed layer by layer, following certain propagation

rules. These rules are designed to allocate relevance to different neurons based on their activation values, weights, and connections within the network. LRP can be implemented using different propagation rules, such as the epsilon-rule, the alpha-beta rule, or the gamma rule, each with its own characteristics and properties. These rules determine how relevance is propagated from one layer to the next, taking into account factors such as positive and negative contributions, biases, and non-linear activation functions. The output of LRP is a relevance map that highlights the importance of different input features or neurons for the model's prediction. By visualizing these relevance scores or overlaying them on the input data, one can gain a better understanding of which regions or features the model focuses on when making predictions.

- **Knowledge distillation.** Knowledge distillation is a technique that transfers the knowledge and predictive capabilities of a complex, black-box model to a simpler and more interpretable model. The process involves training a simpler model, such as a decision tree, logistic regression, or shallow neural network, to mimic the behavior of the complex model with the goal of retain accuracy while gaining interpretability [85]. Initially, a complex model like a deep neural network or an ensemble is trained on a large dataset to achieve high predictive accuracy. Soft targets, which are probability distributions, are generated by the complex model for each input sample. Unlike hard class labels (e.g. encoded vectors representing the true class), these soft targets provide a nuanced understanding of the complex model's predictions. The simpler model, known as the *student model*, is then trained on the same dataset using the soft targets as additional supervision. It learns to mimic the behavior of the complex model, capturing the knowledge encoded in the soft targets. A temperature parameter is introduced during training to control the softness of the targets, allowing the distilled model to focus on capturing underlying patterns rather than specific details. After training, the performance of the distilled model is evaluated using standard metrics. It is expected to achieve comparable accuracy to the complex model while offering increased interpretability. Model distillation provides several benefits, including better understanding of decision rules and important features, increased transparency for interpretation and validation, and improved computational efficiency and resource requirements for deployment.

3.2.5 Post hoc methods

Post hoc interpretability methods refer to techniques used to retrospectively interpret the decisions or predictions made by machine learning models after they have been trained. These methods aim to provide insights into how a model arrived at its output, making it easier for humans to understand and trust the model's behavior without modifying the model's underlying structure or training process. Some of the most common post hoc techniques are: LIME and SHAP that exploit feature importance, PDPs, ICE, ALE

3.3 Projects

3.3.1 NODENS

NODENS is a lightweight machine learning-oriented solution for identifying malicious software, relying on *process signatures* that are generated by programs while they are running. The aim is to ensure the system remains light in order to be easily deployed on budget-friendly hardware. This is done following the subsequent goals:

- Nearly real-time malware detection system that alerts the user immediately to take care of the problem. This prevent a rapid spread of the threats and any further damage.
- Detection of *zero-day* vulnerability, to counter the threat of previously unseen malware.
- **Be interpretable.** This project is the first one that takes care of this aspect, trying to identify *why* a detection alert has occurred and *what* parts of the software have caused it.

- To possess the ability to readjust, make use of user feedback to confirm or counter automated choices and consistently enhance detection capability without depending on an online connection, signature updates or a sandbox environment.

Project methodology

This project takes malware from two of the most famous online malware datasets that are *VX Vault* and *Virus Share*. This malware run on a *Win7 VM* using *PowerShell* to collect processes and especially their details using the *Get-Process* command. This creates a list of the running processes, each one with 64 features, but then the unnecessary fields are deleted and only 22 of them are collected at the end. Subsequently, these extracted features are employed to evaluate a variety of classification methodologies (Random Forest, Gaussian Naive Bayes, Decision Tree, K Nearest Neighbour, Ada Boost, Support Vector Classifier, Gradient Boosting, Logistic Regression, One Class SVM). This evaluation is conducted using a collection of 55 distinct instances of malware through a k-fold cross-validation approach (with k equal to 7), featuring an 80% training and 20% validation data partition 3.3.

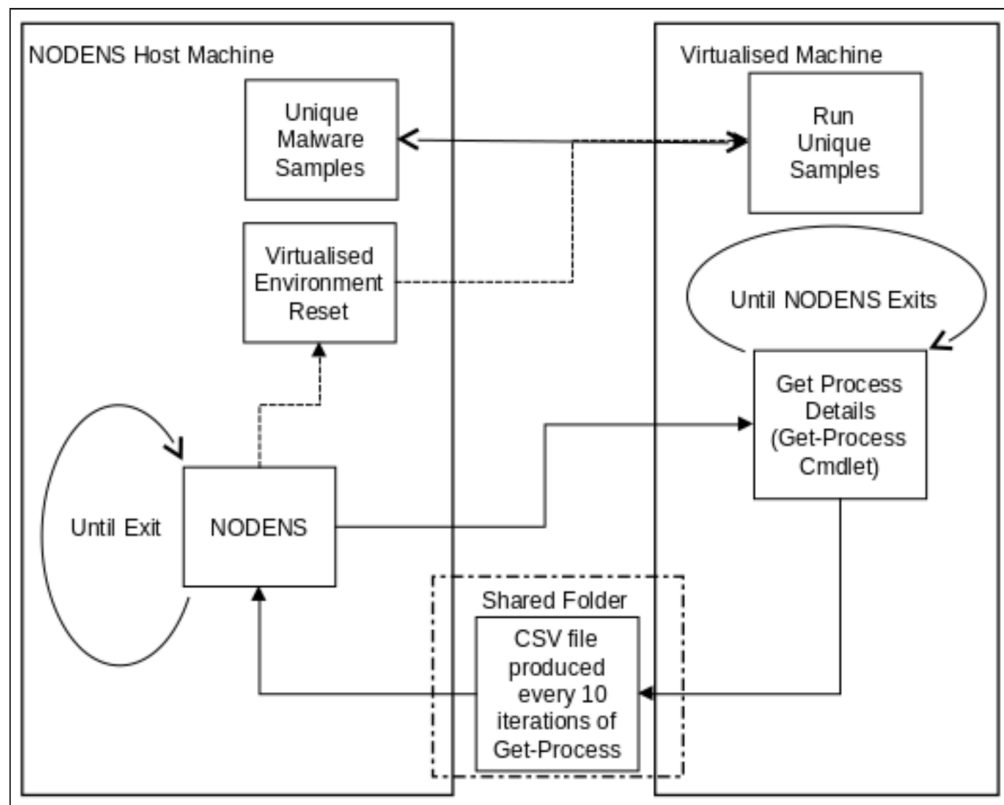


Figure 3.3. Training and algorithm selection process (source: [86])

At the end, the outcome revealed that among the tested techniques, Random Forest exhibited the most promising performance in this experimentation. Consequently, the researchers proceeded to train the Random Forest model using the complete dataset collection. Then the Random Forest Classifier was tested live 3.4, against previously unseen malware samples. Initially the delay was high (approx. 30 seconds between a malicious process being started and it being detected), but after the introduction of a whitelist approach for previously-known processes the delay was shortened to between 3-8 seconds. This demonstrates a future feasibility in a real world application for this project, at least for the live testing aspect. The usage of NODENS starts from a command line interface which allows an end user to utilise multiple different plugin scripts, such as being able to start and stop the data collection and detection process from the command line, as well the termination of malware processes and refitting of the classifier upon identification of previously

unseen malware. The refitting was proven to be effective as it allowed the system to identify malware processes it had previously not detected. This demonstrates that the system was able to increase its understanding of malware processes and points to a generalised link between all malware process behaviour, based on the ability for the system to use readily identified malware and their features to then identify those it had previously missed.

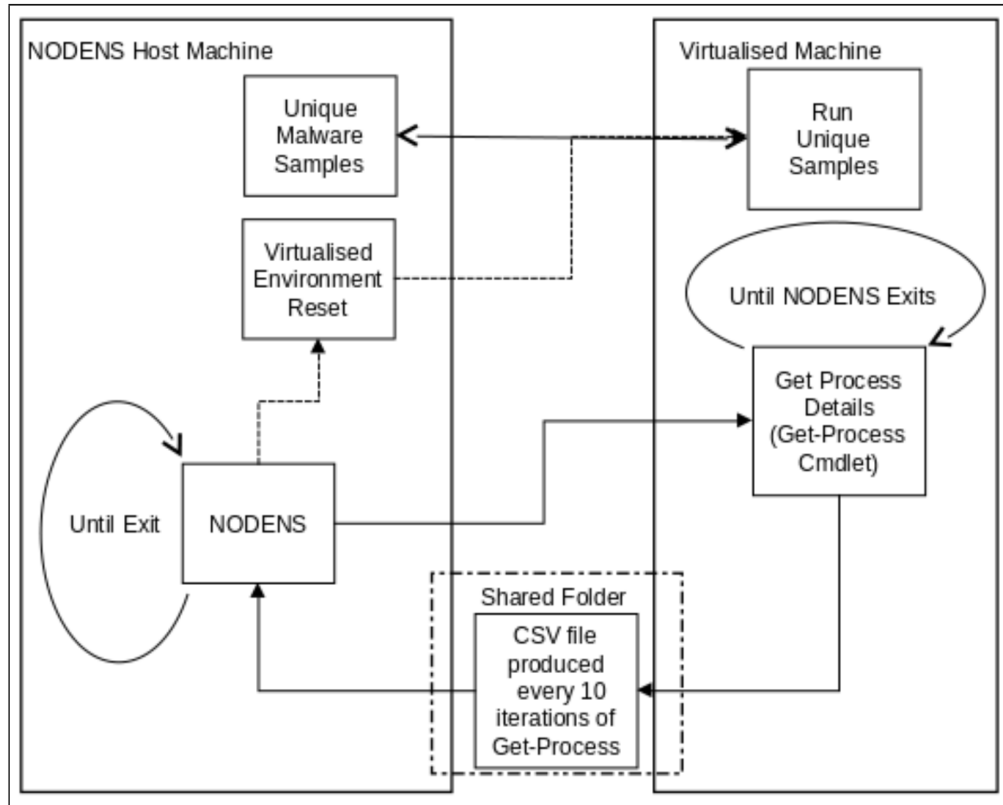


Figure 3.4. NODENS live process (source: [86])

Interpretability aspects

NODENS is capable to automatically output a CSV, or other different graphical files, that describes the decision-specific data. The first CSV output file shows which decision trees had classified a process as either malware or benignware, dependent on the end users selection, this included the nodes within those trees and their decision thresholds 3.5. Then also the capability to output *.json* files, with each file representing a specific decision tree and children nodes. There is also an option to output both a *.png* or *.dot* file 3.6 that are used also for visual interpretation. Using this approach, a final user can perceive the rationale behind classifying a process as malicious or benign, via a easily comprehensible result. This empowers the user to develop insight into the lifecycle of a malware instance, either on its own or as a component of a broader malware group, encompassing shared characteristics among distinct malware instances or categories. This was only a first step into interpretability applied in the malware recognition field. There is not any older project that take care of interpretability in malware identification, all the projects aimed at obtain the best accuracy and other metrics performances using black box approaches. Taking into account this aspect, we can say that this project opens the way to many other researches pointing out the importance of interpretability in malware identification and proposing a model that aims in this way. For sure the level of interpretability that this project proposes is not so powerful, for example we can not go backward the tree to discover the origin of the malware or what portion of the application is malicious.

TREE: 41
0 NODE: feature[ProcessorAffinity] > 0.5 next=76
76 NODE: feature[HasExited] < 0.5 next=77
77 NODE: feature[PeakVirtualMemorySize64] < 94238720.0 next=78
78 NODE: feature[UserProcessorTime] < 0.0150215998292 next=79
79 NODE: feature[PagedSystemMemorySize64] < 103664.0 next=80
80 LEAF: return class=0
TREE: 42
0 NODE: feature[TotalProcessorTime] < 0.675971984863 next=1
1 NODE: feature[HandleCount] < 89.5 next=2
2 NODE: feature[PrivateMemorySize64] > 2400256.0 next=38
38 NODE: feature[PeakWorkingSet64] < 7936000.0 next=39
39 NODE: feature[PeakVirtualMemorySize64] > 33705984.0 next=41
41 LEAF: return class=0
TREE: 43
TREE: 44
0 NODE: feature[TotalProcessorTime] < 0.675971984863 next=1
1 NODE: feature[PrivilegedProcessorTime] > 0.00500719994307 next=71
71 NODE: feature[PeakVirtualMemorySize64] < 66170880.0 next=72
72 NODE: feature[NonpagedSystemMemorySize64] > 2316.0 next=84
84 NODE: feature[HandleCount] < 68.0 next=85
85 NODE: feature[PeakVirtualMemorySize64] < 53284864.0 next=86
86 NODE: feature[PeakWorkingSet64] < 5062656.0 next=87
87 NODE: feature[Description] < 0.5 next=88
88 LEAF: return class=0

Figure 3.5. NODENS .CSV output (source: [86])

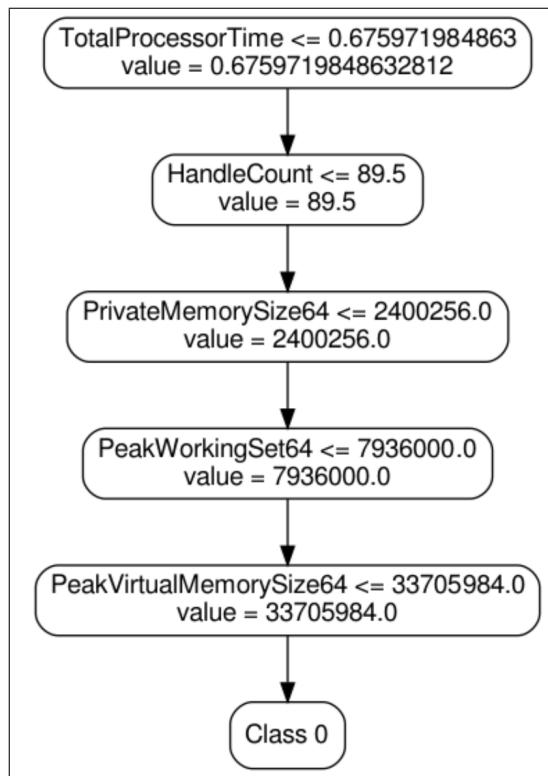


Figure 3.6. NODENS .dot output (source: [86])

3.3.2 PAIRED

The idea behind this project is to design a lightweight malware detection system [87]. This system relies on the analysis of static features extracted from the installed applications to assess their potential threat. The selection of these static features was guided by the aim to expedite decision-making within the system, avoiding the need to wait until an application exhibits malicious behavior, as this delay may result in irreversible harm for the system. The model explainability was carried out through Shapley values that guarantees the interpretable conditions that bring to high accuracy results.

The guiding principles underlying PAIRED can be summarized in these few points:

- **High accuracy:** in the aims of PAIRED’s design philosophy lies the pursuit of high accuracy. Here, the emphasis is placed on selecting a selected subset of features that exhibit the highest classification effectiveness. The crux of this approach is to maximize the model’s predictive power while minimizing the computational consumption.
- **Being lightweight:** another critical facet of PAIRED is its lightweight nature. This principle hinges on a substantial reduction in the number of features, a remarkable reduction to the tune of 84%. This substantially reduces the computational resources, both in terms of processing power and memory, required for feature extraction and classifier execution. Furthermore, this philosophy extends to the choice of a classifier that is inherently resource-friendly, thereby further lightening the system’s overall load.
- **High efficiency:** this is a central pillar of PAIRED’s design system. By reducing the number of features, the system makes rapid decisions an achievable goal. It significantly mitigates the time required for the decision-making process. Perhaps the most notable contribution to enhanced efficiency is the substantial reduction in the time needed for feature extraction.
- **Creating a generalizable classifier:** the final principle guiding PAIRED’s design is the creation of a classifier that can adapt to novel situations. This implies rigorous testing of the trained model against data subsets that were not part of the initial training. The goal here is to ensure that the system exhibits robust generalization, capable of making accurate decisions in diverse and uncharted territory.

In conclusion, the design principles of PAIRED encompass precision through feature selection, lightweight resource management, operational efficiency and the cultivation of a classifier with broad generalization capabilities. These principles collectively form the foundation of a system that balances computational efficiency with accuracy and adaptability.

Project methodology

Figure 3.7 presents an overview of the PAIRED system architecture assuming that the operation is triggered by the installation of a new application on the device where PAIRED system is present. They used the Drebin-215 dataset introduced for the first time in 2014. 215 is the number of features extracted for each sample using static analysis of malicious applications. The dataset included 15036 applications, with a total of 9476 instances that are benign and the remaining 5560 are malicious. The dataset is used to train and test 5 different types of classifier: random forest, logistic regression, decision tree, Gaussian Naive Bayes and support vector machine. The deep neural networks are not taken into consideration because of their high computational requirements compared to the five selected models. The classifiers were trained using 75% of the dataset and tested using the remaining 25%. The one that performed best was the random forest with an accuracy of 0.9865.

The core of this project is the reduction of the number of features taken into consideration without compromising accuracy of the detection system. The method used in the feature selection was recursive feature elimination (RFE) based on the feature importance. In each cycle of RFE, the dataset is randomly split into 75% training subset, and 25% testing subset. A random

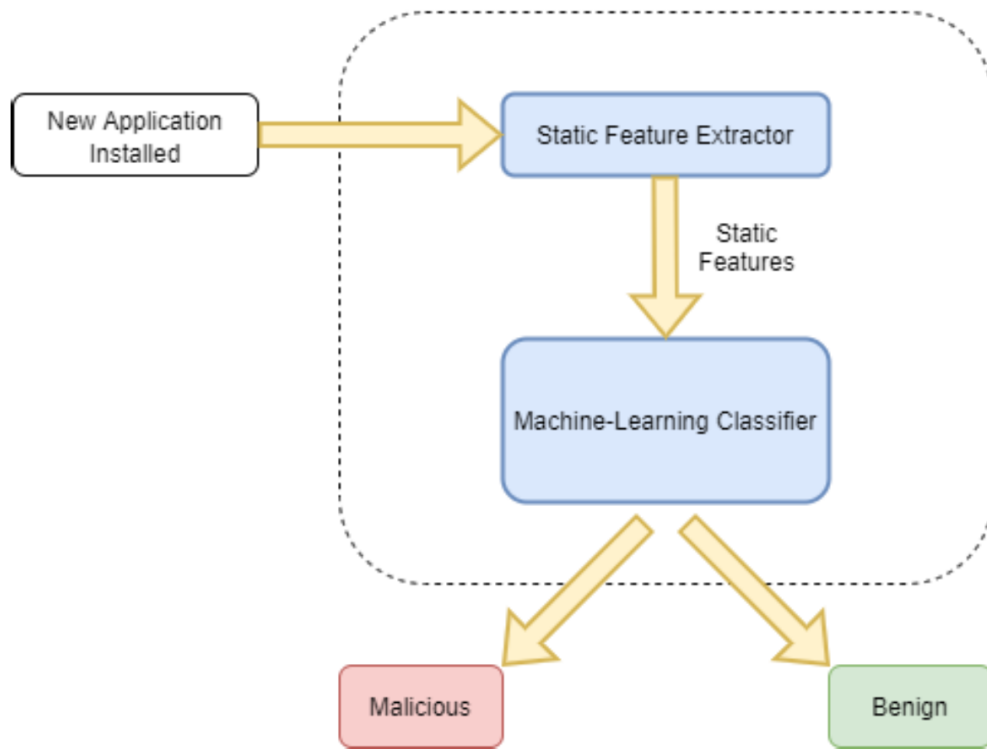


Figure 3.7. Overview of the PAIRED architecture (source: [87])

forest classifier is then created and trained using the training subset. Following this training, the classifier is tested using the testing subset. Afterwards, the feature importance is calculated for all features used in training the classifier. Feature importance can be measured as the averaged impurity decrease computed from all decision trees in the forest. The feature with the lowest importance value is then eliminated. Then, the cycle is repeated while closely monitoring the system’s performance by recording the F_1 score of the tested classifier. The process is repeated until a certain threshold is reached. This threshold is chosen upon the system witnessing a rapid drop in performance, that was 35, as they noticed a rapid drop in the classifier’s F_1 score beyond the 35-feature threshold (fig 3.8). This successive elimination assured that any correlation between the features will not affect the independent feature importance. In other words, the importance of one feature might be affected by the existence (and the elimination of) another feature. Hence, they retrained and re-calculated the importance after each feature was eliminated. Then, the new dataset with 35 features is used to train and test the classifier models and they shows that the random forest outperformed the other classifiers in terms of accuracy and F_1 score.

Interpretability aspects

Interpretability of this model relies on Shapley Additive Explanation because it can provide better insights into the impact of each individual feature on the classification decision compared to feature importance. The latter is calculated through Gini importance using node impurity and so can be used only with linear machine learning algorithms, while Shapley values are model-agnostic, so can be used with any classifier, including deep learning.

In Figure 3.9 there is a summary plot of the Shapley values of the top 10 features within the dataset, from the most important (the one with highest impact) to the lowest one on the classification decision. The prediction is set to 1 when the application is seen as a malware and 0 when it is seen as a benign app. In Figure 3.9 the values on the left side of the axis drag the prediction value down, making the prediction closer to *benign*, while the values on the right side of vertical axis move the prediction up, closer to malware. Red dots represent a high feature

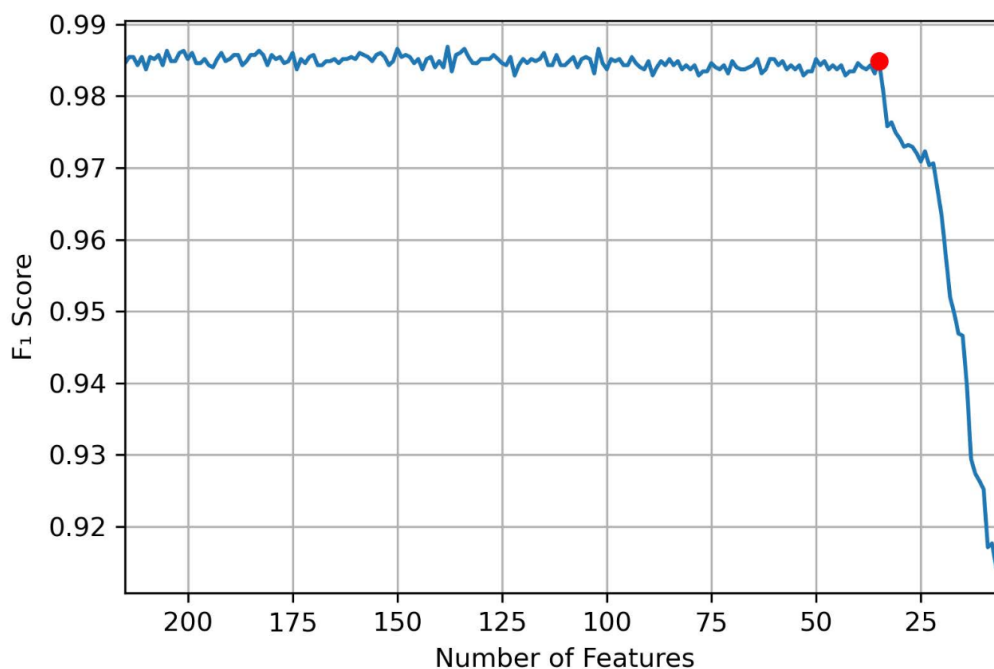


Figure 3.8. Impact of the feature reduction on the generated F_1 score. (source: [87])

value, while blue dots represent low feature value. The features were often binary, so red dots represent 1, while blue dots represent 0. Reading the plot we understand that *SEND SMS* has the highest impact on the decision making process. When it is present with value of 1 it pushes the prediction closer to malicious, while with the value of 0 it brings the prediction closer to benign. This is because there are many malware instances that would send text messages to share the infection between different devices, or to subscribe to premium service numbers that would cause significant financial losses to the user [87]. Similarly, assigning a value of 1 to the *READ PHONE STATE*, *TelephonyManager.getDeviceId* and *android.Telephony.SmsManager* features would drag the prediction towards identifying potential malware. The *READ PHONE STATE* permission grants access to phone state information, including network details, call status, and registered calling accounts, posing a significant privacy concern by potentially disclosing a user's approximate location through cellular data. The *TelephonyManager.getDeviceId* function retrieves the International Mobile Equipment Identity (IMEI), a unique SIM card identifier, which could be exploited for malicious purposes like phone number hijacking and fraudulent activities. On the other hand, *android.Telephony.SmsManager* is responsible for managing SMS operations, like sending various types of SMS messages. Although it connects to the *SEND SMS* feature, it differs in its ability to read and send SMS messages. While each of these three features alone may not definitively label an application as malware, when combined with others, like *SEND SMS*, they collectively raise suspicion. The second feature in terms of impact is *bindService*. In the Android context, setting it to 1 drags the prediction toward benign since it requires proper authorization for a client to bind to a service. Consequently, it's generally not a technique favored by malware creators. The *Ljava.lang.Class.getCanonicalName* and *Ljava.net.URLDecoder* features have a similar benign leaning effect when assigned a value of 1. This stems from their infrequent use by malware. Both functions relate to converting domain names into IP addresses. Most malware communicates with a command and control (C&C) center directly through IP addresses, bypassing the need for domain name resolution. In contrast, benign applications depend on domain names, ensuring seamless operation even when their server IP addresses change.

This project gives us a lot of information on which features contributes the most on the decision making process and how it contributes, if in a bad way pushing the prediction to malware or in a good way dragging the prediction to benign. Although, it does not give us hints on where the real problem inside the application is. For example to detect which is the malicious portion of the

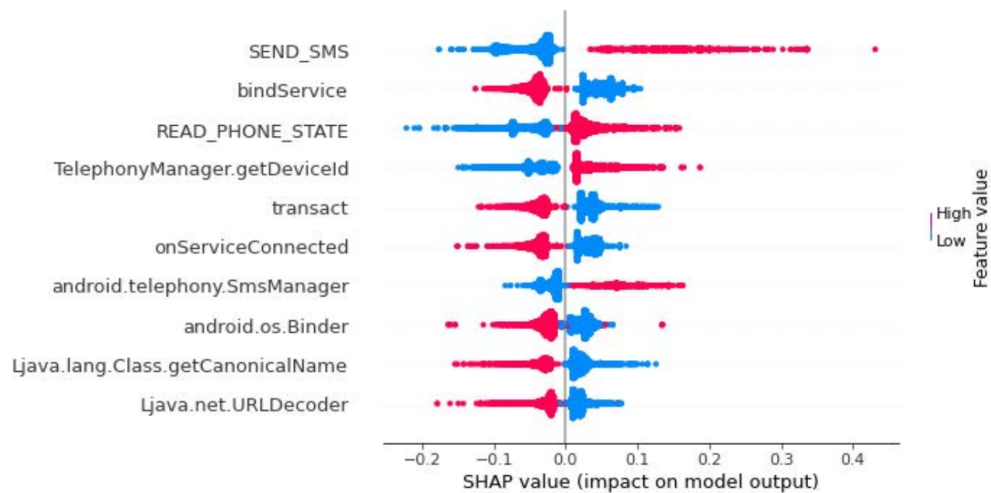


Figure 3.9. Summary plot of the SHAP values for the 10 highest impact features. (source: [87])

code that calls the *SEND SMS* feature we should search through all the classes which one raise that call and then evaluate each of them if they are actually a potential harm for the device or not. Furthermore this project does not perform malware classification, but only identification.

3.3.3 TAMI

Towards an Interpretable Deep Learning Model for Mobile Malware Detection and Family Identification (**TAMI**) method aims to detect malware in Android environment, in which the prediction can be immediately interpreted and evaluated from a security analyst or a generic user thanks to a visual representation of the inference phase (the activation maps) and also a numerical evaluation of the prediction reliability with regard of similar predictions. Moreover, the proposed method is also able to recognize the malware belonging family. They represent applications in terms of images that are subsequently considered as input for a deep learning model designed by the authors. The image representation is considered for the detection task but also with the aim to provide interpretability about the predictions of the designed deep learning model.

Here I itemize some distinctive points of this work:

- **Interpretability as a core point.** Currently, the effectiveness of different models are evaluated using metrics as, for instance, the accuracy: the idea behind this work is to propose the adoption also of interpretability for model comparison with the aim to understand the reason why the model generates a certain prediction.
- **Cumulative heatmap.** They introduce the *cumulative heatmap*, aimed to provide interpretability support to the security analyst in the understanding of classifier decisions. By exploiting the cumulative heatmap the analyst can have an immediate visual impact for understanding the reason why the model outputs a certain label but also why the model performs a misclassification (assessing the model reliability).
- **Grad-CAM.** In addition to the cumulative heatmap, the methodology automatizes the studies of the heatmaps, helping the security analyst in debugging the model without having any prior knowledge on the problem/pattern itself and extending the use of the Grad-CAM (more details at the end of TAMI). This is a key point, because the standard Grad-CAM does not provide useful information for analyzing images of malware (that do not contain any recognizable shape or object).

Gradient-weighted Class Activation Mapping

Gradient-weighted Class Activation Mapping (Grad-CAM) is an algorithm used for visualizing the importance of different regions in an input image that contribute to the predictions made by a convolutional neural network (CNN) model. It helps in understanding the decision-making process of the model and provides insights into which parts of the input image are most influential in determining the output. The key idea behind Grad-CAM is to leverage the gradients flowing into the final convolutional layer of the CNN to generate a class activation map. This map highlights the regions in the input image that are most relevant for predicting a specific class. Grad-CAM achieves this by performing the following steps:

1. **Preparing the CNN Model.** The CNN model to be visualized with Grad-CAM should typically consist of convolutional layers followed by a global pooling layer and a fully connected (dense) layer for classification. The model should be trained on the task of interest, such as image classification, so that it learns to recognize different classes.
2. **Forward Pass.** Select an input image on which you want to visualize the class activation map. Perform a forward pass through the CNN model and obtain the final convolutional layer's output feature maps. These feature maps capture the learned representation of the input image at a higher semantic level.
3. **Compute Gradient.** Calculate the gradients of the predicted class score with respect to the feature maps. These gradients represent the importance of each feature map in influencing the predicted class.
4. **Global Average Pooling.** Take the gradients calculated in the previous step and perform global average pooling (GAP) over each gradient map. GAP reduces the spatial dimensions of the gradient maps to a single value, representing the importance of the corresponding feature map.
5. **Weighted Sum.** Multiply each feature map by its corresponding gradient importance obtained from the previous step. Sum up all the weighted feature maps to obtain the final class activation map.
6. **Visualization.** Apply a ReLU activation function to the class activation map to discard negative values. Resize the class activation map to the size of the input image. Normalize the values in the class activation map between 0 and 1. Overlay the class activation map on the input image to visualize the regions that contribute most to the predicted class.

By visualizing the class activation map obtained through Grad-CAM (fig: 3.13), it becomes possible to understand the CNN model's focus and the regions in the input image that are crucial for making predictions. This can help in interpreting and verifying the model's decisions and identifying potential biases or areas of improvement. Grad-CAM has found applications in various fields, including computer vision, medical imaging, and interpretability of deep learning models.

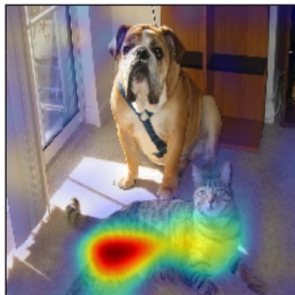
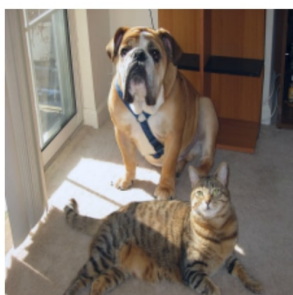


Figure 3.10. Original Image.

Figure 3.11. Grad-CAM *cat*.

Figure 3.12. Grad-CAM *dog*.

Figure 3.13. Example of Grad-CAM visualization (source: [15])

Project methodology

The **TAMI** methodology can be split into two parts as we can see in the Figure 3.14: the first one is related to the standard process to train a deep learning model:

1. **Data preparation.** The initial phase involves assembling, generating, and preparing the dataset. Malware samples are organized into families and transformed into images. All trusted software applications are grouped within the *Trusted* family. For Android applications, they extract the *dex* executable file and convert the byte sequences into 8-bit unsigned integers (ranging from 0 to 255), effectively creating grayscale pixels.
2. **Model Configuration.** Multiple rounds of k-fold cross-validation and hyperparameter optimization are performed to select the most suitable model configuration.
3. **Training and Testing.** The chosen model from the earlier steps is trained on the dataset, with a standard 80/20 data split. The subsequent results are collected and assessed.

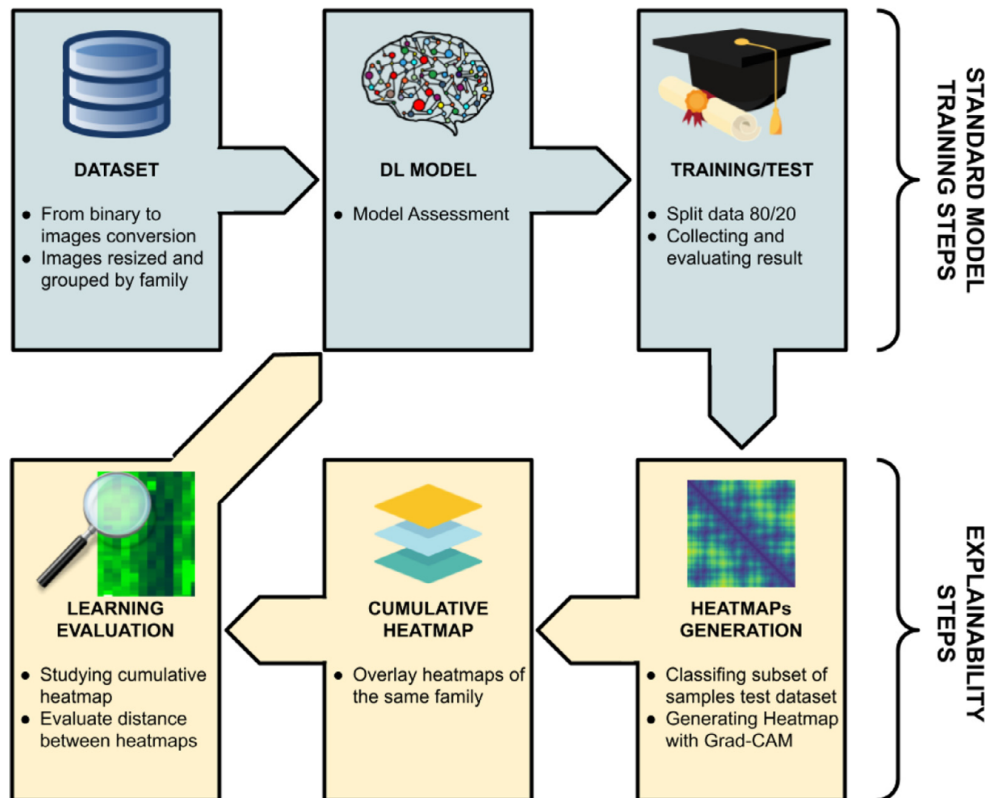


Figure 3.14. Steps of TAMI methodology. (source: [14])

Following this, they move on to the second part of the methodology, focusing on the steps related to model explainability:

1. **Heatmap Generation.** They select a subset of test dataset samples and pass them through the model for classification. In the inference phase, they apply Grad-CAM to identify which portions of the images significantly influence the classification. This process generates a heatmap for each sample in the selected subset.

2. **Cumulative Heatmap.** All the individual heatmaps, categorized by family, are merged together to create a *cumulative heatmap*. This *cumulative heatmap* is generated by calculating the average pixel values across all heatmaps. It effectively represents the distinctive features associated with each specific family.
3. **Learning Assessment.** They analyze and compare both the *cumulative heatmap* and the individual heatmaps to assess similarities. Since the *cumulative heatmap* retains valuable information about the model's output classifications, it provides insights into the learning process itself.

Interpretability aspects

The evaluation of the interpretability for this project include the study of the results by a security analyst that can add relevant knowledge to the analysis. For that reason the last step of the methodology can be divided into 2 parts:

- **Cumulative heatmap assessment.** Security analysts examine the *cumulative heatmap* to assess the patterns extracted by the learning process. Analysts possess knowledge of specific dataset or family characteristics that should or should not manifest during the inference analysis. This phase reinforces the model's robustness through expert manual verification.
- **Euclidean distance calculation.** Heatmaps generated are categorized by malware family, enabling comparisons within the same family. A feature vector summarizing image gradient information is extracted for each heatmap. Consequently, the Euclidean distance is computed between each feature vector and all other feature vectors within the same family, and the average score is determined.

An example of output from this project it is represented by the Figure 3.15. It is composed by three images:

1. The first one is the malware image belonging to the *FakeInstaller* family, where each pixel represent a portion of the malware memory.
2. The second one is the Grad-CAM heatmap of the previous image.
3. The last one shows the relevant areas on the original image by overlaying it with the heatmap. From this image the security analyst could do a reverse engineering action and look into that specific portion of the application memory highlighted by Grad-CAM.

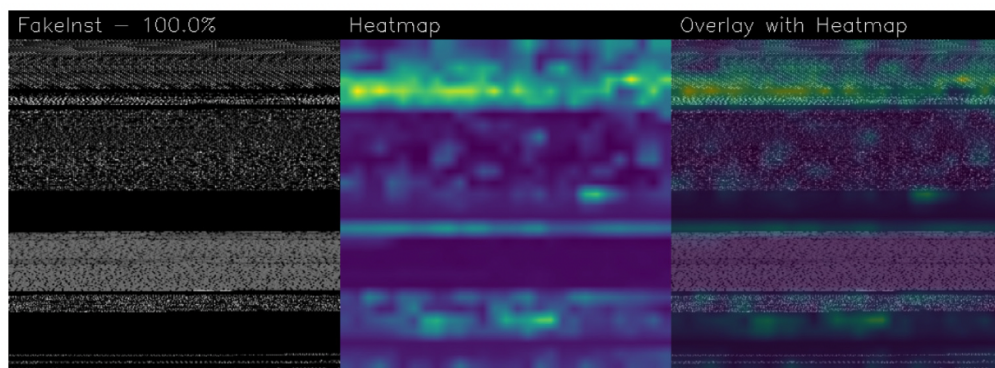


Figure 3.15. Comparison between the malware image and the heatmap. (source: [14])

The heatmap and the *cumulative heatmap* alone do not give an important advantage for our interpretability goal. They help to interpret and understand the outcomes, but they do not provide

fixed information that could be used to any user to evaluate models without any prior knowledge on the architecture or the problem itself. On the other hand, the Euclidean distance on the features vector extracted by the heatmaps can immediately help the security analyst in choosing the best model among a list of trained ones, but this is not better than other works on the topic. The true advantage would be if a reverse engineering phase would be applied at the end of these project's steps. The reverse engineering phase should automatize the process of converting back the image to memory, and so to code, so that the security analyst could immediately have access to the suspicious code. This is what I have carried out in my work of thesis, briefly following what is done in the subsequent work from the same authors of this one that is the **SCIBA** [16] project, where the reverse engineering phase is implemented.

3.4 Summary

In the field of malware classification, the quest for interpretable machine learning techniques is a dynamic landscape, where there is a wide range of state-of-the-art methodologies. This chapter has explored the diverse range of existing techniques, each of which offers unique insights and capabilities for identifying malicious software. However, between this tapestry of established methodologies, the decision to embark on a project that focuses on continuous innovation rather than settling for the already established models comes from a deliberate assessment of the evolving landscape in cybersecurity and machine learning. The appeal of established techniques that have been thoroughly researched and ingrained in the cybersecurity domain, like NODENS and PAIRED, lies in their reliability and maturity. However, the decision to explore an ongoing innovative project like TAMI stems from the recognition that the malware landscape is in a constant state of flux, so there is the need for a different approach that focus even more on interpretability. There is no tool in the wild that is able to return the source code of the malware. In this thesis, which focuses on the interpretability aspects of cybersecurity, I will delve into a project that attempts to fill this void by focusing efforts on continuing the work started with the TAMI project. Moreover, the trajectory of the chosen project fits seamlessly with the ethos of interpretable machine learning. It represents an important shift in which the pursuit of accuracy is aligned with the imperative need for explainability. In essence, the decision to pursue the path of a new approach, that exploits a never used before algorithm, while recognising the reliability of existing state-of-the-art techniques, underscores a commitment to improving the interpretability of malware classification. It represents not just a departure from convention, but a determined effort to bridge the gap between accuracy and comprehensibility in machine learning models applied to cybersecurity. This decision is the result of this era where understanding the *why* behind classification decisions is as important as accurately classifying the *what* in the realm of malicious software.

Chapter 4

Dataset

4.1 Overview of common datasets

In the course of conducting research for this thesis, my objective entailed the procurement of a comprehensive Android malware dataset that would encompass the actual *.apk* files of the target applications. It was apparent, however, that a significant portion of the datasets available online primarily provided descriptive reports or metadata about these applications, rather than granting access to the raw *.apk* files themselves. This dichotomy in the data offerings necessitated a more intricate and exhaustive search for a dataset that aligns with the specific requirements of my research methodology. Another pool of datasets proves to be unsuitable for our project due to the outdated nature of the applications they encompass. The data within this particular collection has not been maintained and updated, rendering it unsuitable for this research and analysis. As a result, I must select alternative data sources that better align with the timeliness and relevance needed for the execution of our research. The initial dataset I delved into during my analysis is the *CIC-AndMal2017* dataset [88], curated and made available by the Canadian Institute for Cybersecurity affiliated with the esteemed University of New Brunswick, nestled in Canada. The *CIC-AndMal2017* dataset stands as a cornerstone in the landscape of cybersecurity and machine learning, cherished by researchers and practitioners alike for its indispensable role in fostering a deeper understanding of the intricate realm of Android malware. Designed with a keen eye for facilitating advanced research endeavors, this dataset serves as an invaluable resource for those dedicated to unraveling the complexities of Android malware detection and analysis. *CIC-AndMal2017* boasts a rich tapestry of benign and malevolent Android applications, encapsulated in the form of APK (Android Package) files. These applications are complemented by a wealth of relevant features and labels, setting the stage for the development and evaluation of robust and sophisticated malware detection models. As such, *CIC-AndMal2017* emerges as a pivotal asset, fostering innovation and progress within the dynamic and ever-evolving field of cybersecurity and machine learning. The dataset composition encompasses a grand total of 10,854 Android applications, categorized into two distinct groups: benign and malware. The benign category is thoughtfully comprised of 6,500 authentic applications that have been sourced directly from the official Google Play Store, ensuring the inclusion of a diverse and comprehensive collection of commonly utilized apps that paint an accurate picture of everyday mobile device usage. On the other hand, the malware category is constructed of 4,354 malicious applications that have been collected from a variety of sources. These sources include third-party app stores, research repositories, and known malware samples. These applications were meticulously selected to represent a broad spectrum of malware types and threats commonly found in the Android ecosystem, making the dataset a valid resource for research and security analysis. The malicious software applications included in the dataset have been classified into four distinct categories (fig: 4.1). These categories have been delineated based on a comprehensive examination of their common attributes and operational patterns. The four malware categories encapsulate various activities, each with its unique *modus operandi*, posing varying degrees of threats to computer systems, networks and the privacy of individuals. These categories encompass, but are not limited to, activities such as data exfiltration, breaching privacy, perpetrating financial fraud, and unauthorized intrusion

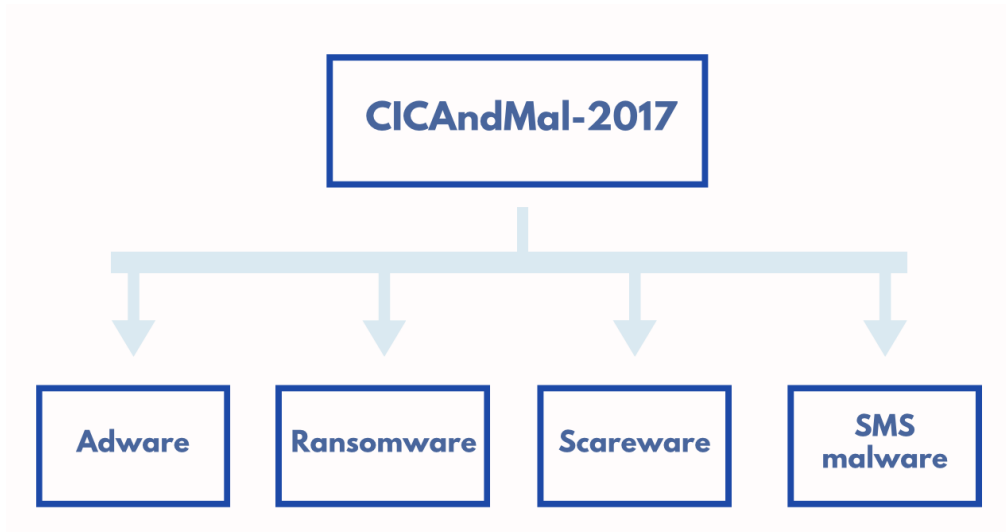


Figure 4.1. CICAndMal-2017 malware classification categories

into digital domains. The four primary categories, which serve as a taxonomy for these malicious software entities, are as follows:

- **Adware:** this category primarily involves the dissemination of unwanted advertisements and pop-ups, often leading to a reduction in system performance and potential privacy concerns for the affected users.
- **Ransomware:** this type of malware encrypts a victim’s data and demands a ransom for its decryption. It is a grave threat to both individuals and organizations, causing data loss and financial repercussions.
- **Scareware:** scareware relies on manipulating users through deceptive tactics, such as false security alerts or fake virus reports, to coax them into purchasing rogue software or services. It preys on the fear and uncertainty of computer users.
- **SMSmalware:** SMS malware operates on mobile devices and can result in unauthorized premium SMS charges or other malicious activities via text messaging. It poses a unique set of challenges to mobile security.

The exhaustive categorization and thorough analysis of these malware types are indispensable for understanding the diverse threats that exist within the digital landscape and developing effective countermeasures to safeguard against them. Following this point each malware category is then furthermore subdivided into distinct families, resulting at the end in a comprehensive collection of 42 malware families. This dataset presents a significant constraint when it comes to its cardinality, as only a relatively small fraction of the dataset is accessible for download. Specifically, for each malware family within the dataset, I have acquired an average of merely 12 applications. This inherent limitation poses a substantial challenge when attempting to employ this dataset for the essential tasks of training, validating and testing a machine learning model. Nevertheless, I have chosen to employ this dataset during the initial phase of my project, wherein I focused on the fine-tuning of my research to align it with my specific computing environment. This decision was instrumental in enabling me to gather an extensive and comprehensive set of insights into the inner workings of my project. The major dataset under consideration is the **MalRadar Dataset** [17], last updated in April 2022 and made available through Zenodo. Zenodo is an online-based platform and digital repository with a core mission of promoting the sharing, preservation, and citation of research outputs across a diverse array of academic disciplines. Serving as a steadfast and openly accessible hub for researchers, scientists and scholars, Zenodo provides them with a secure space to store and disseminate their research data, software, presentations, posters and reports. This platform is an integral part of the larger *OpenAIRE* (Open Access Infrastructure for Research in

Europe) initiative and its operations are conducted under the auspices of CERN (the European Organization for Nuclear Research), symbolizing its dedication to quality and credibility. At Zenodo, researchers have the ability to upload their research outputs and, most notably, assign them persistent identifiers in the form of DOIs (Digital Object Identifiers). These DOIs ensure the discoverability and reusability of their work, establishing a robust foundation for open science practices. Beyond mere data sharing, Zenodo provides an extensive pool of options and features. It accommodates a wide range of file formats, allowing researchers to publish not just their final research papers but also the underlying data, code and supplementary materials. This fosters a culture of transparency and reproducibility, which is vital in advancing the cause of scientific research. Zenodo’s interface is user-friendly, facilitating ease of use for researchers. Moreover, its seamless integration with other scholarly systems, such as GitHub, enhances the efficiency and effectiveness of managing and sharing research outputs. Researchers can efficiently connect their work to a wider academic network, making collaboration and dissemination more streamlined. Crucially, Zenodo also champions the long-term preservation of deposited content. By ensuring that the data and research outputs remain accessible and available over an extended period, Zenodo contributes to the growth of knowledge for future researchers and the broader scientific community. This commitment to archiving and availability plays a pivotal role in maintaining the integrity and continuity of scientific knowledge. MalRadar is a comprehensive repository that houses a collection of 4,534 distinct malware samples, gathered from a wide array of diverse distribution channels. Among these malicious software, a significant proportion amounting to 43.5% [17] was procured from sources extending beyond the realms of the Google Play Store and various third-party app markets. This repository reflects the various paths through which they have infiltrated digital ecosystems, making it a valuable resource for cybersecurity researchers and analysts. The dataset encompasses a wide range of malevolent behaviors, each demonstrating a different facet of the threat landscape:

- **Privacy stealing:** privacy theft stands as one of the most prevalent nefarious activities, where malicious software clandestinely acquires sensitive personal information from unsuspecting users. In the expansive MalRadar dataset, it’s striking to note that nearly 90% of the malware samples, hailing from 105 distinct malware families, are implicated in these covert operations. Such invasions encompass the extraction of highly sensitive data, including phone numbers, contact lists, SMS conversations, and geolocation data, all gleaned from compromised devices.
- **SMS/CALL abuse:** malware frequently leverages permissions related to text messaging and phone calls to execute clandestine, harmful actions without the user’s knowledge. These actions range from sending, blocking, or erasing SMS messages to making unauthorized phone calls. Bad actors often exploit these functionalities to engage in a wide range of malevolent activities, including purloining login credentials, orchestrating fraudulent financial transactions, or disseminating harmful disinformation. For instance, the notorious Agent malware, featured prominently in the MalRadar dataset, ingeniously sidesteps two-factor authentication by intercepting incoming text messages and relaying them to a remote server. This tactical maneuver affords attackers access to SMS-based authentication codes while leaving no trace on the compromised device. In the vast repository of MalRadar data, a staggering 2,287 samples originating from 68 distinct malware families display malicious tendencies linked to SMS interception, deletion, or unauthorized calls.
- **Remote control:** The capacity to communicate with remote Command and Control (C&C) servers is a defining feature of approximately 85% of the malicious applications housed within the MalRadar dataset. These diverse strains of malware establish intricate connections with C&C servers to receive and execute commands, thus enabling them to carry out a spectrum of malicious activities based on the instructions relayed from afar. As an illustrative example, the XLoader malware exploits the WebSocket protocol to establish a communication link with its C&C servers. Upon the receipt of remote directives, this malware executes corresponding actions, including sending SMS messages to designated recipients or producing full-screen phishing interfaces to deceive unsuspecting users. This level of remote control is evident in 3,840 samples originating from 107 distinct malware families within the MalRadar dataset.

Family Name	# reports	# apps	Distribution Channel			Installation				
			GPlay	3rd-party	Others	Repackaging	Update	Drive-by	Library	Standalone
RuMMS	2	796			Website;SMS;Email			√		
Xavier	1	593	√						√	
LIBSKIN	1	290		√					√	
HiddenAd	7	289	√	√						√
GhostClicker	1	248	√						√	
MilkyDoor	1	210	√			√				
EventBot	1	124		√	Website					√
GhostCtrl	1	109			Email					√
Lucy	1	82			Social Media;IM Apps					√
FakeBank	1	80			SMS	√				
FakeSpy	2	74			SMS			√		√
Joker	4	73	√	√					√	√
SpyNote	1	63		√						√
solid	1	61	√							√
ZNIU	1	59			Website					√
KBuster	1	54								√
hiddad	4	54	√							√
Hqwar	4	52								√
Monokle	1	51				√				
hawkshaw	1	48			Forum					√
TOASTAMIGO	1	46	√							√
AdDisplay	1	38	√	√						√
LokiBot	1	33			Website			√		
TERRACOTTA	1	33	√						√	
Click	1	33	√		Website					√
Necro	1	31	√	√	Forum;Blog				√	
Xloader	3	29			Website;SMS			√		
GnatSpy	2	28	√							√
Bahamut	1	27	√		Website	√		√		
meftadon	1	26	√							√
Shopper	2	25	√							√
SpyMax	1	25			Forum					√
Slocker	2	23			Forum					√
Dmisk	1	22			Website			√		
smsthief	1	21			SMS			√		
Maikspy	1	20			Website;Social Media			√		√
Svpeng	2	20			Website					√
donot	2	19		√	SMS					√
campys	1	19			Website;Email;SMS	√				
Banker	6	18	√	√						√
mazarbot	1	18			SMS			√		
ROOTSTV	1	16			Website			√		
Fanta	1	15		√	Website			√		
PhantomLance	1	15	√	√						√
Mapin	1	15	√	√				√		√
raddex	1	15			Website;Email;SMS;Social Media	√				
Exobot	1	14			Website;SMS/MMS					√
SpyAgent	4	14		√						√
Agent	4	13	√	√	Website;Social Media					√
Inazigram	1	12	√							√
Others(98)	127	410	40	9	37	8	5	8	5	75
Total (families)		4,503	61	23	64	14	7	18	11	106
Total (apks)		4,503	2,108	1,073	1,957	443	49	1,100	1,284	1,855

Figure 4.2. Distribution channels and installation methods of the top-50 families in MalRadar. (Source: [17])

- **Bank/Financial data theft:** trojan-banker applications are highly specialized in their operation, with a primary focus on acquiring sensitive information related to users' on-line banking accounts, electronic payment systems and credit or debit card credentials. Within this dataset, there are an alarming 1,522 distinct applications and a network of 45 distinct malware families that are actively engaged in these nefarious activities centered around the theft of financial data. To illustrate, let's take the Exobot malware as a case in point. This malicious software exhibits a sophisticated behavior pattern wherein it deploys a meticulously crafted phishing interface whenever it detects the launch of a specific target application, such as a banking app, on an infected device. This deceptive user interface is

designed to mirror the appearance of legitimate banking login pages, with the explicit intention of deceiving unsuspecting victims into unwittingly divulging their sensitive banking credentials. It is important to note that Exobot’s ambitions are not limited to a handful of banks, rather, it specifically targets more than 30 financial institutions of various scales and reach.

- **Ransom:** ransomware represents a pernicious category of malware that employs a coercive tactic, threatening to either disclose the victim’s confidential data to the public or, worse, to permanently restrict access to it until a ransom is paid. Invariably, ransomware assaults employ a form of concealed Trojan, often masquerading as a seemingly legitimate application to fool users. Once this deceptive software is initiated, it proceeds to systematically encrypt the victim’s files or place a restrictive lock screen, accompanied by a demand for a substantial ransom in exchange for the release of the hostage data. Take, for instance, the case of SLocker, a ransomware variant that goes to great lengths to camouflage itself as a highly appealing and well-known application, thereby baiting unsuspecting users into its trap. Upon activation, SLocker executes a meticulous process of file encryption, withholding the decryption key until the user complies with the ransom demand, typically denominated in the form of Bitcoin. Within this dataset, there are a total of 346 instances cataloged coming from no less than 17 diverse ransomware families, exemplifying the wide-ranging nature of this digital extortion threat.
- **Accessibility abuse:** this is a prevalent concern within the Android ecosystem, primarily stemming from unscrupulous app developers who exploit the enhanced permissions granted to accessibility services for vicious purposes. These services, initially designed to aid users with disabilities in navigating their devices and applications, have become a tempting gateway for malicious actors. These ill-intentioned developers often employ deceptive tactics to gain access to these permissions, enabling them to carry out various illicit activities, ranging from data theft to device hijacking and the installation of malicious content. One prime example of this deceptive behavior is demonstrated by TOASTAMIGO, a seemingly legitimate app locker that boasts PIN code protection. Once installed, these seemingly harmless applications ingeniously request accessibility permissions under the guise of enhancing functionality. However, behind the scenes, they execute a series of malicious commands. These actions may include the exfiltration of sensitive personal information and the installation of unauthorized applications. Research conducted by MalRadars developers has identified 1,356 samples across 30 different malware families actively engaging in accessibility abuse.
- **Privilege escalation:** this is a technique frequently exploited by advanced malware. Privilege escalation can be broadly categorized into two primary types. The first type involves acquiring ROOT access on the targeted system, a feat achieved through the download of root modules or the exploitation of vulnerabilities inherent in the Android platform. ROOT-capable malware poses substantial risks, as the act of rooting an Android device can have far-reaching consequences, potentially compromising the integrity and security of the entire system. The second type of privilege escalation revolves around the relentless solicitation of device administration privileges by malware. In this scenario, the malicious software consistently prompts users to grant such privileges, effectively making it difficult for the user to remove the malware and paving the way for more invasive and damaging malicious activities. In this domain the developers found a substantial 2,028 samples, constituting 45% of the identified malware, belonging to 50 different families, have been observed to exhibit privilege escalation behavior.
- **Stealthy downloading:** malware uses stealthy approach when attempting to install applications on a victim’s device without their knowledge. This often involves exploiting root privileges or manipulating accessibility settings. By silently initiating the download and installation of additional applications, cyber attackers can further propagate malicious software on the already compromised smartphone, or even exploit the pay-per-install advertising networks to generate significant revenue. An illustrative case of this technique can be seen in the actions of the LIBSKIN malware, which, after gaining ROOT privileges, proceeds to download and install applications on the unsuspecting user’s device. Meanwhile, the TOASTAMIGO malware uses the accessibility service to silently download and install other

malware. This dataset includes 1,391 malicious applications, representing a wide range of malicious download behaviours across 35 different families.

- **Aggressive advertising:** malware exhibits a number of aggressive advertising practices that can significantly disrupt the normal functioning of a device. To begin with, malware relies on generating an overwhelming number of fake clicks in the background, with the primary intention of generating illegal profits. It also often floods the device interface with persistent advertisements that the user is unable to close. Some of these ads are not only annoying, but malicious, redirecting unsuspecting users to harmful websites with nefarious intent. One example is the GhostClicker adware, which cleverly injects malicious code into Google’s AdMob library, a widely used mobile ad network, and uses the *dispatchTouchEvent* API to simulate user clicks to generate false network traffic, ultimately increasing the attacker’s revenue. Another example is HiddenAd, which takes a particularly intrusive approach by displaying full-screen ads at regular intervals, severely disrupting the normal user experience of the device by making it virtually impossible for users to close these intrusive ads. The MalRadar dataset has found that more than 42% of malicious applications, comprising an extensive dataset of 1,898 samples and 31 different families, use such disruptive and intrusive advertising practices.
- **Mining:** mining malware is a type of malicious software that uses a smartphone’s processing power to generate revenue by mining cryptocurrencies. This process uses the device’s CPU power to solve complex mathematical problems, and the rewards in the form of cryptocurrency are then sent to the attacker’s wallet. HiddenMiner and JSMiner are examples of this type of malware. HiddenMiner drains a device’s resources by mining Monero, causing increased wear and tear, reduced battery life and slower performance. JSMiner, on the other hand, loads JavaScript library code from Coinhive and uses the attacker’s site key to mine cryptocurrency, resulting in exceptionally high CPU usage.
- **Tricky behaviour:** many malicious applications employ a variety of clever strategies to evade the uninstall process and disguise their true intentions. These stratagems cover a wide range of deceptions, including hiding or altering the application’s visual appearance, manipulating the application’s nomenclature, preventing the user from accessing the application’s detailed information and orchestrating the launch of an imperceptible, translucent background operation that hides the malicious payload from the prying eyes of unsuspecting users. In their bad pursuits, these malicious applications even go so far as to reject notifications and manipulate device configurations, such as muting the ringer or dimming the screen brightness, all in a concerted effort to ensure that their duplicitous activities remain cloaked in darkness, thereby avoiding detection by their hapless victims. Consider, for example, the case of the app carrying the Maikspy malware, which perpetrates an audacious ruse by displaying a false message claiming *Error: 401. Application not compatible. Uninstall...* This clever deception is designed to trick users into believing that the application has been successfully removed from their device. In reality, the malicious spyware continues to operate stealthily, lurking in the background, unknown to its host. Furthermore, MalRadar’s analysis reveals that a significant proportion of these duplicitous applications, 1,872 samples representing 41.5% from a compendium of 52 different families, have effectively used these deceptive techniques to perpetrate their malicious agendas.
- **Premium service:** this novel form of malicious behaviour involves certain applications secretly enrolling unknowing users in premium services without their explicit knowledge or consent. This nefarious behaviour has received limited attention in previous research efforts. These applications autonomously trigger premium service subscriptions on behalf of the user, with financial implications. Typically, they use WAP click, a convenient technological mechanism that facilitates subscription to a range of premium services without the user’s knowledge or authorisation. As an example, consider *Joker*, an Android-based malware that tricks users by using WAP deception techniques. These applications masquerade as legitimate counterparts, such as virtual keyboards, camera utilities or gaming platforms. However, their benign appearance conceals malicious code that secretly opens browser windows and tricks users into subscribing to premium rate phone numbers, providing operators with the opportunity to earn substantial commissions. Within the scope

of MalRadar, there are a total of 162 unique instances catalogued, belonging to 7 different families, each of which is guilty of performing actions of a malicious nature related to premium rate service subscriptions.

4.2 Data characteristics and statistics

After successfully obtaining the MalRadar dataset, I found myself in possession of an extensive collection of *.apk* samples, along with a *.csv* report. This report detailed the various attributes and characteristics of each malware sample. My first task was to isolate the *family* characteristic by extracting it from each line of the report. I then create a *.json* dictionary indexed with the extracted *family*. This process involved organising and categorising the malware samples according to their respective *families* and encapsulating this information within the newly created dictionary. The purpose of this partitioning operation was to create distinct subdirectories, each associated with a specific malware *family*. In this way I introduced an element of organisation and order into what was previously a chaotic collection of digital threats, allowing for more focused and in-depth analysis of each malware. I then undertook the task of culling and sorting the dataset, removing all the *family* containing fewer than 200 samples. The idea behind this process was to ensure the utmost precision and efficiency in the subsequent training of the convolutional neural network. As a result, the *families* that have successfully passed the process are listed below:

	#Apps
RuMMS	756
Xavier	293
LIBSKIN	290
HiddenAd	289
GhostClicker	248
MilkyDoor	210

Table 4.1. MalRadar *families* filtered for 200+ cardinality

- RuMMS:** short for Russian Mobile Malware Syndrome, is a family of Android malware that emerged in 2014. It is known for its ability to steal financial information and login credentials from victims' devices. The RuMMS malware family is typically spread through third-party app stores, phishing campaigns and social engineering tactics. Once installed on a victim's device, the malware can perform a variety of malicious actions, including intercepting and sending SMS messages, stealing contact lists, making unauthorised calls, and downloading and installing additional malicious applications. One of the key features of the RuMMS malware is its ability to intercept SMS messages and forward them to the attackers. This can be used to bypass two-factor authentication measures, allowing attackers to access victims' financial accounts and steal sensitive information. To avoid detection, RuMMS malware often uses advanced obfuscation techniques and can hide its presence on a victim's device by masquerading as a legitimate system process. It can also use encryption to protect stolen data and communications with its command and control servers. RuMMS is a significant threat to Android users, particularly in Russia and neighbouring countries.
- Xavier:** this is a family of mobile malware that targets Android devices. It was first discovered in 2017 and has been active in the wild ever since. Xavier is designed to steal sensitive information from infected devices, such as personal and financial data. The malware is able to evade detection by using advanced obfuscation techniques. It can encrypt its payloads and use dynamic code loading to avoid detection by antivirus programs. It also has the ability to connect to a remote server and download additional malicious code, allowing it to perform a wide range of attacks. Xavier is primarily distributed through third-party app

stores and malicious websites, often disguised as legitimate apps or software. Once installed on a device, it can perform a range of malicious actions, including displaying unwanted advertisements, stealing sensitive information and even installing additional malware. Due to its ability to evade detection and steal sensitive information, Xavier poses a significant threat to Android users. It highlights the importance of using reputable app stores and taking other precautions, such as keeping device software up-to-date and using antivirus software, to protect against mobile malware.

- **LIBSKIN:** once downloaded and installed, the app loads the *libskin.so* shared library, which contains an exploit that roots the device and gives the malicious app absolute power over your device. The LIBSKIN malware family is a type of Android malware that typically masquerades as a legitimate app, often related to entertainment or gaming. The malware is designed to display advertisements, generate revenue for the attackers and steal sensitive user information such as phone numbers, call logs and contacts. Once installed on a victim's device, LIBSKIN malware typically uses obfuscation techniques to evade detection and prevent analysis. The malware often communicates with a remote Command and Control server to receive instructions and transmit stolen data. In addition to displaying unwanted advertisements and stealing user data, LIBSKIN malware can also perform other malicious activities such as downloading additional malware, sending SMS messages to premium rate numbers and taking control of the victim's device. The LIBSKIN malware family is known to evolve over time, with new variants and techniques regularly appearing to evade detection by security researchers. The malware is often distributed through unofficial app stores or by tricking users into downloading it through phishing or social engineering tactics.
- **GhostClicker:** GhostClicker is a family of Android malware that works as a clicker Trojan. It was first discovered in 2017, and has since been actively used in malicious campaigns. GhostClicker is designed to generate revenue for its creators by automatically clicking on ads without the user's knowledge or consent, resulting in increased ad revenue for the attackers. GhostClicker is distributed through a variety of channels, including malicious websites, compromised applications and third-party app stores. Once installed, it runs silently in the background, waiting for the device screen to turn off or the device to go into sleep mode. Once this happens, the malware begins to generate fake clicks on advertisements, inflating the number of clicks and views of the ads. GhostClicker has the ability to evade detection by anti-virus software and security mechanisms. It uses various techniques to disguise its activity and evade detection, including obfuscating its code, using dynamic class loading, and implementing anti-emulation and anti-debugging techniques. GhostClicker can cause a number of problems for users, including increased data usage, reduced battery life and degraded device performance. It can also lead to financial losses for advertisers and app developers who rely on legitimate clicks and views to generate revenue.
- **MilkyDoor:** this is an Android malware family first discovered in 2016. It is known for its ability to bypass Google Play Store security measures, infecting devices through malicious apps distributed outside of the official app store. Once installed on a device, MilkyDoor gains root access and installs a backdoor that allows it to communicate with a remote server and receive commands. This allows attackers to perform a variety of malicious activities, such as stealing sensitive data, displaying unwanted advertisements, and launching additional attacks on other devices or networks. One of the most notable features of MilkyDoor is its ability to use popular applications and games as a disguise to avoid detection. It injects malicious code into legitimate apps and repackages them so that when the user installs the app, the malware is also silently installed in the background. MilkyDoor has been distributed through various channels, including third-party app stores and online forums, as well as through phishing emails and SMS messages. It is an ongoing threat to Android devices and users are advised to only download apps from trusted sources and keep their devices up-to-date with the latest security patches.

My project goes beyond mere malware classification, as I wanted to obtain a comprehensive framework designed to effectively identify the nature of any given application, distinguishing between those that may have malicious intent and those that are benign. In order to achieve this goal, my research involved incorporating a wide range of authentic applications into the dataset,

thereby expanding the scope of the project’s applicability. So, I have selected a total of 4,143 **Benign** applications from the *CIC-AndMal2017* dataset. As a result, the final dataset cardinality is summarized in table 4.2.

	#Apps
Benign	4143
RuMMS	756
Xavier	293
LIBSKIN	290
HiddenAd	289
GhostClicker	248
MilkyDoor	210

Table 4.2. Final dataset cardinality

Chapter 5

Experimental Design and Methodology

The main purpose of this thesis is to try to find one good machine learning algorithm that permits to interpret its results in the clearest possible way. To do this I decided to take one of the most innovative and promising projects developed in literature, adapt it to my data and improve its performance. There was therefore a specific workflow aimed at understanding the result of the machine learning algorithms. The first step was to take all the Android applications, divided in folders organized by malware *family*, and transform them into images. These images are then used to train a neural network model. Before doing it the all dataset applications had to be decompiled to extract *smali* code, that is more resistant to obfuscation than Java source code. This format allows extracting the opcodes for each class and method of the malware. Each opcode was mapped to an ASCII character, and then the characters were stored in a text file, preserving the order in which they appear in the code and keeping track of where classes begin and end, storing this information in a legend text file, which will make it possible to reverse the process from an image area to the corresponding code. We will always be able to identify a specific pixel on the image with its corresponding opcode and line of *smali* code, using the legend text file. This process is the core of the interpretability phase that we will see later. Once I had the images, I fed them as input into the chosen neural network model, InceptionV3 [89], developed by Google researchers, which, after a period of fine tuning, gave me excellent performance results in terms of the chosen metrics. With this trained model I tested it with a various arranged test dataset looking for the best result. In this test dataset is then applied the Grad-CAM algorithm to highlight the more relevant pixels of the image. Comparing the legend file with the Grad-CAM heatmaps we can trace back to the exact opcodes and classes of the source code that contribute the most for the predictions. In this chapter I will describe the algorithms used in each phase of this research and the reasons that are behind each design choice.

My experiments were conducted exploiting the HPC@POLITO cluster in the Legion work environment. HPC@POLITO is a supercomputing initiative under the supervision of DAUIN (Department of Automation and Informatics of the Politecnico di Torino). Its primary mission is to provide high performance computing resources and expert technical support in support of academic research and educational endeavours. In particular, its services extend beyond the boundaries of the university, serving research groups both inside and outside the Politecnico di Torino. Among the three clusters offered by the HPC group, my choice was LEGION. Designed with modularity at its core, LEGION takes the form of an InfiniBand cluster and its key features are detailed in the table 5.1.

Architecture	Linux Infiniband-EDR MIMD Distributed Shared-Memory Cluster
Node Interconnect	Infiniband EDR 100 Gb/s
Service Network	Gigabit Ethernet 1 Gb/s
CPU Model	2x Intel Xeon Scalable Processors Gold 6130 2.10 GHz 16 cores
GPU Model	4x nVidia Tesla V100 SXM2 - 32 GB - 5120 cuda cores (on 6 nodes)
Performance	90 TFLOPS (last update: july 2020)
Computing Cores	1824
Number of Nodes	57
Total RAM Memory	22 TB DDR4 REGISTERED ECC
OS	Centos 7.6 - OpenHPC 1.3.8.1
Scheduler	SLURM 18.08.8

Table 5.1. LEGION - Technical specifications

5.1 From *.apk* to *.png*

In this section I will describe all the actions that the Android applications were subject to while being transformed into images. The first step is the use of a decompiler on each sample of the dataset. For this purpose *APKTool* decompiler was used with the command

```
apktool d {directory}/{label}/{filename}
```

where

- *d* is the command's option to decompile the application
- *directory/label/filename* is the path of the application

APKTool is a powerful open-source utility for decompiling and recompiling Android application packages. It allowed me to reverse engineer APK files, converting them from binary format into a human-readable and editable format. It decompiles resources (e.g., XML files, images and layouts) and the app's bytecode (Java classes) into more accessible formats. The Java bytecode is then converted into *smali* code, as this format allows extracting the opcodes for each class and method of the application. *Smali* is an assembly-like language specific to Android's Dalvik Virtual Machine. It is readable but low-level, and *APKTool* provides tools to work with it. The opcodes were then matched to ASCII characters following a dictionary created by a *.txt* file that establishes the key-value conversion. This file is the *Dalvik.to.OPC.txt* where all the *smali* opcodes find a translation to an ASCII character. It was also saved the number of characters that each class contains. This was done to know exactly where each class starts and ends, so that from the final image we can trace back every pixel to a specific class. The operative code that each Android application is subject to is reported in this listing:

```
general_content = ""
smali_k = {}
end = 0
for smali in smali_paths:
    class_name = smali.replace(f'{{DECOMPILED}}/{{family}}', "")
    file = open(smali, "r")
    encoded_content = converter.encoder(file.read())
    file.close()
```



```
# saving number of characters and encoded content
num_character = len(encoded_content)
smali_k[class_name] = num_character
general_content += encoded_content
```

Then, having all these information stored, the image could be generated and stored, almost ready to be used in the subsequent steps. The very last step that the images are subject to is the image resizing, to make them compatible for being used as input in a neural network training phase. We must be very careful with this step because, unlike normal images where the individual pixel is not so important within the entire image, in our case each pixel corresponds to valuable information, which cannot be lost. To do so, the best resampling algorithm that the *Pillow*, an image processing Python library, offers is the *ANTIALIAS*. This resampling method is designed to produce high-quality results when resizing images, particularly when reducing their dimensions. It is an anti-aliasing filter that helps reduce aliasing artifacts. These are visual distortions in images, often appearing as jagged edges, staircase-like patterns, or unwanted patterns in textures, such as moire patterns, that can occur when an image is scaled down. This algorithm was chosen because the size of the applications was such that they produced very large images, so we needed an algorithm that was particularly efficient for downscaling, deciding to discard those small applications from which too little images were generated. Therefore, in this step, a partial reduction of the dataset was performed by discarding some images that were too small. In particular, 188 images of the *Ghostlicker* malware family were discarded, making this class unusable for the training, validation and testing phases. I then decided to completely exclude this malware family from the next steps of my work, otherwise the bad results it would produce would affect all the other classes. The window function that the *ANTIALIAS* resampling method use is the *Lanczos* window function written as:

$$S(x) = \sum_{i=x-a+1}^{x+a} s_i L(x-i) \quad (5.1)$$

With this function, if we consider a positive integer parameter a , that in our case equal to 3 (fig: 5.1), the *Lanczos* kernel maintains continuity throughout its range and its derivative remains well defined and continuous throughout. Consequently, the reconstructed signal $S(x)$ also exhibits continuity and maintains a continuous rate of change. The *Lanczos* kernel returns a value of zero for all integer inputs of x , except when x equals zero, in which case it returns a value of 1. As a result, the reconstructed signal accurately interpolates the specified data points.

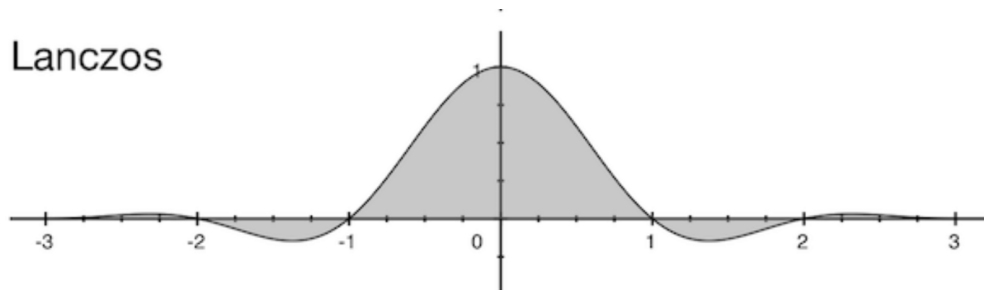


Figure 5.1. Lanczos window function with $a = 3$

Once I had the complete final dataset I had to decide how to split it. At the beginning I tried the most common methodology, that is to split the dataset 80% for training and 20% for testing purposes and then I tried to split it in 90/10 configuration. Moreover, every time the 20% of the training samples were used for validation.

5.2 CNN model evaluation

The dataset of images was used to train a convolutional neural network (CNN). Initially, I had tried to train the same CNN used by the paper’s authors [16] called *BasicCNN*, a sequential model designed by them which is reported in the table 5.2. After a long period of fine-tuning, in which I tried to modify and adapt the dataset or the network, and also worked on the hyperparameters, the results were not good enough to satisfy our intention of obtaining an interpretable tool, but also with great performance. The best results that the tool obtained with this configuration of the CNN is shown in Figure 5.2 which shows the confusion matrix and its normalized version of the results obtained with the test dataset after training the network on the train dataset.

Type	Output Shape	Parameters
InputLayer	(300,300,1)	0
Conv2D	(298,298,32)	320
MaxPooling2D	(149,149,32)	0
Conv2D	(147,147,64)	18496
MaxPooling2D	(73,73,64)	0
Conv2D	(71,71,128)	73856
MaxPooling2D	(35,35,128)	0
Flatten	(156800)	0
Dropout	(156800)	0
Dense	(512)	80282112
Dropout	(512)	0
Dense	(256)	131328
Dropout	(256)	0
Dense	(6)	771

Table 5.2. SCIBA [16] project CNN summary

I take as example an application labeled as a *RuMMS* malware. It produces the following text file where the exact point of beginning and end of each classes in the image is stored as coordinates of the image matrix of pixels.

```

/RuMMS_app_ex/smali/com/a/a/a/a.smali [1,1] [17,1]
/RuMMS_app_ex/smali/com/basket/grateful/FrontAnswer.smali [18,1] [76,2]
/RuMMS_app_ex/smali/com/basket/grateful/BlockAtmosphere.smali [77,2] [36,4]
/RuMMS_app_ex/smali/com/basket/grateful/Dull.smali [37,4] [7,11]
/RuMMS_app_ex/smali/com/basket/grateful/ConceiveSlice.smali [8,11] [80,11]
/RuMMS_app_ex/smali/com/basket/grateful/Alive.smali [81,11] [25,12]
/RuMMS_app_ex/smali/com/basket/grateful/PerformerDiameter.smali [26,12] [78,24]
/RuMMS_app_ex/smali/com/basket/grateful/PersonWilderness$1.smali [79,24] [12,30]
/RuMMS_app_ex/smali/com/basket/grateful/PersonWilderness$2.smali [13,30] [8,31]
/RuMMS_app_ex/smali/com/basket/grateful/ScienceStaff.smali [9,31] [52,42]
/RuMMS_app_ex/smali/com/basket/grateful/PersonWilderness.smali [53,42] [17,47]
/RuMMS_app_ex/smali/com/basket/grateful/SmoothHalf.smali [18,47] [83,48]
/RuMMS_app_ex/smali/com/basket/grateful/Spring$1.smali [84,48] [3,50]
/RuMMS_app_ex/smali/com/basket/grateful/Spring.smali [4,50] [83,71]
/RuMMS_app_ex/smali/com/basket/grateful/StopWash.smali [84,71] [45,94]
/RuMMS_app_ex/smali/com/basket/grateful/b.smali [46,94] [61,98]
/RuMMS_app_ex/smali/com/basket/grateful/a.smali [62,98] [62,99]
/RuMMS_app_ex/smali/com/basket/grateful/d.smali [63,99] [40,101]
/RuMMS_app_ex/smali/com/basket/grateful/c.smali [41,101] [71,107]

```

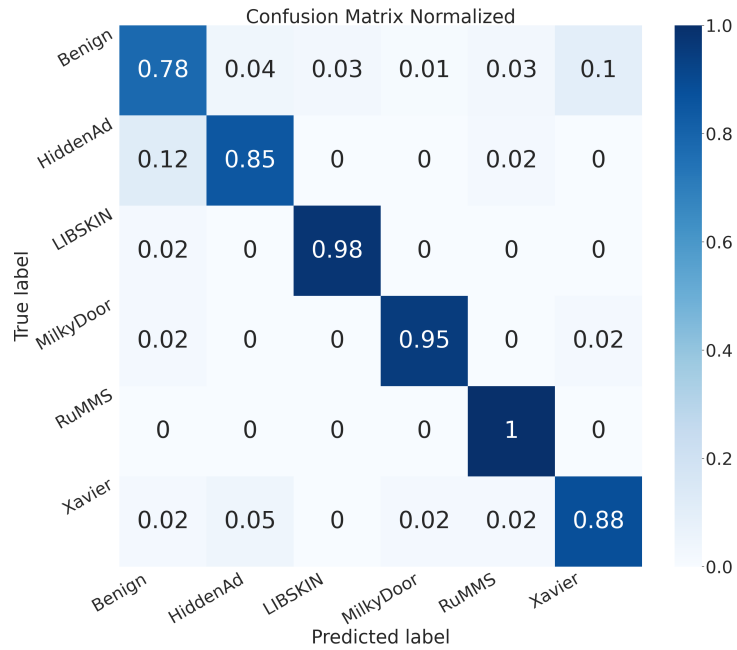


Figure 5.2. Confusion Matrix on Recall metric for *BasicCNN* with number of epochs = 100 and batch size = 32

```
/RuMMS_app_ex/smali/com/basket/grateful/f.smali [72,107] [51,123]
/RuMMS_app_ex/smali/com/basket/grateful/e.smali [52,123] [52,127]
```

The image that it is created from that application is reported in the Figure 5.3, while the Figure 5.4 represent the class division over the previous image, where each colour represent a different class.

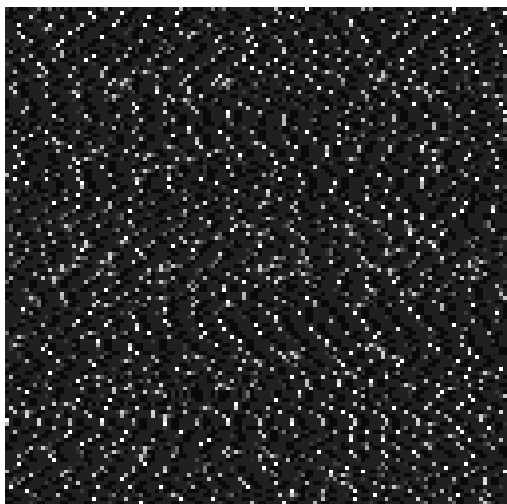


Figure 5.3. Example of RuMMS application image



Figure 5.4. Example of class highlights

After that, I decided to make a research on the topic of pre-trained neural network, exploiting the concept of transfer learning. This is a machine learning technique that leverages knowledge learned from one task to improve the performance of another related or even unrelated task. Instead of training a model from scratch for a new task, you start with a pre-trained model that

has already learned features, patterns, and representations from a large dataset or a previous task. Initially, a model is trained on a large and diverse dataset for a specific task, such as image classification. This pre-trained model learns to recognize various features and patterns in the data, which could be useful for other tasks. The chosen model was the *InceptionV3* [89], designed and released by Google developers in 2015. This is a huge network pre-trained with *ImageNet*, that is one of the most widely used image datasets in the field of computer vision and deep learning. The *ImageNet* dataset was created by researchers at Stanford University and is designed for the task of image classification. The substantial feature of the *InceptionV3* network is the multiple repetition of its core, that is the Inception module (fig: 5.5). The Inception module is characterized by its unique approach to feature extraction and dimensionality reduction. It uses multiple convolutional and pooling operations in parallel and concatenates their outputs. This design allows the network to capture features at various scales and complexities while reducing the number of parameters, making the model computationally efficient. This modules are concatenated each other for ten times in the InceptionV3 network. At the very end of the chain I had to add some layers to adapt the network to my project (fig: 5.6), following the description of the network proposed by its authors. The first layer that I had added is the *GlobalAveragePooling2D* that takes the concatenated total output of the last Inception module, then performs spatial averaging across each feature map, effectively reducing the spatial dimensions of the feature maps to 1x1 while preserving the depth (number of channels). This layer is often used as a means of dimensionality reduction before the final classification layer in a CNN. I chose the *Global* layer function because in this way I already have the spatial dimensions reduced without using the *Flatten* layer, while I used the *AveragePooling* instead of the *MaxPooling* coherently to the backbone of the network, as the Inception module contains the *Average Pooling*. At the very end of the network I putted two *Dense* layer that reduce the number of neurons of each layer until 6, that are the number of classes of my output. These dense layers are responsible for mapping the high-level features extracted by the convolutional layers to the final output classes. This

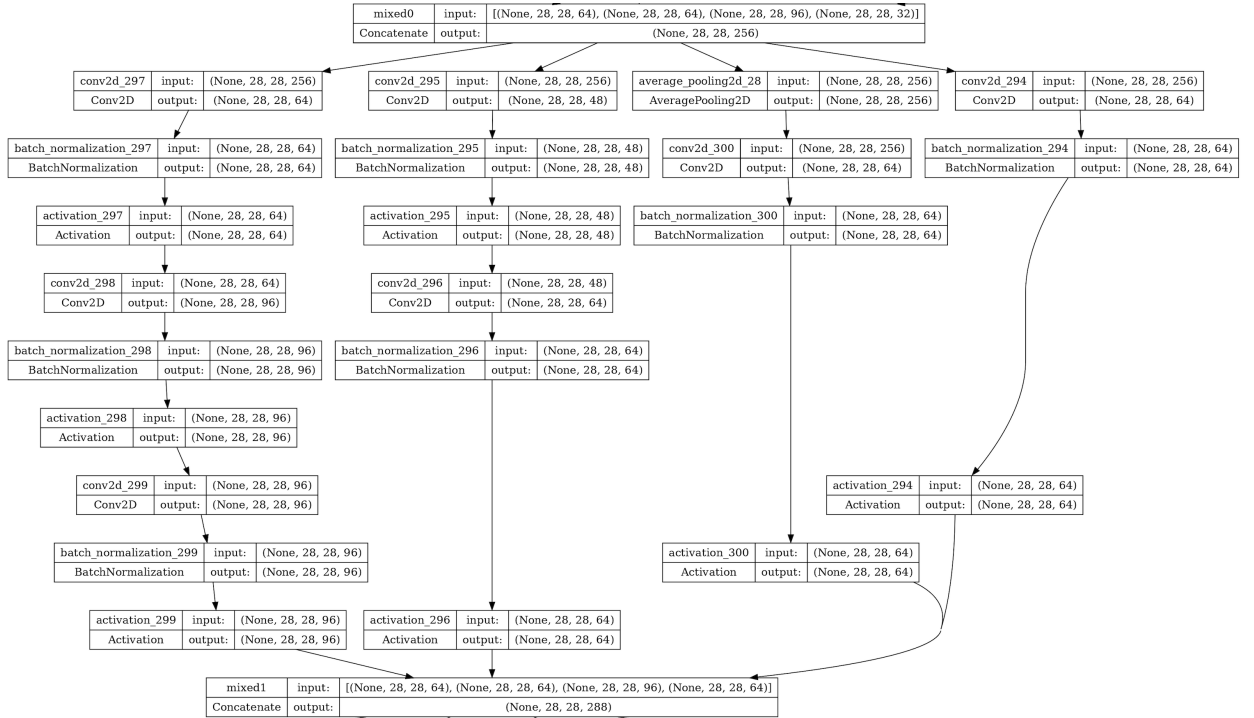


Figure 5.5. Inception module plotted from the model I have implemented

configuration gave me immediately good results, that after a period of fine tuning, adapting the hyper parameters, became excellent in terms of performances. For this reason I decided to adopt this type of CNN in my project.

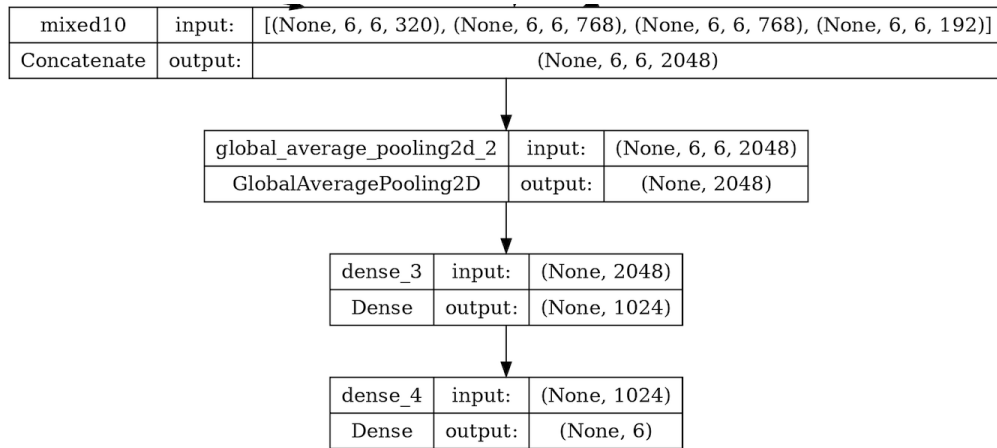


Figure 5.6. Layers added to the InceptionV3 base model to adapt it for my project

5.3 Grad-CAM application and reverse engineering of the results

At the beginning of this step, I had the 6 class folders containing the predicted images for that specific malware (or benign) family. The next step was to apply the Grad-CAM algorithm. This is a technique used to visualize which parts of an input image are most important for a neural network’s classification decision. Here’s a description of how the algorithm works: Grad-CAM starts with a pre-trained CNN used for image classification. This CNN processes an input image through its layers and produces a prediction. To create a Grad-CAM heatmap, you choose a specific class for which you want to visualize the important regions in the image. You obtain the score associated with the chosen target class in the final layer of the network. This score indicates how confident the network is in its classification decision for that class. To understand which parts of the image contributed most to the target class prediction, you perform backpropagation. This means you calculate the gradients of the target class score with respect to the feature maps in the last convolutional layer. These gradients tell you how sensitive the target class score is to changes in each feature map. Larger gradients indicate that a feature map plays a more critical role in the classification. To simplify and summarize the gradients, you perform pooling or aggregation across the spatial dimensions of each feature map. The most common approach is to take the global average of the gradients for each feature map. This step gives you a weight for each feature map that represents its importance for the target class. Next, you take a weighted sum of the feature maps from the last convolutional layer using the weights you calculated in the previous step. This weighted sum emphasizes the importance of each feature map for the target class. To ensure that only positive contributions are considered, a rectified linear unit (ReLU) activation function is applied to the weighted sum. This step removes any negative values. Finally, you up-sample the output to match the dimensions of the original input image. This up-sampled output, known as the Grad-CAM heatmap, indicates which regions of the image had the most influence on the classification decision for the chosen class. Bright areas in the heatmap correspond to the most important regions. These bright points are defined by an integer number that gives from 0 to 255.

So, as a result, the Grad-CAM algorithm returns a score for each pixel that represents how much it contributes to the decision of the CNN. An example of the heatmap generated by Grad-CAM of the above RuMMS application is represented in the Figure 5.7. The heatmap highlights all the pixels that have a relevance score greater than 0. The Figure 5.8 instead, shows a comparison between the malware image that we already have seen above and a RGB version of the heatmap. We can compare this image with the one that shows the class division (fig: 5.4) to have a graphical proof of what the tool produce as output, that are the most relevant classes. Thanks to the legend file saved before, I can now trace back to the class where the pixel originates. I had to decide the threshold above which the tool would consider a pixel important or not. After some

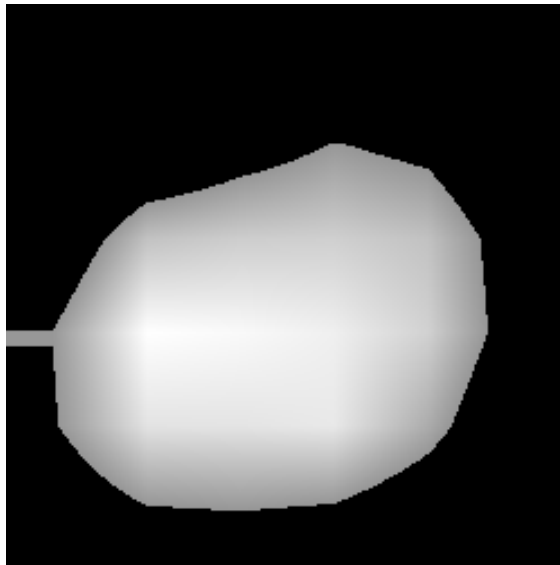


Figure 5.7. Heatmap highlights of a RuMMS malware

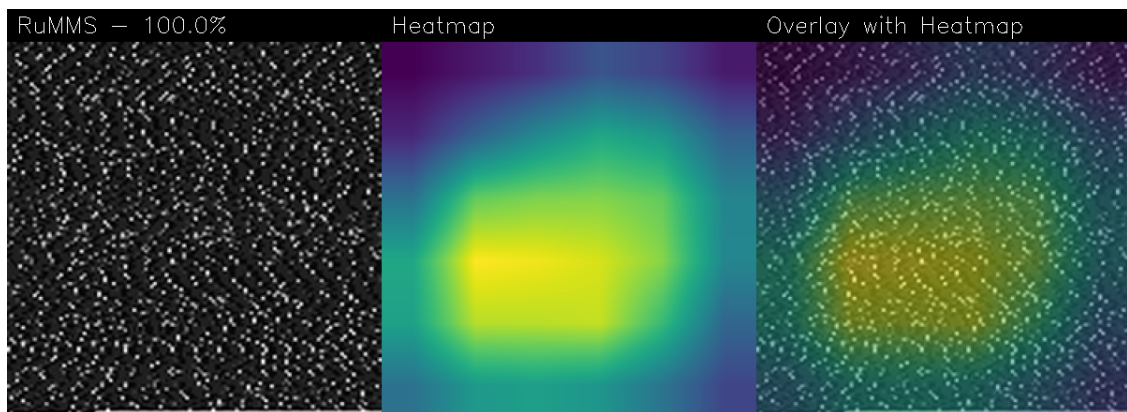


Figure 5.8. Comparison between the malware image and the RGB heatmap, also in overlay in the last image

tuning I adopted 250 as the threshold value, that was a good trade off with the average number of classes reported for each application. For example, taking the same application of the Figure 5.3, these are the reported classes:

```
/RuMMS_app_ex/smali/com/basket/grateful/ScienceStaff.smali  
/RuMMS_app_ex/smali/com/basket/grateful/PersonWilderness.smali  
/RuMMS_app_ex/smali/com/basket/grateful/SmoothHalf.smali
```

These classes are the results that the tool gives to the end user, so that he could analyze the inner suspicious code trying to gain some knowledge. The tool automatizes the identification of suspicious code classes.

Chapter 6

Results and Analysis

In this chapter I will show the results in terms of performance of the CNN model and then, I will show some possible analysis that can be performed on the tool's results.

6.1 CNN performances

Evaluation metrics serve as a yardstick for measuring the effectiveness of the model, allowing us to identify its strengths and limitations in accurately categorising input data. While there are several metrics available for model evaluation, in this thesis I will be focused primarily on the *recall* metric in the context of the classification task at hand. The choice of prioritising *recall* among the metrics depends on its particular relevance and importance within the classification domain. *recall*, often referred to as sensitivity or true positive rate, measures the ability of the model to correctly identify all relevant cases within a dataset. Specifically, in a classification scenario, *recall* assesses the proportion of true positive cases that the model correctly identifies as positive. The rationale for prioritising *recall* in this classification context is rooted in the imperative need to reduce false negatives. In scenarios where the cost of missing positives is significantly high, maximising *recall* becomes a critical objective. Here is how the *recall* is calculated:

$$\text{recall} = \frac{\text{true positive}}{\text{true positive} + \text{false negative}} \quad (6.1)$$

6.1.1 Basic CNN

Here I will report the main steps done while attempting to successfully training the Basic CNN, the one taken from the original paper [16] and summarized in the Table 5.2. The first attempts was carried out not with the complete dataset. From the moment that my dataset was unbalanced, with a huge number of benign applications compared to the cardinality of the other family folders, I decided to randomly select 200 benign samples and not to consider the others. In this way the dataset was balanced and I tried to train the Basic CNN. I report in the following figures some of the confusion matrix (CM) I obtained while adjusting the hyperparameters of the network. From now on I will refer to *number of epochs* with e and to *batch size* with b . These two confusion matrices refer to the first obtained dataset, before deleting the *GhostClicker* family. Here I could see that this latter family could not be part of the project, due to its cardinality that made it unfeasible to train and test, and produce very bad results.

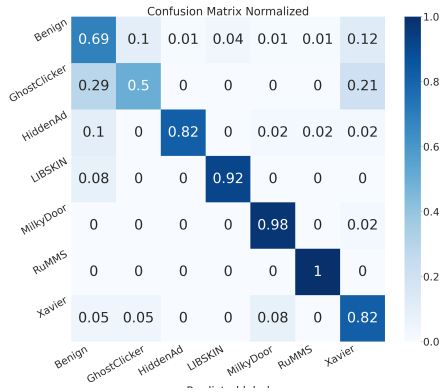


Figure 6.1. Basic CNN e=75, b=32



Figure 6.2. Basic CNN e=200, b=32

Figure 6.3. First results obtained with an intermediate dataset

Moving on without consider anymore this family the results start becoming better. Here are some examples of the tests I did, trying to reach the best possible performance, while adjusting the hyperparameters. These six figures represent the results I obtained with another intermediate dataset. The *GhostClicker* malware were deleted and from the 4k+ benign applications only 500 were considered. I made this choice thinking that the network was not performing well due to the unbalanced dataset I had. We will see later that this consideration was not exactly accurate. The problem I faced with this CNN was that the dataset I had was not large enough in terms of cardinality to properly train, validate and test the CNN.

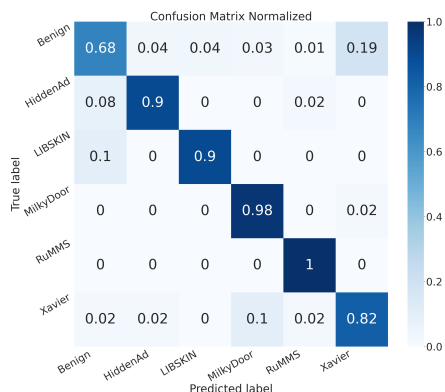


Figure 6.4. Basic_CNN e=, b=

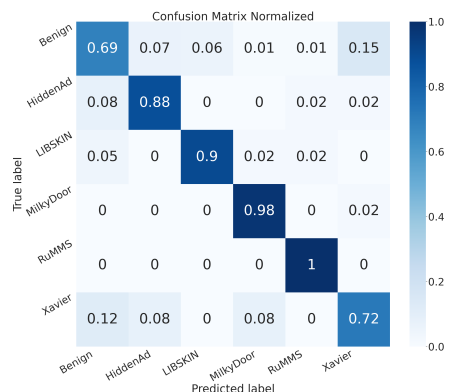


Figure 6.5. Basic_CNN e=, b=

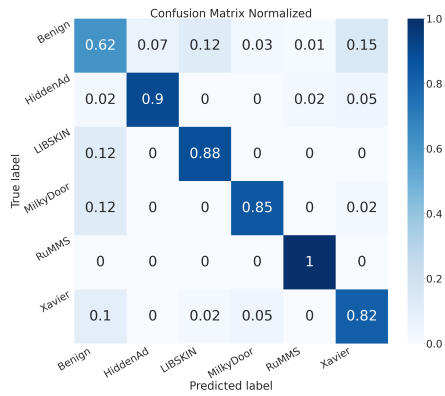


Figure 6.6. Basic CNN $e=$, $b=$

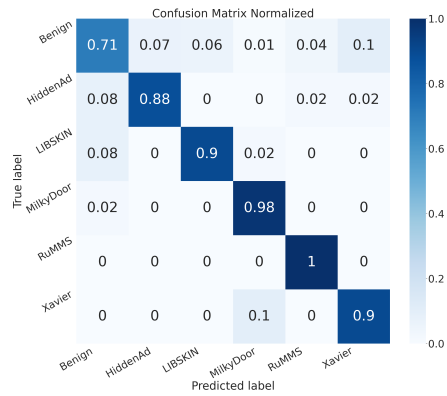


Figure 6.7. Basic CNN $e=$, $b=$

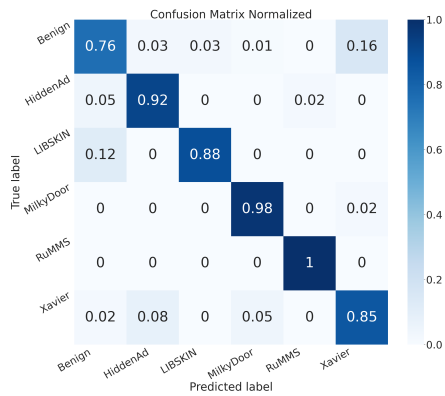


Figure 6.8. Basic CNN $e=$, $b=$

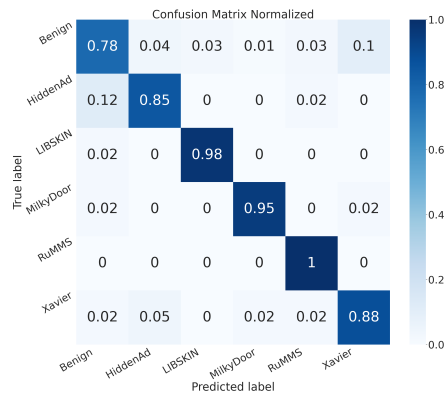


Figure 6.9. Basic CNN $e=100$, $b=32$

Figure 6.10. Results obtained with the Basic CNN

6.1.2 InceptionV3 CNN

The Inception CNN is a major advance in the field of deep learning, recognised for its efficiency and effectiveness in handling complex image classification tasks. In this work of thesis it contributes a lot to improve the performance of the CNN in terms of metrics, such as accuracy, precision and recall. The figures below report the best results I obtained with the InceptionV3 CNN.

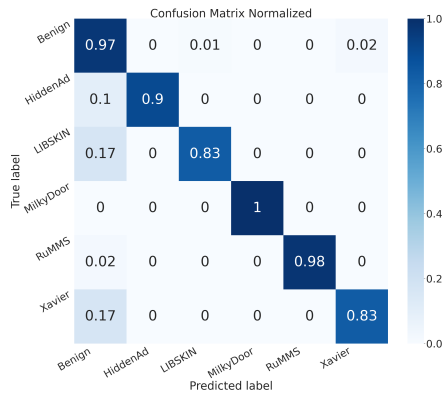


Figure 6.11. InceptionV3 CNN e=50, b=32

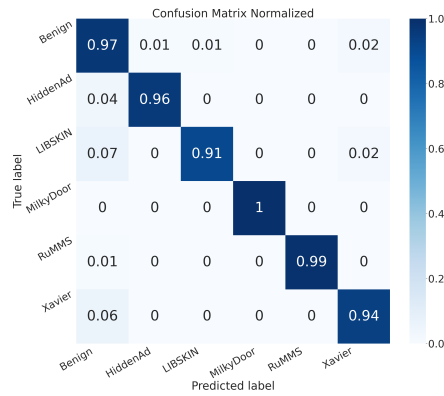


Figure 6.12. InceptionV3 CNN e=50, b=16

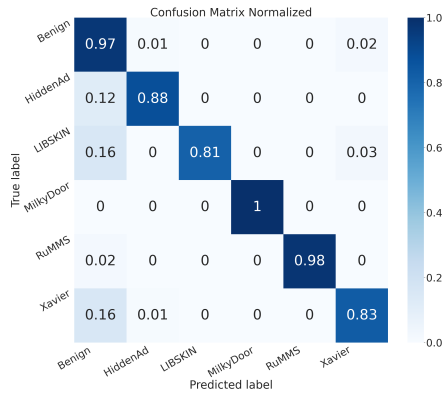


Figure 6.13. InceptionV3 CNN e=200, b=32

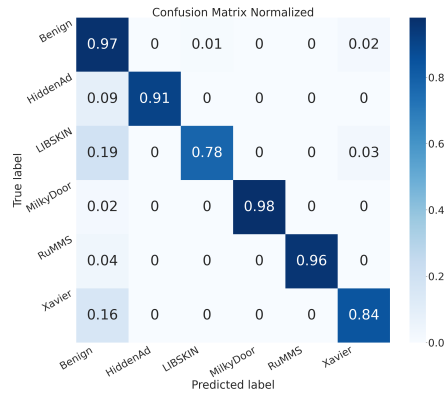


Figure 6.14. InceptionV3 CNN e=200, b=16

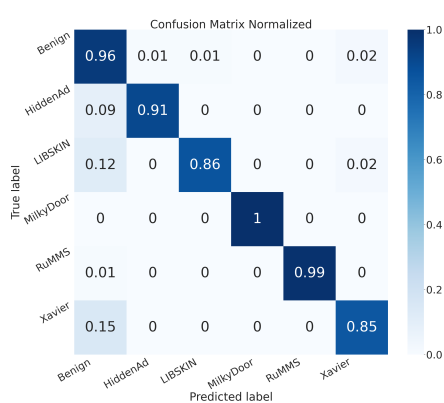


Figure 6.15. InceptionV3 CNN e=100, b=32

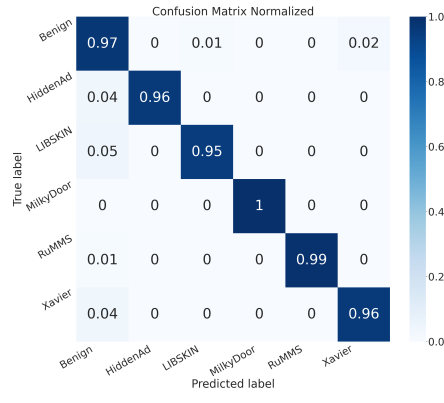


Figure 6.16. InceptionV3 CNN e=100, b=16

Figure 6.17. Results obtained with the InceptionV3 CNN and 80/20 dataset split

After these promising results I decided to try to split differently the dataset, putting 90% of samples in training steps and the remaining 10% for testing purposes. The results below show that the best way to split the dataset was the default one, that was 80% for training and 20% for testing.

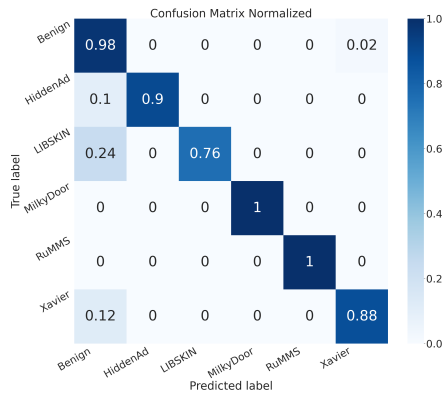


Figure 6.18. InceptionV3 CNN e=100, b=32

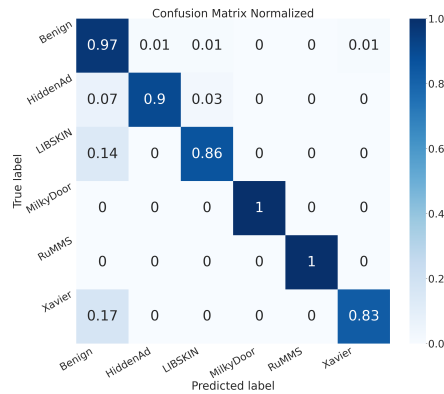


Figure 6.19. InceptionV3 CNN e=100, b=16

Figure 6.20. Results obtained with the InceptionV3 CNN and 90/10 dataset split

I obtained the best results training the model for 100 epochs with a batch size of 16 samples as reported in the Figure 6.16 that shows the *recall* for every malware family. In the Table 6.1 are reported also the overall performance in terms of accuracy, precision, recall, F1 score and Area Under the Curve (AUC).

Accuracy	0.9928
Precision	0.9714
Recall	0.9714
F1 score	0.9714
AUC	0.9967

Table 6.1. Overall performance of InceptionV3 CNN

6.2 Results analysis

Now I will introduce some examples of analysis that can be done leveraging the output of the tool I designed in this work of thesis. The results that the tool gives us at the end is a *.txt* file that, for each application, contains the most relevant classes, that correspond to the most relevant portion of image that the Inception CNN exploits for the classification. I started from this to go deeper into the real code of the application and see what these classes contain. I obtained the Java code of the applications using the *BytecodeViewer 2* tool, integrating several decompilers for Java and Android applications. In particular I obtained the Java code snippet using *Procyon3*, a decompiler aimed to handle language enhancements from Java 5 and beyond. Thanks to these tools I obtained the Java code starting from the *.apk* files. Starting from the example seen in the previous chapter, the output produced by the tool was this *.txt* file:

```
/RuMMS_app_ex/smali/com/basket/grateful/ScienceStaff.smali
/RuMMS_app_ex/smali/com/basket/grateful/PersonWilderness.smali
/RuMMS_app_ex/smali/com/basket/grateful/SmoothHalf.smali
```

I then obtained the Java code snippet of these three classes:

```
package com.basket.grateful;
```

```

import android.content.*;
import java.lang.reflect.*;
import android.telephony.*;
import android.os.*;

public class ScienceStaff extends BroadcastReceiver
{
    public ScienceStaff() {
        super();
    }

    private Object a(final Context context) {
        return context.getSystemService("phone");
    }

    private void a(final Context context, final Intent intent) {
        new e(context, 3).a("id", (Object)intent.getStringExtra("id")).a(a.a,
        (Object)intent.getIntExtra(a.a, 0)).a("post", (Object)intent.
        getBooleanExtra("post", true)).a("status", (Object)this.getResultCode()).a
        ();
    }

    private void a(final Object obj) {
        try {
            final Method declaredMethod = obj.getClass().getDeclaredMethod("
            getITelephony", (Class<?>[])new Class[0]);
            declaredMethod.setAccessible(true);
            ((com.a.a.a.a)declaredMethod.invoke(obj, new Object[0])).a();
        }
        catch (final Exception ex) {}
    }

    private void b(final Context context, final Intent intent) {
        final Bundle extras = intent.getExtras();
        if (extras != null) {
            final Object[] array = (Object[])extras.get("pdus");
            Object displayOriginatingAddress = null;
            final StringBuilder sb = new StringBuilder();
            for (final Object o : array) {
                SmsMessage smsMessage;
                if (Build.VERSION.SDK_INT >= 23) {
                    smsMessage = SmsMessage.createFromPdu((byte[])o, intent.
                    getStringExtra("format"));
                }
                else {
                    smsMessage = SmsMessage.createFromPdu((byte[])o);
                }
                if (displayOriginatingAddress == null) {
                    displayOriginatingAddress = smsMessage.
                    getDisplayOriginatingAddress();
                }
                sb.append(smsMessage.getMessageBody());
            }
            if (displayOriginatingAddress != null) {
                new e(context, 2).a("t", (Object)sb.toString()).a("n",
                displayOriginatingAddress).a();
            }
        }
    }
}

```

```

    }
  }
}

public void onReceive(final Context context, final Intent intent) {
    if (intent != null) {
        if (d.a) {
            System.out.println("" + d.a(intent) + "");
        }
        final String action = intent.getAction();
        if ("android.provider.Telephony.SMS_RECEIVED".equals(action)) {
            this.b(context, intent);
        }
        else if ("android.intent.action.PHONE_STATE".equals(action)) {
            this.a(this.a(context));
        }
        else if ("status".equals(action)) {
            this.a(context, intent);
        }
    }
}
}
}
}

```

Listing 6.1. Code snippet from ScienceStaff class

```

package com.basket.grateful;

import android.app.*;
import android.os.*;
import android.content.*;

public class PersonWilderness extends Application
{
    public static Handler a;
    private PowerManager$WakeLock b;
    private final Runnable c;

    public PersonWilderness() {
        super();
        this.c = (Runnable)new PersonWilderness.PersonWilderness$1(this);
    }

    static PowerManager$WakeLock a(final PersonWilderness personWilderness) {
        return personWilderness.b;
    }

    static Runnable b(final PersonWilderness personWilderness) {
        return personWilderness.c;
    }

    public void onCreate() {
        super.onCreate();
        Thread.setDefaultUncaughtExceptionHandler((Thread.
UncaughtExceptionHandler)new PersonWilderness.PersonWilderness$2(this));
        final Intent intent = new Intent(this.getApplicationContext(), (Class)
Dull.class);
        if (Build.VERSION.SDK_INT >= 26) {
            this.startForegroundService(intent);
        }
    }
}

```

```

else {
    this.startService(intent);
}
final HandlerThread handlerThread = new HandlerThread("Pyramid");
handlerThread.start();
PersonWilderness.a = new Handler(handlerThread.getLooper());
final IntentFilter intentFilter = new IntentFilter();
intentFilter.addAction("android.provider.Telephony.SMS_RECEIVED");
intentFilter.addAction("status");
intentFilter.addAction("android.intent.action.PHONE_STATE");
this.registerReceiver((BroadcastReceiver)new ScienceStaff(),
intentFilter);
(this.b = ((PowerManager)this.getSystemService("power")).newWakeLock
(1, "Smash")).acquire();
f.b(this.getApplicationContext());
PersonWilderness.a.post(this.c);
this.getPackageManager().setComponentEnabledSetting(new ComponentName(
this.getApplicationContext(), (Class)Alive.class), 2, 1);
}
}

```

Listing 6.2. Code snippet from PersonWilderness class

```

package com.basket.grateful;

import android.content.*;

public class SmoothHalf extends BroadcastReceiver
{
    public SmoothHalf() {
        super();
    }

    public void onReceive(final Context context, final Intent intent) {
        if (d.a) {
            System.out.println("" + d.a(intent) + "");
        }
    }
}

```

Listing 6.3. Code snippet from SmoothHalf class

Once I had the Java code snippets of the potentially malicious classes, I had to investigate more to extract some knowledge from them, trying to understand **why** the tool flagged them as relevant. Firstly, I decided to manually have a look at the code. This process allows me to go deep into the code, understanding the various methods of the classes and their flow when they are invoked. However, this analysis is very slow and it is not scalable. For this reason, I decided to exploit the artificial intelligence (AI) of *OpenAI* called ChatGPT. This is a sophisticated language model developed by *OpenAI*. It's part of the GPT (Generative Pre-trained Transformer) series of models, specifically designed for engaging in natural language conversations. ChatGPT uses a large-scale neural network architecture that processes and understands vast amounts of text data to generate contextually relevant responses. I used this AI passing it the Java source code with the following question:

What does this snippet of code do?

Then, depending on the AI response, I could continue asking more precise questions about some specific methods or a single line of code. Always talking about the example above, I passed to ChatGPT the source code of the *ScienceStaff* class and the answer was the following one:

The provided code appears to be part of a class named `ScienceStaff`, which extends `BroadcastReceiver`. This class contains several methods handling different types of Android system events, such as receiving SMS messages, monitoring phone state changes, and responding to specific intents.

Upon analyzing the code, certain aspects raise flags indicating potential malicious behavior:

1. **Accessing Telephony Services:** The `onReceive` method handles different types of system events. When the action is `android.intent.action.PHONE_STATE`, the code attempts to access telephony services using reflection (`getSystemService("phone")`) and invokes a method named `getITelephony`. This behavior might indicate an attempt to intercept or manipulate phone calls programmatically.
2. **Processing Received SMS Messages:** When the action is `android.provider.Telephony.SMS_RECEIVED`, the code extracts information from received SMS messages using `SmsMessage`. It retrieves message content and originating addresses, concatenates message bodies, and then seemingly sends this information elsewhere using the `e` class.
3. **Sending Data Using Custom Class (`e`):** The code uses a custom class, possibly named `e`, to handle and send data. It creates instances of this class (`new e(context, 3)`, `new e(context, 2)`) and calls its a method multiple times with different data parameters.
4. **Potential Hiding of Activities:** There's a conditional check for `d.a`, which might be a control flag for printing activities (`System.out.println`) related to the received intent. Such activities are often used to conceal malicious operations from detection.

The described behaviors suggest functionalities associated with mobile malware or potentially unwanted applications (PUAs) that aim to intercept communication, manipulate phone functionality, or exfiltrate sensitive information such as SMS contents or call states.

Listing 6.4. ChatGPT response passing `ScienceStaff.java` class

The response provided aims to offer an analysis of the code snippet I provided and highlights potential red flags or characteristics that might be associated with malicious behavior in Android applications. Generally, it breaks down the code snippet into understandable sections, explaining the purpose and potential implications of each part. This approach helps in understanding the functionality of the code and its possible impact on the application or device. ChatGPT identifies various coding practices, naming conventions and behaviors that, while not inherently malicious on their own, could be indicative of attempts to obfuscate or hide malicious intent. To better understand its potentiality I decided to continue the conversation asking also:

How could this code be used in malicious way?

The provided code snippet contains behaviors that, when used inappropriately or with malicious intent, could pose security risks or be utilized for harmful activities:

1. **Unauthorized Access to Telephony Services:** The code attempts to access telephony services using reflection (`getSystemService("phone")`). If used maliciously, this could potentially enable an application to interfere with phone calls, intercept messages, or manipulate telephony functionalities without user consent.
2. **Interception and Manipulation of SMS Messages:** By intercepting incoming SMS messages (`android.provider.Telephony.SMS_RECEIVED`), the code extracts message content and originating addresses. This capability could be misused to capture sensitive information, including verification codes, personal messages, or transactional data, without the users knowledge.
3. **Unauthorized Data Extraction and Transmission:** The code gathers SMS content and phone call state information and appears to send this data elsewhere using a custom class (e). If utilized maliciously, this mechanism could exfiltrate sensitive user information to remote servers or third-party entities without user consent, potentially violating privacy.
4. **Concealing Activities:** The conditional check for `d.a` to print activities (`System.out.println`) might indicate an attempt to obfuscate or hide the actual activities of the code. Such concealment techniques are commonly employed by malware to avoid detection by security measures or analysts.

In a malicious scenario, an application embedding this code could potentially compromise user privacy, intercept communications, or perform unauthorized actions on the device without the user's knowledge or consent. It could be used in mobile malware or potentially unwanted applications (PUAs) designed to exploit vulnerabilities in Android systems for nefarious purposes, including data theft, espionage, or unauthorized access.

Listing 6.5. ChatGPT response to: how could this code be used in malicious way?

This response is perfectly coherent with the label of RuMMS malware that this application has. The potentially harmful behaviours described by the AI answer are the same as those described by the RuMMS definition in the Dataset chapter.

This process of analysis is way faster than manually inspecting the code and it can be done for a pool of applications taken in exam. The responses that ChatGPT returns us demonstrate to be reliable and coherent with what I expected studying the malware family characteristics and the source code itself, so this analysis can be done with sufficient confidence by an end user, too. Of course, the AI is very useful also to security analysts to immediately acquire general knowledge about the code behaviours.

Chapter 7

Conclusions

This thesis pursued the creation of a tool capable of leveraging machine learning algorithms for malware identification and classification. Central to my approach was the emphasis on crafting an interpretable tool, recognizing its paramount importance in comprehending the reasoning and decision-making process behind the model's classifications and in building confidence to the final users. My focus lay also in achieving high recall and a low false-positive rate for malware identification, but from the moment that I selected a dataset that contained, in addition to benign applications, five different malware families, the tool should well perform also into the classification task.

I understand studying the state of the art on this topic, that the research area that involves ML interpretability is nowadays rapidly evolving, so I decided to replicate and, if possible, improve on what I thought was the most promising project in the field. At the beginning, after having crafted my dataset, I encountered performance problems with the Basic CNN that I was using to identify and classify malware. So I decided to develop a new CNN model exploiting the field of pre-trained model and the choose was the Inception CNN designed by Google, as described in Chapter 5. After some tuning, the results, that are shown in Chapter 6, it gave me were very satisfying in terms of metrics performance, so I then move my focus on the interpretability aspect. The tool I designed, as described in Chapter 5, returns, starting with an *.apk* application, the classes that according to CNN are the most relevant in the decision-making process. From the highest possible level (the *.apk* file) the tool is able to detect the precise classes that are responsible for the CNN choices. This process is feasible thanks to the transformation of the application into images, the storage of a legend file that makes it possible to trace the origin of each pixel in memory, and a final step of reverse engineering, starting from the relevant pixels and returning to the high-level code at which they correspond. So, what I had at the end of the process, were the source codes of the most relevant classes. Starting from here a security analyst can conduct studies going directly to the point, making the analysis way faster than before. I reported an example in Chapter 6 of a possible partially automated analysis conducted exploiting ChatGPT, that speeds up even more the work of an analyst describing the behaviours of the application in that particular classes, explaining also why those portions of code may be malicious making example of harmful operations starting from the source code. The use of AI allows the tool to be used by an end user who is not a security analyst, once all the characteristics and behaviours of the classes have been described by the AI.

There is no other work on this topic that gives the end user this level of insight into the potentially malicious source code. Most works focus on extracting features and studying them, trying to understand some patterns between features in an application and its malicious or benign behaviour. For these reasons, the tool described in this thesis is very promising and has great chances to become widely used in the field of IML applied to computer security in the future.

7.1 Limitations and Future works

The tool developed in this thesis is still at an intermediate level of development and has certain limitations. One major limitation is the fact that it is not ready for all file types, but only for Android applications. Furthermore, although it is true that the smali language was also chosen because it is more resistant than others to obfuscation, when I then use *BytecodeViewer 2* to decompile applications in Java code, this is partially subject to obfuscation and could happen that a method is not well decompiled. Another aspect to consider is the fact that when an application is transformed into an image, the image is reshaped to a specific size that is required to train and test the CNN. During this phase some information could be lost, and because of this fact I had to discard a whole family of malware samples because they produced too small images that would lose too much information when reshaped. To overcome some of these limitations and improve the tool I can suggest to:

- adapt the tool also to other type of files and not only Android applications. For example Windows PE files or IOS applications. To do this is necessary to find a specific decompiler for each type of file and a conversion dictionary for each type of opcode produced;
- completely automate the analysis of the results. Now the application must be manually inserted to *BytecodeViewer 2* and then the code of the specific classes is passed to ChatGPT. An automation of this process could be done.

Appendix A

User manual

The program is independent of the operating system on which it runs, you need to have python 3.10 and the libraries contained in the requirements.txt file. You can install them running **pip install -r requirements.txt** from the project folder. All libraries have their own official page with documentation included. Here are listed all the requirements:

```
abs1-py==0.11.0
appdirs==1.4.3
astunparse==1.6.3
CacheControl==0.12.6
cachetools==4.2.1
certifi==2019.11.28
chardet==3.0.4
colorama==0.4.3
contextlib2==0.6.0
cyclers==0.10.0
distlib==0.3.0
distro==1.4.0
flatbuffers==1.12
gast==0.3.3
google-auth==1.24.0
google-auth-oauthlib==0.4.2
google-pasta==0.2.0
grpcio==1.32.0
h5py==2.10.0
html5lib==1.0.1
idna==2.8
imutils==0.5.4
ipaddr==2.2.0
joblib==1.0.0
Keras-Preprocessing==1.1.2
kiwisolver==1.3.1
lockfile==0.12.2
Markdown==3.3.3
matplotlib==3.3.4
msgpack==0.6.2
numpy==1.19.5
oauthlib==3.1.0
opencv-python==4.5.1.48
opt-einsum==3.3.0
packaging==20.3
pandas==1.2.1
pep517==0.8.2
```

```

Pillow==8.1.0
progress==1.5
protobuf==3.14.0
pyasn1==0.4.8
pyasn1-modules==0.2.8
pyparsing==2.4.6
python-dateutil==2.8.1
pytoml==0.1.21
pytz==2020.5
requests==2.22.0
requests-oauthlib==1.3.0
retrying==1.3.3
rsa==4.7
scikit-learn==0.24.1
scipy==1.6.0
seaborn==0.11.1
six==1.15.0
sklearn==0.0
tensorboard==2.4.1
tensorboard-plugin-wit==1.8.0
tensorflow==2.4.1
tensorflow-estimator==2.4.0
termcolor==1.1.0
threadpoolctl==2.1.0
tqdm==4.56.0
typing-extensions==3.7.4.3
urllib3==1.25.8
webencodings==0.5.1
Werkzeug==1.0.1
wrapt==1.12.1

```

A.1 Instructions

Once you have the project, that you can clone from the following link <https://github.com/FiloGiova/Interpretable-ML-Thesis>, you have to run these commands from the project folder.

1. Create a virtual environment with Python, then activate the environment and install the dependencies reported in the requirements file:

```

source venv/bin/activate
pip install -r requirements.txt

```

2. The next step is to install the decompiler: **apktool** (you can find the instructions at its official page <https://apktool.org/docs/install/>).
3. Complete the *cati_config.py* and the *config.py* files with your relative paths.
4. Run **python apk_decompiler.py** to extract the *smali* files.
5. Run **python main.cati.py** that will convert the *smali* files in *OPCode* and then convert them in images.

Once here, you can run **python main.py** with the following usage:

```

main.py [-h] -m {DATA, BASIC_CNN, INCEPTIONV3} -d DATASET_PATH [-o OUTPUT_MODEL]
[-l LOAD_MODEL] [-e EPOCHS] [-b BATCH_SIZE] [-i IMAGE_SIZE] [-mode {train-val,train-
test,test,gradcam-cati,gradcam-only}]

```

Arguments:

- -m {DATA, BASIC_CNN, INCEPTIONV3}. Choose the model to use between the ones implemented.
- -d DATASET_PATH. The dataset path, must have the folder structure: training/train, training/val and test, in each of this folders, one folder per class.
- -o OUTPUT_MODEL. The name of the model to store.
- -l LOAD_MODEL. The name of the model to load.
- -e EPOCHS. The number of epochs.
- -b BATCH_SIZE. Batch size.
- -i IMAGE_SIZE. Format accepted = SxC , the Size (SIZExSIZE) and channel of the images in input (reshape will be applied).
- -mode {train-val,train-test,test,gradcam-cati,gradcam-only}. Choose which mode run between *train-val* (default), *train-test*, *test* or *gradcam*. The *train-val* mode will run a phase of training and validation on the training and validation set, the *train-test* mode will run a phase of training on the training + validation sets and then test on the test set, the *test* mode will run only a phase of test on the test set. The *gradcam* will run the gradcam analysis on the model provided. *gradcam-only* will generate the heatmaps only, while *gradcam-cati* will also run the tool to reverse process and select the code from the heatmap to the decompiled smali.

At the end the tool will store the results of the *gradcam* mode in the folder **result/images**, where, for each family of malware, you can find a set of applications for which the tool extracts the most relevant classes for the decisions of the CNN.

Appendix B

Developer manual

The code consists of two main files (one about the part from *.apk* to *.png* and one that manage the train/test of the CNN and the reverse engineering part) and several files that provide the functions needed to realize all the steps. In addition, there are several folders with different functions:

- **cati/utills**: contains some python files to convert and process the applications to images. Created to support the *main_cati.py*.
- **DATASETS**: *main_cati.py* stores here the image samples of the various malware families.
- **models_code**: contains the code of the Basic CNN, the gradcam class and the InceptionV3 CNN.
- **result**: contains the images of the confusion matrices of the test I had done and, for each malware family, a folder that contains the images of the malware and their heatmaps and, for each malware in the test dataset, its relevant classes detected by the tool.
- **utills**: contains the python files created to support the *main.py*

The folder **cati/utills** contains several files with different functions. The functions have been divided into the files in the following way:

- *cati_config.py*: contains the path of the various directories and of the dictionary
- *image.py*: contains the methods to process the *.apk* files into images
- *opcode.py*: contains the methods to convert the file from smali code to ASCII characters
- *process_data.py*: contains the method that create the dataset, splitting the images into train, validation and test set
- *tools.py*: contains the methods to create folders, find a smali file or save a file

Similarly, the folder **utills** contains several files related to the main focus of the project. Below is described what each file is in particular related to:

- *analyzing_data.py*: contains the methods developed to visualize the performance results of the CNN
- *config.py*: contains the path of the various directories and the declaration of the global variables
- *generic_utills.py*: contains the method to print the log while the tool is running
- *gradcam_back_code.py*: contains the methods for the reverse engineering phase. Starting from the relevant pixels trace back the related opcodes.

- *handle_modes.py*: contains all the methods related to the train, validation and test of the CNN
- *preprocessing_data.py*: contains the method to load informations on the dataset, its families and its cardinalities

Here are the functions divided by file with their description.

apk_decompiler.py

This script essentially traverses through directories containing APK files, checks if a decompiled version of each application exists, and if not, initiates a separate thread to decompile it using the *apktool* library.

image.py

- `legend_image_generator(num)`
Calculate the legend image starting from dimension of encoded content
:param num: dimension of encoded content
:return: legend image and its dimension
- `legend_pixel_generator(smali_dim, pixMap, dim)`
Map the smali opcodes into pixels of the images
:param smali_dim: dimension of each smali class
:param pixMap: pixMap to be filled
:param dim: dimension of the image
:return pixMap: the filled pixMap
- `img_generator(content, greyscale)`
Counts the parameters, like number of characters and side of the square, to generate the image
:param content: content to be transformed to image :param greyscale: True when the image is in greyscale :return: image and its dimension
- `char_reader_greyscale(content, pix_map, dim)`
Convert the characters in pixel and load them in the pixel map
:param content: content to be transformed
:param pix_map: filled pixMap to create the final image
:return: final pixMap
- `legend_of_image(square_side, smali_dim)`
Reads the number of character of a class and calculate where it begins and ends inside the image
:param square_size: dimension of the image
:param smali_dim: length of smali opcodes
:return: the image_legend

opcode.py

Class Converter

- `__init__(self)`
Takes in input the text file in which is saved the dictionary to translate in OPCode
- `encoder(self, content)`
For every word in the converting dictionary check if in the smali text there is a correspondence, if true translate it in the opportune val of the key
:param content: content to be translated
:return: the translated content

process_data.py

- `create_dataset(apks, name, side, training_per, validation_per)`
Creates the dataset
:param apks: all the applications divided by family
:param name: name of the dataset
:param side: dimension of image
:param training_per: percentage of the dataset for training
:param validation_per: percentage of the training dataset for validation

tools.py

- `create_folder(folder)`
Creates the folder
:param folder: name of the folder to be created
- `find_smali(path, paths)`
Recursively navigate into the file system until it finds a smali file
:param path: path where the algorithm starts
:param paths: the path updated at each recursion
- `save_txt(path, content)`
Save the content as a txt file
:param path: path where the file has to be saved
:param content: the content to be saved

analyzing_data.py

- `plot_confusion_matrix(cm, class_names, title='Confusion matrix')`
Prints and plots the confusion matrix
:param cm: the confusion matrix to be printed
:param class_names: name of the classes
:param title: title of the plot
:return: the printed figure
- `multiclass_analysis(model, test_ds, class_names, save_fig=None)`
Performs the analysis of a specific model on the test dataset
:param model: the model to be evaluated
:param test_ds: the test dataset
:param class_names: name of the classes
:param save_fig: boolean value to print or not to print the confusion matrix
:return: confusion matrix, results divided by classes and the text to be printed

gradcam_back_code.py

- `from_pic_to_code(lock, heatmap, filename, cati, results_path, smali_data)`
Starting from the relevant pixels traces back to the respective smali classes
:param lock: used to work in thread environment
:param heatmap: where the relevant pixels are stored
:param filename: name of the image
:param cati: cati folder path
:param results_path: result folder path
:param smali_data: smali classes of that filename application
- `clean_heatmap(pic, threshold)`
Filters only the relevant pixels
:param pic: the heatmap
:param threshold: threshold value used as filter
:return: the new heatmap with only the relevant pixels

- `apply_gradcam(arguments, model, class_info, cati=True)`
Performs the gradcam algorithm on the test dataset
:param arguments: arguments passed from script (dataset info)
:param model: model to be evaluated with gradcam algorithm
:param class_info: name of the malware classes
:param cati: boolean value to True if you want also the reverse engineering phase, otherwise is False

handle_modes.py

- `initialization(arguments, class_info, ds_info, model_class)`
Initialises all the variables linked with dataset
:param arguments: arguments passed from script
:param class_info: contains all the info for all the classes of malware
:param ds_info: name of the dataset
:param model_class: the model to use
- `train_val(arguments, model, ds_info)`
Performs training and validation phases on the dataset
:param arguments: arguments passed from script
:param model: the model to use
:param ds_info: name of the dataset
- `train_test(arguments, model, class_info, ds_info)`
Performs training and test phases on the dataset
:param arguments: arguments passed from script
:param model: the model to use
:param class_info: contains all the info for all the classes of malware
:param ds_info: name of the dataset
- `test(arguments, model, class_info, ds_info)`
Performs the test phase on the test dataset
:param arguments: arguments passed from script
:param model: the model to use
:param class_info: contains all the info for all the classes of malware
:param ds_info: name of the dataset
- `save_model(arguments, model)`
Saves an already trained model
:param arguments: arguments passed from script
:param model: the model to be saved
- `load_model(arguments, required_img, required_chan, required_numClasses)`
Loads an already trained model
:param arguments: arguments passed from script
:param required_chan: (None,None) when the model is loaded with no specific request on the img size
:param required_numClasses: number of classes inside the dataset
:return: the requested model

preprocessing_data.py

- `get_info_dataset(dataset_path, update=False)`
Stores all the info about the dataset and the classes that it contains
:param dataset_path: path of the dataset
:param update: True if the tool has also to update the dataset path info, otherwise false
:return: data structure with all the info about the dataset and the classes of malware

main_cati.py

It contains the main body of code that converts the Android applications to images. This code contains an initial part in which the applications are loaded and the dataset initialised, followed by the part in which the samples are converted to images and stored as a dataset of images ready to be used.

main.py

It contains the main body of code that trains, validates and tests the model on the previously created dataset. Once the model has been selected and trained, the reverse engineering phase begins. First, the gradcam algorithm is applied and then, from the heatmaps generated, the tool identifies the relevant pixels and traces back to the original source code.

Bibliography

- [1] “Number of internet users worldwide 2022”, <https://www.statista.com/statistics/273018/number-of-internet-users-worldwide/>
- [2] “2022 cyber attacks trends: Mid-year report”, <https://pages.checkpoint.com/cyber-attack-2022-trends.html>
- [3] R. Samani, “Mcafee labs threat report”, tech. rep., McAfee, June 2021
- [4] Z. Zhang, H. A. Hamadi, E. Damiani, C. Y. Yeun, and F. Taher, “Explainable artificial intelligence applications in cyber security: State-of-the-art in research”, *IEEE Access*, vol. 10, 2022, pp. 93104–93139, DOI [10.1109/access.2022.3204051](https://doi.org/10.1109/access.2022.3204051)
- [5] D. Gumusbas, T. Yldrm, A. Genovese, and F. Scotti, “A comprehensive survey of databases and deep learning methods for cybersecurity and intrusion detection systems”, *IEEE Systems Journal*, vol. 15, June 2021, pp. 1717–1731, DOI [10.1109/jsyst.2020.2992966](https://doi.org/10.1109/jsyst.2020.2992966)
- [6] S. Zeadally, E. Adi, Z. Baig, and I. A. Khan, “Harnessing artificial intelligence capabilities to improve cybersecurity”, *IEEE Access*, vol. 8, 2020, pp. 23817–23837, DOI [10.1109/access.2020.2968045](https://doi.org/10.1109/access.2020.2968045)
- [7] C. Rudin, C. Chen, Z. Chen, H. Huang, L. Semenova, and C. Zhong, “Interpretable machine learning: Fundamental principles and 10 grand challenges”, 2021, DOI [10.48550/ARXIV.2103.11251](https://doi.org/10.48550/ARXIV.2103.11251)
- [8] F. Doshi-Velez and B. Kim, “Towards a rigorous science of interpretable machine learning”, 2017, DOI [10.48550/ARXIV.1702.08608](https://doi.org/10.48550/ARXIV.1702.08608)
- [9] D. V. Carvalho, E. M. Pereira, and J. S. Cardoso, “Machine learning interpretability: A survey on methods and metrics”, *Electronics*, vol. 8, July 2019, p. 832, DOI [10.3390/electronics8080832](https://doi.org/10.3390/electronics8080832)
- [10] C. Xu, Z. Liao, C. Li, X. Zhou, and R. Xie, “Review on interpretable machine learning in smart grid”, *Energies*, vol. 15, June 2022, p. 4427, DOI [10.3390/en15124427](https://doi.org/10.3390/en15124427)
- [11] D. Watson and L. Floridi, “The explanation game: A formal framework for interpretable machine learning”, *SSRN Electronic Journal*, 2019, DOI [10.2139/ssrn.3509737](https://doi.org/10.2139/ssrn.3509737)
- [12] N. Pachhala, S. Jothilakshmi, and B. P. Battula, “A comprehensive survey on identification of malware types and malware classification using machine learning techniques”, 2021 2nd International Conference on Smart Electronics and Communication (ICOSEC), October 2021, DOI [10.1109/icosec51865.2021.9591763](https://doi.org/10.1109/icosec51865.2021.9591763)
- [13] A. Abusitta, M. Q. Li, and B. C. Fung, “Malware classification and composition analysis: A survey of recent developments”, *Journal of Information Security and Applications*, vol. 59, June 2021, p. 102828, DOI [10.1016/j.jisa.2021.102828](https://doi.org/10.1016/j.jisa.2021.102828)
- [14] G. Iadarola, F. Martinelli, F. Mercaldo, and A. Santone, “Towards an interpretable deep learning model for mobile malware detection and family identification”, *Computers and Security*, vol. 105, June 2021, p. 102198, DOI [10.1016/j.cose.2021.102198](https://doi.org/10.1016/j.cose.2021.102198)
- [15] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, “Grad-cam: Visual explanations from deep networks via gradient-based localization”, 2016, DOI [10.48550/ARXIV.1610.02391](https://doi.org/10.48550/ARXIV.1610.02391)
- [16] G. Iadarola, R. Casolare, F. Martinelli, F. Mercaldo, C. Peluso, and A. Santone, “A semi-automated explainability-driven approach for malware analysis through deep learning”, 2021 International Joint Conference on Neural Networks (IJCNN), July 2021, DOI [10.1109/ijcnn52387.2021.9533803](https://doi.org/10.1109/ijcnn52387.2021.9533803)
- [17] L. Wang, H. Wang, R. He, R. Tao, G. Meng, X. Luo, and X. Liu, “Malradar: Demystifying android malware in the new era”, *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, May 2022, pp. 1–27, DOI [10.1145/3530906](https://doi.org/10.1145/3530906)

- [18] I. El Naqa and M. J. Murphy, “What is machine learning?”, pp. 3–11. Springer International Publishing, 2015
- [19] E. Brynjolfsson and T. Mitchell, “What can machine learning do? workforce implications”, *Science*, vol. 358, December 2017, pp. 1530–1534, DOI [10.1126/science.aap8062](https://doi.org/10.1126/science.aap8062)
- [20] J. Vamathevan, D. Clark, P. Czodrowski, I. Dunham, E. Ferran, G. Lee, B. Li, A. Madabhushi, P. Shah, M. Spitzer, and S. Zhao, “Applications of machine learning in drug discovery and development”, *Nature Reviews Drug Discovery*, vol. 18, April 2019, pp. 463–477, DOI [10.1038/s41573-019-0024-5](https://doi.org/10.1038/s41573-019-0024-5)
- [21] P. Gogas and T. Papadimitriou, “Machine learning in economics and finance”, *Computational Economics*, vol. 57, January 2021, pp. 1–4, DOI [10.1007/s10614-021-10094-w](https://doi.org/10.1007/s10614-021-10094-w)
- [22] G. M. and S. C. Sethuraman, “A comprehensive survey on deep learning based malware detection techniques”, *Computer Science Review*, vol. 47, February 2023, p. 100529, DOI [10.1016/j.cosrev.2022.100529](https://doi.org/10.1016/j.cosrev.2022.100529)
- [23] W. Peng, X. Kong, G. Peng, X. Li, and Z. Wang, “Network intrusion detection based on deep learning”, 2019 International Conference on Communications, Information System and Computer Engineering (CISCE), July 2019, DOI [10.1109/cisce.2019.00102](https://doi.org/10.1109/cisce.2019.00102)
- [24] M. Zolanvari, M. A. Teixeira, L. Gupta, K. M. Khan, and R. Jain, “Machine learning-based network vulnerability analysis of industrial internet of things”, *IEEE Internet of Things Journal*, vol. 6, August 2019, pp. 6822–6834, DOI [10.1109/jiot.2019.2912022](https://doi.org/10.1109/jiot.2019.2912022)
- [25] M. F. Ansari, B. Dash, P. Sharma, and N. Yathiraju, “The impact and limitations of artificial intelligence in cybersecurity: A literature review”, *International Journal of Advanced Research in Computer and Communication Engineering*, 2022, DOI [10.48047/ecb/2023.12.8.717](https://doi.org/10.48047/ecb/2023.12.8.717)
- [26] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, “The limitations of deep learning in adversarial settings”, 2016 IEEE European Symposium on Security and Privacy (EuroS&P), March 2016, DOI [10.1109/eurosp.2016.36](https://doi.org/10.1109/eurosp.2016.36)
- [27] G. Rjoub, J. Bentahar, O. A. Wahab, R. Mizouni, A. Song, R. Cohen, H. Otrok, and A. Mourad, “A survey on explainable artificial intelligence for cybersecurity”, 2023, DOI [10.48550/ARXIV.2303.12942](https://doi.org/10.48550/ARXIV.2303.12942)
- [28] M. E. Celebi, “Unsupervised learning algorithms”, Springer International Publishing AG, 2016, ISBN: 9783319242118. Description based on publisher supplied metadata and other sources.
- [29] N. Priyanka and D. Kumar, “Decision tree classifier: a detailed survey”, *International Journal of Information and Decision Sciences*, vol. 12, no. 3, 2020, p. 246, DOI [10.1504/ijids.2020.108141](https://doi.org/10.1504/ijids.2020.108141)
- [30] W. S. Noble, “What is a support vector machine?”, *Nature Biotechnology*, vol. 24, December 2006, pp. 1565–1567, DOI [10.1038/nbt1206-1565](https://doi.org/10.1038/nbt1206-1565)
- [31] D. Berrar, “Bayes theorem and naive bayes classifier”, pp. 403–412. Elsevier, 2019
- [32] A. Parmar, R. Katariya, and V. Patel, “A review on random forest: An ensemble classifier”, pp. 758–763. Springer International Publishing, December 2018
- [33] G. Biau and E. Scornet, “A random forest guided tour”, *TEST*, vol. 25, April 2016, pp. 197–227, DOI [10.1007/s11749-016-0481-7](https://doi.org/10.1007/s11749-016-0481-7)
- [34] J. Han and C. Moraga, “The influence of the sigmoid function parameters on the speed of backpropagation learning”, pp. 195–201. Springer Berlin Heidelberg, 1995
- [35] A. F. Agarap, “Deep learning using rectified linear units (relu)”, 2018, DOI [10.48550/ARXIV.1803.08375](https://doi.org/10.48550/ARXIV.1803.08375)
- [36] K. P. Sinaga and M.-S. Yang, “Unsupervised k-means clustering algorithm”, *IEEE Access*, vol. 8, 2020, pp. 80716–80727, DOI [10.1109/access.2020.2988796](https://doi.org/10.1109/access.2020.2988796)
- [37] M. A. Syakur, B. K. Khotimah, E. M. S. Rochman, and B. D. Satoto, “Integration k-means clustering method and elbow method for identification of the best customer profile cluster”, *IOP Conference Series: Materials Science and Engineering*, vol. 336, April 2018, p. 012017, DOI [10.1088/1757-899x/336/1/012017](https://doi.org/10.1088/1757-899x/336/1/012017)
- [38] M. Ringn  r, “What is principal component analysis?”, *Nature Biotechnology*, vol. 26, March 2008, pp. 303–304, DOI [10.1038/nbt0308-303](https://doi.org/10.1038/nbt0308-303)
- [39] S. Pyatykh, J. Hesser, and L. Zheng, “Image noise level estimation by principal component analysis”, *IEEE Transactions on Image Processing*, vol. 22, February 2013, pp. 687–699, DOI [10.1109/tip.2012.2221728](https://doi.org/10.1109/tip.2012.2221728)

- [40] B. Xiao, “Principal component analysis for feature extraction of image sequence”, 2010 International Conference on Computer and Communication Technologies in Agriculture Engineering, June 2010, DOI [10.1109/cctae.2010.5544358](https://doi.org/10.1109/cctae.2010.5544358)
- [41] J. Fürnkranz and T. Kliegr, “A brief overview of rule learning”, pp. 54–69. Springer International Publishing, 2015
- [42] V. J. Prakash and D. L. M. Nithya, “A survey on semi-supervised learning techniques”, 2014, DOI [10.48550/ARXIV.1402.4645](https://doi.org/10.48550/ARXIV.1402.4645)
- [43] N. N. Pise and P. Kulkarni, “A survey of semi-supervised learning methods”, 2008 International Conference on Computational Intelligence and Security, December 2008, DOI [10.1109/cis.2008.204](https://doi.org/10.1109/cis.2008.204)
- [44] I. Santos, J. Nieves, and P. G. Bringas, “Semi-supervised learning for unknown malware detection”, pp. 415–422. Springer Berlin Heidelberg, 2011
- [45] J. Ran, Y. Ji, and B. Tang, “A semi-supervised learning approach to ieee 802.11 network anomaly detection”, 2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring), April 2019, DOI [10.1109/vtcspring.2019.8746576](https://doi.org/10.1109/vtcspring.2019.8746576)
- [46] S. Fitriani, S. Mandala, and M. A. Murti, “Review of semi-supervised method for intrusion detection system”, 2016 Asia Pacific Conference on Multimedia and Broadcasting (APMediaCast), November 2016, DOI [10.1109/apmediacast.2016.7878168](https://doi.org/10.1109/apmediacast.2016.7878168)
- [47] Y. Li, “Deep reinforcement learning: An overview”, 2017, DOI [10.48550/ARXIV.1701.07274](https://doi.org/10.48550/ARXIV.1701.07274)
- [48] B. Jang, M. Kim, G. Harerimana, and J. W. Kim, “Q-learning algorithms: A comprehensive classification and applications”, IEEE Access, vol. 7, 2019, pp. 133653–133667, DOI [10.1109/access.2019.2941229](https://doi.org/10.1109/access.2019.2941229)
- [49] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, “Deep reinforcement learning: A brief survey”, IEEE Signal Processing Magazine, vol. 34, November 2017, pp. 26–38, DOI [10.1109/msp.2017.2743240](https://doi.org/10.1109/msp.2017.2743240)
- [50] T. T. Nguyen and V. J. Reddi, “Deep reinforcement learning for cyber security”, IEEE Transactions on Neural Networks and Learning Systems, vol. 34, August 2023, pp. 3779–3795, DOI [10.1109/tnnls.2021.3121870](https://doi.org/10.1109/tnnls.2021.3121870)
- [51] R. Sharp, “An introduction to malware”, 2017
- [52] M. Crepaldi, “Automatic malware signature generation”, Master’s thesis, Politecnico di Torino, 2021. <https://webthesis.biblio.polito.it/20400/>
- [53] A. P. Namanya, A. Cullen, I. U. Awan, and J. P. Disso, “The world of malware: An overview”, 2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud), August 2018, DOI [10.1109/ficloud.2018.00067](https://doi.org/10.1109/ficloud.2018.00067)
- [54] P. Mullins, https://www.idc-online.com/technical_references/pdfs/information_technology/Malware%20and%20its%20types.pdf
- [55] N. DuPaul, <https://www.veracode.com/blog/2012/10/common-malware-types-cybersecurity-101>
- [56] S. Ingalls, <https://www.esecurityplanet.com/threats/malware-types/>
- [57] MyraSecurity, <https://www.myrasecurity.com/en/malware/>
- [58] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh, “A survey on heuristic malware detection techniques”, The 5th Conference on Information and Knowledge Technology, May 2013, DOI [10.1109/ikt.2013.6620049](https://doi.org/10.1109/ikt.2013.6620049)
- [59] N. Fox, <https://www.varonis.com/blog/yara-rules>
- [60] P. Arntz, <https://www.malwarebytes.com/blog/news/2017/09/explained-yara-rules#:~:text=YARA%20is%20a%20tool%20that,that%20look%20for%20certain%20characteristics.>
- [61] E. Raff, R. Zak, G. Lopez Munoz, W. Fleming, H. S. Anderson, B. Filar, C. Nicholas, and J. Holt, “Automatic yara rule generation using biclustering”, Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security, November 2020, DOI [10.1145/3411508.3421372](https://doi.org/10.1145/3411508.3421372)
- [62] S. Simon, <https://www.binarydefense.com/what-is-yara-get-to-know-this-malware-research-tool/>
- [63] O. Aslan and R. Samet, “A comprehensive review on malware detection approaches”, IEEE Access, vol. 8, 2020, pp. 6249–6271, DOI [10.1109/access.2019.2963724](https://doi.org/10.1109/access.2019.2963724)
- [64] Z.-U. Rehman, S. N. Khan, K. Muhammad, J. W. Lee, Z. Lv, S. W. Baik, P. A. Shah, K. Awan, and I. Mehmood, “Machine learning-assisted signature and heuristic-based detection of malwares in android devices”, Computers and Electrical Engineering, vol. 69, July 2018, pp. 828–841, DOI [10.1016/j.compeleceng.2017.11.028](https://doi.org/10.1016/j.compeleceng.2017.11.028)

- [65] W. Liu, P. Ren, K. Liu, and H.-x. Duan, “Behavior-based malware analysis and detection”, 2011 First International Workshop on Complexity and Data Mining, September 2011, DOI [10.1109/iwcdm.2011.17](https://doi.org/10.1109/iwcdm.2011.17)
- [66] J. Cloonan, <https://www.infosecurity-magazine.com/opinions/malware-detection-signatures/>
- [67] Y. Ki, E. Kim, and H. K. Kim, “A novel approach to detect malware based on api call sequence analysis”, *International Journal of Distributed Sensor Networks*, vol. 11, June 2015, p. 659101, DOI [10.1155/2015/659101](https://doi.org/10.1155/2015/659101)
- [68] A. Arora, S. Garg, and S. K. Peddoju, “Malware detection using network traffic analysis in android based mobile devices”, 2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies, September 2014, DOI [10.1109/ngmast.2014.57](https://doi.org/10.1109/ngmast.2014.57)
- [69] R. Sihwail, K. Omar, and K. Akram Zainol Ariffin, “An effective memory analysis for malware detection and classification”, *Computers, Materials amp; Continua*, vol. 67, no. 2, 2021, pp. 2301–2320, DOI [10.32604/cmc.2021.014510](https://doi.org/10.32604/cmc.2021.014510)
- [70] D. Bilar, “Opcodes as predictor for malware”, *International Journal of Electronic Security and Digital Forensics*, vol. 1, no. 2, 2007, p. 156, DOI [10.1504/ijesdf.2007.016865](https://doi.org/10.1504/ijesdf.2007.016865)
- [71] Z. Zhao, “A virus detection scheme based on features of control flow graph”, 2011 2nd International Conference on Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC), August 2011, DOI [10.1109/aimsec.2011.6010676](https://doi.org/10.1109/aimsec.2011.6010676)
- [72] M. Sikorski, “Practical malware analysis”, No Starch Press, 2012, ISBN: 9781593274306. Description based on publisher supplied metadata and other sources.
- [73] A. Sharma and S. K. Sahay, “Evolution and detection of polymorphic and metamorphic malwares: A survey”, *International Journal of Computer Applications*, vol. 90, March 2014, pp. 7–11, DOI [10.5120/15544-4098](https://doi.org/10.5120/15544-4098)
- [74] E. Skoudis and L. Zeltser, “Malware”, Prentice Hall series in computer networking and distributed systems, Prentice Hall PTR, 3. pr. ed., 2004, ISBN: 0131014056. Includes bibliographical references and index
- [75] I. You and K. Yim, “Malware obfuscation techniques: A brief survey”, 2010 International Conference on Broadband, Wireless Computing, Communication and Applications, November 2010, DOI [10.1109/bwcca.2010.85](https://doi.org/10.1109/bwcca.2010.85)
- [76] R. Perdisci, A. Lanzi, and W. Lee, “Classification of packed executables for accurate computer virus detection”, *Pattern Recognition Letters*, vol. 29, October 2008, pp. 1941–1946, DOI [10.1016/j.patrec.2008.06.016](https://doi.org/10.1016/j.patrec.2008.06.016)
- [77] C. Molnar, G. Casalicchio, and B. Bischl, “Interpretable machine learning a brief history, state of the art and challenges”, pp. 417–431. Springer International Publishing, 2020
- [78] F. Sperrle, M. El Assady, G. Guo, R. Borgo, D. H. Chau, A. Endert, and D. Keim, “A survey of human centered evaluations in human centered machine learning”, *Computer Graphics Forum*, vol. 40, June 2021, pp. 543–568, DOI [10.1111/cgf.14329](https://doi.org/10.1111/cgf.14329)
- [79] Y. Lin and X. Chang, “Towards interpreting ml-based automated malware detection models: a survey”, 2021, DOI [10.48550/ARXIV.2101.06232](https://doi.org/10.48550/ARXIV.2101.06232)
- [80] A. Silva, T. Killian, I. D. J. Rodriguez, S.-H. Son, and M. Gombolay, “Optimization methods for interpretable differentiable decision trees in reinforcement learning”, 2019, DOI [10.48550/ARXIV.1903.09338](https://doi.org/10.48550/ARXIV.1903.09338)
- [81] L. Wei, L. Liu, J. Qi, and T. Qian, “Rules acquisition of formal decision contexts based on three-way concept lattices”, *Information Sciences*, vol. 516, April 2020, pp. 529–544, DOI [10.1016/j.ins.2019.12.024](https://doi.org/10.1016/j.ins.2019.12.024)
- [82] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate”, 2014, DOI [10.48550/ARXIV.1409.0473](https://doi.org/10.48550/ARXIV.1409.0473)
- [83] S. Cristina, <https://machinelearningmastery.com/the-attention-mechanism-from-scratch>
- [84] E. Lindwurm, <https://towardsdatascience.com/indepth-layer-wise-relevance-propagation-340f9>
- [85] X. Liu, X. Wang, and S. Matwin, “Improving the interpretability of deep neural networks with knowledge distillation”, 2018 IEEE International Conference on Data Mining Workshops (ICDMW), November 2018, DOI [10.1109/icdmw.2018.00132](https://doi.org/10.1109/icdmw.2018.00132)
- [86] A. Mills, T. Spyridopoulos, and P. Legg, “Efficient and interpretable real-time malware detection using random-forest”, 2019 International Conference on Cyber Situational Awareness, Data Analytics And Assessment (Cyber SA), June 2019, DOI [10.1109/cybersa.2019.8899533](https://doi.org/10.1109/cybersa.2019.8899533)

- [87] M. M. Alani and A. I. Awad, “Paired: An explainable lightweight android malware detection system”, *IEEE Access*, vol. 10, 2022, pp. 73214–73228, DOI [10.1109/access.2022.3189645](https://doi.org/10.1109/access.2022.3189645)
- [88] A. H. Lashkari, A. F. A. Kadir, L. Taheri, and A. A. Ghorbani, “Toward developing a systematic approach to generate benchmark android malware datasets and classification”, 2018 International Carnahan Conference on Security Technology (ICCST), October 2018, DOI [10.1109/ccst.2018.8585560](https://doi.org/10.1109/ccst.2018.8585560)
- [89] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision”, 2015, DOI [10.48550/ARXIV.1512.00567](https://doi.org/10.48550/ARXIV.1512.00567)