POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Malware Family Classification with Semi-Supervised Learning

Supervisor
prof. Antonio Lioy
prof. Andrea Atzeni

Candidate
Maria Letizia Colangelo

Academic Year 2022-2023

Summary

In recent years, the spread of malware has increased exponentially, posing a significant challenge for cybersecurity experts. When facing with the constantly evolving world of unknown threats, including zero-day attacks, traditional signature-based approaches for malware detection have been shown to be insufficient. Furthermore, adversaries are adapting by modifying their malicious code, which reduces the efficacy of signature-based detection. As a solution to these problems, supervised machine learning models have been used to develop behaviour-based malware detection systems. These systems are employed to inspect the code in order to identify any malicious or potentially harmful actions performed by that code.

Supervised learning shows promising results in effectively detecting malicious code, but it is significantly limited by the considerable amount of manual effort required for labelling both malware and benign instances. Unsupervised learning methods, while reducing the labelling effort, often yield lower accuracy, especially in complex tasks such as malware detection. A potential resolution lies in the adoption of a hybrid approach, called semi-supervised learning (SsL), which combines labelled and unlabelled data. SsL has the potential to enhance algorithm performance by taking advantage of a limited number of labelled samples and a large number of unlabelled data. However, whether SsL offers advantages over supervised learning in the context of malware detection remains unclear in the current state-of-the-art, given the absence of a rigorous evaluation.

This thesis focused on semi-supervised learning for malware family classification, aiming to explore its benefits compared to the supervised approach. In particular, this research objective is to identify the circumstances under which unlabelled data can be effectively employed to enhance detection accuracy. To achieve this, I employed an evaluation framework from the current state-of-the-art that allows to assess rigorously the benefits of Ssl.

The research begins by conducting an extensive review of the recent literature, examining the various methods proposed in the state-of-the-art for malware detection, with a particular focus on the image-based approach. Furthermore, state-of-the-art methods for malware detection and SsL learning are explained in detail, with a comparative analysis of their relative advantages and drawbacks.

To determine which malware dataset is most suitable for my research objectives, a meticulous evaluation of existing malware sample datasets is performed. This assessment outlines both the strengths and weaknesses of each dataset, offering a comprehensive overview crucial for guiding the selection process. Moreover, this process involves a detailed description of the chosen dataset, MaleVis, including sample distribution, sample features, and available labels.

Following this, I focused on the development of my research methodology. I employed the evaluation framework with a range of different traditional machine learning methods. These algorithms include: Random Forest, K-Nearest Neighbors, Support Vector Machine, Gradient

Boosting. In combinations with these algorithms, I followed two approaches: the direct employment of raw pixel data and the extraction of texture feature descriptors, including but not limited to Local Binary Pattern (LBP), Histogram of Oriented Gradients (HOG), Principal Component Analysis (PCA), and Gray-Level Co-Occurrence Matrix (GLCM). Additionally, I further extended the evaluation framework to encompass the domain of Convolutional Neural Networks.

According to the analysis, it is noteworthy how these algorithms exhibit various behaviours when faced with different challenges such as dataset distribution, feature input, and the proportion of labelled data.

Contents

Chapter 1

Introduction

Malware, short for "malicious software", has experienced an exponential increase over the past few years, although its roots tracing back to the early days of computing. The origins of malware can be found in the 1970s, when viruses began to appear on dedicated networks such as the ARPANET, a precursor to the Internet. One of the earliest known malware instances, the Creeper virus, displayed the message "I'm the creeper, catch me if you can!". It represents the beginning of what would eventually become a complex and continuously evolving threat landscape.

The rapid diffusion of technology directly contributed to the expansion of virus distribution. Furthermore, the widespread use of computers has facilitated the evolution of malware into more intricate and sophisticated forms. This increase in both the distribution and the sophistication of viruses has posed significant challenges in the field of cybersecurity. The 2023 SonicWall Cyber Threat Report showed that 5.5 billion malware attacks were recorded in 2022, a 2% increase year over year. While the increase is small, it is being fueled by massive growth in two areas. In 2022, cryptojacking rose 43% and IoT malware jumped 87%. Together, these increases were more than enough to offset a 21% drop in global ransomware volume, pushing overall malware trends into positive territory for the first time since 2018 [1]. This trend is shown in Fig 1.1.

As cyber threats continue to evolve, cybersecurity must adapt and innovate to protect organisation's systems and networks. Hence, malware detection is essential to prevent malware infections and mitigate the potential damage they can cause. Signature-based method for malware detection is the traditional approach that uses a database of known malware signatures to identify and remove malicious software. This is effective at detecting known malware threats, but it may not be able to detect unknown malware, such as zero-day attacks, which is an attack for which there is no corresponding signature stored in the repository. Additionally, attackers can avoid detection by modifying their malicious code in order to generate new signatures.

On the other hand, behaviour-based malware detection looks for unusual behavior in files or systems that may indicate the presence of malware. This approach is more effective when it comes to detecting unknown threats; however, it may generate more false positives, because legitimate software may perform unusual activities.

To reduce false positives, machine learning is typically involved: in this method, a machine learning model is trained on a large dataset of malware and benign samples to identify patterns of malicious software. In recent times, there has been a significant increase in the use of machine learning to enhance cybersecurity against malware threats. This represents a fundamental change in our approach to defend against these harmful software. Machine learning algorithms can analyse large amounts of data to detect and respond to emerging threats, providing a more proactive and adaptive defence against evolving cyber threats. Machine learning approaches are divided into three categories:

- Supervised learning that uses labelled dataset, where the inputs have already been mapped to the outputs, to train a machine learning model;
- Unsupervised learning that uses unlabelled data and looks for correlation patterns;

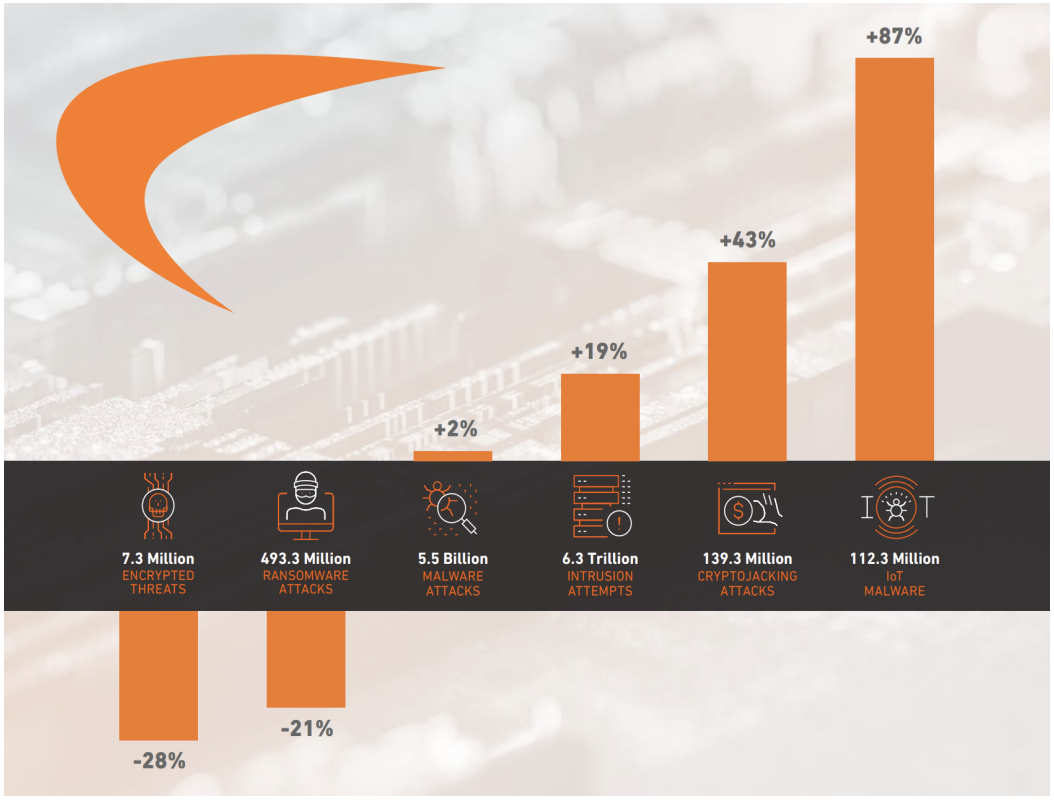

Figure 1.1: 2023 SonicWall Cyber Threat Report [1]

- Semi-supervised learning that is a hybrid approach and combines a small amount of labelled data with a large amount of unlabelled data during training.

The problem with supervised learning for malware detection is the requirement for human experts and specialised tools to distinguish between malware and benign samples; therefore, it is an expensive and time-consuming process. In contrast, while the cybersecurity domain yields a large volume of unlabelled data that can be employed for unsupervised learning techniques (e.g. network traffic data, system logs, endpoints log, etc.), these methods achieve lower accuracy levels. It is possible to overcome the lack of labelled samples by using semi-supervised learning, allowing for more cost-effective and efficient malware classification.

Therefore, the aim of this thesis is to explore the benefits of using unlabelled data for malware family classification. To this end, I employed an evaluation framework previously introduced in a related study. This choice is critical for ensuring a rigorous evaluation of the benefits offered by semi-supervised learning methods, since the framework formalises a set of requirements that are essential for conducting an unambiguous assessment. By systematically applying these requirements, the evaluation process guarantees a robust and reliable comparison of different malware classification techniques.

Fig 1.2 illustrates the project workflow. Starting from the analysis of different malware datasets and considering the corresponding advantages and disadvantages, I opted for the MaleVis dataset, which contains RGB images of both malicious and benign samples. Detailed information on this dataset will be provided in Section 6.5.

Next, choosing an image-based approach, the reasons for which will be discussed later in the Section 7.1, I focused on analysing different image features that could be extracted, and I chose two primary approaches: direct utilisation of raw pixel data and the extraction of texture feature descriptors such as Local Binary Pattern (LBP), Histogram of Oriented Gradients (HOG), Principal Component Analysis (PCA), and Gray-Level Co-Occurrence Matrix (GLCM).

The features mentioned above are then used with a selection of traditional machine learning

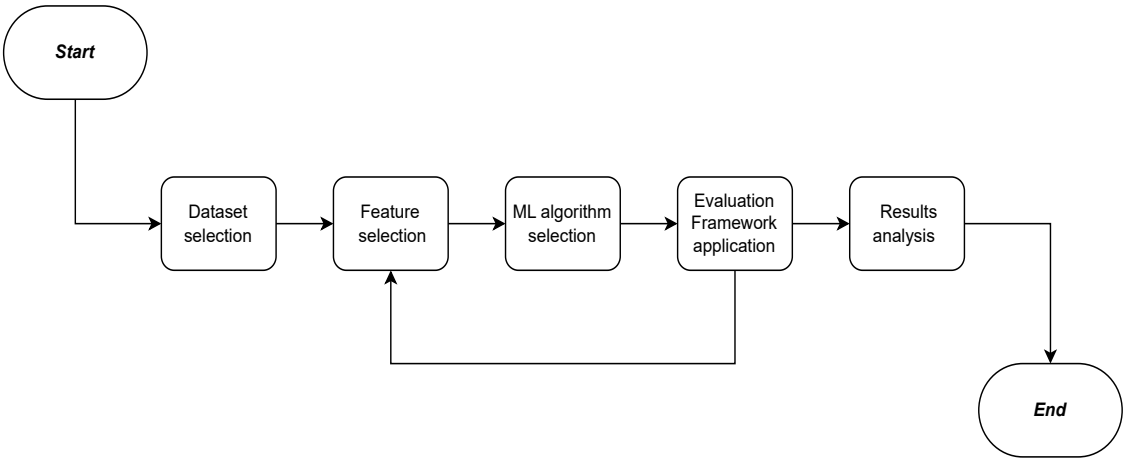

Figure 1.2: Project workflow

methods within the evaluation framework. This results in the generation of outcomes for various feature-classifier combinations.

In addition to employing traditional feature extraction methods, the study leveraged Convolutional Neural Networks to automate the extraction of features directly from images, without the need for manual feature engineering.

This work is structured as follows: The chapter 2 will introduce essential background terms and concepts related to malware, providing a comprehensive understanding of its intent, diverse forms, and the methodologies employed for its detection; the chapter 3 will offer insights into machine learning explaining the core concepts, algorithms, approaches, and its application in the field of cybersecurity, highlighting the current challenges; the chapter 4 will introduce related works, beginning with a comprehensive overview of general approaches to image-based malware classification, as these methods have shown promise in the field of cybersecurity, and then focusing on the works based on semi-supervised learning; the chapter 5 is dedicated to an in-depth exploration of the framework used to assess the benefits of semi-supervised learning; the chapter 6 will describe the commonly used datasets in this task, highlighting both their advantages and limitations, and focusing also on analysing the concept of concept drift associated with outdated datasets; chapter 7 and 8 will be focused on presenting the methodology I implemented, describing the various experiments I carried out and the corresponding results. The algorithms I have employed include both traditional machine learning models (Random Forest, K-Nearest Neighbors, Support Vector Machine, Gradient Boosting) and deep learning models. Conclusions were drawn in the chapter 9, together with outlining the potential future works.

Chapter 2

Background

2.1 Malware

A program that is inserted into a system, usually covertly, with the intent of compromising the confidentiality, integrity, or availability of the victim's data, applications, or operating system [2].

Malware attacks are one of the most common cyber threats nowadays, spanning various industries, including finance, education, and healthcare. It can be distributed through various methods. Emails, websites, software downloads, removable media are common ways employed by attackers to transmit malware. However, it can also be concealed within various file types, including executable files (.exe), images, documents, and archives.

2.1.1 Intent of malware

Malware can be used for different purposes, each designed to carry out specific actions [3]:

- **Intelligence and intrusion**: some malware infiltrates systems to steal private information as emails or passwords;
- **Disruption and extortion**: there is a class of malware that locks users' system to disrupt its normal functioning (it is called ransomware if the objective is financial gain through extortion);
- **Destruction or vandalism**: certain malware destroys computer systems to damage the network infrastructure;
- **Steal computer resources**: malware leverages the computing power of infected systems to run botnets, cryptomining programs (cryptojacking), or send spam emails;
- **Monetary gain**: malware can be used to acquire and consequently sell the organization's intellectual property on the dark web.

2.1.2 Malware forms

There are different types of malware attacks which are categorised differently by various antivirus vendors (e.g. Kaspersky). This classification helps to understand the different nature of malware threats, each one with its own behaviour and characteristics [4]:

- **Adware**: also known as advertisement-supported software, it displays unwanted advertising on the screen. It can also redirect search results to advertising websites and capture user data for potential sale to advertisers without the user's consent. However not all adware is malicious, but some is legitimate;

- **Spyware**: it hides itself on devices monitoring activities and stealing private information;
- **Ransomware and crypto-malware**: ransomware restricts user access to their systems or data until a ransom is paid. Crypto-malware is a specific subset of ransomware that encrypts user files and demands payment within a specified timeframe, often in cryptocurrencies;
- **Trojans**: it looks like legitimate software leading users to run malicious software on their computers. Once installed in user's device, attackers leverage it to delete, copy or modify user data;
- **Worms**: malware that can self-replicate itself to infect other computers without any human action, exploiting system vulnerabilities. It is used to perform various actions e.g. steal information, delete files etc;
- **Virus**: piece of code embedded within an application and executed when the app is running. Once infiltrating a network, it can be used to steal information, perform DDoS attacks, or execute ransomware attacks.

2.2 Malware detection

Malware detection refers to the process of detecting the presence of malware on a host system or of distinguishing whether a specific program is malicious or benign [5]. Malware detection can use two distinct types of analysis: static and dynamic.

2.2.1 Static analysis

Static analysis investigates the code searching for malicious behaviour without executing it, as in signature-based malware detection. This approach explores all possible execution paths of the program. Many tools can be used to perform static analysis: debugger, disassembler, decompiler, and source code analyser. Methods that are used in performing static analysis include File Format Inspection, String Extraction, Fingerprinting, AV scanning, and Disassembly [6]. The problem with static analysis is that attackers can use obfuscation techniques to avoid being detected.

2.2.2 Dynamic analysis

Dynamic analysis runs executable file in a protected virtual environment to monitor the program behaviour and its interaction with the system. It is robust against obfuscation techniques, but it has the limitation that only one execution of the program at a particular instant of time is analysed, unlike static analysis. Moreover malware can attempt to detect whether it is run within a sandbox or virtual machine or actively attack the emulator. Several anti-emulation techniques exist, which check for certain low-level processor features (e.g., undocumented instructions) or timings. Once an executable has determined that it is being emulated, it can terminate execution without performing any malicious actions [5].

2.3 Malware detection approaches

Basic approaches can detect and classify known threats, and they include signature-based approach, behavioural-based approach, heuristic approach, and model-checking approach. However, especially signature-based approach fails to detect unknown malware. On the contrary, behavioural-based and heuristic approaches demonstrate better results in addressing previously unidentified threats. Additionally, there are other more sophisticated approaches that involve Machine Learning that can identify some known and unknown malware.

2.3.1 Signature-based approach

This technique is widely used by antivirus companies. In computer security, a signature is a specific pattern that allows cybersecurity technologies to recognize malicious threats, such as a byte sequence in network traffic or known malicious instruction sequences used by families of malware [7].

Cybersecurity companies maintain large databases of known malware signatures. These databases are frequently updated. When a software needs to be classified as malware or not, each file in this database is scanned to compare the signatures.

The advantage of this technique is that it is fast and effective, but it can only identify known threats. Signature generation can be automated or generated manually by analysts. There are different techniques to generate a signature [8]:

- **String Scanning**: this technique compares the byte sequence in the analysed file with the byte sequences already stored in the database. Since malware nowadays has widely spread, these databases are rapidly increasing in size. String signature is one solution to this problem and it has been largely used by anti-virus companies over past years;
- **Top-and-Tail Scanning**: only the top and the end of the file are taken and the signature is generated. This technique is more efficient because it is faster and relies on the fact that often the virus code is at the beginning or at the end of the files. However it can produce false negatives;
- **Entry Point Scanning**: the entry-point is the point in the program where its execution begins. Malware usually changes the entry point so that malicious code is executed before the real code. This is another faster technique;
- **Integrity Checking (Hash Signatures)**: this technique generates periodically cryptographic checksum (e.g. MD5 and SHA-256) to look for possible changes that may be caused by malware.

2.3.2 Behaviour-based approach

Threat actors can use obfuscation to bypass static detection techniques.

Behaviour-based detection utilises the behaviour information of the malware during its execution as the detection basis, which can get rid of the reliance on the feature of malware file itself. A malware can produce a new variation with different features through shell code, polymorphism, and metamorphism, but its behaviour is still the same as the original one. Therefore, behaviour-based malware detection will not be affected by shell code, Polymorphism and Metamorphism, and therefore can detect new malware [9]. The mechanisms of Polymorphism and Metamorphism are detailed in Section 2.4.1.

However, some malware binaries do not run properly in protected environments, so this approach can generate false negatives. Behaviours are extracted with the following procedures [8]:

- **Automatic analysis by using sandbox**: sandbox is an isolated and controlled environment where the malicious code can be executed to analyse its behaviour without harming the host system;
- **Monitoring of system calls**: tracking system calls made by a process provides insight into its interaction with the operating system. The detection of suspicious system calls allows the identification of potentially malicious intentions;
- **Monitoring of file changes**: this can help in identifying actions such as file creation, modification, or deletion that may indicate malicious behaviour;
- **Comparison of registry snapshots**: by comparing these snapshots, it is possible to detect any unauthorised or malicious changes to the registry, crucial for identifying threats;

- **Monitoring network activities**: analysing network activities enables the detection of anomalies within network traffic that could point to threats;
- **Process monitoring**: by observing the activities of running processes it is possible to identify suspicious processes or their interactions with the system.

In behaviour-based detection, first, behaviours are determined by using one of the technique used above and the dataset is created by subtracting the features using datamining. Then, specific features from the dataset are obtained and classification done by using ML algorithms [8].

2.3.3 Heuristic-based approach

It is a complex detection method which uses experiences and different techniques such as rules and ML techniques. Heuristic-based schema can use both strings and some behaviours to generate rules, and based on that rules it generates signature. It uses API calls, Control Flow Graph (CFG), n-grams, Opcode, and hybrid features when generates a signature [8]. This approach can detect different forms of known and also unknown malware, but not all of the new generation of them. Moreover, it is prone to a high false positive rate.

2.3.4 Model checking-based approach

In this detection approach, malware behaviours are manually extracted and behaviour groups are coded using linear temporal logic (LTL) to display a specific feature. Program behaviours are created by looking at the flow relationship of one or more system calls and define behaviours by using properties such as hiding, spreading, and injecting [8]. As in the previous approach, it can detect some new malware, but not all new generation of them.

2.4 Detection evasion techniques

Malware actors use various evasion techniques to hide themselves in order to avoid being detected. There are many evasion strategies that can be classified into two categories:

2.4.1 Static analysis evasion techniques

The most used techniques targeting static analysis are [10]:

- **Code obfuscation**: obfuscation is when malware deliberately tries to obscure its true intent to potential victims, and/or attempts to hide portions of code from malware researchers performing analysis [11]. Common obfuscation techniques are as follows:
  - **Dead-Code Insertion**: obfuscation technique whereby byte code sequences are inserted into the binary as to not affect the functionality of the program;
  - **Registry Reassignment**: obfuscation technique which swaps unused registers or memory variables with those currently used by the program;
  - **Instruction Substitution**: it replaces instructions with equivalent ones;
  - **Code Transposition**: obfuscation technique which utilizes conditional or unconditional jmp statements to reorder single or blocks of instructions.
- **Encryption**: it was the first technique used to evade static code analysis and traditional signature-based detection approach. The body is usually XORed with a key to make it hard to detect. It infects the system by decrypting itself using a decryption algorithm and a key, after that it again encrypts itself by the encryption algorithm and generates a new key for another variant to avoid the detection mechanism. However, encrypted malware can be detected by analyzing the encryption/decryption algorithm because the decryption/encryption algorithm does not change, e.g., CASCADE was the first encrypted malware [12];

- **Compression**: compression represents an additional level of obfuscation on top of a possible decryption routine and other forms of obfuscation. A Packer is defined as a utility which enacts some form of compression to the executable, either to reduce files size to avoid entropy analysis or introduce a layer of obfuscation to the PE header;
- **Oligomorphism**: since in the encrypted malware the same decryptor is used everytime making it easier to detect, oligomorphism can be used to overcome this limitation. In this malware, decryptors are mutated in the malware variant, i.e., it provides set of obfuscated decryptors [12]. The main limitation is that this set of decryptors is finite.
- **Polymorphism**: it denotes the ability of malicious code to change its appearance or structure while preserving its core functionality. Polymorphic malware is similar to oligomorphic malware, but it can generate millions of decryptors by mutating the instructions in the variant of the malware to evade the signature matching detection technique. In this malware, the mutation engine generates an encryption algorithm and a corresponding decryption algorithm, then the malware code and the mutation engine both get encrypted to generate a new variant of a particular malware. To create this malware obfuscation techniques are used (e.g. dead code insertion, transposition, registry reassignment etc.) Although it can generate a large number of different decryptors, still traditional signature based detection mechanisms can be applied to detect the polymorphic malware by simply finding the original program using the emulation techniques [12].
- **Metamorphism**: metamorphism is a sophisticated technique where the malicious code undergoes complete structural alterations while maintaining its original functionality. In Metamorphic malware instead of mutating the decryptors the malware body is mutated itself (i.e., body-polymorphic) to create a new variant without changing its actions to evade the detection [12]. This malware is very hard to detect by traditional signature scanning techniques.
- **Metamorphic Engine**: a Metamorphic Engine is responsible for the obfuscation and reconstruction of the binary so that the file can remain operational.

2.4.2 Dynamic analysis evasion techniques

There are two modes of dynamic analysis: manual and automated. The manual analysis is when the analysis process is performed by an expert human with the help of a debugger. Automated, on the other hand, is the process that is the similar analysis that is performed automatically by a machine or software, also known as sandbox [13]. On the basis of this, the following can be distinguished:

- **Manual dynamic analysis evasion techniques**: These measures include approaches such as detecting the presence of analysis tools on the system (e.g., Wireshark, TCPDump) or detecting virtual machines as a sign of analysis environment, but, the majority of manual analysis evasion techniques are targeted toward debuggers, which are the primary tools of manual dynamic analysis [13].
  - **Fingerprinting**: most common evasion tactic regarding both manual and automated analysis, but the techniques are different. It aims to detect signs that attest to the presence of an analysis environment or debuggers;
  - **Traps**: codes that when traversed or stepped through by debugger, give specific information if the debugger is present;
  - **Debugger Specific**: tactic that exploits vulnerabilities that are exclusive to a specific debugger;
  - **Targeted**: one of the most advanced tactics, the malware encrypts its malicious payload with a specific encryption key. This encryption key, however, is chosen to be an attribute or variable that could be found only on its target system;
  - **Control Flow Manipulation**: malware exploits the control flow, relying on callbacks, enumeration functions etc.;

  - **Lockout Evasion**: malware continues to execute by interfering with the work of the debugger without searching for its presence;
  - **Fileless Malware**: malware that requires no file, it exploits vulnerability of the target system and injects malicious code directly into the memory; so since dynamic analysis monitors behaviour during execution, it's harder to analyse the malware because there is no executable.

- **Automated dynamic analysis evasion techniques**: Manual analysis is effective, but analysing a large number of malware samples is expensive. Here, the goal is to detect malicious program in a controlled and contained testing environment called sandbox. The techniques are as follows, similar to the manual analysis [13]:
  - **Fingerprinting**: differently the manual analysis scenario, here the concept is to detect signs of the presence of virtual/emulated machine;
  - **Reverse Turing Test**: tactic that looks for human interaction with the system; that is because a sandbox is an automated machine with no interaction, so if there is no interaction malware assumes that it is in a sandbox;
  - **Targeted**: the goal here for the malware is to detect the target environment, not the sandbox as in the previous ones;
  - **Stalling**: based on the concept that sandbox allocates a limited amount of time to analyse each sample, malware postpones its malicious activity to the post-analysis stage;
  - **Trigger-Based**: malware waits for triggers; as in the previous case the concept here is to wait to perform malicious activity out of the sandbox;
  - **Fileless Malware**: as in the manual case, fileless malware can evade also automated analysis.

2.5 Malware Fundamentals: Summary

The chapter focuses on the foundational understanding of malware, elucidating its various forms and the underlying intentions. It then delves into the methodologies of malware analysis, detailing both static and dynamic approaches, along with malware detection strategies, from traditional signature-based methods to behavioural analysis and the evasion tactics employed by malware actors.

This comprehensive review of the background, analysis, detection, and evasion techniques associated with malware serves as a foundational knowledge base for the focus on malware family classification within this thesis.

Chapter 3

Machine Learning and Cybersecurity

The application of machine learning in cybersecurity is increasing over time. There is a general belief among cybersecurity experts that AI-powered antimalware tools will help detect modern malware attacks and improve scanning engines. Evidence of this belief is the number of studies published in the last few years on malware detection techniques that leverage machine learning [14]. There are many advantages in that [15]:

- **Rapidly synthesize large volumes of data**: analysts have to deal with a large amount of data, increasing fast, very hard to manually process in real-time. ML helps to quickly analyse big data;
- **Activate expert intelligence at scale**: cycles of training helps to learn from data decreasing false positive and allowing models to learn and enforce ground truth defined by experts;
- **Automate repetitive, manual tasks**: letting ML to deal with simple repetitive task allow experts to focus on more complex problem;
- **Augment analyst efficiency**: ML gives real-time data to analysts, enabling them to focus the resources on critical vulnerabilities and analyse time-sensitive detections.

Machine learning has been used in a growing set of use cases in cybersecurity (e.g. Malware Detection, Intrusion Detection System, Phishing Detection, etc.)

3.1 Machine Learning

Machine learning is a branch of artificial intelligence (AI) and computer science which focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving its accuracy [16].

It is a field that is growing exponentially in many applications, including cybersecurity. Recent progress in ML has been driven both by the development of new learning algorithms theory, and by the ongoing explosion in the availability of vast amount of data (often referred to as "big data") and low-cost computation [17]. One of the most common uses in cybersecurity is malware detection and classification. The typical supervised machine learning algorithm consists of roughly three components [18]:

1. **A decision process**: A machine learning algorithm will detect a pattern based on some input data. Typically it is used to perform classification or prediction tasks;

2. **An error function**: This function is used to evaluate how good or bad the prediction of the algorithm is;
3. **An updating or optimization process**: Based on the difference between the ground truth and the model prediction, weights are adjusted, and the model retrained until a certain threshold is met.

3.1.1 Machine learning approaches

Machine learning algorithms can be classified into four categories:

- **Supervised learning**: It uses a labelled dataset, where the inputs have already been mapped to the outputs, to train a machine learning model in order to make predictions or to classify data. This is used in many real-world scenarios, as in Spam detection or Fraud detection;
- **Unsupervised learning**: Unlike the previous one, it uses unlabelled data and looks for correlation patterns, e.g. anomaly detection;
- **Semi-supervised learning**: It is a hybrid approach between the previous ones and combines a small amount of labelled data with a large amount of unlabelled data during training. It helps reducing the cost of labelling large amount of data;
- **Reinforcement learning**: Similarly to the supervised one, it is not trained using labelled samples. It is instead trained with a trial and error system, offering feedbacks to the algorithm to learn.

3.1.2 Machine learning algorithms

Common algorithms used include [16]:

- **Neural networks**: models composed of artificial neurons, inspired by the simplification of a biological neural network. These neurons are organised in layers, including an input layer, more hidden layers and one output layer. All nodes are connected and have associated weights and thresholds. If the output of any node exceeds a certain threshold, the node is activated and it sends data to the next level. Applications include natural language translation, image recognition, speech recognition, and image creation;
- **Linear regression**: algorithm used to predict a value of a variable based on another variable. Assuming a linear relationship between the independent variable and the dependent variable, its objective is to find a line that represents this relationship;
- **Logistic regression**: based on a dataset of independent variables, the model makes predictions for categorical response variables. It is commonly used in classification tasks, such as spam detection, for dividing data into different categories;
- **Clustering**: algorithms that search for patterns in data. The data are grouped into clusters on the basis of how similar they are.

3.2 Semi-Supervised Learning

Historically, supervised learning and unsupervised learning have been the two fundamental types of task in machine learning. Alongside these, SsL has emerged as a powerful approach that makes use of both labelled and unlabelled data for training models.

SsL operates in scenarios where the labelled data is sparse or expensive to acquire compared to the availability of abundant, unlabelled data. This scenario is common in many real-world

applications, such as anomaly detection, threat detection, and overall security measures. Many recent research papers and publications have prominently used SsL for malware detection and classification. These works will be examined in more detail in Chapter 4.

The motivation behind SsL lies in the belief that unlabelled data carry valuable information about the underlying data distribution, which, when properly utilised, can improve the model's generalisation and predictive accuracy. Since unlabelled data carry less information than labelled data, they are required in large amounts in order to increase prediction accuracy significantly. This implies the need for fast and efficient SsL algorithms [19].

Semi-supervised learning is based on several fundamental assumptions regarding the data [19]:

- **Cluster Assumption:** It assumes that data can be divided in clusters and that points within the same cluster are likely to belong to the same class;
- **Manifold Assumption:** It assumes that high-dimensional data lie on a low-dimensional manifold. That means that the learning algorithm can work effectively within a space that matches that lower dimension. This way, it can bypass the challenges posed by the curse of high dimensionality;
- **Smoothness Assumption:** It suggests that label changes across the data distribution should occur smoothly. In simpler terms, if two points are situated closely within a densely populated region, their corresponding output labels should also be closely related;
- **Low-Density Separation Assumption:** This assumption extends the Cluster Assumption by considering that decision boundaries between different classes should lie in low-density regions, helping to distinguish between different categories more effectively.

In the context of SsL, two key learning paradigms are often distinguished: transductive learning and inductive learning. These terms refer to different approaches to exploiting labelled and unlabelled data for model training and prediction.

- **Inductive learning**: This is what we commonly refer to as Supervised Learning. In this scenario, the model's training is performed on a known-labelled dataset, allowing it to learn underlying patterns and associations between features and labels. Once trained, the model applies this learnt knowledge to make predictions on new, unseen data instances. There is no need to retrain the model when new data is added;
- **Transductive learning**: It aims at classifying the unlabelled input data by exploiting the information derived from labelled data. However, in this methodology, both the training and testing datasets are encountered during model training. Transductive learning builds a model that fits the training and testing data points that have already been observed. Transduction does not build a predictive model. If a new data point is added to the testing dataset, the algorithm must be re-run from the start, train the model, and then use it to predict the labels.

3.2.1 SsL approaches

There are different approaches in SsL:

Self-Training

It starts with the training process using exclusively labelled data. After the initial training, the trained model is used to predict labels for unlabelled data. These predicted labels are treated as pseudo-labels and appended to the labelled dataset. Subsequently, the supervised method is retrained, incorporating its own predictions as supplementary labelled data points. The process iterates by using the updated model to predict labels for more unlabelled data, generating additional pseudo-labelled instances. These instances are added to the labelled dataset, and the model undergoes further retraining. This iterative loop continues for several cycles.

Co-Training

It involves training a model using multiple views of the data. In the context of Co-Training, the concept of "views" refers to different sets of features or perspectives that offer additional information about each data instance. They are considered sufficient on their own to make accurate predictions and are assumed to offer unique, non-redundant information about the data instances.

It begins with a small set of labelled data instances that are labelled with respect to both views or modalities. A larger pool of unlabelled instances is available for both views. Two separate classifiers or models are trained independently using different views of the data, initially with the labelled dataset. During each iteration, the models make predictions on the unlabelled data, and instances where both models agree confidently are considered trustworthy pseudo-labelled data. The agreed-upon pseudo-labelled instances are added to the labelled dataset for both views, expanding the training set. The models are retrained using the augmented labelled dataset, and the process iterates, with each model learning from the predictions of the other model. The agreement between models reinforces the confidence in pseudo-labelled instances.

Label Propagation

Label propagation is a semi-supervised learning technique that involves propagating labelled information across a graph built from both labelled and unlabelled data. In particular, a graph is constructed where each data point represents a node in the graph. Edges in this graph correspond to similarities or relationships between data points. These similarities can be calculated using various metrics, such as distance measures or affinity scores, establishing connections between nodes. The labelled information is diffused or propagated throughout the graph by iteratively updating the labels of the unlabelled nodes based on the labels of their neighbouring nodes. This propagation occurs through the edges of the graph. As the labels propagate, each node aggregates information from its neighbours, leading to agreement on the labels across the graph. This consensus is reached through iterative label updates.

Minimum Entropy Regularisation

It focuses on high-entropy data points, where the model is uncertain and needs more information, encouraging the model to make confident predictions in order to improve generalisation and overall accuracy.

Consistency Regularisation

It enforces that model predictions on the same data point remain consistent despite minor perturbations or data augmentations. It improves model generalisation better and prevents overfitting.

3.2.2 SsL Challenges

In the realm of SsL, several challenges emerge that impact the effectiveness of these methods in real-world applications. SsL algorithms often make assumptions about data distributions, assuming characteristics such as smoothness or separability of clusters. However, these assumptions may not always hold true in diverse, complex real-world datasets, compromising the applicability of SsL techniques.

Another critical challenge lies in the potential propagation of errors and biases present in the initial labelled data. These inaccuracies can be amplified throughout the SsL iterations, significantly influencing the model predictions. Ensuring the quality and reliability of labelled data becomes paramount to prevent the propagation of biases.

Moreover, evaluating the performance of SsL methods poses a challenge. Conventional evaluation metrics might not adequately capture the enhancements derived from the incorporation

of unlabelled data. The improvements in model performance due to SsL might not be effectively measured or reflected by standard evaluation methodologies.

3.3 Active Learning

In both Active Learning and SsL, unlabelled data is used to improve the performance of the model in situations with a limited amount of labelled data. The key idea behind active learning is that a machine learning algorithm can achieve greater accuracy with fewer training labels if it is allowed to choose the data from which it learns. An active learner may pose queries, usually in the form of unlabelled data instances to be labelled by an oracle (e.g., a human annotator) [20]. The process of active learning can be described as follows:

1. Initial Training: It begins with a small set of labeled data;
2. Query Strategy: The algorithm selects the most informative or uncertain instances from the unlabelled dataset based on a predefined query strategy;
3. Human Annotation: The selected instances are presented to an oracle (could be a human annotator or an automated system) to acquire their labels;
4. Model Update: The newly labelled data is added to the training set, and the model is retrained using this augmented dataset;
5. Iteration: The process iterates, with the model selecting additional instances for labelling in subsequent rounds.

3.3.1 Active Learning approaches

Numerous approaches for defining these query strategies have been suggested in the existing literature:

- **Uncertainty Sampling:** In this approach active learners query the instances they are least confident about labelling, assuming that labelling these uncertain instances will provide the most valuable information for model improvement;
- **Query-By-Committee:** It involves maintaining a committee C of models, all of which have been trained on the current labelled dataset, but each model represents a different and competing hypothesis. Each member of the committee is then allowed to vote on the labellings of query candidates. The most informative query is considered to be the instance about which they most disagree;
- **Expected Model Change:** This method employs a decision-theoretic approach: it focuses on selecting the data instance that would have the most substantial impact on altering the current model if we knew its label;
- **Expected Error Reduction:** This is another decision-theoretic approach that focuses on choosing the data instance not to change the model significantly, but reducing the model's generalization error;
- **Variance Reduction:** It focuses on reducing generalization error indirectly by minimizing output variance. In the context of active learning, variance is a measure of the uncertainty or variability of the model's predictions.

3.3.2 Active Learning challenges

While active learning presents promising opportunities in reducing labelling costs and improving model performance, it also encounters some challenges.

First of all, the choice of query strategy impacts the effectiveness of active learning. Selecting the most suitable strategy for a particular dataset or problem is crucial.

Active learning relies on an oracle for labelling, which could be a human annotator or an automated system. Human involvement can introduce biases or inconsistencies.

Instances that the model finds uncertain might not always be the most informative or beneficial for learning, leading to sub-optimal performance in some cases.

3.4 Classification

Classification attempts to learn the relationship between a set of feature variables and a target variable of interest. The target attribute in classification is a categorical variable with discrete values. Given a set of training data points, along with target labels, classification determines the class label for an unlabelled test case [21]. Two types of classification can be distinguished:

- **Binary classifier**: this type of classifier is employed when the classification problem has only two outcomes, and it is employed when the target variable assumes one of two distinct states;
- **Multi-class classifier**: classifier used if there are more than two categories in the target variable, assigning an appropriate class label to the test case from a multitude of possible categories.

3.4.1 Classification algorithms

Some of the widely used classification algorithms include the following:

Support Vector Machines

The goal of the SVM algorithm is to create the best line or decision boundary that can segregate the n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best-decision boundary is called a hyperplane.

SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called as support vectors, and hence algorithm is termed as Support Vector Machine [22].

The primary objective of SVM is to find the optimal hyperplane that maximises the distance from each class. This hyperplane is considered the best, as it achieves the maximum separation between the classes. There are two types of SVM:

- **Linear SVM**: this is employed when dealing with linearly separable data, where a dataset can be divided into two classes using a single straight line;
- **Non-linear SVM**: this variant is used when working with non-linearly separable data, where no single straight line can efficiently divide the dataset into two classes. Using this method, non-linear relationships in the data are transformed into a higher-dimensional space, where linear separation becomes possible.

SVM performs more effectively in high-dimensional datasets and when the data is linear. Moreover, it is robust to outliers.

On the other hand, on large datasets SVM tends to yield suboptimal results. Furthermore, tuning the hyperparameters of SVM, namely the cost (C) and gamma, is a non-trivial task due to the complexity involved. SVM is primarily designed for binary classification tasks. For multi-class classification, it requires techniques like one-vs-all (OvA) or one-vs-one (OvO) to extend it to multiple classes, which can increase complexity.

K-Nearest Neighbours

The k-nearest neighbours algorithm, also known as KNN or k-NN, is a non-parametric, supervised learning classifier, which uses proximity to make classifications or predictions about the grouping of an individual data point.

The goal of the k-nearest neighbour algorithm is to identify the nearest neighbours of a given query point, so that we can assign a class label to that point [23]. To find it, a distance metric needs to be defined (e.g. Euclidean distance, Manhattan distance, Minkowski distance, Hamming distance).

K is the number of neighbours to check to assign the label. Lower values of k can have high variance but low bias, and larger values of k may lead to high bias and lower variance. The choice of k will largely depend on the input data as data with more outliers or noise will likely perform better with higher values of k [23].

The KNN algorithm is not only one of the simplest but also highly efficient classification algorithms. Furthermore, it requires only a k value and a distance metric, so less hyperparameters compared to other algorithms. Nonetheless, being a "lazy learning algorithm", it only stores the training dataset for later use during prediction. That means that it tends to require greater memory and data storage compared to the other algorithms. Additionally, it does not perform well with high-dimensional data inputs (curse of dimensionality) and so it is also more prone to overfitting:

Naive Bayes

Supervised learning algorithm based on Bayes' Theorem. Bayes theorem provides a way of computing posterior probability $P(c|x)$ from $P(c)$, $P(x)$ and $P(x|c)$. Below the equation:

$$P(c|x) = \frac{P(x|c)P(c)}{P(x)}$$

Where:

- $P(c|x)$ is the posterior probability of class (c, target) given predictor (x, attributes);
- $P(c)$ is the prior probability of class;
- $P(x|c)$ is the likelihood which is the probability of the predictor given class;
- $P(x)$ is the prior probability of the predictor.

This approach is based on the assumption that the features of the input data are conditionally independent given the class, allowing the algorithm to make predictions quickly and accurately [24].

There are many advantages to this algorithm: first of all, it is regarded as a simpler classifier due to the ease of estimating its parameters and also fast and efficient. Additionally, The algorithm can handle a large number of dimensions, which can be challenging for other classifiers to handle effectively.

A limitation of this algorithm is the core assumption: conditional independence does not always hold. Further, another problem is the zero frequency: when a categorical variable does not exist within the training dataset. In this case, the probability will be zero, and multiplied by the other conditional probabilities, the posterior probability will be zero too. A possible solution is to use the smoothing technique.

Decision Tree

They have a tree structure, made of root node, internal nodes, branches and leaf nodes. The leaf nodes represent all the possible outcomes within the dataset [16]. Decision trees can be understood as a series of if-else statements. They evaluate conditions and based on whether the condition is true or false, they proceed to the corresponding next node connected to that decision. There are various methods available for selecting the best attribute at each node; among these information gain (difference in entropy before and after a split on a given attribute) and Gini impurity (probability of misclassifying a randomly selected data point within a dataset if it were labelled based on the class distribution of that dataset) are widely used as popular criteria for splitting decision tree models.

One of the advantages of this algorithm is the interpretability: the tree structure is easy to understand, and it is also straightforward to identify the most important attributes. Moreover, it can handle different data types (i.e. discrete or continuous values).

However, there are some drawbacks: overfitting when the decision tree is complex, and they may not generalise well on new data. Further, small variations in the data can lead to significant differences in the resulting decision tree. Decision trees can be computationally expensive to train compared to other algorithms, due to the greedy search approach used to construct the tree.

Random forests

Random forests are an ensemble learning technique, meaning they combine the predictions of multiple decision trees to make a final prediction. Three are the main hyperparameters, which need to be set before training: node size, number of trees and number of features sampled. The algorithm predicts a value or category by combining the results of all decision trees.

The presence of many decision trees in a random forest solves the overfitting problem; in fact, the process of averaging uncorrelated trees helps to reduce the overall variance and prediction error.

However, this algorithm is obviously more complex, since it consists of many trees, and it requires more memory and processing power for training and prediction. Other than that, it shares pros and cons with the decision tree algorithm: it can handle both classification and regression tasks, but it is a time-consuming process.

Gradient Boosting

Gradient Boosting is another ensemble learning technique. While in Random Forest each tree is independent, here decision trees are built sequentially, and each new tree focuses on correcting the errors made by the previous trees. It employs a gradient descent optimisation process to minimise a loss function: at each iteration, the algorithm computes the gradient of the loss function in relation to the predictions of the current model and then trains a new weak model to minimise this gradient.

Since each iteration corrects the error of the previous tree, it can be more accurate than Random Forest. For the same reason, gradient boosting is well-suited for handling larger dataset.

However, Gradient Boosting might require more computational resources and time compared to Random Forest due to its iterative nature. Moreover, it is more prone to overfitting, requiring the application of regularisation techniques.

Convolutional Neural Network

In deep learning, convolutional neural networks (CNN/ConvNet) is a specialised subclass of Artificial Neural Networks (ANNs) used in classification and computer vision tasks.

Neural networks or artificial neural networks (ANNs) are comprised of a node layers, containing an input layer, one or more hidden layers, and an output layer. Each node, or artificial neuron, connects to another and has an associated weight and threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network [25].

CNN uses a special technique called convolution. Convolution, in mathematical terms, refers to the transformation of two functions into one that expresses how the shape of one is modified by the shape of the other. CNN, using this operation, provides an efficient automated feature extraction, while before CNNs, manual and time-consuming feature extraction methods were used to identify objects in images.

Moreover, CNNs capture the spatial features from an image. Spatial features refer to the arrangement of pixels and the relationship between them in an image. They help us in identifying the object accurately, the location of an object, as well as its relation with other objects in an image. Additionally, convolution technique enhances robustness to noise in the data.

On the contrary it requires a lot of computational effort because of the high number of layers and parameters. Furthermore, if the dataset is too small it may overfit and it may not generalise on new data.

Anyway, the accuracy of malware analysis with machine learning methods appears to be diminishing as malware becomes more sophisticated. While deep learning can provide greater accuracy. For malware image classification, CNN is very common because of its efficiency in image classification. The problem is that malware uses binary data that is one-dimensional, while CNNs solve problems based on two-dimensional images. One usual method is to create two-dimensional images from a one-dimensional image using the width of the two-dimensional images targeted. Further, CNNs are primarily used for supervised learning, and high-quality labelled are fundamental, that is hard in malware classification.

3.4.2 Classification model evaluation

Evaluation is important to understand the performance of the machine learning model. Two of the most common factors that affect this performance are listed below:

- **Overfitting**: it happens when a model fits too well on the training data that it cannot perform well on new data;
- **Underfitting**: it happens when the model is not able to capture meaningful relationships between input and output data.

In the context of malware classification, overfitting and underfitting are common challenges. Dealing with both of these two is crucial. A sufficient and diverse dataset for training is necessary, as well as selecting the right machine learning algorithm, optimising hyperparameters, and using the right feature representation. Additionally, techniques such as cross-validation and regularisation can help mitigate these issues.

There are different metrics to evaluate the performance of a classification model, and it is important to use multiple metrics because a model may perform well on one metric and not on others. The most common are the following:

Accuracy

Accuracy is the easiest way to measure the performance of the classifier. It is defined as the ratio of the correct predictions and the total number of predictions:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

Where:

- True Positive (TP) is the outcome correctly labelled as positive by the model;
- False Positive (FP) is the outcome incorrectly labelled as positive by the model, because it is actually negative;
- True Negative (TN) is the outcome correctly labelled as negative by the model;
- False Negative (FN) is the outcome incorrectly labelled as negative by the model, because it is actually positive;

Even if it is a very simple and easy to interpret measure, in some cases it is misleading. Indeed, it is not a good metric when the dataset is imbalanced.

Confusion matrix

It is a table that represents the classification results. Each column represents one class. So we have a 2-by-2 matrix for binary classification and an n-by-n matrix for multiclass classification where n is the number of classes. It is extremely useful for measuring the Recall, Precision, Accuracy, and AUC-ROC curves.

Precision

Precision describes the proportion of correctly predicted positive cases among all the cases that were predicted as positive:

$$Precision = \frac{TP}{TP + FP}$$

It is a useful metric when False Positives are more significant than False Negatives.

Recall

Recall measures the proportion of actual positive cases that are correctly identified as positive:

$$Recall = \frac{TP}{TP + FN}$$

On the contrary, this is useful when False Negatives are more significant than False Positives.

F1-score

It combines both precision and recall to provide a single measure of a classification model's performance. Useful when the dataset is imbalanced.

$$F1 - score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

The drawback of F1-Score is that it treats both false positive and false negative errors equally, which means it provides a balanced assessment of precision and recall, but does not distinguish between the varying impacts of these errors. As a result, in scenarios where different errors have significantly different consequences, it may be necessary to consider alternative evaluation metrics or to take into account the specific context in which the classification model will be used.

AUC-ROC

ROC curve (receiver operating characteristic curve) that plots two parameters at all classification thresholds:

- $TruePositiveRate(TPR) = \frac{TP}{TP+FN}$
- $TrueNegativeRate(TNR) = \frac{FP}{FP+TN}$

AUC stands for "Area under the ROC Curve". It can be defined as the measure of the ability of a classifier to distinguish between classes. The higher the AUC, the better the performance. So when it is equal to 1, the classifier is perfectly able to distinguish all positive and negative classes. When equal to 0, the classifier predicts all negative classes as positives and vice versa.

3.5 Malware detection using Machine Learning

Malware detection using the traditional approach has the major problem of not being able to detect zero-day attacks. That is why security analysts proposed the behavioral-model approach, but it is not without limitation, as it can be time-consuming.

Machine Learning is a possible solution to both of these limitations. It is important not only to detect malware, but also to determine which family a malware belongs to, in order to have a more comprehensive analysis. The difference between these two tasks, malware detection and malware family classification, is the output of the system. Malware detection outputs a single value that indicates whether a software is malicious or not. Malware classification outputs the probability that a malware belongs to each family.

The traditional machine learning approaches for malware detection or classification do not work on raw malware, instead they require a preprocessing of data to extract a set of features from the executable. The type of features can be classified into two groups, similar to the types of malware analysis: static features and dynamic features [14]:

3.5.1 Static features

Static features are the ones extracted from a piece of the program without executing it. The most common are [14]:

String analysis

String analysis refers to the extraction of every printable string (ASCII and Unicode-printable sequence of characters) within an executable or program [14]. Strings can give clues about the program functionality or about the suspect binary. They can contain references to URLs, filenames, domain names, IP addresses and so on. It is very simple, but it is usually used with other static or dynamic techniques to minimize its pitfalls. Indeed, it is a task that consumes a significant amount of time and it is prone to errors, given that relevant strings appear significantly less frequently compared to irrelevant ones and that large binaries can produce an extensive number of individual strings.

Bytes and opcode N-Grams

N-gram is the most used feature for malware detection and classification. An n-gram is a contiguous sequence of n items from a given sequence of text. N-grams can be extracted from the bytes sequences representing the malware's binary content, by looking at the unique combination of every n consecutive bytes as an individual feature, and from the assembly language source code referring to the unique combination of every n consecutive opcodes as an individual feature [14].

Although they are widely used, there are some issues. First of all it is computationally impossible to enumerate all the n-grams, moreover too many features can lead to curse of dimensionality when the number of samples is lower. Second, this technique is demonstrated to select features that occur frequently enough, namely features with low entropy, such as string and padding. Lastly, minor changes will prevent the feature from appearing, so generalisation cannot be achieved and this will lead to overfitting.

API function calls

API calls are ideal features for characterising malware behaviour, including malicious activities and functionalities. Through API calls, the malware interacts with the operating system components, has access to files and processes, communicates with remote servers, captures user inputs, and so on. However, also this technique could be compromised by code obfuscation: malware authors can hide malicious API calls, making them harder to detect.

Entropy

Since the two most used obfuscation techniques by threat actors are compression and encryption, it is highly interesting for security analysts to understand whether there are encrypted or compressed segments of code within the executable file. Entropy analysis serves this purpose because files containing compressed or encrypted code segments typically exhibit higher entropy compared to native code. In information theory, entropy (or more precisely, Shannon's entropy) quantifies the amount of information in a variable [26]. Generally speaking, the entropy of a bytes sequence refers to its statistical variation. High entropy values indicate that the segment comprises exclusively unique or distinct values. Indeed, if specific byte values have higher occurrence probabilities, the resulting entropy value will be lower. The problem is that malicious actors can use tricks to reduce the entropy value (e.g. to pad "nop" instructions).

Malware representation as a gray scale image

A given malware binary is read as a vector of 8 bit unsigned integers and then organised into a 2D array. Each of these unsigned integers is mapped to a grayscale value within the range [0, 255]. In this representation, 0 corresponds to black, and 255 corresponds to white. The intermediate values represent different shades of gray. The bytes are plotted starting from the left and wrapping to the next row once the end is reached. Fig 3.1 shows this process.

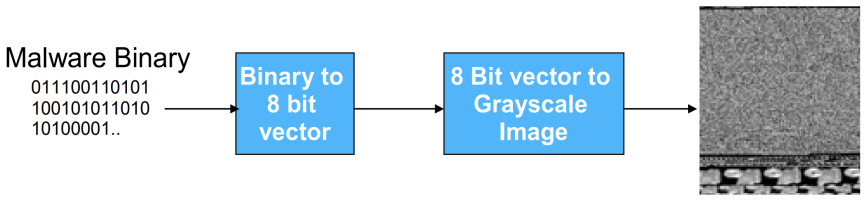

Figure 3.1: Visualizing malware as an image [27]

The width of the image is fixed and the height is allowed to vary depending on the file size [27]. Grayscale is useful in malware classification, as images of various malware samples within the same family exhibit visual similarities while distinct from those in different families. However, there are certain issues associated with this technique. First of all, like most static features, it suffers from code obfuscation: changing the byte structure of the binary will result in the failure of the classification process. Second, to construct an image you need to select an image width which adds a new hyper-parameter to tune. Third, it imposes non-existing spatial correlations between pixels in different rows, which might not be true [14]. Additionally, grayscale images may not capture the full complexity of malware, and their effectiveness depends on the specific features used and the quality of the images.

Function call graphs

A Function Call Graph (FCG) is a directed graph whose vertices represent the functions of which a software program is composed, and the edges symbolise function calls. A vertex is represented by either one of the following two types of functions:

- Local functions, implemented by the programmer to perform specific tasks;
- External functions: provided by the O.S. or system and external libraries [14].

The concept of the call graph in malware classification is that files that share similar call graphs are likely to have been generated from the same family.

Control Flow Graph

A Control Flow Graph (CFG) is a directed graph in which the nodes represent basic blocks and the edges represent control flow paths. A basic block is a linear sequence of program instructions having an entry point (the first instruction executed) and an exit point (the last instruction executed). A CFG is a representation of all the paths that can be traversed during a program's execution [14].

3.5.2 Dynamic features

Dynamic features are those extracted from the execution of malware at runtime [14]. Below are the most common ones:

Memory and Register's usage

The behaviour of a computer program can be represented by the values of the memory contents at runtime [14]. A considerable amount of information can be found in memory, such as active and terminated processes, Dynamic Link Libraries (DLL), running services, registry, and active network connections. Furthermore, the analysis provides accurate information about malware behaviours by extracting memory-based features that can express malware activities and characteristics [28].

Instruction traces

A dynamic instruction trace is a sequence of processor instructions called during the execution of a program. Contrary to the static instruction trace, dynamic traces are ordered as they are executed, while static traces are ordered as they appear in the binary file. Dynamic traces are a more robust measure of the program's behavior, since code packers and encrypters can obfuscate and hinder the code instructions from static analysis [14].

Network traffic

As soon as malware infects a host machine, it may establish communication with an external server to obtain the commands to execute on the victim or to download updates, other malware or to leak private and sensitive information of the user/device. As a result, the monitoring of network traffic entering and exiting the network, the traffic within the network and the host activity, provide helpful information to detect malicious behavior [14].

API call traces

Windows API calls enable virus writers to use these calls for gaining elevated security privileges and executing malicious actions. Compared to the static version, it is more robust to code obfuscation.

3.6 Current challenges

Machine learning for malware detection and classification has to face some issues. The following are the common ones [29]:

Dataset

Data represent the primary challenge in machine learning problems. First of all, the size of the dataset impacts the algorithm performance, but building a larger dataset for malware classification is an expensive task, since it requires for the experts to manually label each sample.

The lack of a standard dataset benchmark also contributed to this issue. Even though malware binaries are shared generously through Web sites such as VirusShare and VX Heaven, benign binaries are often protected by copyright laws that prevent sharing [14].

Moreover, in particular for malware detection and classification, class imbalance is one of the main limitations: the number of malware samples could be disproportionate to the number of benign samples, or the number of samples belonging to one family could be disproportionate to the number of other families samples. Given that, accuracy is not a reliable metric for the model evaluation. Indeed, a model that predicts the majority class for all examples in the test set will have a high classification accuracy. In other words, the classifier might be biased towards the majority classes and achieve very poor classification rates on the minority classes. It might happen that the classifier predicts everything as the major class and ends up ignoring the minor classes. This is called the accuracy paradox [14]. Other more reliable metrics in this case are precision, recall and F1-score.

Outdated datasets are also another issue.

Obfuscated malware

Modern stealthy malware attacks hide their behavior in virtual environments and security tools. This technique makes the malware challenging to be detected [29]. To put it in the machine learning context, an attacker's aim is to fool the machine learning detector by camouflaging a piece of malware in feature space by inducing a feature representation highly correlated to benign behavior. The ability of the attacker to bypass machine learning solutions is related to their knowledge about features and machine learning models to target [14].

Implementation time

This is a common challenge encountered by machine learning experts. Different kinds of algorithms might be used simultaneously in the empirical experiment; it takes longer to detect malware [29]. Furthermore, several elements contribute to this extended processing time. For instance, programs that run slowly or are resource-intensive may reduce the overall efficiency of the detection process. Data overload is another common issue, especially when dealing with extensive datasets, which can extend the time needed for analysis. Additionally, the requirements of certain machine learning algorithms, both in terms of computing power and memory, can lead to longer processing times. This can be a drawback when promptly detection is crucial.

Type of analysis

It might influence the performance of the model, since some malware can't be detected through static analysis. Even though static analysis is popular among researchers, it is also contributed to one of the limitations in detecting malware [29]. To address the drawbacks of static analysis there are many solutions that combine both static and dynamic analysis. By leveraging from each approach these solutions aim to improve the overall efficiency of malware detection.

Malware features

The feature attributes chosen must be independent of each other or have a low correlation coefficient. Highly correlated features contain redundant information, and this can increase the computational complexity. Furthermore, this can make the model sensitive to minor variations in the training data. When the same data are perturbed slightly or when new data is added, the model's coefficients may change significantly. This sensitivity can result in unpredictable and unreliable model behaviour.

Besides, many issues arise due to the large number of unrelated or duplicate characteristics. One of them includes confusion about the learning algorithm, since the algorithm may not be able to understand which attributes are significant for making accurate predictions. In addition, having too many attributes can result in over-fitting. These problems can lead to a reduced classification accuracy [29].

To mitigate these problems and achieve optimal model performance, feature selection techniques are often employed to choose a subset of the most relevant and informative attributes while discarding irrelevant or redundant ones.

Classification method

There are many classification methods used for malware detection as previously explained, and the performance is different based on several critical variables including dataset, features and hyperparameters.

Additionally, distinguishing between malware families is a more challenging problem than a binary classification of malware and benign files [29]. This complexity arises from various factors: for example, malware actors could use polymorphism to change code and behavior of the malicious software, requiring different detection strategies even within the same family. Moreover, some features that are effective to detect a specific family may not be useful for another, due to the significant variations in the characteristics of different malware families. Another crucial problem is related to the out-of-distribution detection, the process of detecting samples that do not belong to the training distribution. The algorithm should be able to detect samples that belong to unknown families.

High false positive rate

False positives in the context of malware detection occur when a benign program is wrongly labelled as malicious one. There are many reasons why this is a concern.

First of all, high false positive rates can result into a waste of important resources, as security teams may need to investigate and handle each false alarm. This can be time-consuming and costly.

Moreover, users may face interruptions or inconveniences when benign files are mistakenly marked as malware, affecting their overall experience.

Additionally, repeated false positives can damage trust in the security software or system. Users may start to disregard alerts, assuming that they are likely false alarms. The acceptable false positive rate in malware detection needs to be very low. There are some methods that are more likely to generate false positives.

Concept drift

Concept drift refers to the change in the underlying relationships between input and output data over time. In the domain of malware detection and classification, using machine learning models that assume that there are static relationships in the data will lead to degrading predictive performance. That's because malware continuously changes over time. Malicious actors continually modify and adapt their techniques to avoid detection, leading to variations in the characteristics

of malware. Therefore, it is important to understand when a model is degrading because it may no longer be capable of accurately identifying new malware threats. Hence, it is essential to periodically assess the model's performance and, when necessary, retrain it with novel malware samples. This approach ensures that the model remains up-to-date and capable of effectively classifying the latest malware variants, reducing the risk of false negatives in malware detection and classification systems. There are also other solutions to address concept drift, and they will be outlined throughout this thesis.

3.7 Cybersecurity and Machine Learning: A recap

This chapter provides an overview of key concepts in machine learning within the context of cybersecurity. It explores various machine learning approaches and algorithms, with a particular emphasis on Semi-Supervised and Active learning, which are the fundamental approaches of this thesis. These are highlighted for their significance in enhancing model efficiency when labelled data is limited.

The chapter further explores the implementation of machine learning in cybersecurity, examining its role in threat detection, and addressing the associated challenges.

The synthesis of these concepts and methodologies allows for a comprehensive understanding of how machine learning can be used to enhance cybersecurity defences in an ever-evolving digital landscape.

Chapter 4

Related works

Machine learning techniques have been widely used in the field of malware detection in recent times. Numerous research papers have been published, adopting a various range of approaches, from traditional machine learning algorithms to deep learning methods. These studies have presented encouraging outcomes, showing the efficacy of employing machine learning in the field of malware detection.

4.1 Image-based approach

Due to promising results and the continuous growth in the field of image processing in machine learning, in recent years researchers have been converting malware executables into images and categorising them based on features from images by applying image processing-based techniques.

C.V. Bijitha et al. [30] presented a detailed study on image processing-based malware classification techniques, focused on three questions:

1. What are the various methods used by researchers to convert executables into images? (Images from Executable Binary Distributions that does not involve any disassembly or execution of samples; Images from Opcodes by Executable Disassembly, from disassembling the executables to construct images, which appears more meaningful compared to binary data; Images from API Calls by Executable Execution, the best way for identifying malware behavioral features that can be executed)

2. What visual features can effectively characterize malware? (Texture Analysis and Image Processing Hashes)

3. What are the image classification techniques adopted by the researchers, and how much accurate they are? (Classification Based on Hand Crafted Features and Deep Learning Based Classification Method).

They identified also open issues in this area, like obfuscation, zero-day malware, cross-family malware, the byte plot (which contains only byte distribution) that cannot capture the behavioural aspects of the malware executable, and the fact that the feature engineering-related workarounds where excessive domain knowledge is required endure significant issues when employing machine learning techniques for such applications. The attackers could utilise the information about the feature set in their favour and exploit the system.

Tran The Son et al. [31] evaluated the classification performance of various machine learning classifiers (i.e. k-NN, NB, SVM, CNN) fed by malware images in various dimensions (i.e., 128×128, 64×64, 32×32, 16×16), on three different datasets including Malimg, Malheur and BIG2015. The experiment results obtained by four different classifiers on the Malimg dataset look very good. Whereas on Malheur and BIG2015 datasets in which there exists a bad classification. It can

be explained that images of a given malware family provided by Malimg dataset are very much similar in texture and dimension; therefore, it is easy for a classifier to do classification. While variants of some malware families in the Malheur dataset are observed to be very different. It is a big challenge for a classifier to learn and classify a new instance of malware. The type of classifier is another factor that affects the accuracy of classification. KNN provides the best performance, NB the worst. CNN instead shows accuracy very similar to SVM but it is more complicated: that is because grayscale malware images are very simple, and using a complicated network for learning and classifying might not be effective. In terms of input size, the highest accuracy value with K-NN is reached with 32x32 and tends to decrease when the size of the input images increases. That is due to the fact that k-NN relies on Euclidean distance to decide the nearest neighbours, so too many pixels make it get confused. Instead, under 32, there is a lack of important features.

Baraa Tareq Hammad et al. [32] proposed a malware classification (MC) method with excellent accuracy and without using any data augmentation or other balancing strategies. It consists of the following steps:

1. Dataset prepation: 2D malware images are created from the malware binary files. Malware binaries are used to construct eight-bit vectors and a grayscale image is created from these vectors;

2. Visualized Malware Pre-processing: the visual malware images need to be scaled to fit the CNN model's input size maintaining the malware images' primary textual characteristics;

3. Feature extraction: both hand-engineering (Tamura) and deep learning (GoogLeNet) techniques are used to extract the features in this step;

4. Classification: to perform malware classification,k-Nearest Neighbor (KNN), Support Vector Machines (SVM), and Extreme Learning Machine (ELM) are employed.

The results indicated that this method can efficiently classify malware even when the dataset is unbalanced. The proposed method's accuracy rate was higher than both the hand-crafted feature and Deep Feature techniques.

4.2 Semi-supervised approach

Because of the limited availability of labels and the high cost of acquiring them, a lot of recent works have adopted Semi-Supervised approach. Several of them are image-based malware classification approaches.

Ding, Y. et al. [33] proposed a malware classification (MC) method based on Portable Executable (PE) file header. The model first converts the malware binary to grayscale, then pre-train the neural network with unlabelled data to learn generic representations of malware samples, and finally finetunes the model with only a small amount of labelled data to learn malware classification capabilities. The proposed algorithm experimented on the open-source Virus-MNIST dataset. Multiple neural network algorithms are used to train the malware classification model (ResNet, DenseNet, MobileNet, VGG etc.). Experiments show that the pre-training process proposed in this method can improve the performance of the model, but for the problem that the performance of traditional malware classification methods continues to degrade with the evolution of malware, this method can only reduce the manual annotation workload required to retrain the model.

Tan Gao et al. [34] proposed a malware classification model based on malware visualisation and co-training of classifiers. First, binary files are processed through visual methods; then the global and local features of the gray images are extracted and fused by a special method (canonical correlation analysis) to eliminate the exclusion between different feature variables. Finally, collaborative learning using a Tri-training algorithm is proposed to continuously train and optimise the classifier by introducing features of unlabelled samples. The datasets used are i.e. Malimg and Microsoft. The results show that this improved collaborative learning reduces the cost of labelling and, moreover, it can continuously improve the model performance through

the input of unlabelled samples, achieving higher classification accuracy, but it increases the time overhead.

Rui Shu et al. [35] proposed a framework that uses hyperparameter optimisation with semi-supervised learning and Bayesian optimization to search a large hyperparameter space. The basis of the framework is Pseudo-labeling: it works by iteratively propagating labels from labelled data to unlabelled data. Moreover, if the percentage of minority class in the training dataset is low (under a certain threeshold), it further adaptively integrates an optimised oversampler into the framework (called SMOTE). The selected classifier is Random Forest. Dapper framework is evaluated with three security datasets that cover different security tasks: Twitter Spam, Malware URLs, CIC-IDS-2017. Standard semi-supervised learning algorithms do not adequately address the class imbalance issue. When decreasing the label rate, the imbalanced issue even gets worse. That is why this framework introduces SMOTE. But SMOTE might bring in an increment of false positive rate. This might result from the pseudo-labelling process, which infers the original minority class even to majority class. Moreover, Dapper with label spreading provides even better performance than supervised learning with 100% labelled training data, but with as low as 10% of original labelled data required.

Jan Koza et al. [36] proposed a comparison between two semi-supervised algorithms for deep neural networks on a large real-world malware dataset. The first one is Pseudo-labeling, which uses unlabelled samples, classified with high confidence, as if they were the actual labels. The second is implemented in two variants: Π-model (compares two predictions of the same state of the network using different inputs and different droppedout neurons; the second prediction is augmented multiplying it with noise) and temporal ensembling (compares the predictions of the network in the current epoch with the prediction obtained in the previous epoch). Based on their experiments, they determined that the performance of the fully supervised learning depends on the number of labelled data. Instead Pseudo-labeling and Π-model have higher accuracy of low ratios of the labels (under 10%). Also, it seems that Π-model outperforms Pseudo-labeling, as its accuracy is higher in most of the measurements.

Salma Abdelmonem et al. [37] proposed an enhanced version of the Π-model, which combines supervised learning with semi-supervised information, making it more accurate and consistent. This work compares three supervised learning models: Four-Layer CNN, ResNet50 and LeNet-8 and selects the best one in terms of accuracy and complexity to be included in the final proposed semi-supervised model. It uses the Malimg dataset. The results show that semi-supervised model performs better than supervised one when the ratio of labelled samples is as low as 20%. This is because supervised learning models overfit when the data is small. On the other side semi-supervised Π-model add regularisation techniques on top of the supervised learning model to prevent overfitting.

Shuwei Wang at al. [38] proposed an improved malware image rescaling algorithm (IMIR) based on local mean algorithm: it extends the sampling range to the edge of sliding window. Then it adjusts the calculation method of step length and adds a fill step to fit the malware images. Its main goal of IMIR is to reduce the loss of information from samples during the process of converting binary files to image files. They designed a classifier model with one-dimensional convolutional neural network which is suitable for malware classification with image features extracted by IMIR. Then they optimised the classifier through weak coupling semi-supervised learning based on SGAN. SGAN (Semi-Supervised GAN) trains both the generator and the semi-supervised classifier at the same time: it uses the supervised learning with labelled samples to train model for judging categories, and the unsupervised learning with unlabelled samples to train model for judging true or false. Through weak coupling they can retain the weak links of supervised part and unsupervised part of SGAN, improving the accuracy of malware classification by making classifiers more independent of discriminators. Identified issues in this paper are the loss of GAN that is hard to optimise and the "mean" operation in IMIR that may be a bit rough.

Giovanni Apruzzese et al. [39] researched previous works utilising the combination of labelled and unlabelled data for Cyber Threat Detection (CTD) and found out that none of such works addressed the benefits of unlabelled data in SsL. Therefore, they formalised the evaluation requirements that enable one to assess the impact of unlabelled data in the development of an SsL model. To address this problem, they proposed an original Cybersecurity Evaluation Framework for SsL

(CEF-SsL) for assessing the benefits of unlabelled data in SsL for previous and future works: this aims to provide a practical assessment of SsL methods by using any fully labelled dataset, while simultaneously considering the deployment budget and ensuring the statistical significance of the results. The chapter 5 will provide a detailed description of this work.

4.3 Concept drift mitigation strategies

As mentioned above, concept drift is one of the primary challenges of machine learning for cybersecurity. Several solutions exist to address and prevent the ageing of classification models.

A prevalent approach found in the literature is to retrain the model. However, a challenge arises when the model is retrained too often, resulting in insufficient new data to effectively enhance the model. Conversely, retraining less frequently can lead to a time period where the model's reliability is compromised. Additionally, manual labelling is expensive, and models need to be updated often with new examples. Due to this reason, researchers have developed different mitigation strategies, for instance to identify a restricted set of optimal items for labelling (active learning).

In the context of network intrusion Andresini et al.[40] proposed INSOMNIA, a semi-supervised intrusion detector which continuously updates the underlying ML model as network traffic characteristics are affected by concept drift. It employs a deep neural network (DNN) as its core classifier. To mitigate the latency caused by model updates, they employ active learning techniques. This approach focuses on updating the model using newly acquired data points that offer the highest potential for information gain.

To evaluate this system they used TESSERACT [41]. Tesseract proposed a new metric named AUT (Area Under Time) to effectively measure model's performance over time within the concept drift context and compare different malware classifiers. The authors also emphasise the importance of adhering to temporal constraints when training and testing models to accurately portray the model's performance. This involves arranging the training data preceding the test data and segmenting the test data into consecutive time periods of equal duration. They proposed a framework designed for evaluating trade-offs between performance and cost of techniques to mitigate time decay such as active learning. Their experiments reveal that Deep Learning is actually the algorithm most robust to time decay. This phenomenon might be attributed to the feature representation within the latent feature space automatically discerned by deep learning, which appears to exhibit enhanced resilience to time decay within the specific experimental context they explore.

Another solution is to design more robust feature spaces, but this is not without challenges. In their work, Adam Duby et al. [42] evaluated feature robustness against concept drift for the malware classification problem. They used an approach based on dynamic features, since they are more challenging for attackers to manipulate. This is due to the fact that dynamic features need to mantain certain semantics to ensure malware's functionality, which can be compromised during manipulation attempts. They used in particular two sets of dynamic features: process handle counts by type, and named memory-mapped files. Their experiments showed that the selected features are more robust against drift over state-of-the-art API based feature sets.

Alternatively, there is another approach, named classification with rejection, where predictions with low confidence resulting from instances affected by drift are rejected. Roberto Jordaney et al. [43] proposed TRANSCEND, a framework that performs classification with rejection to identify ageing classification models in vivo during deployment, much before the machine learning model's performance starts to degrade. This framework uses a conformal evaluator, a tool that evaluates the quality of predictions made by the classifier. Its design is based on conformal prediction theory, which aims to quantify the level of confidence or uncertainty associated with the predictions generated by a model. This tool employs the concept of nonconformity to detect and discard new instances deviating from the training distribution, which are prone to being inaccurately classified. However, this solution does not scale to larger datasets and it is computationally complex. Moreover, the performance of this system relies on the definition of "dissimilarity", and as proved in [44], in settings with high data dimensionality, it could be bad.

Limin Yang et al. [44] proposed CADE, a system that can detect drifting samples that deviate from existing classes, and provide explanations to reason the detected drift. The system uses contrastive learning to learn how to compare and contrast pairs of samples, and uses distance-based explanations to reason the detected drift. Contrastive learning uses the labels in the training data to learn a distance function that measures the similarity of two samples. CADE's computational overhead is smaller than existing methods, and its detection runtime overhead is significantly lower than that of Transcend.

Federico Barbero et al. [45] proposed TRANSCENDENT, a framework built on Transcend to overcome its limitations. They proposed new conformal evaluators that aim to minimise computational overhead.

4.4 Insights from Related Works

In this chapter, an extensive survey explores two primary approaches for malware classification: the Image-based approach and Semi-Supervised Learning. The chapter begins with a critical analysis of various image-based algorithms, highlighting their strengths and limitations in the domain of malware classification. Subsequently, the exploration expands to analyse various SsL techniques applied in the context of malware family identification and categorisation. Several studies and methodologies are discussed, highlighting the efficacy of SsL in handling limited labelled data while leveraging the potential of extensive unlabelled datasets. However, also the limitations and the open issues associated with SSL approaches are emphasised.

Particularly noteworthy is the evaluation framework introduced within the related works, which serves as a principal component in this project. This framework enables a rigorous evaluation of the benefits associated with SsL, addressing a notable gap in previous works.

Furthermore, the chapter provides an in-depth analysis of concept drift, a phenomenon prevalent in dynamic environments, along with various strategies designed to effectively address this challenge. This exploration allows for a comprehensive understanding of the dynamic nature of malware threats and the adaptability required in defence mechanisms to mitigate these evolving adversarial strategies.

Chapter 5

Cybersecurity Evaluation Framework for SsL

Initially, the authors discussed the challenges of labelling data in CTD, which are more difficult than in other domains:

- The intrinsic confidentiality which strongly discourages data sharing;
- Attackers constantly adapt their strategies as well as the environment being protected grows and evolves, leading to the phenomenon of concept drift;
- Even security experts find it challenging to verify the ground truth of a sample;
- Data augmentation is difficult in CTD. For instance, changing a single byte can turn many malicious samples into benign ones.

So, acquiring labels in CTD is an issue that motivates SsL methods to be investigated. Furthermore, they claimed that a dataset that is usable for realistic CTD applications of ML must meet the following criteria:

- It must be *large* enough to capture all the underlying characteristics of the environment to protect, and of the threats to defend against;
- The ratio of benign/malicious samples must be *balanced* enough to allow efficient detection without generating excessive false alarms;
- It must have *accurate ground truth*;
- It must be continuously *updated*.

Imbalanced dataset is a common issue, as introduced before in this thesis. There are many techniques to overcome this issue. The most used one is sampling: a preprocessing technique, so it is applied on training data to reduce the imbalance ratio between classes. The following methods can be distinguished:

- **Over-sampling**: It consists of increasing the size of the minority class by duplicating randomly selected samples;
- **Under-sampling**: It consists of decreasing the size of the majority class by removing randomly selected samples;
- **Hybrid sampling**: It consists of applying both previous techniques.

The problem with sampling is that there is no guarantee that achieving good accuracy on a balanced dataset will necessarily result in reliable performance in a real-world highly imbalanced scenario, especially when dealing with highly imbalanced data.

In fact, as stated by Harsurinder Kaur et al. [46], over-sampling tends to increase the size of the data space, which results in a situation of overfitting and takes more time in the training phase. However, under-sampling removes some set of samples randomly from the majority class where lies the possibility of losing informative samples.

Although, as cited above, Apruzzese et al. [39] in their paper asserted that a dataset must be balanced in CTD, the oversampling techniques that they applied in the presence of data-imbalance did not lead to significant changes. This problem requires further analysis, and it will be expounded upon through a detailed explanation of the implemented methodology in Section 7.3.1, followed by the presentation of the correlated results in Section 8.3.

5.1 Requirements to assess SsL benefits

CEF-SsL framework can be applied to assess the benefits of SsL in CTD. Below is an explanation of the foundational terminology employed to define specific requirements used for evaluating the advantages of SsL, as referenced in the cited paper:

Let $\mathcal{L}$ be a given labelling budget. Let $\mathbb{L}$ be any labelled dataset containing sample-label pairs, obtained by using $\mathcal{L}$. Let $\overline{\mathbb{L}}$ be a superset of $\mathbb{L}$. Let $\mathbb{U}$ be an unlabelled dataset containing samples of which the ground truth is not known. Finally, let $\mathbb{F}$ be another labelled dataset which represents future data. Let $\mathbb{D}$ be a fully labelled dataset, used to derive $\mathbb{F}$, $\mathbb{U}$, $\mathbb{L}$, $\overline{\mathbb{L}}$. Fig 5.1 shows the diagram of the partitions creation process.

Let μ be any performance metric, e.g., accuracy or F1-score. Let SL be a model trained on $\mathbb{L}$, and $\overline{SL}$ be a model trained on $\overline{\mathbb{L}}$. The goal of a semi-supervised learning (SsL) method is using $\mathbb{U}$ alongside any $\mathbb{L}$ obtained with $\mathcal{L}$ to devise a model SsL. After deployment, SsL should predict the ground truth of the samples in $\mathbb{F}$ by achieving a performance $\mu(SsL)$ s.t.: $\mu(SL) < \mu(SsL) < \mu(\overline{SL})$.

The deployment of a semi-supervised model SsL requires to invest some budget, $\mathcal{B}$. Such budget can be seen as the sum of three elements $\mathcal{U}$ (investment for obtaining $\mathbb{U}$), $\mathcal{L}$ (investment for generating $\mathbb{L}$) and ϵ (extra operation for developing SsL that is not related to labelling the samples in $\mathbb{L}$). By using $\mathcal{L}$, the organisation eventually obtains a labelled dataset $\mathbb{L}$, whose composition depends on the cost of labelling each individual sample x in $\mathbb{L}$; let $\mathcal{C}_x$ denote such cost, $\mathcal{L}$ can be defined as: $\sum_{x \in \mathcal{L}} \mathcal{C}_x$.

The ROI (return on investment) of any solution can be expressed as the ratio between its expected performance and its development budget; in the case of SsL: $ROI(SsL) = \frac{\mu(SsL)}{\mathcal{B}(SsL)}$.

The 7 requirements that must be maintained for any CTD task to ensure a clear and unequivocal assessment of the advantages of SsL methods are:

1. **Lower bound**: It is necessary to evaluate a lower bound model that only uses $\mathcal{L}$ and makes no use of $\mathbb{U}$. If the performance metric achieved by SsL is approximately the same of the one achieved by SL, there is no practical benefit in using the unlabelled data. It may also be that $\mu(SsL) < \mu(SL)$, meaning that using $\mathbb{U}$ is detrimental;

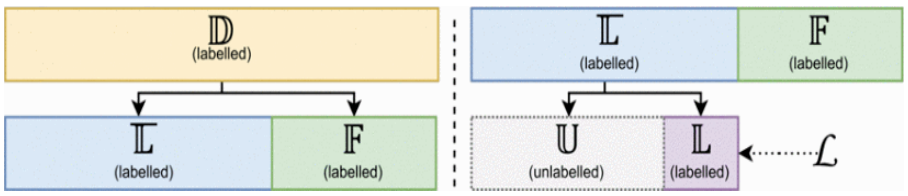

Figure 5.1: $\mathbb{D}$ is first split into $\mathbb{F}$ and $\overline{\mathbb{L}}$. Then $\overline{\mathbb{L}}$ is further split into $\mathbb{L}$ according to $\mathcal{L}$, and the leftout samples are considered as unlabelled $\mathbb{U}$ [39]

2. **Ablation Study**: It is necessary to always consider a 'vanilla' model $\underline{SsL}$ that uses the unlabelled data in a trivial way together with a $\mathbb{L}$ randomly sampled from $\mathbb{D}$. The 'vanilla' $\underline{SsL}$ allows to gauge (i) the smallest improvement provided by $\mathbb{U}$ via comparisons with SL; and also (ii) the smallest cost induced by using $\mathbb{U}$, because the randomness of $\mathbb{L}$ and the lack of supervision make the corresponding ϵ minimal. Moreover, $\underline{SsL}$ serves as a baseline for an ablation study to simulate worst case scenarios in which any operation that relies on $\mathbb{U}$ to refinely compose $\mathbb{L}$ is not functional in practice;

3. **Upper Bound**: It is necessary to train an upper bound model $\overline{SL}$ on $\overline{\mathbb{L}}$ and evaluate its performance on $\mathbb{F}$ as $\mu(SL)$. Motivation: The SL serves to assess the performance achievable by augmenting $\mathbb{L}$ until it reaches $\overline{\mathbb{L}}$. Moreover, if $\mu(SL) \approx \mu(\overline{SL})$, then investing in $\mathbb{U}$ to develop any SsL may not be worth it in the first place;

4. **Statistical Significance**: It is necessary to verify the statistical significance of any evaluation result. The huge search space for composing $\mathbb{L}$ from $\mathbb{D}$ may lead to erratic results. Hence, different draws of $\mathbb{L}$ (and, preferably, also of $\overline{\mathbb{L}}$ and $\mathbb{F}$) must be assessed and conclusions must be drawn after statistically significant comparisons;

5. **Transparency**:It is necessary to ensure full transparency on the composition of $\mathbb{L}$, $\overline{\mathbb{L}}$, $\mathbb{U}$ and $\mathbb{F}$ (size and balance ratios in terms of class composition);

6. **Reproducibility**:Any evaluation must be supported with information that allow its reproducibility;

7. **Multiple Settings**: It is necessary to evaluate any model by considering multiple deployment settings.

5.2 Implementation

The unlabelled data $\mathbb{U}$ used to develop any SsL is beneficial if it is shown that: (i) $\mu(SL) \ll \mu(\overline{SL})$, and (ii) ROI($SsL$) is better than both ROI(SL) and ROI($\underline{SsL}$). Let $\overrightarrow{ML}$ denotes an array of ML methods, and (n, k) is a pair of integers that regulate the 'runs'. The proposed framework can be divided into three stages, as illustrated in Fig 5.2:

- **Prepare**, that uses $\mathcal{L}$ to partition $\mathbb{D}$ into $\mathbb{L}$, $\mathbb{U}$ and $\mathbb{F}$;
- **Run**, which (i) train SL on $\mathbb{L}$ and test it on $\mathbb{F}$ as $\mu(SL)$; account for all operational costs as $\epsilon(SL)$; (ii) train $\overline{SL}$ on $\overline{\mathbb{L}}$ and test it on $\mathbb{F}$ as $\mu(\overline{SL})$; account for all operational costs as $\epsilon(\overline{SL})$; (iii) use $\mathbb{L}$ and $\mathbb{U}$ to devise $\underline{SsL}$ and test it on $\mathbb{F}$ as $\mu(\underline{SsL})$; account for all operational costs as $\epsilon(\underline{SsL})$; Then, if the SsL method does not make any assumptions on $\mathbb{L}$, then this framework uses $\mathbb{U}$ and the previously drawn $\mathbb{L}$ as input for the SsL method, otherwise it generates a new $\mathbb{L}$ considering the specifications of the SsL method and use it with $\mathbb{U}$ as inputs, finally SsL is trained and tested on $\mathbb{F}$ accounting for all operational costs as $\epsilon(SsL)$;
- **Iterate**, to obtain statistically significant results, it performs multiple 'runs' Finally the aggregated results of each method can be validated via e.g. Student t-test.

The output are two (n, k)-dimensional arrays, whose elements include the results of all the models devised through $\overrightarrow{ML}$ on each run, i.e.: $\overrightarrow{\mu}$, containing the performance on 'future' data; and $\overrightarrow{\epsilon}$, containing any cost incurred during the development (not related to labelling).

The authors used CEF-SsL to assess binary classifiers in their evaluation. Consequently, in some cases, they aggregated all samples from different malicious families into a single malicious class. But this thesis aims not only to separate malicious software from legitimate software, but also to categorise them. Evaluations of SsL methods in multiclass classification settings are challenging as stated in this paper.

Applying CEF-SsL in similar settings is possible, i.e., by manually specifying the cost to label each malicious sample and performing hundreds of runs of CEF-SsL, but this is empirically difficult from a research perspective. This paper outlines the following challenges:

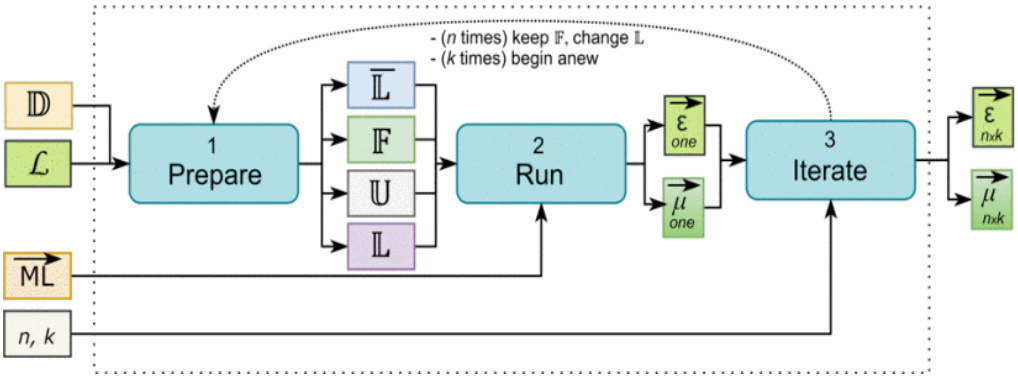

Figure 5.2: Evaluation Framework for SsL

- Randomly choosing a limited amount of samples from N malicious classes: some datasets only have a very limited number of samples for some classes; there are many approaches to mitigate this problem, e.g. aggregate samples in macro classes, removing classes with few samples or aggregate these in another class; however all these methods may introduce some errors;
- Composing appropriate partitions $\mathbb{L}$, $\mathbb{U}$ and $\mathbb{F}$: since in the context of real-world scenarios benign events are typically more prevalent than malicious ones, a limited labeling budget raises the question, "How many samples per class should be included in $\mathbb{L}$?" And what about $\mathbb{U}$ and $\mathbb{F}$? There is the possibility of using the relative distribution in a dataset, but it may introduce bias or skew the results to favour the majority class.

In conclusion, they used this framework to perform the first statistically validated benchmark of 9 selected SsL methods on 9 well-known datasets for CTD. These SsL techniques represent variations of two established SsL approaches: *"self learning via pseudo-labelling"* and *"active learning via uncertainty sampling"*. In particular, the study examines three distinct pseudo-labelling techniques, three distinct active learning techniques, and three hybrid methods that combine pseudo-labeling with active learning:

Pseudo Labelling

They defined:

- $\underline{SsL}$: using all pseudo labels regardless of their confidence;
- πSsL: using only the pseudo labels with the highest confidence $c \geq 99\%$;
- $\hat{\pi} SsL$: which repeats the previous operation another time. They use πSsL to predict the remaining $\mathbb{U}$, and insert the corresponding pseudo-labelled samples with $c \geq 99\%$ in the 'mixed' $\mathbb{L}$.

Active Learning

The budget for labelling samples is divided into two equal parts. The first half of the budget is used to train the initial learner, and the second half is used to assign the correct labels to specific samples based on the threshold c. To prevent any bias, random selection is used to choose the 'actively labeled' samples from among those that meet the specified criteria. Due to this random selection process, the framework repeats the selection five times for each active learning method. They defined:

- αSsL_l: using low confidence samples $c < 1\%$;
- αSsL_h: using high confidence samples $c \geq 99\%$;
- αSsL_o: using the other samples $1\% < c < 99\%$.

Pseudo-Active Learning

Pseudo Labelling and Active Learning are combined by using $\hat{\pi}SsL$ as the initial learner (but developed with half of the initial $\mathcal{L}$), which produces three 'pseudo-active' methods ($\alpha^{\pi}SsL_l$, $\alpha^{\pi}SsL_o$, $\alpha^{\pi}SsL_h$) in the same way as in the 'pure' active learning.

5.3 Performance Assessment and Statistical Validation

The considered ML methods use the Random Forest (RF) learning algorithm because (i) they have been widely adopted by previous work, (ii) they perform well at low computational costs and can be parallelised, (iii) their performance is similar to other algorithms, but they require less training time. Additional evaluation of other algorithms can be done. The results reveal that SsL methods are better than supervised ones, however, in some cases they can degrade performance. Thus, they proved that there exists a substantial margin for improvement of the SsL methods for CTD.

They used *F1-score* as a performance metric, computing the average F1-score achieved by each method in $\overrightarrow{ML}$ on each dataset, across all the different combinations of $\mathcal{L}$ and $\mathcal{C}$. They evaluated *execution time* as the metric for ϵ.

For statistical validation, they used the Wilcoxon ranksum [47]. This test involves the comparison of two populations to assess a specific null hypothesis, denoted as H_0. The test outputs a z-value which is used to determine a p-value: H_0 can be accepted or rejected on the basis of p, according to a target significance level. H_0 assumes that the two populations are statistically equivalent: a large p-value suggests that the two groups are similar(so H_0 should be accepted), while a small p-value suggests that the two groups are different, favoring H_0 rejection. The significance level is set to 0.05, implying that if $p > 0.05$ then the two populations are equivalent; conversely, if $p \leq 0.05$, the two populations are different. For each dataset, they compared the populations containing the performance of the baseline SL against: (i) the best 'pure' pseudo-labelling method; and (ii) the best active learning method.

5.4 CEF-SsL Overview

This chapter includes an exploration of the framework's foundational concepts. It delineates the underlying assumptions of the framework's application, while providing an in-depth analysis of the requirements used for evaluating the advantages of SsL. Furthermore, it provides a detailed breakdown of the methodology chosen for the assessment, describing the step-by-step implementation process. It provides the explanation of the self-learning and active learning approaches, offering insights into the selection and application of evaluation metrics alongside the statistical validation methods.

In this thesis, this framework was tailored to suit the adopted image-based approach, incorporating various algorithms beyond the Random Forest model employed in the original one, including deep learning methodologies.

Chapter 6

Dataset

As mentioned above, datasets pose significant challenges when utilizing machine learning for malware classification. Above some of the most recent and widely used datasets in the literature:

6.1 SOREL-20M

SOREL-20M (Sophos-ReversingLabs-20 Million) dataset is a large-scale dataset consisting of nearly 20 million files with pre-extracted features and metadata, high-quality labels derived from multiple sources, information about vendor detections of the malware samples at the time of collection, and additional "tags" related to each malware sample to serve as additional targets [48]. Despite it is a large dataset which makes it interesting for analysis, the main limitation is the absence of labels for malware families. Therefore, this dataset is deemed unsuitable for the purpose of this thesis.

6.2 Microsoft Malware Classification Challenge

As part of the Microsoft Malware Classification Challenge announced in 2015, a nearly 0.5 terabyte dataset was released, consisting of 21,741 malware samples. This dataset is divided into two parts, 10,868 samples for training and the other 10,873 samples for testing. In addition to its role in the Kaggle competition, this dataset has become widely adopted as a standard benchmark for researching and modelling malware behaviour. It consists of a set of known malware files representing a mix of 9 different families. Each malware file has an identifier, a 20-character hash value uniquely identifying the file, and a class label, which is an integer representing one of the 9 family names to which the malware may belong. For each file, the raw data contain the hexadecimal representation of the file's binary content, without the header (to ensure sterility). The dataset also includes a metadata manifest, which is a log containing various metadata information extracted from the binary, such as function calls, strings, etc [49]. This dataset also suffers from class imbalance. But it provides the possibility to extract various features (opcodes, function calls, and ngrams) and also to generate grayscale graphical representations.

6.3 BODMAS

Researchers have faced limitations in studying issues like concept drift and malware family evolution due to the absence of two critical capabilities in existing open PE malware datasets: recent/timestamped malware samples and well-curated family information. For these reasons, the BODMAS dataset was released. It contains 57,293 malware samples and 77,142 benign samples collected from August 2019 to September 2020, with carefully curated family information (581

families) [50]. For each sample there is both the original PE binary as well as a pre-extracted feature vector that shares the same format with existing datasets such as Ember [51] and SOREL20M [48]. The family label information provided in this dataset is significantly more comprehensive compared to existing datasets. For example, the Microsoft dataset [49] only includes 9 families, whereas this dataset offers a much broader range of family labels. Datasets such as SOREL-20M do not contain family labels. Combining this dataset with existing datasets such as Ember and SOREL-20M, researchers could have malware samples span over three years to study malware evolution and potential concept drift of classifiers [50]. But a large number of training families will immediately increase the difficulty of training an accurate multi-class classifier. Moreover, it is highly imbalanced i.e., only 14 families have samples greater than 1000 and more than 100 malware families have only 1 sample in the whole dataset.

6.4 Malimg

As previously explained, researchers are increasingly using image processing-based techniques for malware classification in recent years. The first step is to convert the malware binary into an image. Then, extract the features from the image. Next, based on the extracted features, classify the executable into different families. The Malimg Dataset contains 9,339 malware byteplot images from 25 different families. Executable binaries represented as a sequence of bits are grouped into 8-bit vectors, where each element can have a value between 0 and 255. When generating a grayscale image, the value 0 is transformed into black, 255 becomes white, and the remaining values are assigned various shades of gray. Following the guidelines proposed by Nataraj et al. [27], the width of the images is typically maintained at a constant value, while the height is adjusted to correspond to the size of the executable file. Fig 6.1 illustrates samples from different malware families in the dataset. An advantage of this dataset is that it comprises images, so

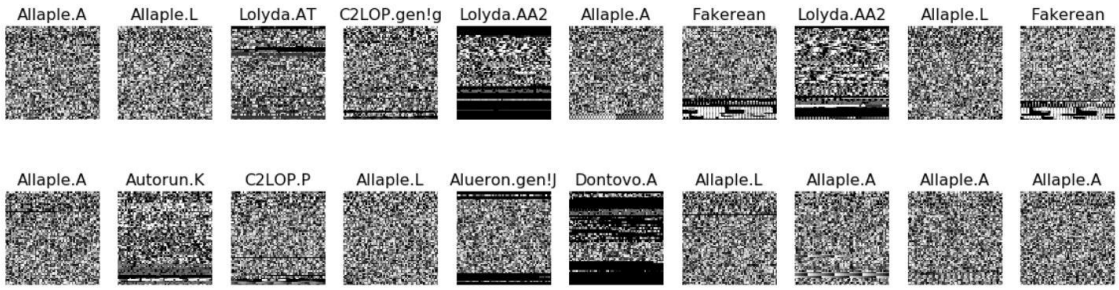

Figure 6.1: Variants of malware from Malimg

samples don't require pre-processing when applying image-based analysis. On the other hand, the dataset exhibits significant class imbalance. For instance, there are 2949 samples representing the Allaple.A family, whereas only 80 samples pertain to the Skintrim.N family. Moreover, malware binaries are not available.

6.5 MaleVis

It is a corpus involving byte images of 26 (25+1) classes. Here, 1 class represents the "legitimate" samples while the rest of the 25 classes correspond to different malware types. To construct this corpus, they first extracted the binary images from malware files (supplied by Comodo Inc) in 3 channel RGB form. Following to having vertically long images, the images are resized in 2 different square sized resolution (224x224 and 300x300 pixels) [52]. MaleVis dataset involves a total of 9100 training and 5126 validation RGB images. Therefore, it is necessary to convert these RGB images into grayscale images. All the training classes involve 350 image samples while the validation set has various numbers of images. Further, the binaries are available if you contact them. Details of this dataset are shown in Table 6.1 where the distribution of the training set is illustrated.

Class ID	*Family*	*Malware Category*	*Sample Number*
#1	Adposhel	Adware	350
#2	Agent	Trojan	350
#3	Allaple	Worm	350
#4	Amonetize	Adware	350
#5	Androm	Backdoor	350
#6	Autorun	Worm	350
#7	BrowseFox	Adware	350
#8	Dinwood	Trojan	350
#9	Elex	Trojan	350
#10	Expiro	Virus	350
#11	Fasong	Trojan	350
#12	HackKMS	Riskware	350
#13	Hlux	Worm	350
#14	Injector	Trojan	350
#15	InstallCore	Adware	350
#16	MultiPlug	Adware	350
#17	Neoreklami	Adware	350
#18	Neshta	Virus	350
#19	Other	-	350
#20	Regrun	Trojan	350
#21	Sality	Virus	350
#22	Snarasite	Trojan	350
#23	Stantinko	Trojan	350
#24	VBA	Macro Malwares	350
#25	VBKrypt	Trojan	350
#26	Vilsel	Trojan	350

Table 6.1: MaleVis dataset.

6.6 Virus-MNIST

Image classification dataset consisting of 10 executable code varieties and 51,880 examples (48,422 training and 3,454 test). Specifically, the dataset comprises thumbnail images that bear resemblance to the well-known MNIST handwritten digits dataset. However, in this case, the thumbnails are created by reshaping potential malware code into image arrays. The designation of 9 virus families for malware derives from unsupervised learning of class labels (K-Means clustering) to match the standard MNIST format and multi-class [53]. It explores a dataset that contains static analysis data: Raw PE byte stream rescaled to a 32 x 32 greyscale image using the Nearest Neighbor Interpolation algorithm and then flattened to a 1024 bytes vector. The PE malware samples were downloaded from virusshare.com while PE non malicious examples from portableapps.com [54]. By testing 10 or more MD5 hash values against the VirusTotal.com database, they assigned the broad types and example executable names. It is available on Kaggle and Github. Fig 6.2 shows the abstract images derived for each of the 10 classes, with "0" as the only one that is non-malicious.

It is a large scale dataset and choosing a 32x32 pixel representation for the PE header offers advantages to deep learning algorithms that rely on powers of 2 (stride length), to construct their convolutional layers. However, this dataset exhibits an imbalanced distribution of samples. Moreover there is a 20:1 ratio between malware and beneware, so non-malicious executables should be added. Additionally, results from benchmarking on this dataset shows the highest false-negative rate for malware is class "0" or beneware that gets flagged as potentially malicious. This behaviour may reflect the inexactness of the PE header as an indicator of malware, or the hijacking of benign header characteristics to disguise malware in the first 1024 bytes. The higher false-negative rates may also originate in the KMeans approach, given the clustering excluded the non-malicious cases to simplify the generation of 9 malware families as independent groups

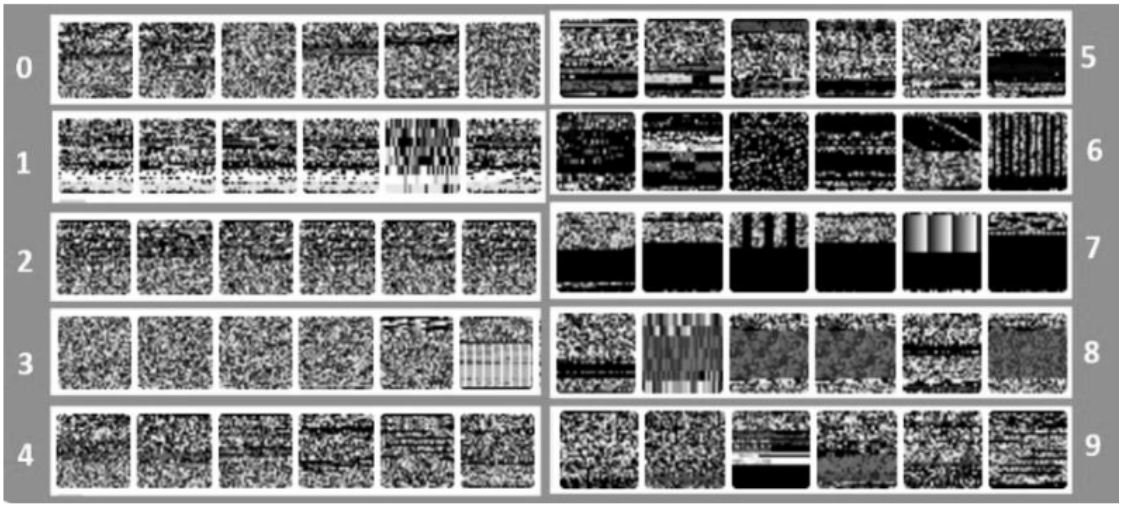

Figure 6.2: Virus-MNIST showing 10 classes [53].

from beneware. Further investigation may explore whether other clustering approaches benefit the class distinctions [53].

6.7 VirusShare

VirusShare is a repository of malware samples to provide security researchers, incident responders, forensic analysts, and the morbidly curious access to samples of live malicious code [55]. All samples are delivered in password-protected zip-files for safety. Each zip file contains 131,072 samples if the number in the zip name falls between 0 and 148. However, if the zip file is numbered 149 or higher, it contains 65,536 samples. The higher-numbered zip files contain more recent samples, allowing for the study of both new and old malware. However, the zip files do not include any labels indicating the specific malware contained within or their respective families. To assign tags to the malware, external support is required, specifically the assistance of vx-underground. The main drawback is that the website explicitly prohibits the use of automated scripts for downloading malware, making the process of constructing the dataset time-consuming.

6.8 VirusTotal

VirusTotal is a website that provides the capability to analyse files and detect potential malware within them. It offers a platform where users can submit files for scanning and obtain reports on any identified malicious content or suspicious behaviour. Additionally, VirusTotal offers access to a vast dataset of malware samples. The main drawbacks of using the VirusTotal dataset are the limited number of samples available and the requirement to submit a form on the website to obtain information about the family of a specific malware by providing its hash.

6.9 Concept Drift Analysis through Malware Datasets

The most used datasets have been collected years ago, which makes them old in terms of malware evolution. It is indeed a fact that malicious actors continuously develop new malware to evade detection, and these new samples might have characteristics and behaviours that differ from the older ones. This results in changes in the data distribution, a phenomenon known as concept drift.

As already explained before, concept drift affects the performance of the machine learning models over time causing them to become obsolete. Employing older datasets enables the simulation of historical threat environments and allows evaluating the adaptability of the model to shifting distributions. By training on older data and validating on recent data, the model's ability to handle concept drift can be assessed effectively.

Moreover, ensuring an accurate train-test split is essential; otherwise, the model could suffer from temporal bias, a phenomenon that arises when experiments assume a consistent feature distribution between test-time and train-time, leading to potentially poor performance due to the dynamic evolution of malware behaviours and techniques. On the contrary, training a model only on recent data might overfit to the specific characteristics of that time period. Predictions are more robust when older data are incorporated into the model, since it generalises better to different eras.

Distinct from the other datasets, BODMAS is more recent, but it spans a short time-frame (2019-2020). In [50] the authors used this dataset to study the impact of concept drift on malware classifiers. In particular, they trained a multi-class Gradient Boosting Decision Tree (GBDT) classifier on the malware samples in the first month of BODMAS and then they evaluated the performance of the classifier in the remaining 12 months.

Additionally, Alejandro Guerra-Manzanares et al. [56] merge different data sources to create an expansive dataset containing samples spanning all years of Android history. Moreover, it is also the first dataset to include the time variable in the Android malware detection issue. As Android malware is a non-stationary phenomenon, it is an alive and constantly evolving phenomenon that should be placed into its temporal context. In this regard, the timestamp of apps allows the investigation and characterisation of concept drift in Android apps and its proper detection to build more effective and robust models.

Addressing label bias, since each AV engines' own naming rules and there does not exist any convention on malware family naming, the authors scanned all the dataset samples using VirusTotal AV engine and they labeled each sample based on the majority of the vote. The issue with this approach, as discussed in [57], is its limited effectiveness due to its failure to consider the independence of AV engines, thereby decreasing the model's performance.

Their experiments demonstrated that choosing a reputable AV engine outperforms label aggregation, despite the possibility of a higher false positive rate, which could potentially lead researchers to lean towards the label aggregation approach. However, the drawback is associated with consumer engines, as they can give rise to a misleading perception of majority support for the label aggregation approach. In fact, researchers noted that certain AV engines exhibited close F1 scores and showed significant similarities in their training logs, suggesting the possibility of the existence of analogous AV engines. However, while the labels provided by consumer-supplier engines may seem standardised and reliable, this could lead researchers to place unwarranted trust in consumer engines as the ground truth, even when they might not be suitable for label aggregation. The authors recommended to remove these consumer engines, but this action could result in the discarding of a significant amount of data, inflating the number of singleton labels and minority families.

Another approach is to elect one reputable AV engine. However, the issue with this approach is the feature preference problem, since an AV engine that prefers one feature might give a different family name than the one that picks another feature.

6.10 Comparative Analysis of Malware Datasets: Summary

The exploration of various malware datasets has provided insights into their diverse characteristics, strengths, and limitations. Among the various datasets explored, several critical limitations have been identified that impact their applicability to specific research objectives.

- **Absence of Malware Family Labels**: Some datasets are limited to the malware detection process due to the absence of comprehensive labels distinguishing different malware families;

- **Lack of Malware Binaries**: Certain datasets lack access to malware binaries, preventing the in-depth analysis required for understanding malicious code behaviours and functionality;
- **Imbalanced Class Distributions**: Unbalanced class distributions within datasets pose challenges, impacting the effectiveness of training models, potentially leading to biased learning outcomes and inaccurate predictions;
- **Absence of a Benign Class**: Datasets that lack a benign class essential for comparative analysis limit their utility in distinguishing between malicious and non-malicious instances;
- **Absence of Recent and Timestamped Data**: An additional limitation prevalent in several malware datasets is the absence of recent or time-stamped data. This limitation restricts the analysis of concept drift, a critical aspect in understanding the evolving nature of malware over time.

Each of these constraints highlights the importance of careful selection of datasets specific to particular research objectives. Understanding these limitations enables researchers to align the attributes of the dataset with the analysis objective.

Chapter 7

Proposed method

The main purpose of this thesis is to analyse which are the limitations of the Semi-Supervised approach compared to the Supervised one and how to overcome these. More in detail the aim is to identify which are the circustamces under which the unlabelled data can be used to improve the performance accuracy.

To assess the benefits of unlabelled data, I used the CEF-SsL framework explained in the Chapter 5. The CEF-SsL implementation code is available at `https://github.com/hihey54/CEF-SsL`. This framework was designed for binary classification, so I needed to modify it in order to suit the goal of this work, which involves multi-class classification.

To experiment with the limitations of the SsL (Semi-Supervised Learning) approach, the initial step involved understanding and determining the specific features that need to be extracted. Then I experimented with different algorithms beyond the Random Forest implemented in the framework, and not limited to traditional machine learning, in order to make comparisons between their different results.

In Section 7.1 of this chapter, the motivations that prompted me to start with an image-based approach will be outlined, encompassing various approaches and techniques employed in extracting meaningful features from images specifically tailored for the classification of malware.

In Section 7.2, I will introduce the work environment I used and the support libraries. In the upcoming sections, which represent the core of this research, various algorithmic approaches will be explored while delving into their implementation specifics.

Section 7.3 will detail the modifications made to the original framework to align it with my specific problem, including the set of specific parameters and the incorporation of various techniques.

Moving to Section 7.4, an overview detailing the feature inputs will be provided, including an explanation of the handling of raw pixels and the methodologies for extracting texture features.

Additionally, Section 7.5 will encompass the implementation details of the adopted traditional machine learning algorithms. Subsequently, in Section 7.6, comprehensive insights into the CNN architecture and associated hyperparameters will be presented.

Lastly, Section 7.7 will present the combined dataset utilised for subsequent experiments.

7.1 Image-based malware classification

The use of visualisation-based malware analysis approaches to analyse malware and classify them has grown significantly over the past few years. As explained in [30], the popularity stems from two factors: First, effectiveness in detecting new variants of malware. Malware writers create new variants by reusing distinctive modules from existing malware, so the similarities in binary code could be effectively captured and aid in the detection process. Secondly, the image processing

field is constantly being the hot area of research, and new feature engineering techniques keep evolving, which could help the cybersecurity community combat malware.

In image-based malware classification, machine learning algorithms can be either traditional models based on handcrafted features or deep learning models that learn features on their own.

The use of image processing methods in the field of malware detection and/or classification initially focused on analysing texture. In the context of images, texture describes the variations in colour, intensity, or surface characteristics within an image such as smoothness, softness, roughness, etc. There are many feature extraction techniques, in my experiment the ones that I used are:

- **Local Binary Pattern(LBP)**: This technique computes a local representation of the texture. To create this local representation, each pixel is compared with its neighbouring pixels. It tests whether the surrounding points are greater than or less than the central point: if the intensity of the center pixel is greater-than-or-equal to its neighbour, then the value assigned is 1, otherwise 0. This binary number is generated for each set of neighbours, and the results are collected into a binary string. Subsequently, this binary string is converted into a decimal value, and this numerical representation is stored in a 2D LBP array with the same width and height as the input image. This entire procedure is repeated for each pixel within the image. Next, a histogram that represents the distribution of these quantized LBP values is created, where the x-axis corresponds to the different LBP pattern (integers), and the y-axis represents the frequency of occurrence of each value;
- **Gray Level Co-occurrence Matrix (GLCM)**: This technique calculates how often pairs of pixel intensities with specific spatial relationships or directions occur in the image, creating a matrix. In particular, the procedure involves determining the intensity value of the reference pixel and the neighbouring pixel in a specified direction (commonly defined by angles like 0, 45, 90, and 135 degrees). This process is repeated for all pixels in the image, resulting in a matrix where each element corresponds to the number of times a specific intensity pair occurs in the defined spatial relationship and direction. Once you have the GLCM matrix, you can compute various statistical properties that serve as descriptors of the texture or patterns in the image. Common properties include:
  - `Contrast`: high contrast values indicate the presence of significant variations between pixel intensities;
  - `Homogeneity`: the degree of similarity between pixels within an image. Higher values indicate more uniform regions;
  - `Correlation`: the degree to which changes in pixel intensity values are connected or correlated with each other;
  - `Energy`: it quantifies the uniformity of the image. Higher energy values indicate a more uniform texture;
  - `Dissimilarity`: it measures how different the gray levels of neighboring pixels are.
- **Histogram of Oriented Gradient (HOG)**: This technique computes histograms of gradient orientations in an image. First, the image is divided into small cells and for each pixel in the cell gradients are computed. Gradients represent the intensity changes in both the x (horizontal) and y (vertical) directions. The magnitude and orientation of the gradients are calculated for each pixel within the cell. Next, the gradient orientations are grouped into predefined bins or histograms and for each pixel in the cell the gradient magnitude is accumulated into the corresponding histogram bin based on its orientation. The cells are then grouped into larger blocks, and the histograms of the cells within a block are concatenated to form a single feature vector. This feature vector is then normalised to make it more robust to changes in contrast. The feature vectors of all blocks are concatenated to form the final HOG descriptor for the image.

In the literature, the most commonly employed conventional algorithms for malware classification include K-Nearest Neighbours (KNN), Support Vector Machine (SVM), and Random Forest (RF).

Architecture	Cluster Linux Infiniband-EDR MIMD Distributed Shared-Memory
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 Node	24x nVidia Tesla V100 SXM2 - 32 GB - 5120 cuda cores
Performance	90 TFLOPS (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

Figure 7.1: Legion - technical specification [58].

While in deep learning, Convolutional Neural Networks (CNNs) have emerged as the preferred choice among researchers. Many researchers have utilised various pre-trained CNN models such as VGG16, ResNet50, Inception v3, among others. The transfer learning approach, which involves fine-tuning these pre-trained CNN models on specific malware datasets, has become a popular and effective strategy in achieving high accuracy in malware classification tasks.

7.2 Work environment and tools used

In this section, I will describe the working environments that were used to develop the machine learning tool, as well as the libraries that were utilised to implement the algorithms.

7.2.1 Legion

The HPC@POLITO supercomputing initiative, managed by the Department of Automation and Informatics of the Polytechnic University of Turin (DAUIN), provides resources for advanced computing and technical support for academic and educational research activities [58]. Three clusters are available at this center: CASPER, HACTAR, and LEGION. The one I used was LEGION. It is an InfiniBand cluster with a performance exceeding 21 TFLOPS (measured with only 14 nodes) with the characteristics illustrated in the Fig 7.1.

7.2.2 Google Colaboratory

Colab is a hosted Jupyter Notebook service that requires no setup to use and provides free access to computing resources, including GPUs and TPUs [59]. It is designed for data science, machine learning, and artificial intelligence (AI) research. The Jupyter notebooks run on Google's cloud servers. Users can create, edit, and run Python code directly in the browser. Libraries commonly used in Data science and machine learning are pre-installed in Colab, including NumPy, Pandas, and Matplotlib.

7.2.3 Support library: Scikit-learn and PyTorch

Scikit-learn is a Python module integrating a wide range of state-of-the-art machine learning algorithms for medium-scale supervised and unsupervised problems. This package focuses on bringing machine learning to non-specialists using a general-purpose high-level language. Emphasis is put on ease of use, performance, documentation, and API consistency. It has minimal dependencies and is distributed under the simplified BSD license, encouraging its use in both academic and commercial settings [60]. It depends only on NumPy and SciPy to facilitate easy distribution.

Family	*Sample Number*	*Family*	*Sample Number*
Adposhel	494	Injector	495
Agent	470	InstallCore	500
Allaple	478	MultiPlug	499
Amonetize	497	Neoreklami	500
Androm	500	Neshta	497
Autorun	496	Other	1832
BrowseFox	493	Regrun	485
Dinwood	499	Sality	499
Elex	500	Snarasite	500
Expiro	501	Stantinko	500
Fasong	500	VBA	500
HackKMS	499	VBKrypt	496
Hlux	500	Vilsel	496

Table 7.1: Samples distribution of MaleVis dataset.

NumPy is a Python library that provides multidimensional arrays and matrices along with along with a comprehensive set of high-level mathematical functions to operate on these objects efficiently.

SciPy is another popular Python library built on top of NumPy that provides a wide range of modules for various scientific computations, including optimisation, integration, interpolation, linear algebra, statistical functions, and much more. Scikit-learn is the library used in the implementation of the CEF-SsL framework.

However, Scikit-learn is mostly used for machine learning, while PyTorch is used for deep learning. Deep learning frameworks have often focused on either usability or speed, but not both. PyTorch is a machine learning library that shows that these two goals are in fact compatible: it provides an imperative and Pythonic programming style that supports code as a model, makes debugging easy, and is consistent with other popular scientific computing libraries, while remaining efficient and supporting hardware accelerators such as GPUs [61]. In the context of image-based malware classification, deep learning has been an effective solution. To implement neural networks, this library was used.

7.3 CEF-SsL implementation for multi-class classification

In my experiments, I used the MaleVis dataset [52]. This choice derives from the necessity of including a benign class within the dataset to enable the process of malware family classification. The Malimg dataset, widely employed in image-based malware classification, lacks the inclusion of a benign class. Hence, the MaleVis dataset emerged as the most fitting option to meet the requirements of this research. It comprises a total of 14,226 samples, which are further classified into 12,394 malicious samples and 1,832 benign samples.

The MaleVis dataset is well balanced within its malicious classes, with no class having a significantly smaller number of samples, with consistent sample counts among the classes. This addresses the challenges outlined earlier regarding the expansion of this framework into multi-class classification scenarios. In particular, the lack of classes with inadequate sample sizes mitigates any concerns surrounding the random selection of a limited number of samples from the malicious classes. However, there is a moderate difference between the number of samples in the malicious classes and the number of samples in the benign class ("Other" family), as it is illustrated in the Table 7.1

Consistent with the original framework, I have preserved the 80:20 split, as it is common in CTD tasks. The other parameters I set are: n = 2, k = 3 which are a pair of integers that regulate the runs of CEF-SsL; different costs of labelling: 1=malicious and benign samples have same cost; 2=malicious samples have twice the cost of benign samples; 5=malicious samples have five times

the cost of benign samples. These are the costs of labelling defined by the original framework, that is tailored for binary classification.

Other parameters defined are two different values of the labelling budget, 50 and 100. The active budget is 0.5 meaning that half of the labelling budget is used to develop the first learner, and then the remaining half is used to assign the correct label to those samples that meet a specific confidence threshold: samples whose confidence is above $(100 - high\_confidence\_window/2)$ are considered to be of "high confidence" and samples whose confidence is below $(low\_confidence\_window/2)$ are considered to be of "low confidence" where $high\_confidence\_window$ and $low\_confidence\_window$ are 0.2. The confidence thresholds remain consistent across all family classes. The active learning trials are 2.

7.3.1 Multiple binary classifiers

To adapt the original framework to multi-class classification settings, one approach is to use multiple binary classifiers. This strategy iterates over all different malicious families, one at a time, performing a binary classification One-vs-One each time. Each iteration in detail is a binary classification between a specific malware family class and the benign class. The results are aggregated from all the binary classifiers.

Nevertheless, when dealing with imbalanced classes, this may lead to biased models or poor classification results. Similarly to the original framework, to mitigate the imbalance between malicious and benign classes, I also used an oversampling technique called SMOTE, which stands for Synthetic Minority Over-sampling Technique, provided by the imbalanced-learn (imblearn) library, that extends scikit-learn. SMOTE creates "synthetic" examples rather than by oversampling with replacement. The minority class is over-sampled by taking each minority class sample and introducing synthetic examples along the line segments joining any/all of the k minority class nearest neighbours. Synthetic samples are generated in the following way: Take the difference between the feature vector (sample) under consideration and its nearest neighbour. Multiply this difference by a random number between 0 and 1, and add it to the feature vector under consideration. This causes the selection of a random point along the line segment between two specific features [62].

However, using multiple binary classifiers can be computationally expensive, especially with a large number of malware families. Moreover, it simplifies the classification task and does not consider potential relationships or shared characteristics between families, not capturing the full complexity of malware classification. To determine which approach is most suitable for this specific problem, the outcomes of this strategy were compared with the results obtained using the method detailed in the following section.

7.3.2 Multi-class classifier

As the previous assumptions can introduce limitations in the malware classification process, I have also adapted this framework to perform a multi-class classification approach considering all the malware families simultaneously.

Hence, I considered as the *F-score* value the weighted mean between the values for all the classes. Furthermore, to modify active learning techniques for multi-class classification, I follow the approach of selecting the sample with the greatest probability value among all classes and subsequently comparing this score to the threshold to determine its placement within a specific confidence range.

For the sake of practicality, I also maintained a distinct labelling cost only between the benign and malware classes, since it is not feasible due to the extensive time required. However, in the context of multi-class classification, these labelling costs can vary, especially among different malware families. For instance, a rarer malware family may incur different labelling costs compared to a more common one. Nevertheless, conducting an exhaustive search for all possible combinations of varying costs across numerous malware families can be unfeasible due to the vast number of potential combinations. The number of combinations increases exponentially with the number of

classes, leading to significant computational complexity and time requirements. To illustrate, in a scenario with 26 distinct classes, each having three possible cost factors (1x, 2x, and 5x), the total number of combinations would be approximately $3^{26} \approx 2.541.865.828.329$. For this reason, I considered a uniform cost per malicious sample, assigning the same labelling cost to all malicious samples. This would mean that each malicious sample has a labelling cost of, for example, 100, and the benign class could have a cost of 100 or 200 or 500 based on the defined criteria (the same, twice or five times the cost of malicious samples).

Also in this scenario, I employed SMOTE to address the issue of severe dataset imbalance, therefore when the cost of labelling malicious data is five times higher compared to labelling benign data.

7.4 Image-based with handcrafted features

As RGB images are used, the initial step involved converting them into grayscale images. Afterward, they were resized to dimensions of 128x128. The following steps depend on the specific features employed:

7.4.1 Raw pixel data

Machine learning models can be trained using directly grayscale malware images, where in the grayscale pixel values serve as the features of the input images, rather than relying on features extracted by image descriptors.

In this process, the resized images were flattened into 1D arrays and each of the resulting arrays was then labelled with its corresponding malware family. These labelled arrays were subsequently combined and appended together to form a new training set for the machine learning algorithm.

7.4.2 HOG feature descriptor and PCA

To compute the HOG feature descriptor, I used the *hog()* function provided by the Scikit-image library. I configured the HOG parameters as follows: I specified the number of gradient orientations as 4, set the cell size to (8, 8) pixels, defined the number of cells for each block as (2, 2) and applied block normalisation using the *L2-Hys* method.

PCA (Principal Component Analysis) is a commonly used dimensionality reduction technique. I used the *PCA()* function available in the scikit-learn library to reduce the dimensionality of the HOG features, setting the number of components to 50, to keep only the most informative principal components. Algorithm 1 illustrates the application of these functions to extract hog features with reduced dimensions.

Algorithm 1 HOG Feature Extraction + PCA

Input: Grayscale Image
Output: Reduced-dimension HOG features

1: Define HOG parameters (e.g., cell size, block size, etc.)
2: Compute HOG features using scikit-image:
3: $F_{\text{HOG}} \leftarrow \textbf{hog}(image)$
4: Apply PCA for dimensionality reduction
5: Initialise a PCA object with the desired number of components($n\_components$):
6: $pca \leftarrow \textbf{PCA}(n\_components = 50)$
7: Compute PCA features using the **fit_transform** method:
8: $F_{\text{PCA}} \leftarrow \text{pca.}\textbf{fit\_transform}(F_{\text{HOG}})$

7.4.3 LBP feature descriptor

I used the *local_binary_pattern()* function from the Scikit-image library to compute the LBP feature descriptor. A radius of 1 was chosen, which defined the radius of the circular neighbourhood used to compute LBP. The number of sampling points was set to 8 times the selected radius, allowing for comprehensive sampling within the neighbourhood. In order to simplify the texture analysis process, the 'uniform' method was used for LBP computation. Then, a histogram of LBP patterns was created. This is shown in the Algorithm 2

Algorithm 2 LBP Feature Extraction

Input: Grayscale Image
Output: Histogram of LBP features

1: Specify LBP parameters: the radius R and the number of neighbors P
2: Compute LBP features:
3: $F_{\text{LBP}} \leftarrow$ **local_binary_pattern**$(image, P, R, method =$ 'uniform') using scikit-image
4: Compute LBP histogram:
5: $LBP_{\text{hist}} \leftarrow$ **numpy.histogram**$(F_{\text{LBP}}, bins, range)$
6: Normalize LBP_{hist}

7.4.4 GLCM feature descriptor

To compute the gray-level co-occurrence matrix I used the *graycomatrix()* function provided by the Scikit-image library. The function takes several parameters to specify: (i) the distances between the reference pixel and its neighbouring that I set to [1] focusing only on adjacent pixel pairs; (ii) the angles at which pixel pairs are considered and I selected different directions (0, 45, 90 and 135 degrees) to capture information in different directions; (iii) symmetric, set to true, means that the glcm is computed symmetrically for each angle; (iv) normed set to true means that the values in the glcm are divided by the sum of all values resulting in a probability matrix where each element is the probability of a specific pixel pair occurring in the specific direction.

Then I used the *graycoprops()* function to calculate a feature from this matrix as a compact representation of it. The texture properties I considered are: energy, correlation, dissimilarity, homogeneity, and contrast. Algorithm 3 shows the glcm features extraction process.

Algorithm 3 GLCM Feature Extraction

Input: Grayscale Image
Output: GLCM properties

1: Calculate the Gray-Level Co-occurrence Matrix (GLCM) using **graycomatrix**:
2: $glcm \leftarrow$ **graycomatrix**$(image, distances, angles, levels)$
3: Extract GLCM properties using **graycoprops**:
4: $properties \leftarrow$ **graycoprops**$(glcm)$

7.5 ML models with handcrafted features

Any classifier that supports the *predict_proba* method provided by scikit-learn library can be employed. In particular, I conducted tests with some of the most prevalent algorithms commonly employed in the field of malware classification. The adopted implementations are as follows, with adjustments tailored to specific cases in the experiments, detailed and documented in Chapter 8.

7.5.1 Random Forest

The classifier I first used is Random Forest, that is, as already mentioned, the implementation provided by the CEF-SsL framework, using the scikit-learn library. The Random Forest model

consists of 100 trees and the 'gini' criterion is chosen for splitting nodes. There is no imposed limit on the maximum depth of the individual decision trees, and no specific class weights are assigned, meaning that all classes are treated equally in terms of importance.

7.5.2 Support Vector Machine

The Support Vector Machine (SVM) is another simple algorithm in the field of machine learning. It was originally designed to perform binary classification, but subsequent advancements have enabled it to execute multiclass classification tasks as well. The SVM classifier is provided with scikit-learn, and I configure it to use the Radial Basis Function (RBF) kernel, which is a suitable choice for handling non-linear data.

7.5.3 K-Nearest Neighbors

As outlined in the relevant literature, numerous investigations have employed the k-Nearest Neighbors (k-NN) algorithm for image-based malware classification and have achieved good results. I used the KNeighborsClassifier from scikit-learn setting the number of nearest neighbours to 5.

7.5.4 HistGradientBoosting

The use of a Gradient Boosting Classifier is not as prevalent in the field of malware classification compared to the other methods mentioned, but recent research has demonstrated its remarkable accuracy.

Histogram-Based Gradient Boosting, as other gradient boosting algorithms, builds an ensemble of decision trees where each tree corrects the errors of the previous ones. It minimise a loss function by iteratively adding decision trees. However, it uses histogram-based binning to divide continuous features into discrete bins, making it more memory-efficient and faster when compared to traditional gradient boosting algorithms. I employed this type of Gradient Boosting classifier, since it is much faster than GradientBoostingClassifier for big datasets ($n\_samples >= 10000$).

I used the HistGradientBoostingClassifier provided by the scikit-learn library. The hyperparameters have been set as follows: a maximum of 100 iterations of the boosting process; a learning rate of 0.1 is used to control each tree's influence on the final prediction for stability; a fixed random seed of 42 for reproducible output across multiple function calls; and 1 as the maximum number of leaves for each tree.

7.6 Image-based with Neural Networks

The convolutional neural network (CNN) I used in this study comprises several layers, including 2 convolutional layers and 2 fully connected layers. The purpose of this network is to classify grayscale images into one of 26 different classes.

In addition to the network architecture, I set the parameters as follows: I used the Cross-Entropy Loss as the objective function during training; the Adam optimizer is employed for updating the model's parameters; I experimented with different image sizes (32,64,128), different batch size (64,128,256) and different learning rate (0.1, 0.01, 0.001).

As the framework requires the classifier to support the *predict_proba* method, that is not directly available for CNN models in PyTorch like it is in scikit-learn for some classifiers, I used the trained model's output to which is applied the Softmax function to obtain class probabilities.

Family	*Sample Number*	*Family*	*Sample Number*
Adposhel	494	Injector	495
Agent	470	InstallCore	500
Allaple	478	Instantaccess	431
Alueron.gen!J	198	Lolyda.AA1	213
Amonetize	497	MultiPlug	499
Androm	500	Neoreklami	500
Autorun	496	Neshta	497
BrowseFox	493	Regrun	485
Dinwood	499	Sality	499
Elex	500	Snarasite	500
Expiro	501	Stantinko	500
Fakerean	381	VB.AT	408
Fasong	500	VBA	500
HackKMS	499	VBKrypt	496
Hlux	500	Vilsel	496

Table 7.2: Samples distribution of the blended dataset.

7.7 Blended dataset

An additional experiment was conducted to create a more comprehensive dataset, in which a combined dataset was generated by merging data from MaleVis and Malimg. Specifically, this dataset includes all the malware families from the MaleVis dataset along with five families from the Malimg dataset. This blended dataset is illustrated in Tab 7.2.

The objective was to enhance the dataset's diversity. This addition could potentially provide the model with a broader spectrum of malicious characteristics for more comprehensive learning and improved generalisation.

The selection of the five classes for integration was based on their sample sizes, with the aim of similarity to MaleVis. This choice allows us to incorporate a diverse set of images while maintaining a balanced dataset without causing significant imbalance.

Chapter 8

Results

In this chapter, I will present the results of the experiments that were detailed in Chapter 7. The primary objective of this study is to conduct an in-depth comparison between traditional machine learning models with hand-crafted features and deep learning models. To illustrate the results I will present tabulated data that showcases:

1. The average *F1-scores* achieved by each method during their application to machine learning;

2. The average *execution time*, considered as the "cost" of each method, without taking into account other additional costs for the sake of simplicity;

These scores are determined based on a range of different combinations of labelling budget $\mathcal{L}$ and the cost of labelling $\mathcal{C}$.

To perform a statistical comparison between methods, I employed the Wilcoxon test, following the approach in the original framework, utilising the *mannwhitneyu()* function available in the SciPy library. The pseudocode in Algorithm 4 illustrates the procedural steps to conduct the Wilcoxon test. This was done to evaluate the performance of the baseline SL compared to (i) the most effective pure pseudo-labelling method and (ii) the most effective active learning method. The resulting p-values and z-values from these comparisons are documented in tables.

Algorithm 4 Mann-Whitney U Test

function MannWhitneyUTest(group1, group2)
    all_data ← combine(group1, group2)
    ranked_data ← rank(all_data)
    n1 ← len(group1)
    n2 ← len(group2)
    U1 ← sum ranks for group1
    U2 ← sum ranks for group2
    U ← min(U1, U2)
    N ← $n1 + n2$
    z ← $\frac{U - \frac{n1 \cdot n2}{2} + 0.5}{\sqrt{\frac{n1 \cdot n2 \cdot (N+1)}{12}}}$
    p ← $2 \cdot norm.cdf(z)$
    return U, z, p-value
end function

The experimental setup, as previously outlined in Chapter 7, involved specific resource requirements from the cluster. Each task was executed using a single 8-core CPU operating at 2.10 GHz, and a 10 GB RAM allocation. Additionally, it made use of one GPU. These configurations were consistently applied across all experiments.

In the tables reporting all the methods F1-score and execution times averages, dark blue cells outline the best 'pure' pseudo-labelling method, while blue cells the best active learning method.

In tables that display Wilcoxon statistics, a green cell signifies that the use of unlabelled data leads to performance enhancement, red cells indicate a decrease in performance, and grey cells represent equivalency.

8.1 Binary Classifiers for Malware Family Classification results

This section will present the results obtained from four distinct machine learning models (RF, SVM, KNN, HGBM) when applied to four different feature inputs (Raw pixels, HOG features, GLCM features, LBP features).

As mentioned above, malware instances originating from the same family tend to exhibit comparable texture features. However, I have observed that certain malware families show an even higher degree of feature similarity. This phenomenon becomes even more evident when employing raw pixel data as input, in contrast to the utilisation of texture feature descriptors. The pixel values across different images within the same class are nearly identical. However, when texture feature descriptors are applied, even though the distinctions are minimal, they remain discernible.This emphasises that texture features can capture subtle distinctions between images, while raw pixel data tend to be highly uniform within the same class. In particular, the malware families that exhibit identical texture features include VBA and Vilsel.

Figures 8.1, 8.2, 8.3, 8.4. 8.5 and 8.6 visually highlight the remarkably similar features exhibited by these two classes. This similarity becomes even more pronounced when using raw pixel values in the classification process as shown in Figures 8.7 and 8.8

	glcm_0	glcm_1	glcm_2	...	glcm_18	glcm_19	Label
7700	0.016765	0.424228	26.420829	...	0.051526	1745.037448	VBA
7701	0.016762	0.424223	26.420829	...	0.051526	1745.032984	VBA
7702	0.016762	0.424223	26.420829	...	0.051526	1745.032984	VBA
7703	0.016765	0.424217	26.421075	...	0.051526	1745.040424	VBA
7704	0.016763	0.424225	26.420829	...	0.051526	1745.034038	VBA
...	...	...	...	...	...	...	...
12097	0.016764	0.424230	26.420706	...	0.051526	1745.035092	VBA
12098	0.016763	0.424215	26.421075	...	0.051526	1745.037014	VBA
12099	0.016765	0.424223	26.420952	...	0.051526	1745.038874	VBA
12100	0.016765	0.424217	26.421075	...	0.051526	1745.040424	VBA
12101	0.016765	0.424242	26.420460	...	0.051526	1745.033914	VBA

Figure 8.1: GLCM features of VBA family

	glcm_0	glcm_1	glcm_2	...	glcm_18	glcm_19	Label
8400	0.110848	0.743283	34.695805	...	0.097538	6136.103354	Vilsel
8401	0.111046	0.743338	34.687685	...	0.097846	6136.013144	Vilsel
8402	0.109093	0.741019	35.094734	...	0.096230	6176.597495	Vilsel
8403	0.111227	0.743343	34.687562	...	0.097710	6136.021762	Vilsel
8404	0.110350	0.742503	34.795706	...	0.097963	6206.880650	Vilsel
...	...	...	...	...	...	...	...
12389	0.110485	0.743227	34.698819	...	0.097381	6136.198710	Vilsel
12390	0.103720	0.740028	34.898314	...	0.090612	6195.414409	Vilsel
12391	0.111040	0.745025	34.462168	...	0.099191	6167.356687	Vilsel
12392	0.110782	0.743226	34.697466	...	0.097454	6136.280551	Vilsel
12393	0.104980	0.740289	34.887734	...	0.091759	6194.902970	Vilsel

Figure 8.2: GLCM features of Vilsel family

	hog_0	hog_1	hog_2	...	hog_48	hog_49	Label
7700	-11.511365	2.663467	0.872738	...	-0.778858	0.148291	VBA
7701	-11.511385	2.663646	0.872407	...	-0.778684	0.148381	VBA
7702	-11.511385	2.663646	0.872407	...	-0.778684	0.148381	VBA
7703	-11.511383	2.663762	0.872127	...	-0.778482	0.148466	VBA
7704	-11.511368	2.663604	0.872501	...	-0.778240	0.148474	VBA
...	...	...	...	...	...	...	...
12097	-11.511363	2.663462	0.872740	...	-0.778869	0.148293	VBA
12098	-11.511386	2.663900	0.871892	...	-0.777865	0.148649	VBA
12099	-11.511374	2.663614	0.872432	...	-0.778670	0.148378	VBA
12100	-11.511383	2.663762	0.872127	...	-0.778482	0.148466	VBA
12101	-11.511337	2.663022	0.873659	...	-0.779422	0.148028	VBA

Figure 8.3: HOG features of VBA family

	hog_0	hog_1	hog_2	...	hog_48	hog_49	Label
8400	-5.249645	-30.959083	-11.970644	...	0.345125	0.388426	Vilsel
8401	-5.250565	-30.963961	-11.967267	...	0.355520	0.384916	Vilsel
8402	-4.311890	-31.078504	-12.315847	...	-1.179034	0.275594	Vilsel
8403	-5.250514	-30.964784	-11.966130	...	0.354692	0.384263	Vilsel
8404	-4.105214	-30.866754	-12.222275	...	-2.203542	0.621032	Vilsel
...	...	...	...	...	...	...	...
12389	-5.247238	-30.963873	-11.968648	...	0.348591	0.386658	Vilsel
12390	-5.865047	-30.579916	-9.581132	...	1.678618	-0.013180	Vilsel
12391	-4.657934	-32.317829	-11.436745	...	-0.607150	1.100923	Vilsel
12392	-5.246108	-30.964686	-11.971768	...	0.357120	0.391320	Vilsel
12393	-5.871463	-30.572383	-9.581126	...	1.664369	0.004687	Vilsel

Figure 8.4: HOG features of Vilsel family

	lbp_0	lbp_1	lbp_2	...	lbp_8	lbp_9	Label
7700	0.146667	0.114868	0.035461	...	0.171082	0.260620	VBA
7701	0.146667	0.114868	0.035461	...	0.171143	0.260620	VBA
7702	0.146667	0.114868	0.035461	...	0.171143	0.260620	VBA
7703	0.146667	0.114868	0.035461	...	0.171082	0.260681	VBA
7704	0.146667	0.114868	0.035461	...	0.171082	0.260620	VBA
...	...	...	...	...	...	...	...
12097	0.146667	0.114868	0.035461	...	0.171143	0.260620	VBA
12098	0.146667	0.114868	0.035461	...	0.171082	0.260681	VBA
12099	0.146667	0.114868	0.035461	...	0.171082	0.260620	VBA
12100	0.146667	0.114868	0.035461	...	0.171082	0.260681	VBA
12101	0.146667	0.114868	0.035461	...	0.171082	0.260620	VBA

Figure 8.5: LBP features of VBA family

	lbp_0	lbp_1	lbp_2	...	lbp_8	lbp_9	Label
8400	0.081665	0.098022	0.056885	...	0.205933	0.139771	Vilsel
8401	0.081543	0.098145	0.056824	...	0.206238	0.139709	Vilsel
8402	0.080566	0.098145	0.058289	...	0.201904	0.138000	Vilsel
8403	0.081482	0.098267	0.056519	...	0.206421	0.139954	Vilsel
8404	0.081848	0.095581	0.058105	...	0.202454	0.140991	Vilsel
...	...	...	...	...	...	...	...
12389	0.081848	0.098022	0.056885	...	0.205566	0.139771	Vilsel
12390	0.085510	0.094482	0.057556	...	0.197083	0.142944	Vilsel
12391	0.081543	0.098877	0.054626	...	0.205078	0.139587	Vilsel
12392	0.081726	0.098083	0.056763	...	0.205872	0.139771	Vilsel
12393	0.085449	0.094299	0.057495	...	0.198730	0.142639	Vilsel

Figure 8.6: LBP features of Vilsel family

	Image	Label
7700	[[0, 0, 1, 0, 0, 4, 5, 0, 1, 2, 1, 2, 3, 2, 0,...	VBA
7701	[[0, 0, 1, 0, 0, 4, 5, 0, 1, 2, 1, 2, 3, 2, 0,...	VBA
7702	[[0, 0, 1, 0, 0, 4, 5, 0, 1, 2, 1, 2, 3, 2, 0,...	VBA
7703	[[0, 0, 1, 0, 0, 4, 5, 0, 1, 2, 1, 2, 3, 2, 0,...	VBA
7704	[[0, 0, 1, 0, 0, 4, 5, 0, 1, 2, 1, 2, 3, 2, 0,...	VBA
...	...	...
12097	[[0, 0, 1, 0, 0, 4, 5, 0, 1, 2, 1, 2, 3, 2, 0,...	VBA
12098	[[0, 0, 1, 0, 0, 4, 5, 0, 1, 2, 1, 2, 3, 2, 0,...	VBA
12099	[[0, 0, 1, 0, 0, 4, 5, 0, 1, 2, 1, 2, 3, 2, 0,...	VBA
12100	[[0, 0, 1, 0, 0, 4, 5, 0, 1, 2, 1, 2, 3, 2, 0,...	VBA
12101	[[0, 0, 1, 0, 0, 4, 5, 0, 1, 2, 1, 2, 3, 2, 0,...	VBA

Figure 8.7: Raw pixels of VBA family

	Image	Label
8400	[[0, 0, 0, 0, 26, 117, 70, 89, 10, 136, 145, 2...	Vilsel
8401	[[0, 0, 0, 0, 26, 117, 70, 89, 10, 136, 145, 2...	Vilsel
8402	[[0, 0, 0, 0, 26, 119, 71, 90, 10, 137, 146, 2...	Vilsel
8403	[[0, 0, 0, 0, 26, 117, 70, 89, 10, 136, 145, 2...	Vilsel
8404	[[0, 0, 0, 0, 26, 119, 71, 90, 10, 137, 146, 2...	Vilsel
...	...	...
12389	[[0, 0, 0, 0, 26, 117, 70, 89, 10, 136, 145, 2...	Vilsel
12390	[[0, 0, 0, 0, 26, 117, 70, 89, 10, 136, 145, 2...	Vilsel
12391	[[0, 0, 0, 0, 26, 117, 70, 89, 10, 136, 145, 2...	Vilsel
12392	[[0, 0, 0, 0, 26, 117, 70, 89, 10, 136, 145, 2...	Vilsel
12393	[[0, 0, 0, 0, 26, 117, 70, 89, 10, 136, 145, 2...	Vilsel

Figure 8.8: Raw pixels of Vilsel family

In the context of multiple binary classifiers, when the features are nearly identical, the classifiers become highly skilled at making accurate predictions for those families. Nevertheless, averaging F1 scores from different classifiers, especially when some families have perfect F1 scores, can significantly impact the overall performance measure. Consequently, the resultant average F1 score might not accurately reflect the effectiveness of the classifiers in handling the entire dataset or all distinct categories.

8.1.1 Random forest results

As the model that uses GLCM features with 100 trees shows overfitting, a more simplified version of the random forest model was employed, utilising 80 trees, with the objective of reducing the complexity of the model and improving generalisation capabilities. Transitioning to a different feature set, such as HOG and LBP features, provides alternative insights: the models using HOG and LBP features did not show overfitting issues even with a higher number of trees; this could suggest that these features might generalise better.

The results presented in Tables 8.1, 8.3, 8.5, 8.7 and 8.9 can be summarised as follows:

- Among all the features, the most effective 'pure' pseudo-labelling technique is the model that uses all pseudo-labels, except for the model that uses GLCM features where the most effective one is the model that repeats the operation of the one with high-confidence labels;
- Active learning stands out as the primary strategy for optimising the utilisation of the labelling budget. However, there is an exception in raw pixel analysis, where the utilisation of unlabelled data doesn't yield discernible benefits;
- Models trained with high-confidence labels consistently deliver the poorest performance, deviating from the expected trend;

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.985	0.631(s)
SL	0.970	0.249(s)
$\underline{SsL}$	0.972	0.974(s)
πSsL	0.968	0.762(s)
$\hat{\pi} SsL$	0.971	1.316(s)
αSsL_l	0.976	0.580(s)
αSsL_o	0.972	0.601(s)
αSsL_h	0.961	0.591(s)
$\alpha^{\pi} SsL_l$	0.973	1.133(s)
$\alpha^{\pi} SsL_o$	0.972	1.156(s)
$\alpha^{\pi} SsL_h$	0.960	1.146(s)

Table 8.1: F1-score and Execution Time average Random Forest with Raw pixel.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.297	-1.042
Best active learning	αSsL_l	0.081	-1.740

Table 8.2: Wilcoxon Statistics Random Forest with Raw Pixels

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.976	0.265(s)
SL	0.954	0.224(s)
$\underline{SsL}$	0.955	0.583(s)
πSsL	0.953	0.575(s)
$\hat{\pi} SsL$	0.955	0.838(s)
αSsL_l	0.963	0.521(s)
αSsL_o	0.963	0.525(s)
αSsL_h	0.947	0.525(s)
$\alpha^{\pi} SsL_l$	0.96	0.886(s)
$\alpha^{\pi} SsL_o$	0.962	0.887(s)
$\alpha^{\pi} SsL_h$	0.95	0.885(s)

Table 8.3: F1-score and Execution Time average RF with LBP features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.341	-0.951
Best active learning	αSsL_o	<0.001	-3.71

Table 8.4: Wilcoxon Statistics RF with LBP features

- The model demanding the most time across all cases is the pseudo-active learning model employing mid-confidence labels. However, for raw pixels and GLCM features, the second pseudo-labelling method utilising high-confidence labels displays similarly high time requirements.

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.99	0.292(s)
SL	0.977	0.218(s)
$\underline{SsL}$	0.977	0.598(s)
πSsL	0.975	0.563(s)
$\hat{\pi} SsL$	0.975	0.838(s)
αSsL_l	0.982	0.51(s)
αSsL_o	0.976	0.516(s)
αSsL_h	0.965	0.515(s)
$\alpha^{\pi} SsL_l$	0.979	0.863(s)
$\alpha^{\pi} SsL_o$	0.976	0.867(s)
$\alpha^{\pi} SsL_h$	0.964	0.863(s)

Table 8.5: F1-score and Execution Time average RF with HOG features .

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.824	-0.222
Best active learning	αSsL_l	0.034	-2.123

Table 8.6: Wilcoxon Statistics RF with HOG features

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.983	0.225(s)
SL	0.963	0.185(s)
$\underline{SsL}$	0.964	0.481(s)
πSsL	0.963	0.482(s)
$\hat{\pi} SsL$	0.964	0.7(s)
αSsL_l	0.969	0.43(s)
αSsL_o	0.971	0.433(s)
αSsL_h	0.956	0.433(s)
$\alpha^{\pi} SsL_l$	0.967	0.729(S)
$\alpha^{\pi} SsL_o$	0.97	0.73(s)
$\alpha^{\pi} SsL_h$	0.959	0.733(s)

Table 8.7: F1-score and Execution Time average RF with GLCM features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\hat{\pi} SsL$	0.202	-1.276
Best active learning	αSsL_o	<0.001	-4.267

Table 8.8: Wilcoxon Statistics RF with GLCM features

The Wilcoxon Statistics presented in Tables 8.2, 8.4, 8.6, 8.8 and 8.10 reveal a consistent pattern in the results. In each case, active learning techniques consistently demonstrate statistically significant improvements. But when it comes to raw pixels, both models perform similarly, showing no clear difference in their effectiveness.

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.993	0.301(s)
SL	0.982	0.229(s)
$\underline{SsL}$	0.982	0.621(s)
πSsL	0.979	0.591(s)
$\hat{\pi} SsL$	0.979	0.878(s)
αSsL_l	0.986	0.53(s)
αSsL_o	0.981	0.536(s)
αSsL_h	0.97	0.535(s)
$\alpha^{\pi} SsL_l$	0.983	0.901(s)
$\alpha^{\pi} SsL_o$	0.982	0.906(s)
$\alpha^{\pi} SsL_h$	0.969	0.902(s)

Table 8.9: F1-score and Execution Time average RF with all features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.801	-0.252
Best active learning	αSsL_l	0.004	-2.888

Table 8.10: Wilcoxon Statistics RF with all features

8.1.2 SVM results

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.935	0.232(s)
SL	0.856	0.011(s)
$\underline{SsL}$	0.854	0.126(s)
πSsL	0.865	0.065(s)
$\hat{\pi} SsL$	0.857	0.141(s)
αSsL_l	0.89	0.025(s)
αSsL_o	0.894	0.029(s)
αSsL_h	0.855	0.025(s)
$\alpha^{\pi} SsL_l$	0.89	0.068(s)
$\alpha^{\pi} SsL_o$	0.89	0.077(s)
$\alpha^{\pi} SsL_h$	0.862	0.071(s)

Table 8.11: F1-score and Execution Time average SVM with LBP features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	πSsL	0.329	-0.976
Best active learning	αSsL_o	0.002	-3.158

Table 8.12: Wilcoxon Statistics SVM with LBP features

The findings extracted from Tables 8.11, 8.13, and 8.15 can be summarised as follows:

- For all the features, the most effective 'pure' pseudo-labelling techniques is the one that uses high-confidence labels;

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.922	0.3(s)
SL	0.869	0.013(s)
$\underline{SsL}$	0.87	0.163(s)
πSsL	0.877	0.076(s)
$\hat{\pi} SsL$	0.872	0.163(s)
αSsL_l	0.886	0.029(s)
αSsL_o	0.889	0.035(s)
αSsL_h	0.863	0.028(s)
$\alpha^{\pi} SsL_l$	0.888	0.078(s)
$\alpha^{\pi} SsL_o$	0.889	0.093(s)
$\alpha^{\pi} SsL_h$	0.873	0.083(s)

Table 8.13: F1-score and Execution Time average SVM with GLCM features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	πSsL	0.001	-3.423
Best active learning	αSsL_o	<0.001	-7.285

Table 8.14: Wilcoxon Statistics SVM with GLCM features

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.923	0.512(s)
SL	0.865	0.019(s)
$\underline{SsL}$	0.865	0.279(s)
πSsL	0.872	0.125(s)
$\hat{\pi} SsL$	0.867	0.273(s)
αSsL_l	0.886	0.047(s)
αSsL_o	0.889	0.055(s)
αSsL_h	0.858	0.046(s)
$\alpha^{\pi} SsL_l$	0.888	0.127(s)
$\alpha^{\pi} SsL_o$	0.886	0.152(s)
$\alpha^{\pi} SsL_h$	0.87	0.137(s)

Table 8.15: F1-score and Execution Time average SVM with all features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	πSsL	<0.001	-3.501
Best active learning	αSsL_o	<0.001	-9.134

Table 8.16: Wilcoxon Statistics SVM with all features

- The best active learning method is in all cases the active learning approach with mid-confidence labels;
- Active learning appears as the standout strategy for optimising the use of the labelling budget when using LBP features; while using GLCM and all features both pseudo-labelling technique and active learning techniques provide improvements;

```
1       0.0
2       0.0
3       0.0
4       0.0
5       0.0
      ...
1823    1.0
1824    1.0
1827    1.0
1830    1.0
1831    1.0
Name: sl_probability, Length: 1263, dtype: float64

Samples with high confidence: 1263
Samples with mid confidence: 0
Samples with low confidence: 0
```

Figure 8.9: Probability estimates for SVM algorithm with HOG features

- Interestingly, models trained with the high-confidence labels consistently deliver the poorest performance;

- There are overfitting issue when dealing with raw pixels and hog features. Figure 8.9 shows the predicted probabilities attributed to each sample for classification. The model demonstrates an extraordinary degree of confidence in its predictions. The probabilities are consistently nearly 0 for the negative class and nearly 1 for the positive class. These results were derived employing both *'linear'* and *'rbf'* kernel;

- In all the cases, the upper bound model is the one requiring the highest time.

The Tables 8.12, 8.14, and 8.16 containing Wilcoxon Statistics highlight that across all instances, the active learning methods consistently exhibit statistically significant enhancements. And for GLCM features and the model with the combined features also pseudo-labelling improves performance.

```
1       0.0
2       0.0
3       0.0
4       0.0
5       0.0
      ...
1823    1.0
1824    1.0
1827    1.0
1830    1.0
1831    1.0
Name: sl_probability, Length: 1263, dtype: float64

Samples with high confidence: 1263
Samples with mid confidence: 0
Samples with low confidence: 0
```

Figure 8.10: Probability estimates for KNN algorithm with LBP features

8.1.3 KNN results

The KNN model shows overfitting issues when dealing with both raw pixel values and features extraction predictors. In particular this challenge manifests prominently when the classification task involves distinguishing two specific malware families: VBA and Vilsel. The Figures 8.10 provide insights into the predicted probabilities assigned by the model to each sample for classification.

It is possible to observe that the model exhibits an exceptionally high level of certainty in its predictions. The probabilities are consistently 0 for the negative class and 1 for the positive class, corresponding only to samples with high confidence.

These outcomes were obtained using both the Euclidean distance and the Manhattan distance with varying values of k, specifically k = 3,5,10,20. Despite the alterations in k and the distance metrics, the model consistently exhibited this remarkable certainty in its predictions.

8.1.4 HGBM results

To address the overfitting problem that the HistGradientBoosting model showed, the following steps were taken:

- Model complexity was reduced by imposing a maximum depth limit on the trees (max_depth=3);
- The maximum number of iterations was reduced to 80;
- The learning rate was decreased to 0.01;
- The implementation of early stopping was also applied.

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.975	4.129(s)
SL	0.937	2.793(s)
$\underline{SsL}$	0.942	6.975(s)
πSsL	0.937	5.801(s)
$\hat{\pi} SsL$	0.939	8.999(s)
αSsL_l	0.929	4.667(s)
αSsL_o	0.950	4.917(s)
αSsL_h	0.911	4.538(s)
$\alpha^{\pi} SsL_l$	0.928	7.461(s)
$\alpha^{\pi} SsL_o$	0.951	7.607(s)
$\alpha^{\pi} SsL_h$	0.911	7.252(s)

Table 8.17: F1-score and Execution Time average GB with raw pixels.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.09	-1.7
Best active learning	$\alpha^{\pi} SsL_o$	0.002	-3.076

Table 8.18: Wilcoxon Statistics GB with raw pixels.

The insights from the Tables 8.19, 8.21, 8.23 and 8.25 are as follows:

- Surprisingly, for most features, utilising all pseudo-labels proves to be the most effective approach among the 'pure' pseudo-labelling methods;

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.955	0.148(s)
SL	0.936	0.123(s)
$\underline{SsL}$	0.936	0.297(s)
πSsL	0.934	0.266(s)
$\hat{\pi} SsL$	0.933	0.401(s)
αSsL_l	0.94	0.242(s)
αSsL_o	0.942	0.254(s)
αSsL_h	0.926	0.243(s)
$\alpha^{\pi} SsL_l$	0.939	0.378(s)
$\alpha^{\pi} SsL_o$	0.938	0.385(s)
$\alpha^{\pi} SsL_h$	0.922	0.38(s)

Table 8.19: F1-score and Execution Time average GB with LBP features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.92	0.1
Best active learning	αSsL_o	0.483	-0.701

Table 8.20: Wilcoxon Statistics GB with LBP features.

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.97	0.22(s)
SL	0.95	0.173(s)
$\underline{SsL}$	0.948	0.408(s)
πSsL	0.948	0.363(s)
$\hat{\pi} SsL$	0.946	0.551(s)
αSsL_l	0.947	0.299(s)
αSsL_o	0.954	0.328(s)
αSsL_h	0.93	0.311(s)
$\alpha^{\pi} SsL_l$	0.946	0.486(s)
$\alpha^{\pi} SsL_o$	0.951	0.492(s)
$\alpha^{\pi} SsL_h$	0.926	0.47(s)

Table 8.21: F1-score and Execution Time average GB with HOG Features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.287	1.065
Best active learning	αSsL_o	0.385	-0.868

Table 8.22: Wilcoxon Statistics GB with HOG Features

- The best active learning method is the one that uses mid-confidence labels, except for the model using raw pixel where the most effective one is the pseudo-active learning model that uses mid-confidence labels;
- Active learning stands out as the most efficient approach to make the best use of the labelling budget only using raw pixels;

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.964	0.188(s)
SL	0.946	0.153(s)
$\underline{SsL}$	0.946	0.35(s)
πSsL	0.944	0.333(s)
$\hat{\pi} SsL$	0.944	0.503(s)
αSsL_l	0.946	0.275(s)
αSsL_o	0.952	0.287(s)
αSsL_h	0.934	0.281(s)
$\alpha^{\pi} SsL_l$	0.946	0.435(s)
$\alpha^{\pi} SsL_o$	0.949	0.433(s)
$\alpha^{\pi} SsL_h$	0.931	0.425(s)

Table 8.23: F1-score and Execution Time average GB with GLCM features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.909	-0.114
Best active learning	αSsL_o	0.194	-1.3

Table 8.24: Wilcoxon Statistics GB with GLCM features

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.978	0.262(s)
SL	0.961	0.175(s)
$\underline{SsL}$	0.959	0.441(s)
πSsL	0.959	0.371(s)
$\hat{\pi} SsL$	0.956	0.57(s)
αSsL_l	0.956	0.3(s)
αSsL_o	0.964	0.327(s)
αSsL_h	0.941	0.303(s)
$\alpha^{\pi} SsL_l$	0.954	0.496(s)
$\alpha^{\pi} SsL_o$	0.962	0.511(s)
$\alpha^{\pi} SsL_h$	0.938	0.488(s)

Table 8.25: F1-score and Execution Time average GB with all features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.414	0.817
Best active learning	αSsL_o	0.273	-1.096

Table 8.26: Wilcoxon Statistics GB with all features

- Notably, models trained on the high-confidence labels consistently exhibit the poorest performance;

- The model requiring the highest execution time is the 'pure' pseudo-labelling model that repeats the operation of the one with high-confidence labels.

However, the Wilcoxon statistics provided in Tables 8.18, 8.20, 8.22, 8.24 and 8.26 show that in all cases using both pseudo-labelling and active learning don't provide any enhancements, except for the raw pixels where active learning is the best strategy.

8.2 Multi-Class Malware Family Classification results

This section presents the same results of the previous one, but in a multi-class classification context. Several experiments were made:

- *'macro'* and *'weighted'* average to calculate metrics
- *'entropy'* and *'max'* for choosing the probability along all the classes probability

Nevertheless, across all scenarios, the outcomes consistently revealed that incorporating unlabelled data did not yield any enhancements. Furthermore, in the context of active learning, it even resulted in a performance decline.

8.2.1 Random Forest results

The Random Forest model in the context of multi-class classification has 100 trees and there is no imposed limit on the maximum depth of the individual decision trees.

Tables 8.27, 8.29, 8.31, 8.33 and 8.35 collectively reveal several key findings:

- Across all scenarios, the most effective 'pure' pseudo-labelling technique is observed in the model utilising all pseudo-labels, except for the raw pixels where the most effective is the model that uses high-confidence pseudo-labels;

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.912	19.38(s)
SL	0.84	3.6(s)
$\underline{SsL}$	0.826	22.889(s)
πSsL	0.829	10.515(s)
$\hat{\pi} SsL$	0.807	20.973(s)
αSsL_l	0.791	4.578(s)
αSsL_o	0.817	4.67(s)
αSsL_h	0.791	4.546(s)
$\alpha^{\pi} SsL_l$	0.781	11.909(s)
$\alpha^{\pi} SsL_o$	0.803	11.93(s)
$\alpha^{\pi} SsL_h$	0.785	11.653(s)

Table 8.27: F1-score and Execution Time average multi-class RF with Raw Pixels.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	πSsL	0.086	1.729
Best active learning	αSsL_o	0.003	2.99

Table 8.28: Wilcoxon Statistics multi-class RF with Raw Pixels

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.825	0.733(s)
SL	0.75	0.313(s)
$\underline{SsL}$	0.75	1.137(s)
πSsL	0.747	0.827(s)
$\hat{\pi} SsL$	0.746	1.344(s)
αSsL_l	0.72	0.639(s)
αSsL_o	0.723	0.693(s)
αSsL_h	0.705	0.717(s)
$\alpha^{\pi} SsL_l$	0.715	1.183(s)
$\alpha^{\pi} SsL_o$	0.719	1.177(s)
$\alpha^{\pi} SsL_h$	0.702	1.176(s)

Table 8.29: F1-score and Execution Time average multi-class RF with LBP features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.915	0.118
Best active learning	αSsL_o	0.001	3.486

Table 8.30: Wilcoxon Statistics multi-class RF with LBP Features

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.885	1.162(s)
SL	0.808	0.348(s)
$\underline{SsL}$	0.808	1.584(s)
πSsL	0.805	1.097(s)
$\hat{\pi} SsL$	0.807	1.966(s)
αSsL_l	0.782	0.75(s)
αSsL_o	0.783	0.714(s)
αSsL_h	0.765	0.713(s)
$\alpha^{\pi} SsL_l$	0.779	1.574(s)
$\alpha^{\pi} SsL_o$	0.78	1.549(s)
$\alpha^{\pi} SsL_h$	0.762	1.548(s)

Table 8.31: F1-score and Execution Time average multi-class RF with GLCM features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.853	-0.186
Best active learning	αSsL_o	<0.001	3.756

Table 8.32: Wilcoxon Statistics multi-class RF with GLCM Features

- The active learning strategy that shows the best performance is the one that involves mid-confidence labels;
- In the most instances, models relying on high-confidence labels exhibit poorer performance;
- The model that demands significant computational time corresponds to the 'pure' pseudo-labelling technique that repeats the operation of the one that uses high-confidence labels;

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.917	2.331(s)
SL	0.85	0.484(s)
$\underline{SsL}$	0.849	2.877(s)
πSsL	0.845	1.534(s)
$\hat{\pi} SsL$	0.833	3.054(s)
αSsL_l	0.807	0.823(s)
αSsL_o	0.826	0.829(s)
αSsL_h	0.807	0.822(s)
$\alpha^{\pi} SsL_l$	0.799	1.901(s)
$\alpha^{\pi} SsL_o$	0.818	1.907(s)
$\alpha^{\pi} SsL_h$	0.803	1.896(s)

Table 8.33: F1-score and Execution Time average multi-class RF with HOG features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.996	0.017
Best active learning	αSsL_o	<0.001	3.992

Table 8.34: Wilcoxon Statistics multi-class RF with HOG Features

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.939	2.319(s)
SL	0.872	0.488(s)
$\underline{SsL}$	0.871	2.855(s)
πSsL	0.86	1.636(s)
$\hat{\pi} SsL$	0.855	3.219(s)
αSsL_l	0.83	0.829(s)
αSsL_o	0.85	0.835(s)
αSsL_h	0.824	0.833(s)
$\alpha^{\pi} SsL_l$	0.819	2.081(s)
$\alpha^{\pi} SsL_o$	0.839	2.111(s)
$\alpha^{\pi} SsL_h$	0.814	2.058(s)

Table 8.35: F1-score and Execution Time average multi-class RF with all features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.791	0.276
Best active learning	αSsL_o	<0.003	2.979

Table 8.36: Wilcoxon Statistics multi-class RF with all features

except for the model that uses raw pixels where the model requiring more time is the one that uses all pseudo-labels.

The Wilcoxon statistics provided in Tables 8.28, 8.30, 8.32, 8.34 and 8.36 show that in all cases using pseudo-labelling does not provide any enhancements, while active learning is detrimental.

8.2.2 SVM results

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.374	22.485(s)
SL	0.242	1.153(s)
$\underline{SsL}$	0.233	14.27(s)
πSsL	0.232	5.461(s)
$\hat{\pi} SsL$	0.205	8.012(s)
αSsL_l	0.182	2.641(s)
αSsL_o	0.189	2.676(s)
αSsL_h	0.181	2.631(s)
$\alpha^{\pi} SsL_l$	0.183	5.671(s)
$\alpha^{\pi} SsL_o$	0.187	5.685(s)
$\alpha^{\pi} SsL_h$	0.179	5.602(s)

Table 8.37: F1-score and Execution Time average multi-class SVM with GLCM features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.319	1.008
Best active learning	αSsL_o	<0.001	4.758

Table 8.38: Wilcoxon Statistics multi-class SVM with GLCM Features

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.428	19.61(s)
SL	0.271	1.055(s)
$\underline{SsL}$	0.263	13.309(s)
πSsL	0.246	5.706(s)
$\hat{\pi} SsL$	0.239	9.673(s)
αSsL_l	0.233	2.335(s)
αSsL_o	0.224	2.365(s)
αSsL_h	0.2	2.37(s)
$\alpha^{\pi} SsL_l$	0.211	9.399(s)
$\alpha^{\pi} SsL_o$	0.215	9.424(s)
$\alpha^{\pi} SsL_h$	0.2	9.366(s)

Table 8.39: F1-score and Execution Time average multi-class SVM with LBP features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.468	0.738
Best active learning	αSsL_l	0.01	2.573

Table 8.40: Wilcoxon Statistics multi-class SVM with LBP Features

Tables 8.37, 8.39, 8.41 and 8.43 present the following observations:

- In all scenarios, the most effective 'pure' pseudo-labelling technique is observed in the model that utilizes all pseudo-labels;

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.863	14.823(s)
SL	0.783	1.172(s)
$\underline{SsL}$	0.787	14.864(s)
πSsL	0.778	6.125(s)
$\hat{\pi} SsL$	0.781	11.947(s)
αSsL_l	0.728	3.237(s)
αSsL_o	0.74	3.237(s)
αSsL_h	0.728	3.148(s)
$\alpha^{\pi} SsL_l$	0.729	5.954(s)
$\alpha^{\pi} SsL_o$	0.739	5.959(s)
$\alpha^{\pi} SsL_h$	0.725	5.866(s)

Table 8.41: F1-score and Execution Time average multi-class SVM with HOG features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.414	-0.817
Best active learning	αSsL_o	<0.001	4.015

Table 8.42: Wilcoxon Statistics multi-class SVM with HOG Features

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.38	35.097(s)
SL	0.229	1.755(s)
$\underline{SsL}$	0.221	22.984(s)
πSsL	0.218	8.593(s)
$\hat{\pi} SsL$	0.194	12.805(s)
αSsL_l	0.18	4.099(s)
αSsL_o	0.184	4.13(s)
αSsL_h	0.169	4.089(s)
$\alpha^{\pi} SsL_l$	0.178	9.731(s)
$\alpha^{\pi} SsL_o$	0.185	9.758(s)
$\alpha^{\pi} SsL_h$	0.177	9.625(s)

Table 8.43: F1-score and Execution Time average multi-class SVM with all features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.454	0.76
Best active learning	$\alpha^{\pi} SsL_o$	<0.001	3.61

Table 8.44: Wilcoxon Statistics multi-class SVM with all features

- The best active learning method is for all the cases the one that uses mid-confidence labels, except in the case of the model using LBP features, where the most effective technique involves low-confidence labels and int the case of the model using all the features, where the most effective is the pseudo-active learning technique that uses mid-confidence labels;
- Models relying on high-confidence labels exhibit poorer performance in most instances.

- The model that demands the most time is consistently the upper bound model in all scenarios, except for the model utilising HOG features, where the pseudo-labelling model using all pseudo-labels takes that position.

The Wilcoxon statistics provided in Tables 8.38, 8.40, 8.42 and 8.44 depict consistent trends across different scenarios. These analyses reveal that the utilisation of pseudo-labelling fails to yield any discernible enhancements across all cases examined. Additionally, the incorporation of active learning appears to have a detrimental effect on the outcomes, undermining the performance rather than providing any notable improvements.

8.2.3 KNN results

The KNN model employed in the multi-class classification scenario is configured with parameter 'k' set to 5 and utilises the Euclidean distance metric.

The Tables 8.45, 8.47, 8.49, 8.51 and 8.53 reveal the following insights:

- For all the scenarios, the best 'pure' active learning technique is the one that uses all the pseudo-labels;
- The active learning technique that proves most effective appears to be the utilisation of mid-confidence labels in three cases (raw pixels, HOG features and combined features) and low-confidence labels in the remaining two;
- Similar to the other models, the methods utilising high-confidence labels consistently showed poorer performance;
- The models demanding more time are consistently the ones combining pseudo-labelling with active learning.

As reflected in the Wilcoxon Statistics outlined in Tables 8.46, 8.48, 8.50, 8.52 and 8.54 the trend aligns with the previously observed patterns in Random Forest and Support Vector Machine

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.757	0.002(s)
SL	0.671	0.001(s)
$\underline{SsL}$	0.672	8.474(s)
πSsL	0.67	8.474(s)
$\hat{\pi} SsL$	0.67	8.476(s)
αSsL_l	0.643	5.853(s)
αSsL_o	0.644	5.853(s)
αSsL_h	0.629	5.853(s)
$\alpha^{\pi} SsL_l$	0.638	18.958(s)
$\alpha^{\pi} SsL_o$	0.638	18.958(s)
$\alpha^{\pi} SsL_h$	0.624	18.958(s)

Table 8.45: F1-score and Execution Time average multi-class KNN with Raw Pixels.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.8	-0.253
Best active learning	αSsL_o	<0.001	3.587

Table 8.46: Wilcoxon Statistics multi-class KNN with Raw Pixels

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.894	0.005(s)
SL	0.806	0.003(s)
$\underline{SsL}$	0.814	0.525(s)
πSsL	0.806	0.525(s)
$\hat{\pi} SsL$	0.809	0.529(s)
αSsL_l	0.776	0.549(s)
αSsL_o	0.778	0.549(s)
αSsL_h	0.754	0.549(s)
$\alpha^{\pi} SsL_l$	0.775	0.949(s)
$\alpha^{\pi} SsL_o$	0.775	0.949(s)
$\alpha^{\pi} SsL_h$	0.756	0.949(s)

Table 8.47: F1-score and Execution Time average multi-class KNN with HOG features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.127	-1.526
Best active learning	αSsL_o	0.001	3.362

Table 8.48: Wilcoxon Statistics multi-class KNN with HOG Features

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.793	0.015(s)
SL	0.692	0.004(s)
$\underline{SsL}$	0.699	0.539(s)
πSsL	0.685	0.534(s)
$\hat{\pi} SsL$	0.691	0.546(s)
αSsL_l	0.66	0.552(s)
αSsL_o	0.659	0.552(s)
αSsL_h	0.64	0.552(s)
$\alpha^{\pi} SsL_l$	0.65	0.979(s)
$\alpha^{\pi} SsL_o$	0.649	0.98(s)
$\alpha^{\pi} SsL_h$	0.634	0.98(s)

Table 8.49: F1-score and Execution Time average multi-class KNN with LBP features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.292	-1.053
Best active learning	αSsL_l	0.001	3.474

Table 8.50: Wilcoxon Statistics multi-class KNN with LBP Features

analyses. In these instances, both active learning and pseudo-labelling follow a similar trajectory to what was observed previously, showcasing a lack of positive impact from pseudo-labelling strategy on the model's efficacy, and a detrimental impact of active learning.

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.783	0.004(s)
SL	0.673	0.003(s)
$\underline{SsL}$	0.678	0.504(s)
πSsL	0.668	0.503(s)
$\hat{\pi} SsL$	0.674	0.507(s)
αSsL_l	0.641	0.54(s)
αSsL_o	0.64	0.54(s)
αSsL_h	0.623	0.54(s)
$\alpha^{\pi} SsL_l$	0.633	0.994(s)
$\alpha^{\pi} SsL_o$	0.631	0.994(s)
$\alpha^{\pi} SsL_h$	0.619	0.994(s)

Table 8.51: F1-score and Execution Time average multi-class KNN with GLCM features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.547	-0.603
Best active learning	αSsL_l	<0.001	3.508

Table 8.52: Wilcoxon Statistics multi-class KNN with GLCM Features

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.809	0.006(s)
SL	0.689	0.003(s)
$\underline{SsL}$	0.693	0.536(s)
πSsL	0.683	0.534(s)
$\hat{\pi} SsL$	0.689	0.54(s)
αSsL_l	0.65	0.537(s)
αSsL_o	0.651	0.537(s)
αSsL_h	0.631	0.537(s)
$\alpha^{\pi} SsL_l$	0.64	0.958(s)
$\alpha^{\pi} SsL_o$	0.641	0.958(s)
$\alpha^{\pi} SsL_h$	0.626	0.958(s)

Table 8.53: F1-score and Execution Time average multi-class KNN with all features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.562	-0.58
Best active learning	αSsL_o	<0.001	3.812

Table 8.54: Wilcoxon Statistics multi-class KNN with all features

8.2.4 HGBM results

The HistGradientBoosting has been set with a maximum of 100 iterations of the boosting process and a learning rate of 0.1.

Tables 8.55, 8.57, 8.59 and 8.61 provide the following observations:

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.818	6.221(s)
SL	0.745	9.286(s)
$\underline{SsL}$	0.749	18.218(s)
πSsL	0.751	20.685(s)
$\hat{\pi} SsL$	0.749	28.59(s)
αSsL_l	0.717	15.884(s)
αSsL_o	0.717	15.933(s)
αSsL_h	0.702	15.925(s)
$\alpha^{\pi} SsL_l$	0.712	29.012(s)
$\alpha^{\pi} SsL_o$	0.713	29.935(s)
$\alpha^{\pi} SsL_h$	0.712	29.905(s)

Table 8.55: F1-score and Execution Time average multi-class GB with LBP features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	πSsL	0.303	-1.03
Best active learning	αSsL_o	<0.001	3.745

Table 8.56: Wilcoxon Statistics multi-class GB with LBP Features

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.878	7.044(s)
SL	0.811	9.239(s)
$\underline{SsL}$	0.814	17.169(s)
πSsL	0.814	20.74(s)
$\hat{\pi} SsL$	0.815	27.298(s)
αSsL_l	0.791	17.105(s)
αSsL_o	0.791	17.042(s)
αSsL_h	0.772	16.748(s)
$\alpha^{\pi} SsL_l$	0.783	30.027(s)
$\alpha^{\pi} SsL_o$	0.786	29.741(s)
$\alpha^{\pi} SsL_h$	0.784	29.64(s)

Table 8.57: F1-score and Execution Time average multi-class GB with GLCM features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\hat{\pi} SsL$	0.489	-0.693
Best active learning	αSsL_o	0.004	2.923

Table 8.58: Wilcoxon Statistics multi-class GB with GLCM Features

- No single dominant model exists for the optimal 'pure' pseudo-labelling technique;
- In most instances, the preferred active learning approach utilizes mid-confidence labels, except in combined features, where the low-confidence label method proves most effective;
- Coherently with the other models, the methods that use high-confidence labels show the poorest performance;

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.913	12.037(s)
SL	0.841	8.663(s)
$\underline{SsL}$	0.848	20.562(s)
πSsL	0.845	22.284(s)
$\hat{\pi} SsL$	0.848	32.726(s)
αSsL_l	0.819	16.949(s)
αSsL_o	0.819	16.993(s)
αSsL_h	0.792	16.585(s)
$\alpha^{\pi} SsL_l$	0.813	33.064(s)
$\alpha^{\pi} SsL_o$	0.815	32.86(s)
$\alpha^{\pi} SsL_h$	0.809	33.077(s)

Table 8.59: F1-score and Execution Time average multi-class GB with HOG features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\hat{\pi} SsL$	0.13	-1.515
Best active learning	αSsL_o	0.006	2.787

Table 8.60: Wilcoxon Statistics multi-class GB with HOG Features

Method	*F1-score*	*Execution time*
$\overline{SL}$	0.938	12.611(s)
SL	0.879	10.158(s)
$\underline{SsL}$	0.884	22.489(s)
πSsL	0.882	23.401(s)
$\hat{\pi} SsL$	0.883	34.856(s)
αSsL_l	0.861	18.17(s)
αSsL_o	0.861	18.063(s)
αSsL_h	0.835	17.506(s)
$\alpha^{\pi} SsL_l$	0.856	35.552(s)
$\alpha^{\pi} SsL_o$	0.857	35.79(s)
$\alpha^{\pi} SsL_h$	0.851	35.118(s)

Table 8.61: F1-score and Execution Time average multi-class GB with all features.

	Method	*p-value*	*z-value*
Best 'pure' pseudo-labelling	$\underline{SsL}$	0.205	-1.267
Best active learning	αSsL_l	0.004	2.878

Table 8.62: Wilcoxon Statistics multi-class GB with all Features

- The models requiring more time are the ones that combine pseudo-labelling and active learning.

The Wilcoxon statistics provided in Tables 8.56, 8.58, 8.60 and 8.62 consistently uphold the observed trends, showing that pseudo-labelling fails to offer enhancements while active learning poses a detrimental impact across these scenarios.

8.3 Summary: Traditional Machine Learning

This section aims to summarise the results obtained from both multiple binary classification and multi-class classification, focusing on highlighting the prevailing trends observed across various experiments and analyses.

Multiple Binary Classifiers

From the analysis of the results of the traditional machine learning algorithms reported in Sections 8.1.1, 8.1.2, 8.1.3 and 8.1.4 it is possible to observe that:

- For the binary classification, the Random Forest model achieved the greatest F1-score, followed by the Gradient Boosting but with a larger execution time. Support Vector Machine has smaller performance and overfitting issues when using HOG features and raw pixels, while K-Nearest Neighbours overfit in any case;
- In the majority of the scenarios, active learning provide a small enhancement on the baseline SL, except for most of the cases regarding Gradient Boosting;
- In Random Forest and Gradient Boosting, the standout performing texture feature is HOG. In both cases, using the combined features improve the performance;
- For both the Random Forest model and GB, using raw pixels displays slightly lower performance compared to the most effective feature, but with a marginally higher execution time for the first one and a considerably higher execution time for the second one;
- Interestingly, active learning and pseudo-active learning methods using high-confidence labels lead to poorest results;
- Employing SMOTE occasionally yields small enhancements. Specifically, its utilisation alongside the Gradient Boosting, where initially the Wilcoxon test indicated a lack of discernible benefits, a notable positive effect emerged, improving the active learning method and augmenting its efficacy.

Multi-class classification

In reviewing the results of the traditional machine learning algorithms as detailed in Sections 8.2.1, 8.2.2, 8.2.3, and 8.2.4, the following trends emerge:

- Random Forest and Gradient Boosting provide the better scores, but GB with a longer execution time;
- SVM performs exceptionally poorly, except when utilizing HOG features where its performance significantly improves;
- Once again, HOG emerges as the superior performer among texture features. There is also a performance enhancement observed upon integrating combined features, except for SVM and KNN;
- As the previous binary scenario, both active learning and pseudo-active learning techniques employing high-confidence labels yield the least favorable outcomes;
- Across all scenarios, active learning proves detrimental, and pseudo-labelling fails to demonstrate any improvements.
- Implementing SMOTE occasionally leads to slight enhancements in the results, although there is not a significant alteration.

Another important aspect to note is that in handling raw pixel data, the considerable dimensionality introduces substantial obstacles, notably in terms of execution time, especially when employing complex models like SVM and GB. In achieving notable results in distinguishing between malware families, the computational overhead due to the high dimensionality of the raw pixel data was particularly pronounced. The models exhibited significantly extended execution times during training and prediction phases. This increase in computational demand derived from the inherent nature of raw pixel data, which inherently translates into a vast number of features.

8.4 Convolutional Neural Network results

The CNN model, despite its success in achieving competitive classification accuracies and effectively discerning between diverse malware families, presented an inherent challenge regarding computational demands. The intricate architecture and depth of the convolutional neural network introduced a substantial computational overhead, due to the iterative process of the framework.

This challenge persisted despite attempts to mitigate complexity, such as changing hyperparameters or simplifying the model architecture significantly. Even with these adjustments, the computational resources required for the CNN remained notably high, posing limitations in terms of scalability and computational efficiency.

8.5 Blended dataset results

Using the blended dataset with the multiple binary classifiers approach does not solve the overfitting issue with the algorithms; this is due to the high degree of similarity within the malware families in the MaleVis dataset.

Therefore, I employed this dataset in the multi-class classification context with the traditional machine learning algorithms and the CNN, in order to increase diversity and improve generalisation.

The results of these experiments for both the traditional machine learning and deep learning scenarios are as follows:

8.5.1 Traditional machine learning algorithms

- In the Random Forest model, the performance improved across all cases except for the model using GLCM features;
- Support Vector Machine model performance showed improvement consistently with the blended dataset in all cases;
- For the Gradient Boosting model, performance remained equivalent or slightly better overall, except for a decrease observed in HOG features;
- K-Nearest Neighbours model performance improved consistently, except for the model using raw pixels.

However, the trend remains consistent with the model trained only on MaleVis dataset. The utilisation of pseudo-labelling failed to yield improvements, while active learning techniques showed counterproductive effects on the outcomes.

8.5.2 Convolutional Neural Network

The incorporation of a combined dataset from MaleVis and Malimg into the CNN model presents several advantageous potentials for enhancing performance compared to the use of the MaleVis dataset alone.

The merging of these datasets offers a wider range of malware samples, encompassing diverse families and variations present in both repositories. This wider diversity increases the model's learning capacity, enabling it to capture a more comprehensive understanding of malware features. The expanded dataset size and diversity augment the model's ability to generalise across various malware families.

However, the inclusion of diverse datasets necessitates meticulous preprocessing to address inconsistencies in labelling, feature representations, and data quality, ensuring the model's robustness and effectiveness in discerning malware characteristics as reported next in Section 9.2.

Chapter 9

Conclusions

In this thesis, the objective was to evaluate the efficacy of a Semi-Supervised Learning (SsL) approach within the domain of image-based malware classification. The evaluation employed the CEF-SsL framework, which had previously been introduced in a related work within the current state-of-the-art. This framework was utilised to conduct a statistically validated benchmark of 9 SsL approach.

To adapt this framework, originally designed for a binary classification task, two distinct approaches were implemented. The first approach employed multiple binary classifiers, each responsible for distinguishing a specific malware family from the benign class. This One-vs-One strategy was iteratively applied for each malware family, and the results from these binary classifiers were aggregated to make predictions for multi-class classification. However, this approach had limitations, particularly in oversimplifying the classification task by not considering potential relationships or shared characteristics between malware families. To address these limitations, the framework was further adapted for multi-class classification, allowing for the simultaneous consideration of all malware families and providing a more comprehensive perspective on the classification task.

The traditional algorithms employed in the evaluation included Random Forest, Gradient Boosting, K-Nearest Neighbours, and Support Vector Machine. These algorithms were trained directly on raw pixel values extracted from grayscale image representations of malware, as well as on texture feature descriptors such as Local Binary Pattern, Histogram of Oriented Gradients, Gray Level Co-occurrence Matrix, and combinations thereof.

While in the context of deep learning, a simple Convolutional Neural Network comprising two convolutional layers followed by two fully connected layers was employed. The network is fine-tuned through the use of a cross-entropy loss function, coupled with the optimisation power of the Adam optimizer.

The MaleVis dataset served as the basis for training and testing these algorithms.

In Chapter 8, the outcomes and analysis of the results are presented, revealing distinctive patterns for each algorithm. In the context of multiple binary classifiers, for the random forest algorithm, it is possible to observe that using raw pixels does not yield benefits from incorporating unlabelled data. This observation may be attributed to the high-dimensionality of raw pixels, posing a challenge for the model to extract meaningful information from unlabelled instances. Additionally, a sophisticated model such as Random Forest may derive more significant advantages when the features effectively capture the underlying patterns or structures in the data.

Conversely, the K-Nearest Neighbours algorithm encounters overfitting issues when utilising both texture features and raw pixels. The local structure captured by KNN may lack sufficient distinctiveness for effective classification.

The results of the Support Vector Machines algorithm indicate its susceptibility to overfitting in high-dimensional feature spaces, such as those involving HOG features and raw pixels. On the contrary, active learning emerges as the most effective strategy in the other case. Using GLCM features and combined features, pseudo-labelling also provides improvements.

As for the Gradient Boosting algorithm, it fails to showcase any discernible benefits when incorporating unlabelled data, except in instances involving raw pixels.

In the multi-class context using active learning is always detrimental: this highlights the fact that in a multi-class context the complexity of the task increases and pseudo-labels introduce errors that are more complex to correct when dealing with multiple classes that have complex interaction, or more complex decision boundaries. Also, trying different metrics, macro and weighted *F1-score*, different model hyperparameters does not improve the results.

The deployment of Convolutional Neural Networks in malware classification, while promising, has encountered challenges as evidenced by the long execution time. Experimentation with hyperparameters, including variations in batch size, learning rate, and image size, did not yield improvements.

9.1 Challenges

Numerous challenges emerged throughout the course of this research, covering various aspects:

Dataset

A significant challenge comes from the MaleVis dataset. Grayscale images often exhibit high degree of similarity within a family, and the model becomes too able to distinguish the samples. This difficultness has already been outlined in related work [32] specifically addressing the similarities among images within the Malimg dataset. This raises concern about overfitting and potential lack of generalisation. This challenge becomes especially critical in the context of real-world scenarios, where diverse samples are essential for robust model training and deployment. To address this challenge, considerations are made towards augmenting the dataset with a more diverse set of images. The goal is to introduce variability that better simulates the complex nature of malware instances in practical settings.

Classifier

Overfitting issues were observed, especially with the KNN algorithm, particularly when dealing with two specific malware families that demonstrated remarkably similar features. This problem is emphasised when using raw pixels, which are more similar than texture feature descriptors. Even if pixel values are similar, variations in texture features can reveal subtle differences in patterns, providing more discriminative information. In fact, as can be seen from the results, the efficiency of the classification process is highly related to the effectiveness of the feature extraction techniques, which is another problem in this context. Changing the hyperparameter or the image size did not solve this issue.

Execution Time

Execution time remains a significant challenge in this work, particularly when employing algorithms such as SVM and Gradient Boosting within a high-dimensional space. The complexity further amplifies due to the iterative nature of the framework, necessitating various methods and conducting numerous iterations, which inevitably intensifies computational demands.

Similar concerns extend to Convolutional Neural Networks as well. While exceptionally potent in their predictive capabilities, CNNs involve extensive computations.

9.2 Future Works

Combining Diverse Malware Datasets

Expanding the dataset used for training and evaluation is a critical step to address the limitations associated with the relatively small size of the MaleVis data set and the homogeneity within the same family of images. By incorporating a more extensive and diverse dataset, researchers can introduce a wider range of scenarios and instances, making the model more resilient to variations and better preparing it for real-world challenges. However, it is essential to carefully curate and annotate the additional data to ensure its quality and relevance to the problem. Moreover, considerations should be given to potential biases introduced by the new dataset and how they might impact the model's performance.

The absence of a standardised convention to name malware families further complicates the combination of diverse datasets. Different datasets often employ distinct labelling conventions or taxonomies to categorise malware. Integrating these labels is critical to creating a unified and coherent dataset, necessitating efforts to map or standardise the diverse family names.

Moreover, disparities in the sets of features or attributes used to describe malware samples across different datasets pose challenges during integration. Variations in structures, formats, and metadata further increase the complexity. Proper alignment of these differences is essential to ensure a consistent and compatible dataset for meaningful analysis and model development.

Multi-modal data

The exploration of features beyond image-based characteristics could be useful for enhancing the model's understanding of malware samples. Integrating alternative features, such as behavioural patterns, network activity, or code structures, can provide a more comprehensive view of malware, capturing aspects that may not be evident in image representations alone. However, the challenge lies in selecting and extracting meaningful features that complement the image data. Careful consideration should be given to feature engineering techniques and the incorporation of multi-modal data.

Concept drift considerations

Incorporating timestamped data into the framework offers a valuable approach to investigate the impact of concept drift on machine learning models. By considering the temporal dimension into the train-test split, it is possible to evaluate the performance of the model over time.

However, the main challenge encountered in this work is related to the acquisition of time-stamped data, particularly for benign samples. Although some data sets offer time-stamped malware binaries, the unavailability of time-stamped benign files poses significant challenges.

Moreover, handling timestamps in the context of machine learning for cybersecurity comes with its own set of challenges. The timestamp approach is a critical issue, as it is not straightforward to determine with absolute precision and reliability. For example, in VirusTotal each file may have the following timestamps associated:

- `creation_date`: extracted when possible from the file's metadata. Indicates when it was built or compiled;
- `first_submission_date`: date when the file was first seen in VirusTotal;
- `last_analysis_date`: most recent scan date;
- `last_modification_date`: date when the object itself was last modified;
- `last_submission_date`: most recent date the file was posted to VirusTotal.

The choice of timestamp methodology plays a crucial role in influencing the overall performance of concept drift models. While these timestamps are sourced from an external service, making them resistant to manipulation, certain issues persist. For instance, the creation date can be faked by malware creators. Additionally, the first submission date and the last submission date are vulnerable to significant delays and temporal misalignments due to users being required to submit files for analysis.

Appendix A

User Manual

The original CEF-SsL framework is available at https://github.com/hihey54/CEF-SsL.

A.1 System Requirements

This program is compatible with all the operating systems. It requires Python 3.10 or later. Additionally, the following dependencies need to be installed:

- Numpy
- Scipy
- scikit-learn
- scikit-image
- imbalanced-learn
- pandas
- Opencv-python
- Matplotlib
- Pytorch

You can install these dependencies using the following command in your terminal:

```
$ pip install <library>
```

Replace $<$ *library* $>$ with the name of the library you want to install.

A.2 Installation

The framework is available at https://github.com/MLetiziaColangelo/CEF-SsL-MCC. You can download it with the following command:

```
$ git clone https://github.com/MLetiziaColangelo/CEF-SsL-MCC.git
```

A.3 Configuration

Before executing the framework it is essential to configure it. According to your needs, you can specify various parameters:

classification_type: It specifies whether the malware classification framework is executed with a direct multi-class classification approach or through the use of multiple binary classifiers;

classifier_name: A string specifying the type of classifier you want to use, e.g., 'all' for executing all available classifiers comprehensively, 'svm' (Support Vector Machine), 'knn' (K-Nearest Neighbors), 'rf' (Random Forest), or 'gb' (HistGradientBoosting);

feature_name: A string indicating the feature extraction method, e.g., 'lbp' (Local Binary Pattern), 'hog' (Histogram of Oriented Gradients), 'glcm' (Gray-Level Co-occurrence Matrix), or 'combined' to use a combination of them.

Use command-line options to specify the desired settings. The available options are:

- `--classification`:
  - `multiclass`: Run multiclass classification;
  - `binary`: Run binary classification.
- `--features`:
  - `lbp`: Use lbp features for classification;
  - `glcm`: Use glcm features for classification;
  - `hog`: Use hog features for classification;
  - `combined`: Use all the features for classification;
  - `raw`: Use raw pixels directly for classification;
- `--classifier`:
  - `all`: Use all the classifiers;
  - `knn`: Use only the K-Nearest Neighbors classifier;
  - `svm`: Use only the Support Vector Machine classifier;
  - `gb`: Use only the HistGradientBoosting classifier;
  - `rf`: Use only the Random Forest classifier;

You can execute your script from the command line by combining these options. For example:

```
$ python main.py --classification multiclass --features lbp --classifier rf
```

To run multi-class classification with lbp features data using the Random Forest classifier. Or:

```
$ python main.py --classification binary --features raw --classifier all
```

To run binary classification with raw pixels using all the classifiers.

A.4 Result

The function outputs evaluation metrics such as F1-score, and execution times for each combination of classifier and feature extraction method.

Moreover, it will show the Wilcoxon RankSum test comparing the baseline supervised learning vs. the pure semi-supervised learning and the baseline supervised learning vs. the best active learning method.

The averages of the F1-scores and execution times for each model are plotted and saved in the **f1_score_execution_time/** directory, while the results of the Wilcoxon tests for each methods are saved in the **wilcoxon_text/** directory. These directories can be found within the **result/output_images_multiclass/** folder for multi-class classification and the **result/output_images_binary/** folder for binary classification.

Appendix B

Developer Manual

The CEF-SsL-MCC project directory has the following structure:

```
CEF-SsL-MCC/
├── data/
│   ├── malware_dataset.csv
│   └── benign_dataset.csv
├── results/
│   ├── output_images_multiclass/
│   │   ├── f1_score_execution_time/
│   │   │   └── ...
│   │   └── wilcoxon_test/
│   │       └── ...
│   └── output_images_binary/
│       ├── f1_score_execution_time/
│       │   └── ...
│       └── wilcoxon_test/
│           └── ...
├── utils/
│   ├── functions
│   ├── feature_extraction.py
│   └── plotting_functions.py
├── classification/
│   ├── features_classification.py
│   └── rawpixels_classification.py
├── main.py
└── README.md
```

Here's an overview of each directory:

1. **data/**: this directory contains the datasets in CSV format, that associates each image path to the family label;
2. **results/**: this directory includes images generated plotting average values for f-scores and execution time for each model. Additionally, this directory contains the results of the Wilcoxon test, which is used to compare the performance of different methods. These results consider both direct multi-class classification and the utilisation of multiple binary classifier for multi-class classification;
3. **utils/**: this folder contains functions used throughout the project for both multi-class and binary classification scenarios. It also includes the feature extraction code and the functions to plot both Wilcoxon test statistics table and the table containing the average values of *F1-score* and *execution time* for each method;

4. **classification/**: this directory contains the main functions for different scenarios. These functions handle both direct multi-class classification and using multiple binary classifiers for multi-class classification. Furthermore, they are subdivided to manage malware classification with both feature extraction and with raw pixels directly as input;

5. `main.py` is the main entry point for running the framework.

B.1 List of Functions

In this section, I will provide a detailed overview of the functions in the Utils directory:

- `feature_extraction.py`: this file contains various functions to extract the different image feature descriptors I employed in my project;
- `plotting_functions.py`: this file consists of two functions, one for plotting the table of Wilcoxon statistics and another for displaying the table of average values for *F1-score* and *execution time* for each method implemented in the framework;
- `functions.py`: this is derived from the original framework, it contains general functions used in the algorithm, including data preprocessing. For the model training and testing, as well as active learning, it handles both binary and multi-class scenario. For the first case, it is modified to handle the training One-vs-One between each specific malware family and the benign class; for the second one it is modified to consider all the classes simultaneously.

B.1.1 feature_extraction.py

B.1: Function 1

```
def extract_features_from_dataframe(df, feature):

    This function iterates over the dataframe and calls the feature-specific
        extraction function based on the input feature type.

    :param image: Dataframe containing the image paths and their
        corresponding labels.
    :param feature: Feature to extract
    :return: Features extracted
```

B.2: Function 2

```
def extract_lbp_features(image):

    This function extracts the LBP features and then computes the histogram
        of LBP features.

    :param image: Grayscale image.
    :return: LBP histogram.
```

B.3: Function 3

```
def extract_hog_features(image):

    This function extracts the HOG features.

    :param image: Grayscale image.
    :return: HOG features.
```

B.4: Function 4

```
def extract_pca_features(hog, n_components):

    This function applies PCA for dimensionality reduction to the HOG
        features.

    :param image: HOG features.
    :param n_components: number of components to initilise the PCA object
    :return: PCA features.
```

B.5: Function 5

```
def extract_glcm_features(image):

    This function extracts the GLCM features.

    :param image: Grayscale image.
    :return: GLCM features.
```

B.1.2 plotting_functions.py

B.6: Function 6

```
def plot_average(avg_f1, avg_time):

    This function creates a table containing the average values for f1-score
        and execution time for each method.

    :param avg_f1: List of the average f1-score for all methods.
    :param feature: List of the average execution times for all methods.
    :return: None
```

B.7: Function 7

```
def plot_wilcoxon(best_pseudo, best_active, pseudo_p, pseudo_z, active_p,
    active_z):

    This function creates a table for the Wilcoxon statistics comparing the
        baseline Supervised Model with the best "pure" pseudo-learning model
        and the best active learning model.

    :param best_pseudo: The best "pure" pseudo-learning model.
    :param best_active: The best active learning model.
    :param pseudo_p: The p-value of the best "pure" pseudo-learning model.
    :param pseudo_z: The z-value of the best "pure" pseudo-learning model.
    :param active_p: The p-value of the best active learning model.
    :param active_z: The z-value of the best active learning model.
    :return: None
```

B.1.3 functions.py

As already mentioned, the content of this file is based on the original framework. The following functions have been added or modified to align with my objectives, while the remaining functions that I am not reporting in this section remain unchanged. The functions `active_learning_binary` and `active_learning_multiclass` share identical structures to the original ones except for the train_test functions they invoke.

B.8: Function 8

```
def load_and_preprocess_image(image_path):

    This function loads the image using OpenCV converting it to grayscale,
        and then resizes it to the desired dimension.

    :param image: The image path extracted from the CSV dataset file.
    :return: NumPy array representing the loaded image.
```

B.9: Function 9

```
def convert(df):

    The purpose of the function is to load and preprocess each image from a
        dataframe containing the image path and the corresponding label. For
        images that are loaded successfully, it creates a new DataFrame with
        columns "Image" and "Label" and appends the preprocessed image along
        with its label.

    :param image: The dataframe containing the path of an image along with
        its label.
    :return: New dataframe containing the loaded image and the corresponding
        label
```

B.10: Function 10

```
def train_test_binary(train_df, train_labels, test_df, test_labels, messages,
    base_clf):

    This function performs training and testing and compute different
        performance metrics.

    :param train_df: The dataframe containing the training data.
    :param train_labels: Numpy array containing the labels associated with
        the training samples.
    :param test_df: The dataframe containing the evaluation data.
    :param test_labels: Numpy array containing the labels associated with the
        evaluation samples.
    :param message: Depending on this variable, the function will print
        different informations.
    :param base_clf: The classifier
    :return: Sets of classification performance metrics.
```

B.11: Function 11

```
def train_test_multiclass(train_df, train_labels, test_df, test_labels,
    messages, base_clf):

    This function performs training and testing and compute different
        performance metrics. It is the same as the above function but in a
        multi-class classification scenario. So the performance metrics are
        the weighted means of those values.

    :param train_df: The dataframe containing the training data.
    :param train_labels: Numpy array containing the labels associated with
        the training samples.
    :param test_df: The dataframe containing the evaluation data.
    :param test_labels: Numpy array containing the labels associated with the
        evaluation samples.
```

```
    :param message: Depending on this variable, the function will print
        different informations.
    :param base_clf: The classifier
    :return: Sets of classification performance metrics.
```

B.12: Function 12

```
def assign_confidence_binary(df, high_confidence_window,
    low_confidence_window, probability_column_name = "Probability",
    confidence_column_name = "Confidence", debug=True, split=True):

    This function computes the range boundaries for the numerical confidence
        scores. Based on these values, it associates each probability value
        with a confidence score among "high", "mid" and "low".

    :param df: The dataframe containing the probability values for each class
        associated to the images.
    :param high_confidence_window: value to define the range boundaries of
        the confidence score.
    :param low_confidence_window: value to define the range boundaries of the
        confidence score.
    :param probability_column_name: The column name of the dataframe that
        contain the probability values.
    :param confidence_column_name: The column name of the dataframe in which
        the confidence score will be stored.
    :param debug: Parameter to print different informations.
    :param split: If it is set to True, the function will create three
        separate DataFrames: df_high, df_mid, and df_low. Each DataFrame
        contains the rows of the original DataFrame df that correspond to
        data points falling into the respective confidence categories (high,
        mid, or low).
    :return: The dataframe containing the new column "confidence" and the
        three dataframes corresponding to the samples with high, mid and low
        confidence.
```

B.13: Function 13

```
def assign_confidence_multiclass(df, high_confidence_window,
    low_confidence_window, probability_column_name = "Probability",
    confidence_column_name = "Confidence", debug=True, split=True):

    This function is the same of the one described earlier, with the key
        difference that it is tailored for multi-class classification
        scenarios. In this context, it identifies the maximum probability
        value across all classes and assigns a corresponding confidence score.

    :param df: The dataframe containing the probability values for each class
        associated to the images.
    :param high_confidence_window: value to define the range boundaries of
        the confidence score.
    :param low_confidence_window: value to define the range boundaries of the
        confidence score.
    :param probability_column_name: The column name of the dataframe that
        contain the probability values.
    :param confidence_column_name: The column name of the dataframe in which
        the confidence score will be stored.
    :param debug: Parameter to print different informations.
```

```
    :param split: If it is set to True, the function will create three
        separate DataFrames: df_high, df_mid, and df_low. Each DataFrame
        contains the rows of the original DataFrame df that correspond to
        data points falling into the respective confidence categories (high,
        mid, or low).
    :return: The dataframe containing the new column "confidence" and the
        three dataframes corresponding to the samples with high, mid and low
        confidence.
```

B.14: Function 14

```
def active_learning_binary_raw(confidences_df, baseTrain_df, validation_df,
    trials = 5, ssl = True, samples=50, label_name = "label", base_clf=None,
    messages=0):

    The function is designed for active learning with multiple trials, where
        samples are selected based on confidence values and integrated into
        the training process to improve model performance. Differently from
        the versions with texture features, before invoking train_test
        function the raw pixels are reshaped to the correct form.

    :param confidences_df: The dataframe containing the samples associated to
        a specific confidence value (low, mid, high).
    :param baseTrain_df: The dataframe containing the labelled samples.
    :param validation_df: The dataframe used for validation.
    :param trials: Numbers of trials for active learning.
    :param ssl: If set to true, the model uses both labeled and unlabeled
        data to learn.
    :param samples: Number of samples selected from the confidence_df.
    :param label_name: The column name associated to the label.
    :param clf: The classifier.
    :param messages: For debugging messages.
    :return: Precision, recall, F1-score, and training time.
```

B.2 features_classification.py.

It includes the implementation of the framework evaluating and comparing the performance of different classifiers and feature extraction methods when applied to a specific dataset. This updated version of the framework is tailored for handling two approaches: (i) the original strategy, that iterates over each malicious family, effectively treating the problem as a series of binary classifications; (ii) a multi-class classification task addressing the challenge of classifying instances into multiple categories simultaneously.

B.3 rawpixel_classification.py.

Like the previous one, but the input is the image directly as raw pixels without any feature extraction method. This is done by flattening each image from a 2D array into a 1D array.

B.4 main.py

The entry point of the framework.

Bibliography

[1] SonicWall, "2023 SonicWall Cyber Threat Report", https://www.sonicwall.com/2023-cyber-threat-report/

[2] NIST, "Malware definition", https://csrc.nist.gov/glossary/term/malware

[3] Cisco, "What is the intent of malware?", https://www.cisco.com/site/us/en/products/security/what-is-malware.html

[4] Kaspersky, "What are the different types of malware?", https://www.kaspersky.com/resource-center/threats/types-of-malware

[5] S. Katzenbeisser, J. Kinder, and H. Veith, "Malware Detection", Encyclopedia of Cryptography and Security (H. C. A. van Tilborg and S. Jajodia, eds.), pp. 752–755, Springer, 2011, DOI 10.1007/978-1-4419-5906-5_838

[6] R. Tahir, "A Study on Malware and Malware Detection Techniques", International Journal of Education and Management Engineering, vol. 8, March 2018, pp. 20–30, DOI 10.5815/ijeme.2018.02.03

[7] MalwareBytes, "Signature", https://www.malwarebytes.com/glossary/signature

[8] O. A. 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

[9] W. Liu, P. Ren, K. Liu, and D. Hai-xin, "Behavior-Based Malware Analysis and Detection", 2011 First International Workshop on Complexity and Data Mining, 2011, pp. 39–42, DOI 10.1109/IWCDM.2011.17

[10] K. Brezinski and K. Ferens, "Metamorphic Malware and Obfuscation -A Survey of Techniques, Variants and Generation Kits", Secur. Commun. Netw., October 2021, DOI 10.13140/RG.2.2.19702.52802

[11] MalwareBytes, "Obfuscation", https://www.malwarebytes.com/glossary/obfuscation

[12] S. Sahay, A. Sharma, and H. Rathore, "Evolution of Malware and Its Detection Techniques", Information and Communication Technology for Sustainable Development, July 2020, DOI /10.1007/978-981-13-7166-0_14

[13] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste, "Malware Dynamic Analysis Evasion Techniques: A Survey", ACM Comput. Surv., vol. 52, November 2019, DOI 10.1145/3365001

[14] D. Gibert, C. Mateu, and J. Planes, "The rise of machine learning for detection and classification of malware: Research developments, trends and challenges", Journal of Network and Computer Applications, vol. 153, 2020, p. 102526, DOI 10.1016/j.jnca.2019.102526

[15] Crowdstrike, "Machine learning for cybersecurity", https://www.crowdstrike.com/cybersecurity-101/machine-learning-cybersecurity

[16] IBM, "Machine learning", https://www.ibm.com/topics/machine-learning

[17] R. Pugliese, S. Regondi, and R. Marini, "Machine learning-based approach: global trends, research directions, and regulatory standpoints", Data Science and Management, vol. 4, 2021, pp. 19–29, DOI 10.1016/j.dsm.2021.12.002

[18] UC Berkeley, "Machine learning process", https://ischoolonline.berkeley.edu/blog/what-is-machine-learning

[19] O. Chapelle, B. Schölkopf, and A. Zien, "Semi-Supervised Learning", Adaptive computation and machine learning, MIT Press, September 2006

[20] B. Settles, "Active Learning Literature Survey", Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009

[21] IBM, "Overview of classification in machine learning", https://community.ibm.com/community/user/ibmz-and-linuxone/blogs/subhasish-sarkar1/2020/04/04/overview-of-classification-in-machine-learning
[22] Javatpoint, "Support vector machine algorithm", https://www.javatpoint.com/machine-learning-support-vector-machine-algorithm
[23] IBM, "K-nearest neighbors", https://www.ibm.com/topics/knn
[24] Analytics Vidhya, "Naive bayes", https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/#What_Is_the_Naive_Bayes_Algorithm?
[25] IBM, "Artificial neural network", https://www.ibm.com/topics/neural-networks
[26] S. Vajapeyam, "Understanding Shannon's Entropy metric for Information", arXiv preprint arXiv:1405.2061, March 2014, DOI 10.48550/arXiv.1405.2061
[27] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware Images: Visualization and Automatic Classification", Proceedings of the 8th International Symposium on Visualization for Cyber Security, New York, NY, USA, 2011, DOI 10.1145/2016904.2016908
[28] R. Sihwail, K. Omar, and K. A. Zainol Ariffin, "An Effective Memory Analysis for Malware Detection and Classification", Computers, Materials and Continua, vol. 67, February 2021, pp. 2301–2320, DOI 10.32604/cmc.2021.014510
[29] N. Z. Gorment, A. Selamat, L. K. Cheng, and O. Krejcar, "Machine Learning Algorithm for Malware Detection: Taxonomy, Current Challenges and Future Directions", IEEE Access, 2023, pp. 1–1, DOI 10.1109/ACCESS.2023.3256979
[30] C. V. Bijitha and H. V. Nath, "On the Effectiveness of Image Processing Based Malware Detection Techniques", Cybernetics and Systems, vol. 53, no. 7, 2022, pp. 615–640, DOI 10.1080/01969722.2021.2020471
[31] T. T. Son, C. Lee, H. Le-Minh, N. Aslam, M. Raza, and N. Q. Long, "An Evaluation of Image-Based Malware Classification Using Machine Learning", Advances in Computational Collective Intelligence (M. Hernes, K. Wojtkiewicz, and E. Szczerbicki, eds.), Cham, 2020, pp. 125–138, DOI 10.1007/978-3-030-63119-2_11
[32] B. T. Hammad, N. Jamil, I. T. Ahmed, Z. M. Zain, and S. Basheer, "Robust Malware Family Classification Using Effective Features and Classifiers", Applied Sciences, vol. 12, no. 15, 2022, DOI 10.3390/app12157877
[33] Y. Ding, X. Zhang, B. Li, J. Xing, Q. Qiang, Z. Qi, M. Guo, S. Jia, and H. Wang, "Malware Classification Based on Semi-Supervised Learning", Science of Cyber Security (C. Su, K. Sakurai, and F. Liu, eds.), Cham, 2022, pp. 287–301, DOI 10.1007/978-3-031-17551-0_19
[34] T. Gao, L. Zhao, X. Li, and W. Chen, "Malware detection based on semi-supervised learning with malware visualization", Mathematical Biosciences and Engineering, vol. 18, no. 5, 2021, pp. 5995–6011, DOI 10.3934/mbe.2021300
[35] R. Shu, T. Xia, H. Tu, L. Williams, and T. Menzies, "Reducing the Cost of Training Security Classifier (via Optimized Semi-Supervised Learning)", arXiv preprint arXiv:2205.00665, 2022, DOI 10.48550/arXiv.2205.00665
[36] J. Koza, M. Krcál, and M. Holena, "Two Semi-supervised Approaches to Malware Detection with Neural Networks", Proceedings of the 20th Conference Information Technologies - Applications and Theory (ITAT 2020), Slovakia, 2020, pp. 176–185
[37] S. Abdelmonem, S. Seddik, R. El-Sayed, and A. S. Kaseb, "Enhancing Image-Based Malware Classification Using Semi-Supervised Learning", 2021 3rd Novel Intelligent and Leading Emerging Sciences Conference (NILES), 2021, pp. 125–128, DOI 10.1109/NILES53778.2021.9600511
[38] S. Wang, Q. Wang, Z. Jiang, X. Wang, and R. Jing, "A Weak Coupling of Semi-Supervised Learning with Generative Adversarial Networks for Malware Classification", 2020 25th International Conference on Pattern Recognition (ICPR), 2021, pp. 3775–3782, DOI 10.1109/ICPR48806.2021.9412832
[39] G. Apruzzese, P. Laskov, and A. Tastemirova, "SoK: The impact of unlabelled data in cyberthreat detection", 2022 IEEE 7th European Symposium on Security and Privacy, June 2022, DOI 10.1109/eurosp53844.2022.00010
[40] G. Andresini, F. Pendlebury, F. Pierazzi, C. Loglisci, A. Appice, and L. Cavallaro, "INSOMNIA: Towards Concept-Drift Robustness in Network Intrusion Detection", Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security, New York, NY, USA, 2021, pp. 111–122, DOI 10.1145/3474369.3486864

[41] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: Eliminating experimental bias in malware classification across space and time", 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, August 2019, pp. 729–746

[42] A. Duby, T. Taylor, G. Bloom, and Y. Zhuang, "Evaluating Feature Robustness for Windows Malware Family Classification", 2022 International Conference on Computer Communications and Networks (ICCCN), 2022, pp. 1–10, DOI 10.1109/ICCCN54977.2022.9868914

[43] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro, "Transcend: Detecting Concept Drift in Malware Classification Models", 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, 2017, pp. 625–642

[44] L. Yang, W. Guo, Q. Hao, A. Ciptadi, A. Ahmadzadeh, X. Xing, and G. Wang, "CADE: Detecting and Explaining Concept Drift Samples for Security Applications", USENIX Security Symposium, 2021

[45] F. Barbero, F. Pendlebury, F. Pierazzi, and L. Cavallaro, "Transcending Transcend: Revisiting Malware Classification with Conformal Evaluation", arXiv preprint arXiv:2010.03856, 2020, DOI 10.48550/arXiv.2010.03856

[46] H. Kaur, H. S. Pannu, and A. K. Malhi, "A Systematic Review on Imbalanced Data Challenges in Machine Learning: Applications and Solutions", ACM Comput. Surv., vol. 52, August 2019, DOI 10.1145/3343440

[47] S. Datta and G. Satten, "Rank-Sum Tests for Clustered Data", Journal of the American Statistical Association, vol. 100, February 2005, pp. 908–915, DOI 10.1198/016214504000001583

[48] R. E. Harang and E. M. Rudd, "SOREL-20M: A Large Scale Benchmark Dataset for Malicious PE Detection", arXiv preprint arXiv:2012.07634, 2020, DOI 10.48550/arXiv.2012.07634

[49] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft Malware Classification Challenge", arXiv preprint arXiv:1802.10135, 2018, DOI 10.48550/arXiv.1802.10135

[50] L. Yang, A. Ciptadi, I. Laziuk, A. Ahmadzadeh, and G. Wang, "BODMAS: An Open Dataset for Learning based Temporal Analysis of PE Malware", 2021 IEEE Security and Privacy Workshops (SPW), 2021, pp. 78–84, DOI 10.1109/SPW53761.2021.00020

[51] H. S. Anderson and P. Roth, "EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models", arXiv preprint arXiv:1804.04637, 2018, DOI 10.48550/arXiv.1804.04637

[52] Malevis, "Malevis dataset", https://web.cs.hacettepe.edu.tr/~selman/malevis/

[53] D. Noever and S. E. M. Noever, "Virus-MNIST: A Benchmark Malware Dataset", arXiv preprint:2103.00602, 2021, DOI 10.48550/arXiv.2103.00602

[54] A. Oliveira, "Malware Analysis Datasets: Raw PE as Image", IEEE Dataport, 2019, DOI 10.21227/8brp-j220

[55] VirusShare, "VirusShare", https://virusshare.com/about

[56] A. Guerra-Manzanares, H. Bahsi, and S. Nömm, "KronoDroid: Time-based Hybrid-featured Dataset for Effective Android Malware Detection and Characterization", Computers & Security, vol. 110, 2021, p. 102399, DOI 10.1016/j.cose.2021.102399

[57] J. Yan, X. Jia, L. Ying, J. Yan, and P. Su, "Understanding and Mitigating Label Bias in Malware Classification: An Empirical Study", 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS), 2022, pp. 492–503, DOI 10.1109/QRS57517.2022.00057

[58] "Computational resources provided by hpc@polito", http://www.hpc.polito.it/

[59] "Google Colaboratory", https://colab.google/

[60] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python", Journal of Machine Learning Research, vol. 12, 2011, pp. 2825–2830

[61] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library", arXiv preprint arXiv:1912.01703, 2019, DOI 10.48550/arXiv.1912.01703

[62] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique", Journal of Artificial Intelligence Research, vol. 16, June 2002, pp. 321–357, DOI 10.1613/jair.953