POLITECNICO DI TORINO

Master degree course in Ingegneria Informatica

Master Degree Thesis

Machine Learning for malware characterization and identification

Supervisor
prof. Antonio Lioy
prof. Andrea Atzeni

Candidate
Marco Saracino

March 2023

Summary

Nowadays, one of the most important threats that needs to be addressed is malware. Malicious programs have evolved over time, becoming more numerous and complex. Zero-day malwares are the new malware that are already widespread on the Internet but have not yet been identified. Traditional signature-based malware detection systems fail to detect these new malicious files because they have not yet been analyzed, so the systems will not have a valid signature with which to identify them and will cause false negatives when placed under examination. To identify and classify malware without the need of the malware signatures, I tried using different machine learning techniques to understand which algorithm was best suited for the task.

First, datasets were sought that were suitable for my task, and then the available malware had to be analyzed to see what features could be extracted. These features were then adapted to build sensible data structures to be given as input to the selected algorithms. Then four machine learning algorithms were selected to be used for testing. In order to choose the algorithms, a study of the state of the art was made, the results obtained from the different research already done on this type of search were compared, and it was determined which four algorithms were the most promising. During the state-of-the-art study, it was noted that there were few features extracted from the datasets per search. Usually, in fact, the authors were going to extract one or two features from the malware to be used with a single machine learning algorithm. I therefore decided to use a different approach. I set as my goal to extract as many available features as possible from my dataset and try to employ them with my chosen algorithms. In this way, I was able to conclude what was the best algorithm to use in malware detection and classification for each feature.

The features chosen were binary file size, n-grams of bytes, n-grams of opcodes, count of occurrences of each individual opcode, entropy, n-grams of APIs, and the check for the presence of each individual API function. The algorithms chosen were Random Forest, K-nearest neighbors, Support Vector Machine and Gradient Boosting Classifier. The results showed that Random Forest and Gradient Boosting Classifier algorithms perform better in terms of accuracy. In addition, SVMs by performing a training phase for each class in the dataset take a long time to be ready to perform for the learning phase. The n-grams of bytes was the feature with which the algorithms performed most promisingly, having said that the n-grams of opcodes also performed excellently. To confirm that everything was generalizable for any dataset, the same procedure was tried again with another dataset, obtaining very similar results. The results are very significant, leading one to think that the use of machine learning algorithms in malware identification and classification may be a solution to one of the biggest threats in the modern world.

Acknowledgements

I would like to express my sincere thanks to Professor Lioy and Engineer Andrea Atzeni for allowing me to work on the following thesis and for the help they gave me throughout the whole process.

I would like to thank my parents for their support and trust throughout my schooling. I thank them for all the sacrifices they made to raise and educate me, allowing me to reach this very important milestone for me. I hope one day I can reciprocate by making them proud of me.

Special thanks go to my brother Antonio, who supported me in a special way as I grew up, offering me advice in difficult times and helping me overcome the obstacles I encountered during my academic journey.

I would also like to give special thanks to my beloved Clare, who always believed in me, encouraging me and and cheering for me whenever I had to face an obstacle, such as exams or the following thesis. I hope someday I can support you as you did me during this period.

I would like to thank Andrea Sindoni, Paolo Rabino and Matteo Ferrenti who provided me with technical support when I had doubts during the realisation of my thesis.

Finally, I would like to thank my friends who accompanied me during my studies, alleviating the tiredness and tension caused by the exams, allowing me to face the university course with more lightness.

Contents

Chapter 1

Introduction

In recent years, the number of users on the Internet has grown exponentially. As of January 2023, there were 5.16 billion (about 64% of the world's population) users connected to the Internet [1]. The rapid growth of the Internet has made the development of cybersecurity threats possible and thus a new difficult challenge to be faced. Software with malicious intentions such as spying, damaging etc. is called malware.

Malware first appeared in the 1960s, initially created for fun inside laboratories. One of the first malware in history was Creeper (1971); it would pop up the following message on the screen: "I'M THE CREEPER : CATCH ME IF YOU CAN". It was a worm, a type of malware that reproduces itself by going on to infect other devices connected to the same network. Subsequently, malware slowly became more numerous, beginning to infect even more specific devices or operating systems. The first malware to emerge from a laboratory environment was Elk Cloner, it was created in 1982 by a 15-year-old initially as a joke. He inserted the malicious code inside a game disk, a floppy disk to be precise. This malware attacked devices running the Apple II operating system, in fact the 50th time the game was started inside the device, instead of starting the game a blank screen was shown with a poem about the malware itself. Subsequently, the first malware against IBM devices, Brain, was born. Brain infects a PC by replacing the boot sector of a floppy disk with a copy of the virus (a type of malware). The real boot sector is moved to another part of the disk's memory and marked as bad. Infected disks have five kilobytes of malicious memory. Also for the following malware, a particular message created by the authors was shown on the screen.

After a few years malware began to have much more damaging effects, malware capable of spying on the victim's device, stealing information, damaging industrial machinery began to spread, a real threat was created. In recent decades, an exponential growth of malware recorded on the Internet could be observed. To be precise, the AV-Test Institute records more than 450,000 malware and potentially unwanted applications (PUA) under Windows every day. As you can see from the graph in figure 1.1 malware affecting Windows is continuously growing, just think that from 2012 to 2022 we went from 82.8 million (approximately) to 1 billion malware registered on the internet.

The graph obviously shows the total amount of malware found on the Internet, so one column contains malware from the previous year as well. If you want to view only the malware that is detected each year you have to look at the graph in figure 1.2. Of course, it should also be noted that some of the malware spread on the Internet after a certain period is no longer a threat since patches are found to the vulnerabilities that they exploit to damage the victim's device. Moreover, malware is not only more numerous but also more complex; in fact, malicious software authors use obfuscation and/or other sophisticated techniques such as polymorphism or metamorphism to evolve the code structure, making malware identification difficult if not impossible.

For all these reasons, the signature-based detection techniques typically used by most commercial antivirus solutions (such as manually crafted Yara rules) are rendered ineffective in the current scenario. To date, it is impossible for cyber security analysts to go and analyze every single malware variant that can be found on the Internet. Nor is it certain that once a new malware family is sufficiently identified and analyzed, the generated signature employed in anti-viruses is

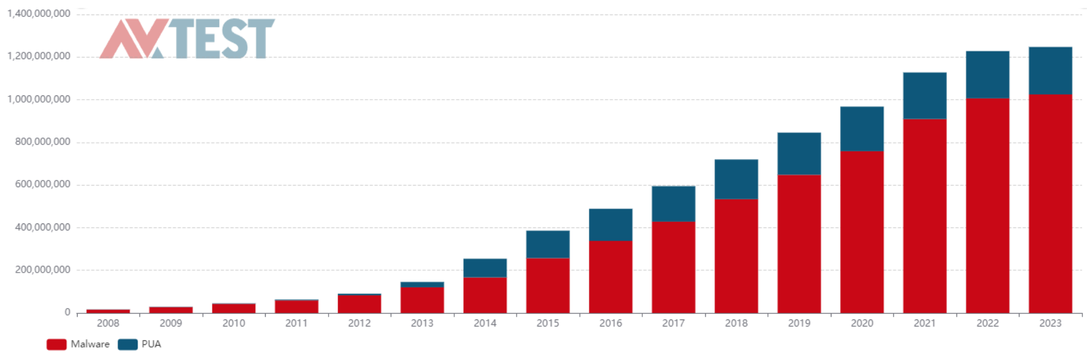


Figure 1.1: Total amount of malware and PUA

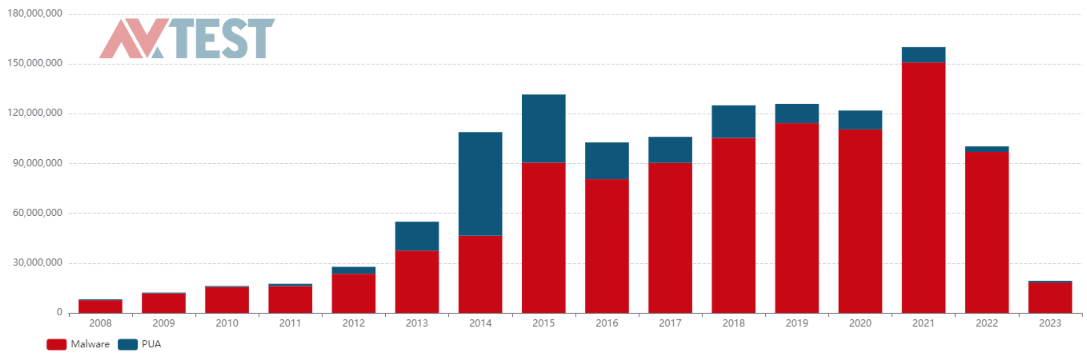


Figure 1.2: Annual growth of malware and PUAs

capable of detecting all new malware variants of the same family. The use of obfuscation techniques can go a long way toward rendering useless all the effort it would take to go into creating a signature to be used for malware identification and classification. Indeed, signature-based systems are geared more towards an "identify and then respond" approach, an approach that might have been acceptable as long as zero-day malware was a manageable number. Today, this approach is no longer acceptable, with the development of the World Wide Web we have moved on to putting a lot of data on the Internet or locally on our personal devices. Security is therefore sought that is capable of responding to attacks even before they occur. Machine learning can address these problems. Indeed, much current research has proven that machine learning has promising results in the field of malware identification and classification. Many researchers have succeeded in realising tools that can excellently classify malware belonging to certain data sets searchable on the Internet. These solutions are still under development, but many papers have already been produced trying to support this thesis concerning the use of machine learning in this type of work.

This thesis addresses a topic that has been much discussed recently in the field of cyber security, namely going to employ machine learning algorithms in the field of malware detection and classification. Machine learning in recent years has developed a great deal in the field of classification, in so many areas. This is precisely why many researchers have also begun to use it in the field of cyber security. In particular, this thesis seeks to better explore what information that can be gleaned from malware can best be exploited by machine learning algorithms in order to fully accomplish the task previously described. In addition, the machine learning algorithm to be used has also been extensively researched, through various experiments with different algorithms. The ultimate goal, then, is to figure out which is the best algorithm to use along with the best features that can be extracted from malware.

The development process can be divided into 4 stages (as shown in Figure 1.3): there is the **choice of the dataset**, the **definition of the features to be extracted** from the malware, the **definition of the machine learning algorithms** to be used, and finally the **evaluation of the various algorithms**.

In the first phase we tried to understand what was the best dataset available on the Internet for

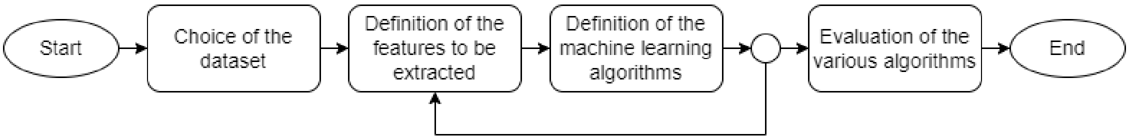

Start
Choice of the dataset
Definition of the features to be extracted
Definition of the machine learning algorithms
Evaluation of the various algorithms
End

Figure 1.3: Development process

our work. Different datasets such as EMBER, SOREL-20M, MalwareBazaar etc. were compared but in the end we chose to use the dataset used in the Microsoft Malware Classification Challenge and the ST-WinMal dataset created by a colleague of mine at the Politecnico di Torino. In Chapter 4 can be found all the considerations regarding the datasets evaluated during my research, there is a precise description of their characteristics and the reasons why they were evaluated as suitable or unsuitable for my type of work.

The second phase was the part where choices were made not only about the performance of my machine learning tool but also with respect to the time available. In the initial stages where the state of the art of this subject was studied, I realized that the features that can be extracted from executable files that are useful for malware recognition and classification are numerous. Therefore, a critical choice had to be made based on the dataset available and the time required to perform the extraction. In addition, for some features it is necessary to have specific tools (sometimes for a fee) to perform the extraction. In the initial part of Chapter 5, the rationale behind the choice of features to be used and the reasoning performed to extract them from the dataset can be found.

In the final part of Chapter 5, on the other hand, the third stage of the development process of this thesis was described. Indeed, this step involved studying which algorithms were most widely used in the field of this research and understanding which had the most potential for my type of study. Four algorithms were chosen to compare: Random Forest, K-NN, SVM and Gradient boosting. These algorithms have already been used and reported excellent results, so they were chosen to be used with other types of features, namely those extracted by me from the two datasets mentioned above. Obviously the previous two steps were not performed totally sequentially, there were in fact times when after extracting two/three features we would already try some of the algorithms mentioned above, so as to see if they had potential or were totally unsuitable for our field of research.

In the concluding part of the thesis, the results were recorded. To optimize the classification, techniques such as k-fold cross validation were used to make the results as accurate as possible. Obviously, the datasets contain only a few malware families so the results can only be generalized in part but these are compromises that must be taken into account. In Chapter 6 you can find all the discussions regarding the results of the research while in Chapter 7 you can find the conclusions I came to once I finished the study.

Chapter 2

Background

2.1 Malware

Malware, short for “malicious software”, refers to any intrusive software developed by cybercriminals (so called “hackers”) to steal data and damage or destroy computers and computer systems [2]. The purposes that malware can have are varied, such as disrupting the normal computer operations, gathering sensitive and confidential information from an unwitting user, gaining access to private computer networks and/or showing unwanted advertisements or spam [3].

Malware attacks can crack weak passwords, bore deep into systems, spread through networks, and disrupt the daily operations of an organization or business. Other types of malware can lock up important files, spam you with ads, slow down your computer, or redirect you to malicious websites.

Malicious software is at the root of most cyberattacks, including the large-scale data breaches that lead to widespread identity theft and fraud. Malware is also behind the ransomware attacks that result in millions of dollars in damages. Hackers aim malware attacks against individuals, companies, and even governments [4].

2.1.1 Why are malware used?

Cybercriminals use malware for different purposes, the common goal is to obtain an economic gain or cause consequences from which they can profit [4]:

- **Data theft**: Cybercriminals can steal data stored in databases, servers or personal devices and use it to commit identity theft or sell them on the dark web to other cybercriminals or people interested in them.
- **Corporate espionage**: Data theft on a corporate scale is known as corporate espionage. Companies can steal classified and sensitive information from their competitors, and governments often target large corporations as well.
- **Cyberwarfare and international espionage**: Governments use cyber attacks against another nation, with the aim of disrupting activities of a nation without getting caught.
- **Sabotage**: One possible purpose of an attacker is to cause harm to an organization. Attackers can delete files, wipe records, or shut down entire organizations to cause millions of dollars of damage.
- **Extortion**: Some types of malware encrypts a victim’s files or device and demands payment for the decryption key. The purpose is to get the victim - a person, institution, or government - to pay the ransom. It is not certain that the victim after paying the ransom will not be blackmailed in the future.

- **Law enforcement**: Police and other government authorities can use spyware to monitor suspects and harvest information to use in their investigations.

- **Entrepreneurship**: Hackers can sell the malware they have written. The developer licenses their malware in exchange for an up-front payment or a subscription payment.

- **DDoS attacks**: Hackers can use malicious software to create botnets - linked networks of "zombie computers" under the attacker's control. The botnet is then used to overload a server in a distributed denial of service (DDoS) attack.

- **Mining cryptocurrency**: Cryptominers force a victim's computer to generate, or mine, bitcoin or other cryptocurrency for the attacker.

2.1.2 Classification of malware

Malware is commonly divided into a number of classes, depending on the way in which it is introduced into the target system and the sort of policy breach caused [5]:

- **Virus**: Malware which spreads from one computer to another by embedding copies of itself into files, which by some means or another are transported to the target. The means of transport is often known as the vector of the virus. The transport may be initiated by the virus itself (for example, it may send the infected file as an e-mail attachment) or rely on an unsuspecting human user (who for example transports a CD-ROM containing the infected file).

- **Worm**: This is a self-replicating and active malicious program that can spread over the network by exploiting various system vulnerabilities. It uses targeting vulnerabilities in the operating system or installed software. It contains harmful routines but can be used to open communication channels which serve as active carriers. The Worm consumes a lot of bandwidth and processing resource through continuous scanning and makes the host unstable which can sometimes cause the system to crash. It may also contain a payload that are pieces of code written to affect the computer by stealing data, deleting files or create a bot that can lead the infected system being part of a botnet. While viruses require human activity to spread, worms have the ability spread and replicate independently [6].

- **Trojan horse**: Malware which is embedded in a piece of software which has an apparently useful effect. The useful effect is often known as the overt effect, as it is made apparent to the receiver, while the effect of the malware, known as the covert effect, is kept hidden from the receiver. Trojan horses usually are used to give remote access to the attacker, in this way the attacker could perform any malicious activity that is inters to him.

- **Logic bomb**: Malware which is triggered by some external event, such as a specific date or time, or the creation or deletion of a specific data item such as a file or a database entry.

- **Rabbit**: Malware which uses up all of a particular class of resource, such as message buffers, file space or process control blocks, on a computer system.

- **Backdoor**: Malware that when it reach the target device permits to the attacker to gain access to system resources. Usually the attacker sends the malware to multiple devices to create a botnet to perform a DDoS attack.

- **Spyware**: This is a malicious program that uses functions of an operating system with the intention of spying on user's activity. They sometimes have additional capabilities like interfering with network connections to modify security settings on the infected system. They spread by attaching themselves to legitimate software, through a Trojan horse or even taking by exploiting known software vulnerabilities. Spyware can monitor user behaviour, collect keystrokes, internet usage habits and send the information to the program author [6].

- **Ransomware**: Ransomware is a program that infects a host or network and holds the system captive while requesting a ransom from the system/network users. The program normally encrypts the files on the infected system or locks down the system so that the users have no access. It then displays messages that force the users to pay to have access to their systems again. Ransomware use the same propagation means as a computer worm to spread and therefore user awareness [6].

There are many different definitions of malware classes. It is in fact possible to find other types of classes because the dividing line between different classes of malware is very thin. The classes are obviously not exclusive, for example, a virus can contain logic bomb functionality, if its malicious effect is not triggered until a certain date or time is reached. Or a trojan horse may contain ransomware functionality, and so on [5].

2.2 Malware analysis

The process of extracting information about their behavior from malware is called malware analysis. When a malicious sample is discovered in the wild or on a machine, it is usually an executable which has been compiled and therefore presented in machine language. The main goal of malware analysis is to extract as much information from the discovered sample to understand the threat associated [6]. Malware analysis is used to develop effective detection techniques for malicious code [7]. Malware analysis can be grouped roughly into two categories:

- Static analysis
- Dynamic analysis

2.2.1 Static analysis

Static analysis is a simple and quick analysis technique. In this analysis, the malware is decompiled and its source code is examined using several tools, like Pestudio, IDA Pro, etc. [8].The executable has to be unpacked and decrypted before doing static analysis. The disassembler/debugger and memory dumper tools can be used to reverse compile executables [9]. When compiling the source code of a program into a binary executable, some information gets lost. This loss of information further complicates the task of analyzing the code.

Static malware analysis is commonly done by hand for various reasons, for example, if the source code is available several interesting information, such as data structures and used functions can be extracted. This information gets lost once the source code has been compiled into a binary executable and thus impedes further analysis [7].

The detection patterns used in static analysis include string signature, byte-sequence n-grams, syntactic library call, control flow graph and opcode (operational code) frequency distribution etc [9].

There are different techniques used for static malware analysis. Some of are described below [7].

- File fingerprinting: Beside examining obvious external features of the binary this includes operations on the file level such as computation of a cryptographic hash (e.g., md5) of the binary in order to distinguish it from others and to verify that it has not been modified.
- File format: By leveraging metadata of a given file format additional, useful information can be gathered. This includes the magic number on UNIX systems to determine the file type. For example from a Windows binary, which is typically in PE format (portable executable) a lot of information can be extracted, such as compilation time, imported and exported functions as well as strings, menus and icons.

- AV scanning: If the examined binary is well-known malware it is highly likely to be detected by one or more AV scanners. To use one or more AV scanner is time consuming but it becomes necessity sometimes.

- Packer detection: Nowadays malware is mostly distributed in an obfuscated form e.g., encrypted or compressed. This is achieved using a packer, whereas arbitrary algorithms can be used for modification. After packing the program looks much different from a static analysis perspective and its logic as well as other metadata is thus hard to recover. While there are certain unpackers, such as PEiD2, there is accordingly no generic unpacker, making this a major challenge of static malware analysis.

- Disassembly: The major part of static analysis is typically the disassembly of a given binary. This is conducted utilizing tools, which are capable of reversing the machine code to assembly language, such as IDA Pro. Based on the reconstructed assembly code an analyst can then inspect the program logic and thus examine its intention.

The main advantage of static malware analysis is that it allows a comprehensive analysis of a given binary. That is, it can cover all possible execution paths of a malware sample. Additionally, static analysis is generally safer than dynamic analysis as the source code is not actually executed. However, it can be extremely time-consuming and thus requires expertise.

Limitations: Generally, the source code of malware samples is not readily available. That reduces the applicable static analysis techniques for malware analysis to those that retrieve the information from the binary representation of the malware [7]. In addition, cybercriminals sometimes use obfuscation or encryption techniques to make static malware analysis very complex.

2.2.2 Dynamic analysis

Dynamic malware analysis consists in extracting information about the behavior of the sample during its execution. To do this, you need to create an isolated environment in which you can try to run the malware without having to worry about the consequences.

Dynamic malware analysis manages to overcome the limit of static analysis regarding malware obfuscation techniques, extracting information not from the binary file but from the execution of the malware itself. However, this technique also has limitations, such as the possibility for the attacker to have introduced in the code a check on the environment in which the malware is executed. In this way, the attacker can insert a command into the code that terminates the execution in case the malware is not in the desired environment.

There are two basic approaches for dynamic malware analysis which are as below [7]:

- Analyzing the difference between defined points: A given malware sample is executed for a certain period of time and afterwards the modifications made to the system are analyzed by comparison to the initial system state. In this approach, Comparison report states behavior of malware.

- Observing runtime-behavior: In this approach, malicious activities launched by the malicious application are monitored during runtime using a specialized tool

There are different approaches and techniques that can be applied to perform dynamic analysis [10]:

- **Function Call Monitoring**: the property that makes functions interesting for program analysis is that they are commonly used to abstract from implementation details to a semantically richer representation. For example, the particular algorithm which a sort function implements might not be important as long as the result corresponds to the sorted input. When it comes to analyzing code, such abstractions help gain an overview of the behavior of the program.

- **Function Parameter Analysis**: dynamic function parameter analysis focuses on the actual values that are passed when a function is invoked. The tracking of parameters and function return values enables the correlation of individual function calls that operate on the same object. For example, if the return value (a file-handle) of a `CreateFile` system call is used in a subsequent `WriteFile` call, such a correlation is obviously given.
- **Information Flow Tracking**: The goal of information flow tracking is to shed light on the propagation of "interesting" data throughout the system while a program manipulating this data is executed. In general, the data that should be monitored is specifically marked (tainted) with a corresponding label. Whenever the data is processed by the application, its taint-label is propagated. Assignment statements, for example, usually propagate the taint-label of the source operand to the target.
- **Instruction trace**: A valuable source of information for an analyst to understand the behavior of an analyzed sample is the instruction trace. That is, the sequence of machine instructions that the sample executed while it was analyzed.

There are different dynamic analysis tool like Anubis, GFISandbox etc. These tools permit to create sandbox where analyst could perform analysis operation end extract the information they want. Sandboxes can be used in several ways. It can act as a real machine with some limitation: for example with limited network access or with no network access at all.

2.3 Detection evasion

To understand how is made malware detection and malware analysis is strongly recommended to study how malware try to avoid detection techniques. Cybercriminals try to camouflage their malware with different techniques, to be certain that when they send the malware to the target the victim device could not identify the malicious software and stop his execution.

Initially, the malware code was not protected because attackers knew that the defenses used today had not yet been developed. This, however, allowed security analysts to be able to analyze malware without any difficulty and to be able to create the first malware identification systems. One of the first pieces of information that was used to identify malware, for example, is the sequence of opcodes present in the code. The sequence was used to understand if one file could be malicious or not.

After the advent of the first defense systems, the attackers decided to develop techniques to disguise their malware. Today's malware in fact use various techniques to not be identifiable by antivirus and other identification systems.

Cybercriminals do not just want to prevent malware from being detected by the security system owned by the target device. They also want to avoid that, in case it is identified and captured by security analysts, they cannot analyze it to understand its functioning and extract information to be able to it stop in the future.

Hackers are also interested in using obfuscation techniques for another reason: in addition to not wanting to be able to detect and analyze malware, they want to keep the algorithms used by them to create malicious software confidential. They basically want to keep the "intellectual property" behind the malware confidential.

Obfuscation techniques can be divided into two categories anti-static and anti-dynamic analysis techniques [11]. The techniques are described in the following sections.

2.3.1 Anti-static analysis techniques

Malware authors tried to create techniques to counter static analysis tools to prevent their malicious software from being investigated. The static analysis tools, as explained in section 2.2.1, look in binary files or assembly files for information in order to understand the behavior of the software under analysis. Below can be find the most used techniques.

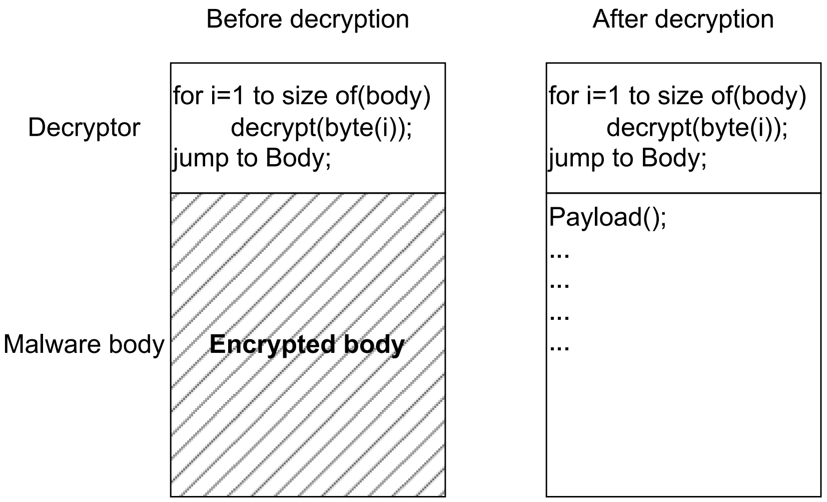


Figure 2.1: Encrypted malware.

Packing the code

Early malware only use encryption to evade detection and analysis. The encrypted malware is mainly composed of two parts: the decryption and encrypted code body. When code is executed, two parts are loaded into memory simultaneously. The decryption first obtains the executive power to decrypt the encrypted code, and then performs the actual function code. Compression and encryption together are referred to as packers [12].

- **Encrypted Malware**

  The first approach to evade the signature based antivirus scanners is to use encryption. In this approach, an encrypted malware is typically composed of the decryptor and the encrypted main body (as shown in Figure 2.1) [13]. Usually the body is XORed with a key to make it difficult to detect. For each infection, encrypted malware makes the body unique by using different key to hide the signature. However, the decryption routine remain same, hence it can be detected by analyzing the decryptor [14].

  In this method when the malware code executes; first the decryption part is executed to decrypt the body of the malware and then the code is executed for the action. The main purpose of this technique is to avoid antivirus detection and static code analysis. This method also delay the process of investigation [15]. However, the main problem of this approach is that the decryptor remains constant from generation to generation.

- **Oligomorphic Malware**

  The short comings of the encrypted malware led to the development of different occultation techniques. In Oligomorphic malware decryptors are mutated from one variant to other. A different key is used when encrypting and decrypting malware payload, in this way a malware could contain few hundred different decryptors. Although oligiomorphism provides different decryptor from a list of decryption for each new attack, still there are chance to caught by antivirus by checking all the decryptor [15] even if it is not a simple process [14]. For overcoming the limitation, the malware authors developed the polymorphic malware.

- **Polymorphic Malware**

  Malware of this type is equipped with the same evasion technique as the other malware discussed previously; however, it contains an encrypted body with several copies of the decryptor (polymorphic decryptor) [16]. The encrypted portion of the payload contains several copies of the decryptor and can be encrypted at multiple levels. Therefore, polymorphic malware is more difficult to detect than oligomorphic malware. Also, some polymorphic viruses apply obfuscation techniques to only their decryptor to evade detection. For instance, the

```
    mov dx, msg
    mov ah, 9
    int 0x21
    jmp C

A: mov ah, 0x4c
    int 0x21

B: mov dl, 0x0a
    mov ah, 2
    int 0x21
    jmp A

C: mov dl, 0x0d
    mov ah, 2
    int 0x21
    jmp B
```

Figure 2.2: Code Transportation example.

decryptor code may be reordered by placing jump instructions or inserting garbage code to change the malware signature whilst preserving its semantics [16].

- **Metamorphic Malware**

  Metamorphic malware are the first types of malware that don't use encryption techniques to avoid detection. In order to further evade detection, malware writers have extended the above mentioned malware types by applying code obfuscation to the entire malware body [16]. It also implements a mutation engine like polymorphism but it change its whole body rather by only altering the decryptor [15]. The basic idea is the syntax change on each new copy while semantics remains the same i.e. the apparently virus change on each infection but the meaning or working remains the same. It is very difficult to detect such malware because each new copy has a completely different signature.

Code Transportation

Code Transportation change the order of the instructions of the original code without change the behavior of the malware. To make this, there are 2 techniques.

The first technique shuffle in a random way the sequence of instruction and recover the original execution thanks to unconditional branches or jumps instructions. Clearly, it is not difficult to defeat this method because the original program can be easily restored by removing the unconditional branches or jumps [13]. The second technique reorders independent instructions. As it is difficult to find independent functions, this method is difficult to implement, but it guarantees a high level of protection. An example is shown in Figure 2.2

Dead-Code Insertion

This obfuscation technique inserts dead code into the sequence of instructions to change the appearance of the program without altering its behavior. This approach can evade the signature-based detection systems. When the redundant code is inserted then the different signature is generated [11].

This technique slows down the work of malware analysts as they have to waste time figuring out which instruction is really useful to the program and which one was inserted only for disturbance. An example of an instruction widely used for this purpose is the NOP (Not operation like

```
    mov dx, msg
    mov ah, 9
    int 0x21
    jmp A
    mov edx,len ;useless code
    mov ecx,msg
    mov ebx,1
    mov eax,4
    int 0x80

A: mov dl, 0x0d
    mov ah, 2
    int 0x21

    mov dl, 0x0a
    mov ah, 2
    int 0x21

    mov ah, 0x4c
    int 0x21
```

Figure 2.3: Code Transportation example.

```
    mov dx, msg
    mov bh, 9
    int 0x21

    mov cl, 0x0d
    mov bh, 2
    int 0x21

    mov cl, 0x0a
    mov bh, 2
    int 0x21

    mov bh, 0x4c
    int 0x21
```

Figure 2.4: Register reassignment example.

instruction in x86). Another way to implement this technique is to insert a jump instruction to an x number of successive lines by inserting useless code in the middle as in the example shown in Figure 2.3

Register reassignment

This technique replaces the use of a register in an instruction with another unused instruction. Register replacement requires that no register dependencies in control flow are affected [17]. However, it is expensive obfuscation approach because it requires manual transformation of identifiers of constants, registers, and variables [11].

An example is shown in Figure 2.4 (The original code is shown in Figure 2.2 and registers ah and dl are reassigned to bh and cl respectively)

```
mov dx, msg
mov bh, 5
add bh, 4
int 0x21

mov cl, 0x0d
mov bh, 7
sub bh, 2
int 0x21

mov cl, 0x0a
mov bh, 2
int 0x21

mov bh, 0x4c
int 0x21
```

Figure 2.5: Instruction substitution example.

Subroutine reordering

Subroutine reordering obfuscates an original code by changing the order of its subroutines in a random way [18]. This technique can generate n! different variants, where n is the number of subroutines. For example, Win32/Ghost had ten subroutines, leading to 10! = 3628800 different generations [18].

Instruction substitution

Instruction substitution is a technique that allow to change instructions with other equivalents. Every sequence of original instructions can be replaced by some arbitrary instructions.

Thereby, numbers of variants of same malware files can be created. To handle this problem for every possible variant of same malware the unique signatures is required to detect these variants as well. It is not an impossible task but with the face of increasing new variants of same malware is not an easy task [11].

An example is shown in Figure 2.5 (The original code is shown in Figure 2.2)

2.3.2 Anti-dynamic analysis techniques

The analysis technique could overcome the limits of static analysis, not being influenced by anti-static analysis techniques. Anti-dynamic analysis mainly detects the fingerprint information of the current environment and finds analysis tools. The anti-dynamic analysis method commonly used includes detection of debuggers and virtual machines [12].

Detection of virtual machine

It is not likely to execute the malware files onto the host computer as such because the malware files can harm the host computer. Normally for analysis we setup the virtual environment [11]. Malware authors sometimes use anti-virtual machine (anti-VM) techniques to thwart attempts at analysis. With these techniques, the malware attempts to detect whether it is being run inside a virtual machine. If a virtual machine is detected, it can act differently or simply not run. This can, of course, cause problems for the analyst [19]. With these techniques, a malware attempts to detect whether it is being run inside a VM or on a real machine. If the VM is detected, it can act differently or simply does not run [20].

The popularity of anti-VM malware has been going down recently, and this can be attributed to the great increase in the usage of virtualization. Traditionally, malware authors have used anti-VM techniques because they thought only analysts would be running the malware in a virtual machine. However, today both administrators and users use virtual machines in order to make it easy to rebuild a machine (rebuilding had been a tedious process, but virtual machines save time by allowing you to go back to a snapshot). Malware authors are starting to realize that just because a machine is a virtual machine does not necessarily mean that it isn't a valuable victim. As virtualization continues to grow, anti-VM techniques will probably become even less common [19].

There are different anti-VM techniques, but typically they target VMware because it's the most used VM. Below can be found the various techniques used:

- **Hardware fingerprinting**: Hardware fingerprinting consists in looking for special virtualized hardware patterns unique to VMs [20]. For example, The first three bytes of a MAC address are typically specific to the vendor, and MAC addresses starting with 00:0C:29 are associated with VMware. VMware MAC addresses typically change from version to version, but all that a malware author needs to do is to check the virtual machine's MAC address for VMware values [19].
- **Registry check**: The registry contains system configuration information of the OS in the machine. Usually the information are stored hierarchically in key/value pairs and contains information like: Windows version number, build number, and registered users, the computer processor type, number of processors, memory and so on;

  Registry contains also VM specific key/value pair. Some tools, like for example ScoopyNG search for a certain keys within the Windows registry to determine whether the machine is virtual or not [21].
- **Memory check**: This technique involves looking at the values of specific memory locations after the execution of instructions such as store interrupt descriptor table (SIDT), store local descriptor table, store global descriptor table, or store task register [20]. Some hypervisors, such as VMware, leave artifacts in memory that cybercriminals can exploit to prevent malware from running in a controlled environment. Some are critical processor structures, which, because they are either moved or changed on a virtual machine, leave recognizable footprints [19]. VMware create dedicated registers for each VM. These registers have different address than the one used by the host system, and by checking the value of this address, the virtual systems' existence can be detected [20]. This technique does not work on multicore processors as each process has its own interrupt descriptor table.

Detection of debuggers

Debuggers are key tools for malware analysis, enabling the study of the software dynamically fashion, through a step-by-step execution of the code to examine its internals and impact [22].

Anti-debugging is a popular anti-analysis technique used by malware to recognize when it is under the control of a debugger or to thwart debuggers. Malware authors know that malware analysts use debuggers to figure out how malware operates, and the authors use anti-debugging techniques in an attempt to slow down the analyst as much as possible [19]. Debuggers may also manipulate the execution environment by altering, for instance, memory, registers, values of variables, configurations, among others [22]. Through debuggers, analysts can check the binary code much more thoroughly. This is because debuggers offer the opportunity to manipulate the low-level behavior of the malware. However, even though it is possible to perform very thorough analysis with debuggers, malware authors have developed several anti-debugging techniques.

- **Detection of API**: Windows operating systems have many API functions that can be used to determine if the running program is debugged. Some of these functions were designed for

```
DWORD errorValue = 12345;
SetLastError(errorValue);

// try outputting string on debugger;
// if no debugger is present, it will set
// the last-error code to a new value
OutputDebugString(Test for Debugger");

if(GetLastError() == errorValue)
{
    ExitProcess();
}
else
{
    RunMaliciousPayload();
}
```

Figure 2.6: OutputDebugString anti-debugging technique example.

debugger detection; others were designed for different purposes but can be repurposed to detect a debugger [19]. The following Windows API functions can be used for anti-debugging: `IsDebuggerPresent`, `CheckRemoteDebuggerPresent`, `NtQueryInformationProcess` and `OutputDebugString`.

- **Manually Checking Structures**: Using the Windows API may be the most obvious method for detecting the presence of a debugger, but manually checking structures is the most common method used by malware authors [19]. Malware authors have various reasons for not trusting API functions to check for a debugger. For example, if an analyst runs a rootkit on their operating system, this software can return false information to API calls made by the malware, such as the absence of a debugger. For these reasons, malware authors choose to perform the functional equivalent of the API call manually, rather the rely on the Windows API. There are several flags that are checked for information about the presence of debuggers:
    - **BeingDebugged flag**: For each process running on Windows, a structure called the Process Environment Block (PEB) is created by the operating system. In the following structure there is a flag called BeingDebugged, which indicates whether the related process is connected to a debugger or not. By checking the value within this parameter, you can figure out whether to stop or change the behavior of the malware or let it run.
    - **ProcessHeap flag**: The ProcessHeap parameter present in the PEB structure collects information about the heap allocated for that particular process. Within this parameter you can find out whether the heap was created by a debugger or by the normal execution of the operating system.
    - **NTGlobalFlag**: Processes started within a debugger run slightly differently than others, therefore they create memory heaps differently. The information needed to determine how to create heap structures is stored at an undocumented location in the PEB.
- **Checking for System Residue**: When analyzing malware, we typically use debugging tools, which leave residue on the system. Malware can search for this residue in order to determine when you are attempting to analyze it, such as by searching registry keys for references to debuggers. Malware can also search the system for files and directories, such as common debugger program executables, which are typically present during malware analysis [19].

The techniques presented up to here allow to identify the debuggers used on Windows. In addition to these techniques that exploit the functionality of the operating system, there are others that recognize the presence of a debugger thanks to its behavior. Several anti-debugging techniques are used by malware to detect this sort of debugger behavior: INT scanning, checksum checks, and timing checks.

- **INT scanning** INT 3 is the software interrupt used by debuggers to temporarily replace an instruction in a running program and to call the debug exception handler a basic mechanism to set a breakpoint. The opcode for INT 3 is 0xCC. Whenever you use a debugger to set a breakpoint, it modifies the code by inserting a 0xCC. This technique can be overcome by using hardware breakpoints instead of software breakpoints [19].
- **Performing code checksum** Malware can calculate a checksum on a section of its code to accomplish the same goal as scanning for interrupts. Instead of scanning for 0xCC, this check simply performs a cyclic redundancy check (CRC) or a MD5 checksum of the opcodes in the malware. This technique can be overcome by using hardware breakpoints or by manually modifying the execution path with the debugger at runtime.
- **Timing checks** There are a couple of ways to use timing checks to detect a debugger [19]:
  - Record a timestamp, perform a couple of operations, take another timestamp, and then compare the two timestamps. If there is a lag, you can assume the presence of a debugger.
  - Take a timestamp before and after raising an exception. If a process is not being debugged, the exception will be handled really quickly; a debugger will handle the exception much more slowly

Chapter 3

Detection and Classification of malware

A Malware detector 'D' is defined as a function whose domain and range are the set of executable program 'P' and the set malicious, benign [17]. The general malware detection function can be defined as [15]

$$D(s) = \begin{cases} \text{malicious if s contains malicious code} \\ \text{benign if s is a normal program} \\ \text{undecidable if D fails to determine s} \end{cases} \tag{3.1}$$

D is the function that checks whether certain software (s) is malicious or not. It may happen that function D cannot decide if the program is malicious or not, in these cases D must be able to give as much information as possible to the analyst so that he can go and analyze by hand if the software under analysis is malicious or not.

It is difficult to identify malware since numerous approaches have been developed throughout time to get beyond the barriers that the malware's developers erected as they produced new harmful software. We can also consider how challenging it is to find new malware before using analysis tools on a sample that has already been intercepted. Indeed, zero-day malware is currently the biggest challenge facing cybersecurity analysts.

Nowadays, malware detectors are an endpoint protection, that is, antivirus. These software are used to analyze the files that arrive on the system in which they are installed, and to recognize whether the subjects analyzed are malware or not. Furthermore, antivirus usually provide other information in addition to the one mentioned above. In fact, they must classify the malware and update information (such as the number of detections) in special databases. The general malware classification function C can be defined as

$$C(s) = \begin{cases} \text{Trojan if s contains specific trojan features} \\ \text{Keylogger if s contains specific keylogger features} \\ \textbf{...} \\ \text{undecidable if C fails to classify s} \end{cases} \tag{3.2}$$

The malware detector detects the malware based on signatures of malware.The binary pattern of the machine code of a particular virus is called as signature. Antivirus programs compare their database of virus signatures with the files on the hard disk and removable media (including the boot sectors of the disks) as well as within RAM [17]. The signatures are generated by security analysts after an analysis phase executed with specific tools. Obviously antivirus are not perfect, like any other testing tools there is the possibility that it returns false positives and false negatives. Below you can understand how we can find ourselves in front of errors during the test phase of a sample:

- False positive: A false positive occurs when a virus scanner erroneously detects a 'malware' in a non-infected file. False positives result when the signature used to detect a particular malware is not unique to the malware - i.e. the same signature appears in legitimate, non-infected software [17].
- False negative: A false negative occurs when a virus scanner fails to detect a virus in an infected file. The antivirus scanner may fail to detect the virus because the virus is new and no signature is yet available, or it may fail to detect because of configuration settings or even faulty signatures.

Malware detection is the process of investigating the content of the program and deciding whether the analyzed program malware or benign. The malware detection process includes 3 stages: Malware analysis, feature extraction, and classification.

The malware analysis phase has already been extensively discussed in section 2.2 so now the feature extraction phase will be addressed.

3.1 Malware Feature extraction

The information that can be extracted from a malware through the analysis phase are called malware features, they can be divided into two main groups: static features and dynamic features.

Static features are those features that can be extracted without the execution of malware. Such features are extracted by studying binaries of malware. While dynamic features are those which are extracted after executing a binary file. These features are extracted through dynamic analysis [23].

3.1.1 Static features

Static features are extracted from executable binaries or assembly source files. In Android applications, on the other hand, you have to disassemble the APKs. To do these operations there are tools created specifically for this purpose such as IDA Pro or Radare 2. Below you will find a description of the most important static feature.

Bytes and opcode N-Grams

The most common feature for identifying and classifying malware is the N-gram. An N-gram is a contiguous sub-sequence of n elements placed one in a row to the other in a sequence, for example if we take a sequence of letters of the alphabet such as a, b, c and d, if we are talking about 2-gram we obtain some sub-sequences which are: a-b, b-c and c-d. In the malware field, N-grams are extracted from a sequence of bytes from the malware binary. By treating a file as a sequence of bytes, the n-grams are extracted by looking at the unique combination of each n consecutive bytes as an individual feature of the malware type. In the case of assembly files, the elements used as grams are not bytes but instructions such as "ADD", "MUL", "PUSH" etc., for this reason they are defined not as bytes N-grams but as Opcode N-grams.

String Analysis

String analysis refers to the extraction of each printable string within an executable or program. Searching for strings is the simplest way to get clues about a program's functionality. The information that can be found in the strings can be, for example, the URLs to which the program connects, or the paths of the files that are accessed and that change during the execution of the malware, etc.

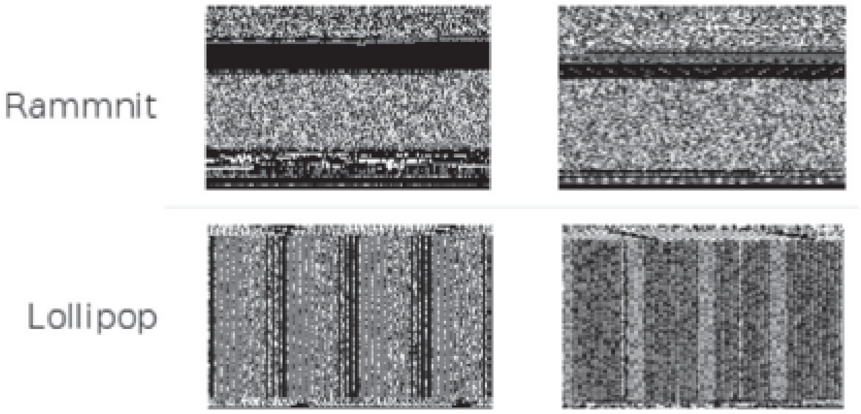


Figure 3.1: Gray scale representation of the binary content of malware samples belonging to the Ramnit and the Lollipop families [24].

API function calls

Application Programming Interfaces (API) and their function calls are regarded as very discriminative features [24]. APIs are the functions that must be called by a malware in order to access system information such as networking, security, file management, and so on. By studying which APIs are called by various malware, it is possible to identify which family they belong to as a common feature. As there is no other way for software to access the system resources without using API functions, the invocation of particular API functions provides key information to represent the behavior of malware [24].

Entropy

As described in section 2.3.1, malware authors use various encryption and packaging techniques to obfuscate malware from malicious software detectors on victims' computer systems. Consequently, it is of great interest for the information security industry to be able to detect the presence of encrypted or compressed segments of code within executable files [24]. This is why the study of the entropy present in the analyzed executables was born. In fact, it has been noted that the entropy in the native code is much higher when the files are compressed or encrypted. In the context of information theory, the entropy of a bytes sequence reflects its statistical variation. In particular, zero entropy would mean that the same character has been repeated over the analyzed segment [24]. On the contrary, if a block of bytes all different is found, it means that the entropy of that block is very high.

Malware representation as a grey scale image

An interesting approach to viewing malware was first introduced in 2011, where an attempt was made to convert a binary to a gray scale image. These images are obtained by interpreting each byte as a pixel of an image, where the values range from 0 to 255 (0: black, 255: white). Finally, the resulting array is transformed into a matrix.

As can be seen from Figure 3.1 two malware from different families create 2 totally different images, while 2 malware from the same family are very similar. This similarity stems from the fact that many malware have pieces of code copied from other malware.

However, this feature also has flaws. The disadvantages are:

- To construct an image you need to select an image width which adds a new hyper-parameter to tune. Notice that selecting the width consequently determines the height on the image depending of the size of the binary [24].

- Forces a spatial correlation between two pixels of different lines even if it does not exist in the code.
- It suffers from code obfuscation techniques like all static features.

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 symbolize function calls [24]. The call graph doesn't give information on how control of program flows instead it provides information on calling of various procedures. A node vertex in the graph can represent two different types of functions:

- Local functions, implemented by the software publisher to perform a specific function.
- An external function, obtained from the operating system or from an external library.

One particularity of the graph is that only local functions can invoke external functions, not the other way around [24].

Control Flow Graph

A Control Flow Graph (CFG) is the graphical representation of control flow or computation during the execution of programs or applications [25]. A Control Flow Graph (CFG) is a directed graph in which the nodes represent basic blocks and the edges represent control flow paths [24]. Each block in the graph represents a linear sequence of instructions having an entry point (the first executed instruction of the block) and an exit point (the last executed instruction of the block). When an instruction allows to have two different flows of execution, two exit points are inserted which thus allow the reader to understand the different execution paths that the program can have.

3.1.2 Dynamic features

Dynamic feature extraction consists of running the executable usually in an insulated environment which can be a virtual machine (VM) or an emulator and then extract features from the memory image of the executable or from its behaviors [26]. Dynamic Malware Analysis monitors the behavior of malicious software by observing the actual sequence of instructions executed. It reveals process creation, file and registry manipulation and modifications of memory values, registers and variables [24].

Memory and Register's usage

The behavior of a program can be represented by the values that are assumed by some portions of the memory during the execution of the malware. In other words, the values stored in the logs while the program is running can tell us whether this is malware or not.

Instruction traces

The instruction trace is the sequence of processor instructions called during the execution of a software. Unlike the static version, this trace does not respect the order of how they are written on the binary file but the order in which they are executed. The trace collected through dynamic methods, in fact, allows for a more robust measure that also avoids the obfuscation techniques used by the authors of the malware. It, respecting the call order, allows the reconstruction of the execution flow, even if the instructions of the malware are interchanged in the binary file. The cause of this advantage is that the control flow graph is built during a fake execution going to see the real call order.

Network traffic

Malware can not only interact with the operating system and its data, but also with the network, infecting multiple machines connected to the same router for example. A targeted study of network traffic can be used to extract features from various malware. All this first requires a thorough analysis of the "normal" behavior of the network without ongoing attacks. 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. Approaches in the literature extract events at several abstraction levels, from raw packets to network flows, detailed protocol decoding such as HTTP and DNS requests, to host-based events and metadata such as IP addresses, ports and packet counts [24].

API call traces

Software programmers use the Windows API to access basic resources available to a Windows system including, but not limited to, file systems, devices, processes, threads and error handling, and also to access functions beyond the kernel such as the Windows registry, start/stop/create a Windows service, manage user accounts and so on [24]. Obviously all the advantages that can be obtained from the static feature are also inherited from the extraction performed dynamically.

3.2 Malware detection approaches

The features seen in the previous section are all information that is used by malware detectors to understand if they are analyzing malicious software or not. In recent years, many researchers have tried to find a solution that would be able to detect any malware that comes into their analysis. In this section we will deal with the already existing solutions that have been used up to now and that continue to be present in antivirus. In the early days, signature-based detection method was widely used. This method works fast and efficiently against the known malware, but does not perform well against the zero-day malware [27].

Subsequently, researchers to try to solve the signature-based problem have developed new techniques such as behavior-based, heuristic-based and model-checking based. Today the most widespread research is on identification techniques through deep learning or ML algorithms. These two new techniques will be addressed in a specific section. In each approach, feature extracting method is different one from another. It could not have been proven one detection method works better than another because each method has its own advantages and disadvantages. By using behavior-, heuristic-, and model checking-based detection approaches; huge number of malware can be detected with a few behaviors and specifications. In addition, new malware can be detected by using these approaches as well. However, they cannot detect all malware [27].

3.2.1 Signature-based malware detection

Signature is a malware feature which encapsulates the program structure and identifies each malware uniquely [27]. The signature is nothing more than a sequence of bits generated by the malware binary. This technique consists in the generation of this signature from the file under analysis, followed by a comparison phase with the signatures of the already known malware present in a database. As you can guess if a malware has already been analyzed and recognized this technique will be very effective against its replicas, if instead there were to identify and classify a new malware (zero-day malware for example) this technique would be insufficient.

Signature generation process

Antivirus have a database in which all the signatures of malware already scanned are collected. In fact, during the analysis phase some features are extracted from the executables, which are then

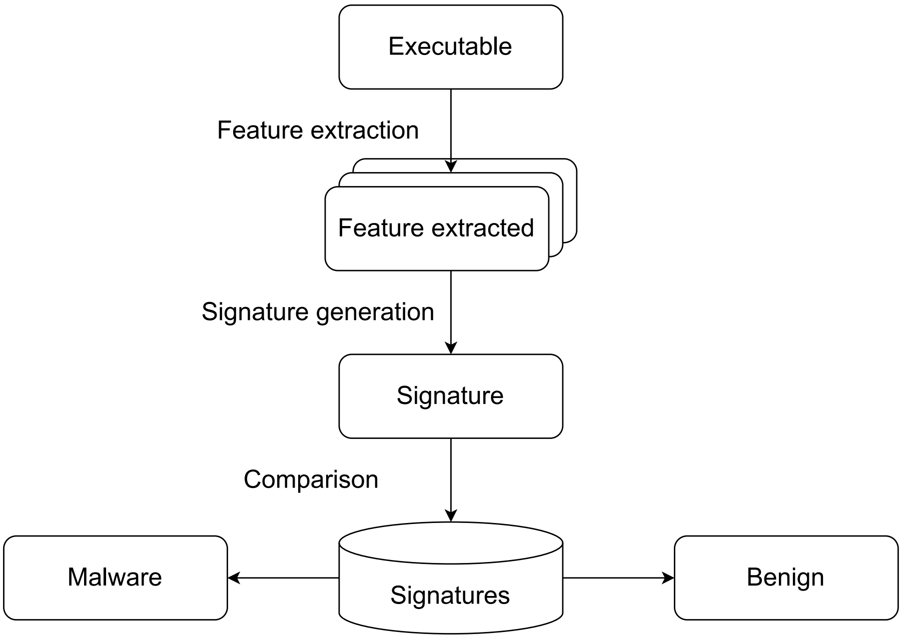


Figure 3.2: Signature-based malware detection schema.

```
rule ExampleRule
{
    strings:
    $signature: {90 FF 16 83 EE 04 83 EB 01 75 F6}

    condition:
    $signature
}
```

Figure 3.3: Byte signature in Yara format [27].

used to generate the signature. When sample program needs to be marked as malware or benign, signature of the related sample is extracted as the same way before and compared with signatures on the database (General view of signature based detection schema can be seen in Figure 3.2) [27]. It has the advantage of accurately identifying known malware instances and requiring fewer resources to do so. The disadvantage is that it cannot detect the new and unknown malware instances because we cannot find the signatures for those types of malware [28].

There are many different techniques to create a signature such as string scanning, top-and-tail scanning, entry point scanning, and integrity checking [27]:

- **String Scanning**: This technique involves comparing a sequence of bytes in the scanned file with the byte sequences of known malware previously saved in a database. String scanning was used a lot by the first antivirus released on the market as it was very fast and effective when few families of malware were widespread. This technique could be accompanied by the use of the Yara rules. In fact, these rules could be saved in the database to keep track of all malicious sequences (Example of Yara rule in Figure 3.3 referring to malware example in Figure 3.4).

- **Top-and-Tail Scanning**: To generate the signature, only the initial and final part of the file under examination are taken. In this way it is easy to identify all types of malware that to remain hidden are attached to the beginning or end of other harmless files.

```
Start: 0x401A2E length: 0xC
90 nop
FF 16 call dword ptr [esi]
83 EE 04 sub esi, 4
83 EB 01 sub ebx, 1
75 F6 jnz short loc_401A30
```

Figure 3.4: Example byte sequence of a malware [27].

- **Entry Point Scanning**: The entry point is the point in a program, module or function where the code begins; specifically, the memory address where it begins. Specifically, in a file the entry point indicates where is the first instruction that must be execution when the file starts to run. Therefore, certain malware can be detected by extracting the signature from the sequences at the program entry points [27].
- **Integrity Checking**: The integrity check is nothing more than the generation of the file hash. In fact, a hash is generated using algorithms such as SHA-256 or MD5 to create a unique identifier for each malware, to be kept in a database for future comparisons.

As shown, there are several techniques for generating malware signatures. All of the ones shown are very fast and efficient but are not resistant to obfuscation techniques. For example, malware can easily change the strings and program entry point in its instruction set. By this, generated signature may mislead the detecting schema [27]. Some researchers have tried to create new techniques to generate more effective signatures but still have to admit some limitations. The limits may be, for example, techniques that were applicable only to specific classes of malware, or the tests they carried out covered only a small slice of malware existing on the web.

Evaluation of signature-based detection

As analyzed in this section, signature-based detection is a very fast and effective technique with malware belonging to the same family. It has in fact been used for many years by the antivirus on the market. Unfortunately, however, it is not effective against zero-day malware, malware that uses obfuscation techniques or polymorphic malware. Furthermore, it is very easy to get false positives if the rules are ineffective. This problem arises from the fact that extracting signatures is a long and complex process and it is easy to make mistakes during the feature extraction phase. Despite these limitations, it is still a very widespread technique, in fact, in order to be effective, rules have been defined to be taken into consideration [27]:

- Signature should be as short as possible and can represent many malware with single signature,
- Effective automatic signature generation mechanism must be built,
- During the signature generation, data mining and ML techniques need to be used more,
- Signature should be resistant to packing and obfuscation techniques

3.2.2 Behavior-based malware detection

Behavior-based malware detection approach observes the program behaviors with monitoring tools and determines whether the program is malware or benign [27]. This technique is based on the assumption that two malware belonging to the same family will have similar behavior. It is therefore easy to understand that once the behaviors of known malware families are analyzed, even new malware will be easy to detect. This is not entirely true as some new malware are not yet detectable through behavioral methods. For example, all malware that uses anti-VM and anti sandbox techniques cannot be detected because to extract the behavior of a malware, it must first be run in a controlled environment. It is therefore possible to obtain false negatives.

Behavior detection process

In section 3.1 was discussed the various feature extraction techniques and which ones are most used in the field of malware detection and classification. Among these techniques there are some that have been used extensively in the field of malware detection based on their behavior. When establishing a behavior-based detection system, behaviors are obtained by using one the following procedure [27]:

- Automatic analysis by using sandbox;
- Monitoring of system calls;
- Monitoring of file changes;
- Comparison of registry snapshot;
- Monitoring network activities;
- Process monitoring.

In behavior-based detection, first, behaviors are determined by using one of the technique used above and the dataset is created by subtracting the features using data mining. Then, specific features from the dataset are obtained and classification done by using ML algorithms [27]. After the training and testing phase with the ML tool, the decision maker is developed which will be the final product installed on the endpoints to detect potential malware.

Evaluation of Behavior based detection

The scheme of Behavior-based detection techniques can be divided into 3 phases:

- Determine behaviors (data mining can be used),
- Extract features from behaviors (data mining is used),
- Apply classification (machine learning is used).

When extracting features in ML models, it is also necessary to find a way to transform the extracted values (which can be strings, registers, etc.) into values that can be understood by an ML tool and therefore into numbers. To build the similarities between different features there are different techniques such as the Hellinger distance, the cosine coefficient etc.

The difficulties in defining a behavior, the large number of extracted features (when using n-grams, etc.), and the difficulties in identifying the similarities and differences among the extracted properties have prevented the creation of an effective detection system [27]. In addition, there are several obfuscation techniques that prevent malware from running in a monitored environment. Only today, as ML is starting to spread in the field of cybersecureity, this technique is starting to develop and to be studied by several researchers.

3.2.3 Heuristic-based malware detection

As has been shown, signature-based and behavior-based malware detection have some drawbacks. Hence, heuristic malware detection methods are proposed to overcome these disadvantages [29]. It is a complex detection method which uses experiences and different techniques such as rules and ML techniques. Although it has a high accuracy rate to detect zero-day malware to a certain degree, it cannot detect complicated malware [27]. The heuristic detector developed by cyber analysts is based on several features:

- **File based**: can be use the path of files opened and/or wrote from malware.

- **Weight based**: the weight of the file can be useful information to understand if the suspicious file can belong to a specific family of malware.
- **Generic signature**: It is also possible to integrate the use of the signatures seen in the signature-based detection.
- **Rule based**: rules created by behavior-based detection can also be included.

Evaluation of heuristic-based detection

Heuristic-based schema can use both strings and some behaviors to generate rules, and based on that rules it generates signature. It uses API calls, CFG, n-grams, Opcode, and hybrid features when generates a signature. Although the heuristic-based detection can detect various forms of known and unknown malware, it is insufficient to detect all new generation of malware. In addition, heuristic-based approaches are prone to high false positive rate [27].

3.2.4 Model checking-based malware detection

Although model checking is originally developed to verify the correctness of system against specifications, it has been used to detect malware as well [27]. With this detection technique, the behavior of the malware is extracted thanks to the use of malware analysis tools (static and/or dynamic). These features are encoded using the logic applied in model checking such as linear temporal logic, compute tree logic etc.

The behavior of the sample under study is created by looking for the flow of execution of the system calls for example, or other properties such as the values of the registers. Model checking-based detection can detect some new malware to a certain degree, but cannot detect all new generation of malware [27]. These features are then used to build the model checker which, unlike detection based on heuristic methods, does not use machine learning or data mining.

Evaluation of model checking-based detection

This approach is generally used for program verification and not used sufficiently for malware detection. Although it is effective to detect some new malware variants, it is still insufficient to detect all complex malware [27]. Furthermore, it is a long and complex process as the extraction of features and transformation into models understandable by the model checker must be done manually.

3.3 Malware detection and classification using Machine Learning and Deep Learning

In the last decade or so, the use of machine learning as a solution to the problem of malware detection and classification has increased exponentially. The problem of zero-day malware has led scholars to find new solutions as those shown above are not very effective against this threat.

The success and consolidation of machine learning approaches would not have been possible without the confluence of three recent developments [24]:

- The first development concerns the growing spread of labeled malware available on the web. As of today, it is no longer just the security community that has possession of the malware binary files with labels that catalog the malware, but also the research community. This implies that more people can try their hand at creating their own ML tool to achieve promising results for the future. An example of what has just been said is the dataset distributed by Microsoft for the Big Data Innovators Gathering Anti-Malware Prediction Challenge [30] or the VirusShare [31] site which gives access to many malware.

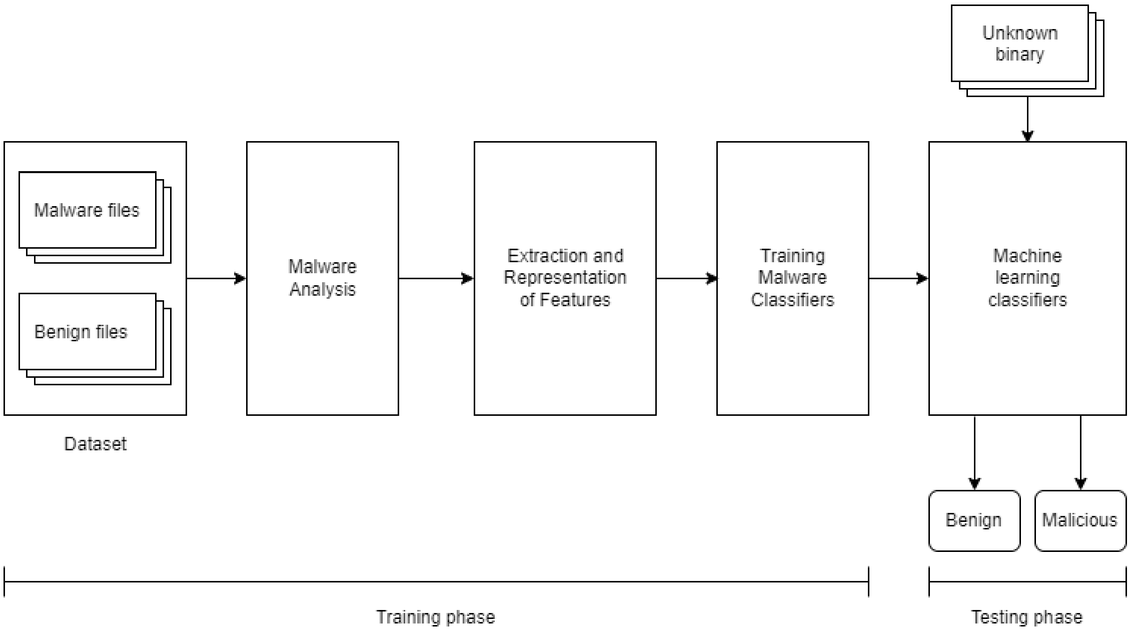


Figure 3.5: Schematic workflow of Malware Detection System using Machine Learning.

- The second development is the increase in computational power that has occurred in recent years. The rapid growth in computational power has also lowered hardware costs and this has led to powerful computers in the hands of researchers. Consequently, it allowed researchers to speed-up in the iterative training process and to fit larger and more complex models to the ever increasing data [24].

- Third, the machine learning field has evolved at an increased pace during the last decades, achieving breakthrough success in terms of accuracy and scalability on a wide range of tasks, such as computer vision, speech recognition and natural language processing [24].

Machine learning algorithms provide more option and space for developing a more accurate model by considering more features of malware and benign samples. ML has a very wide scope in the field of computer security like for malicious URL detection, intrusion detection and malware detection [32].

If you want to understand the complete scheme of the workflow you can look at the Figure 3.5. As in many machine learning researches, the first step is to find or create a dataset that can contain as many samples as possible and that is as representative of reality as possible. The second phase occurs if there are no features already in the dataset obtained during the previous step. In fact, if the dataset contains only binary files, it will be necessary to analyze them in order to decide which features to extract and how to do it. Then we look for an automatic way to extract the features if they are not already available. Finally comes the main phase in which the tool is trained and subsequently tested to check that the training phase has been carried out correctly. At this point, if the tool has reached acceptable levels, you can decide to use it to classify future unknown files.

As it has been said several times, a major benefit of using machine learning in malware detection is that it can develop a model to detect unknown malware. The reason is that it consists of several algorithms which can be applied to the wide variety of malware features set and produces better malware detection results [32]. Besides, there are many other advantages:

- Existing anti-virus and sandbox techniques can be subverted. As mentioned several times, these techniques are very complicated to use

- Automates extracting insight from malware samples.

- Can better generalize at identifying unknown variations.

- It can reduce human effort and time for analyzing the malware.
- Research work has been done by many researchers which shown the significant confidence in the malware detection.

3.3.1 Challenges to implement ML in malware detection

These solutions seem to bring many benefits but obviously there are obstacles to overcome. In particular, these limitations stem from the physical limitations of today's computers and the ability of attackers to continuously improve their products. There are five challenges in particular that are still a problem.

Class imbalance

Obtaining good training data is one of the most challenging aspects of any machine learning problem. Machine learning classifiers are only as good as the data used to train them, and reliable labeled data is especially important for the task of malware detection, where the process of labeling a file can be a very time-consuming process [24].

Another problem is the proportionality of the samples. It is possible that in your dataset there are a number of benign samples not equal to the number of malicious samples, or that in a dataset there are more samples of a certain class than samples of another. This is known as the class imbalance problem. A concrete example can be seen in the Microsoft dataset [30] in which there are 2478 samples of the Lollipop family and only 42 samples of the Simda family. This kind of distribution, where one class much larger than the other(s) can lead to a model that predicts the value of the majority classes for all predictions and still achieve high classification accuracy while lacking predictive power [24].

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 [24].

Open and public dataset

The task of malware detection and classification has not received the same attention in the research community as other applications, where rich benchmark datasets exist [24]. This situation is got worse thanks to the legal restrictions that are around benign binaries. Copyright laws prevent the free sharing of software produced by official companies, such as all Adobe, Microsoft, etc. software. Malicious binaries instead can be found on some sites, like VirusShare. Nevertheless, both benign and malicious binaries may be obtained in volume for internal use only through services such as VirusTotal, but subsequent sharing is prohibited [24]. If one looks what is being shared by VirusShare (or other source like it), it can be seen that only binaries are easy to obtain. In fact, sites like VirusTotal that provide data detected by anti-malware software about a certain malicious file have many restrictions on their use. For example, data about malware and information obtained about them cannot be shared. Also, trying to manually extract data from malware takes a long time.

All this has led to a lot of research in this field but all very difficult to compare. Since everyone develops their own method for extracting features, it is not easy to evaluate only the machine learning solution used. For example, if a researcher extracts the features using a certain tool, he will be able to obtain excellent results on his samples but if he will carry out the same test on other datasets used by other researchers he will be able to obtain poor results. At the present time, the only standard benchmark available to the research community regarding Windows Portable Executables is the one provided by Microsoft for the Big Data Innovators Gathering Anti-Malware Prediction Challenge [24]. Unfortunately, this dataset also has limitations. To neutralize them, binaries without headers have been inserted, in this way those who download them should not run into danger on their computer. In consequence, researchers are constrained to using only the provided byte code and disassembly files (generated with the IDA Pro disassembler) [24].

Concept drift

In the machine learning literature, the term “concept drift” has been used to describe the problem of the changing underlying relationships in the data [24]. In machine learning, the output is mapped starting from samples given as input to the algorithm. There is therefore a mapping function that, received at the input of the data, predicts the output values. In machine learning (especially in digit classification, text categorization or speech recognition) it is assumed that the training data are a good representation of reality and who therefore once the model has been sufficiently trained it will no longer have to be updated. In other words, they assume that the mapping learning from historical data will be valid for new data in the future and that the relationships between input and output do not change over time. This is not true for the problem of malware detection and classification [24].

As software updates, the malware that exploits their vulnerabilities also evolves accordingly. During the life phase of a software, many updates are carried out, usually just to fix bugs that can lead cyber criminals to exploit them in order to damage or steal information from the software. This can therefore lead to an obsolete mapping function that needs to be updated according to the new standards. Obviously it must be said that this change is very slow. With an update you change a few lines of code, the behavior remains very similar so a system based for example on API function calls will continue to work. But in the long run it can be a big problem. Thus, in order to build high-quality models for malware detection and classification, it is important to identify when the model shows signs of degradation and thereby it fails to recognize new malware [24].

Cybercriminals have also understood the dangers of this malware identification and classification technique, so they have begun to develop techniques to counter it. They also started using machine learning to increase the threat of their malware.

Adversarial machine learning

Adversarial machine learning is a big problem in machine-based malware classifiers. Adversarial machine learning means when malware developer uses the tactics of ML to bypass the malware detector [32]. 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 [24].

Defenders therefore find themselves having malware that possess the ability to understand if there is a detection system in the targeted device or if it does not have one. This is an undeniable fact that there is no other way except machine learning to detect the present malware which is highly complex. Also, the pace of malware development is very high. Now the thing is how we tackle these challenges to implement ML in the cybersecurity domain [32].

Attacker ML systems require knowledge of the identification and classification system on the target computer. For instance, consider a machine learning approach that relies on the program’s invocations of API functions or the DLLs dynamically loaded by the executable. An attacker might use this information to conceal the usage of any suspicious API function by packing the executable and leaving only the stub of the import table or perhaps even no import table at all. These modifications to the feature space can be manually performed or not.

Defenders in turn have found ways to combat these malware. The first way is to decrease the number of data samples used for training. If you use too many samples paradoxically our defensive system will be much more specific, so changes like the ones mentioned in the previous example will work very well. Only this is not enough, it is also necessary to use hybrid systems, that is, using both static and dynamic features. Malware detector based on a single ML algorithm can be bypassed however the ensemble malware detector can be more resilient to handle the adversarial machine learning [32].

Interpretability of the models

The interpretation of machine learning models is a new and open challenge. Most of the models used at the present time are treated as a black box [24]. In the field of cybersecurity we might think that it is a good way for cyber criminals not to have access to the functioning of detection and classification systems but in reality they do not. This could pose a problem in cybersecurity applications when a false alarm occurs as analysts would like to understand why it happened. The interpretability of the model determines how easily the analysts can manage and assess the quality and correct the operation of a given model [24].

3.4 Machine learning classification algorithms

In the literature, various machine learning algorithms have been used to train malware classifiers such as Naive Bayes (NB), k-Nearest Neighbours (k-NN), Support Vector Machine (SVM), Decision Tree (DT), Random Forest (RF) and Artificial Neural Network (ANN). This section presents a brief description of how they work, their advantages and disadvantages.

3.4.1 Naive Bayes

Naive Bayes classifiers are a family of classifiers which work on probability and are based on the Bayes theorem [23]. They assume that the set of features used as input to the algorithm are independent of each other. Bayes' theorem (from which the classifiers take their name) is used to calculate the probabilities:

$$P(A_i|E) = \frac{P(E|A_i)P(A_i)}{P(E)} = \frac{P(E|A_i)P(A_i)}{\sum_{j=1}^{n} P(E|A_j)P(A_j)} \tag{3.3}$$

As shown by the equation above, the probability of each class and the conditional probability of each data instance with class are calculated. Then, the prediction of a class is done with cumulative probability which is calculated by multiplying the class probability and conditional probability of each contributing data instance [32].

Advantages

Implementing Naive Bayes classifiers is very simple, and they are also simple to understand if someone needs to study examples that have already been created. It can perform well with irrelevant dataset. Also, the classifier can be trained using the small size of the dataset [32].

Disadvantages

The main limitation of NB classifier is that it does not perform well if the data features in trained data are correlated to each other [32]. In fact, as previously mentioned, independent features must be used. For example, using API function calls in conjunction with register values will yield poor results as the register values are extracted immediately after an API function call

3.4.2 K-Nearest Neighbour

K-Nearest Neighbour (K-NN) classification algorithm classifies the input instance by considering the class label of k nearest training instances [32]. The basic fundamental behind this algorithm is that two objects which belong to the same class have some commonalities which can be detected based on some distance metric [23]. The class of input instance is predicted as of the class of majority instances. Distance measures Euclidean, Manhattan, Hamming and Minkowski are used to find the class label of an input instance from nearest K nearest instances [32]. **K** here represents the number of nearest neighbors which is the main factor and its value is generally small odd numbers like 1, 3, 5, or 7 [23]. An example is shown in the Figure 3.6.

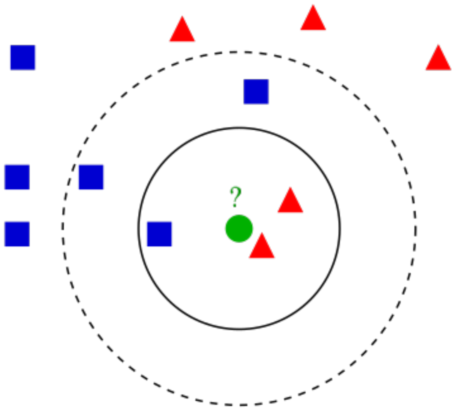


Figure 3.6: Example of K-NN with k = 3 with the contiguous line or k = 5 with the dashed line .

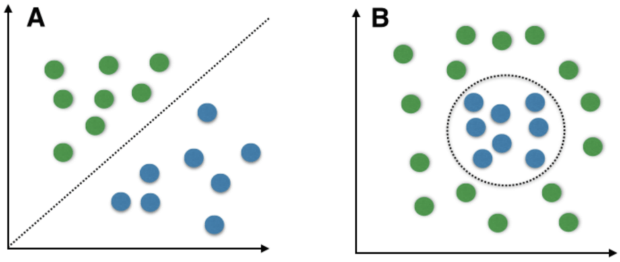


Figure 3.7: Linear (A) and polynomial (B) SVM.

Advantages

Implementation of K-NN is very simple and can be updated at a very low cost as new instances with known class labels. K-NN algorithm does not make any assumption about the dataset. It is more robust to search space means do not need to be linear separable dataset [32].

Disadvantages

The main drawback of the K-NN algorithm is that it does not perform well if a dataset is unevenly distributed. Also the selecting the appropriate value of K is unpredictable [32]. t it becomes significantly slow when there is a large number of entries [12].

3.4.3 Support Vector Machine

The support vector machine is an ensemble that tries to create a hyperplane to create a clear separation in space between the various different classes. There are 2 types of SVM (Figure 3.7): linear SVM and polynomial SVM transforming the polynomial problem into a linear problem. The first type uses linear hyperplanes to divide the space into 2 different classes, while the second type uses polynomial hyperplanes.

For binary classification, a vector of points on two-dimensional input space can be visualized which separate the input data instance into two different classes benign class and malware class [32]. In the case of malware classification, it is very difficult to use linear SVMs because the samples are very dispersed in the sampling space.

Advantages

SVM is the most promising Classification algorithm which produces good accuracy in classification. It can perform well with high dimensional dataset and can classify the non-linear separable

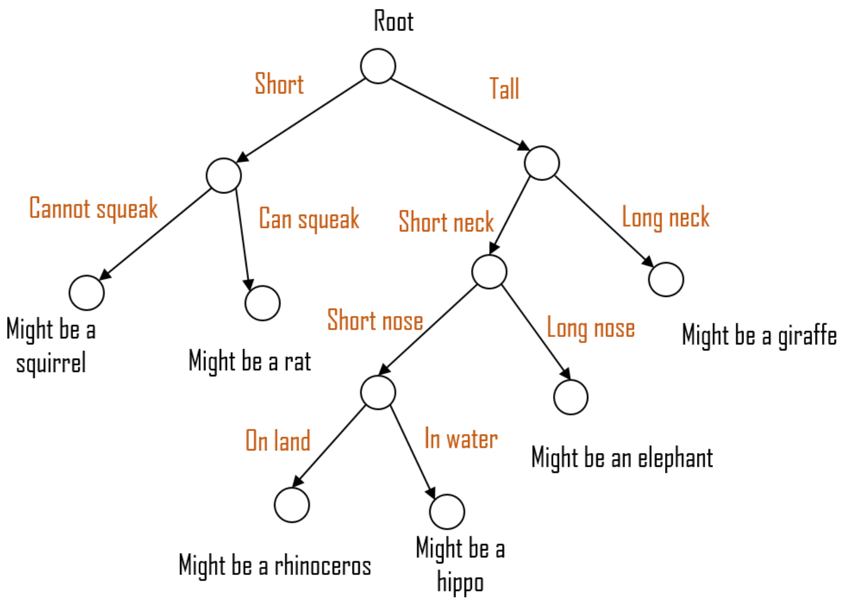


Figure 3.8: Example of decision tree.

data as well. Selection of the regularization parameter varies from problem to problem [32].

Disadvantages

Training time is getting large in SVM with a large value of penalty parameter (Penalty parameter of the error term. Can be considered as the degree of correct classification that the algorithm has to meet or the degree of optimization the the SVM has to meet). Also, choosing the value of C is a trade-off between testing and training error.

3.4.4 Decision Tree

This classifier is represented as a tree in which the internal nodes are represented as conditions to be evaluated, while the leaves represent the class to which to assign the sample (Figure 3.8). In decision tree classification, a decision tree is created by computing the info gain of each attribute in datasets. The attribute has maximum info gain becomes the root. Then other becomes a leaf of the root. Then, the created decision tree is used for making class predictions [32].

This tree structure helps a lot in making decisions using the divide and conquer technique. We consider an unknown variable A1 whose class must be assigned, starting from the root node, one must choose the path according to the characteristics of A1 and go down the tree until a leaf node is encountered and upon reaching the leaf node assign the 'label' of the class corresponding to A1.

Advantages

Decision Tree classifiers can perform well with high dimensional dataset also with noisy data. Unlike K-NN and SVM classifiers, it works as a white box. The interpretation of trained can be done. Because of this analysis of the trained model can be elaborated. Training speed of Decision Tree classifiers is also fast [32].

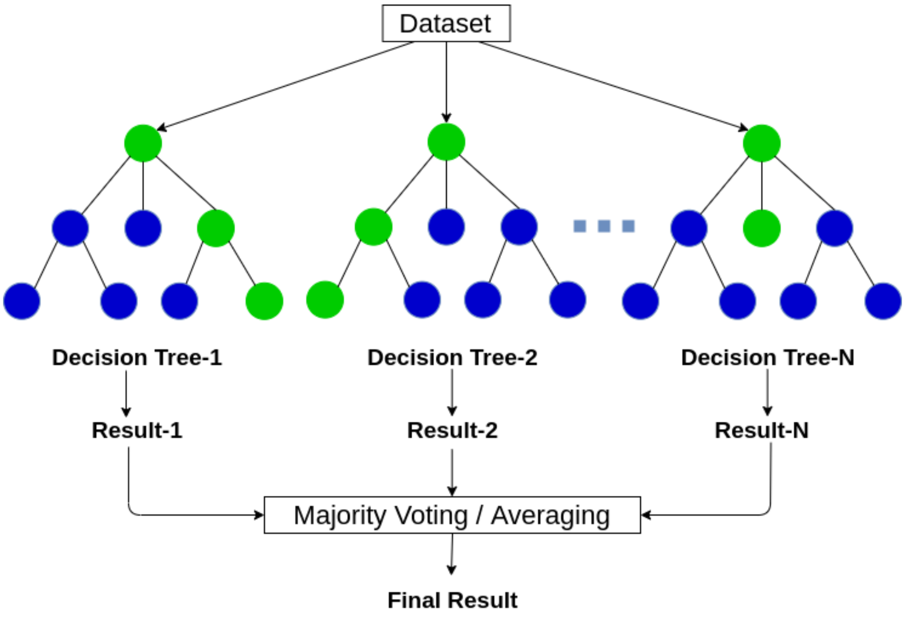


Figure 3.9: Example of random forest decision process.

Disadvantages

A small change in the dataset can cause a large change in the structure of the decision tree which causes instability of the model. It does not perform well with the small number of data features [32].

3.4.5 Random Forest

It is an updated version of the decision tree in which instead of creating a single tree, a certain number are created, creating a collection of decision trees. There are two types of randomness, the selection of input variables and the bootstrap of samples. In fact, the selected variables will not be the same for all trees but a certain amount of samples will be assigned to each tree.

Advantages

RF machine learning algorithm is immune to variation in the dataset because selects the multiple subsets randomly which reduce the risk of over fitting that why It gives the best classification results. Unlike the decision tree, it can provide good results if there is a change in the dataset [32].

Disadvantages

Training speed is slow. It is directly dependent on the number of classifiers which are trained to build the strong classifier. It is not transparent as decision tree because of the number of decision trees are computed to builds a strong modal. That is why it becomes difficult to interpret the results [32].

3.4.6 Artificial Neural Network

Artificial Neural Networks are the process models which mimics the human brains working to facilitate for finding the decision boundaries by minimizing error rate [32]. Neural networks have a layered structure, including an input layer and an output layer, plus other intermediate layers called hidden layers. A neural network, therefore, obtains input information that is transmitted to all neurons (the individual nodes of the neural network) by training the network for the testing phase. These neurons are divided into levels, the first level contains the input nodes, the intermediate levels contain the nodes where the input is transformed and processed and the output level contains the nodes in which the answers are provided.

There are several types of neural networks, in the malware classification mainly 2 are used (other techniques have never been successful):

- **Recurrent Neural Network (RNN)**: It is a class of neural networks in which information from the past remains in the memory of the nodes, creating a kind of effect similar to feedback. Obviously this increases the accuracy of the neural network, however, increasing the performance cost.
- **Convolutional Neural Network (CNN)**: Widely used in the field of image classification, in fact in the field of malware classification it is mainly used with the use of malware transformed into images. Filters are used to search patterns in images taking into account different areas of the picture in order to find a common factor between the various samples of the same class. This type of neural network uses a certain number of resources based on the size of the images given as input. For this, techniques to reduce the resolution of the images are used.

Advantages

Artificial Neural Network can model the non-linear dataset of large number of input features. The Artificial Neural Network can be used almost every kind of problem specially for the optimal problem [32].

Disadvantages

Artificial Neural Network might be lead to over-fitting, the weights established for the training data may not be generalize the other datasets, even from the same populations. In the end, this technique is computationally expensive [32].

Chapter 4

Dataset

This chapter will introduce the main datasets that have been evaluated to carry out this research and which ones have actually been used. The datasets most commonly used in this type of research were evaluated, in addition an attempt was made to use another dataset created specifically by another thesis student of the Politecnico di Torino called ST-WinMal. Finding a dataset was not a trivial operation, since the goal was to find one that could provide as many features as possible having already calculated labels. Furthermore, we wanted to look for a dataset that also provided the binaries, so that we could also try to manually extract some features such as byte ngrams. In summary, the goals were:

- Own the binaries and if possible the disassemblies of malicious and benign files
- Have access to features extracted using tools like IDA Pro
- Have a large number of files available.

The datasets evaluated with all their characteristics, advantages and disadvantages will be listed below.

4.1 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 [33]. Labels derived from multiple sources are also provided, i.e. from different vendors of antivirus or malware detection tools. The complete dataset counts 12,699,013 training samples, 2,396,822 validation samples and 4,195,042 test samples, more than enough numbers to be able to carry out many types of analyzes on them. Unfortunately, not all of these samples have their respective binary file associated. However, the authors of the dataset managed to solve this problem by providing 9,919,251 samples of malware binaries. The distribution of benign and malware files can be viewed in Table 4.1.

The samples are identified by hashes, calculated with sha256 to be precise. In fact, the primary key in the database is the hash. The data was collected from January 1, 2017 to April 10, 2019 [33].

Unfortunately this dataset had a problem that should not be underestimated for this type of research. In fact, the labels associated with the malware were not the families as I researched, but the type of malware (for example, adware, ransomware, etc.). The malware present in this dataset are in fact part of the following types of malware: adware, flooder, ransomware, dropper, spyware, packed, crypto miner, file infector, installer worm and downloader.

In summary, the advantages of this dataset are its size and the presence of already extracted features which, however, were not the ones that interested me the most. In fact, the disadvantages I found are the lack of labels on the malware families and the features regarding API or ngram i opcode. I have therefore decided not to use this dataset.

Set	*Malicious*	*Benign*
Training set	1,360,622	5,102,606
Validation set	962,222	1,533,579
Test set	1,360,622	2,834,441

Table 4.1: Distribution of malware and benign samples across SOREL-20M dataset

4.2 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 [31]. For safety reasons, access to the site is granted by invitation only. To request access, it is therefore necessary to write an email to the site administrators explaining the reasons why you want to access. The administrators will check the request and decide whether to send you the credentials to be able to access. All samples are delivered in password-protected zip-files for safety (For example, VirusShare_00000.zip).

Inside each zip are 131,072 samples if the number in the zip name is between 0 and 148, while there are 65,536 in zips numbered 149 and up. Virus share then provides many malware run them samples to download via torrent. Higher-numbered zips contain newer samples, so you can study new and old malware.

Even with this dataset, therefore, you have access to a large number of malware, such as for SOREL-20M but unfortunately there is no label on which malware is inside the zip and which family it belongs to. To be able to tag malware, you need external support, specifically the help of vx-underground. On this site it is possible to access the txt files associated with the individual VirusShare zips. Inside the text file there are as many lines as the malware present inside the zip, and in each line there is the corresponding hash of the malware and the family it belongs to, together with other useful information such as the size of the malware.

The main disadvantage of this dataset is the prohibition of creating scripts to quickly download the malware needed to compose a dataset. In fact, it is necessary to try to choose families in advance to compose a dataset that can be used later in the research. The site therefore forbids launching scripts for the automatic download of malware and this means that the construction of the dataset takes too much time. Furthermore, the features are to be extracted later, this requiring other work outside the scope of this thesis. It was therefore chosen not to use this dataset, but the information was useful for the creation of the ST-WinMal dataset mentioned in the introduction of this chapter.

4.3 VirusTotal

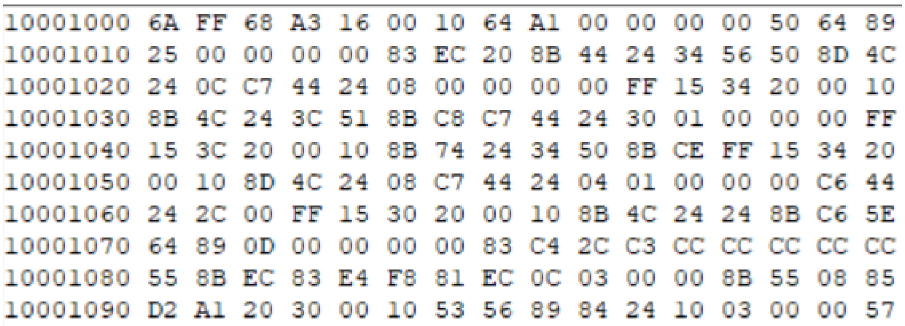
VirusTotal [34] is not a real dataset, but a site used for the analysis and study of malware. On their web page, in fact, it is possible to upload files to obtain a report on their danger. The site also provides a very large dataset of malware samples. In order to access the VirusTotal dataset, you must contact the administrators through the appropriate section, and motivate your request. For research reasons it is easy to obtain their approval.

VirusTotal provides information that has been extracted from different antivirus vendors, such as Kaspersky, Avg etc. After making a request they give access to a Google Drive Folder with different malware cataloged in different zip files. A significant advantage is the possibility of having direct access to the malware code, therefore to C files, java etc.

Here the main disadvantage is the number of samples and the fact that you have to request a form on the site in order to know the family of a particular malware by entering its hash. This disadvantage led me not to use this dataset as there was a limit on the possible requests to be made on a daily basis.

Family Name	*Number Train Samples*	*Type*
Ramnit	1,541	Worm
Lollipop	2,478	Adware
Kelihos_ver3	2,942	Backdoor
Vundo	475	Trojan
Simda	42	Backdoor
Tracur	751	Trojan Downloader
Kelihos_ver1	398	Backdoor
Obfuscator.ACY	1,228	Any kind of obfuscated malware
Gatak	1,013	Backdoor

Table 4.2: Malware families in Microsoft Malware dataset



```
10001000 6A FF 68 A3 16 00 10 64 A1 00 00 00 00 50 64 89
10001010 25 00 00 00 00 83 EC 20 8B 44 24 34 56 50 8D 4C
10001020 24 0C C7 44 24 08 00 00 00 00 FF 15 34 20 00 10
10001030 8B 4C 24 3C 51 8B C8 C7 44 24 30 01 00 00 00 FF
10001040 15 3C 20 00 10 8B 74 24 34 50 8B CE FF 15 34 20
10001050 00 10 8D 4C 24 08 C7 44 24 04 01 00 00 00 C6 44
10001060 24 2C 00 FF 15 30 20 00 10 8B 4C 24 24 8B C6 5E
10001070 64 89 0D 00 00 00 00 83 C4 2C C3 CC CC CC CC CC
10001080 55 8B EC 83 E4 F8 81 EC 0C 03 00 00 8B 55 08 85
10001090 D2 A1 20 30 00 10 53 56 89 84 24 10 03 00 00 57
```

Figure 4.1: Example of bytes file of Microsoft Malware dataset.

4.4 Microsoft Malware Classification Challenge

Microsoft announced in 2015 the Microsoft Malware Classification Challenge, with which it published a dataset of the size of 0.5 terabytes, containing different malware belonging to 9 families. In addition to being used in the competition, this dataset was also used by various researchers, in fact in 2018 the papers that cited this dataset were more than 50 [30].

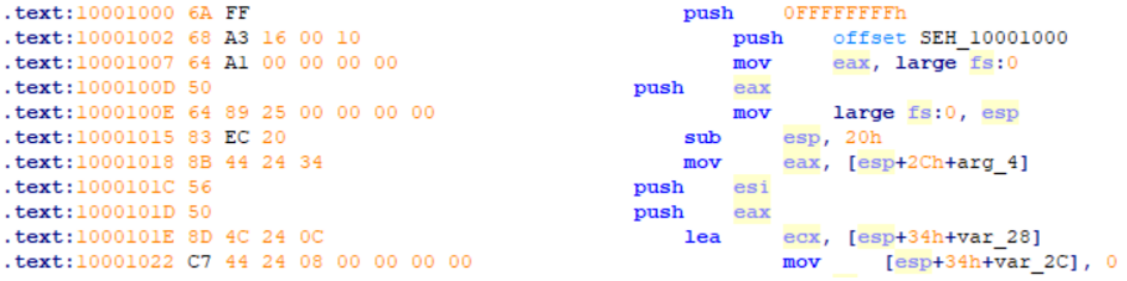
The dataset consists of a set of known malware files belonging to 9 different families. Each malware file has a unique identifier, namely a 20-character hash value, along with a tag (an integer between 1 and 9) that identifies the family of that malware. Unfortunately, since it was designed for a competition, the test set does not have labels so that the participants did not have the solution and had to insert a document that associated each malware of the test set which family had been predicted.

Because of this, only the train set is available and you have to split it in your code into train, validation and test sets. There was an attempt to request access to the test set labels but there was no response from the authors of the paper [30] that described this dataset. The distribution among the 9 families of available already tagged malware can be seen in the Table 4.2

For each malware two files are given, one contains the hexadecimal representation of the content of the binary file (see Figure 4.1) of the malware without header (to make it harmless), the other contains instead the disassembled extracted using the IDA disassembler tool [35] (see Figure 4.2). The disassembled file contains various metadata such as function calls, strings etc.

As you can already see this dataset offers a lot of versatility as you can extract many features such as ngrams of bytes, opcodes and function calls, entropy can be calculated and also the grayscale graphical representation. It is also easily downloadable from the Kaggle website [36] and being designed for a classification competition it was created specifically for the purpose of this thesis.

The main disadvantages of this dataset are the uneven distribution of malware among the various families, in fact having only 42 samples of the Simda family against 2,942 of Kelihos version 3 is certainly not a real representation of the malware world.



```
.text:10001000 6A FF                                      push     0FFFFFFFFh
.text:10001002 68 A3 16 00 10                                 push     offset SEH_10001000
.text:10001007 64 A1 00 00 00 00                              mov      eax, large fs:0
.text:1000100D 50                                     push     eax
.text:1000100E 64 89 25 00 00 00 00                           mov      large fs:0, esp
.text:10001015 83 EC 20                                   sub      esp, 20h
.text:10001018 8B 44 24 34                                mov      eax, [esp+2Ch+arg_4]
.text:1000101C 56                                     push     esi
.text:1000101D 50                                     push     eax
.text:1000101E 8D 4C 24 0C                                lea      ecx, [esp+34h+var_28]
.text:10001022 C7 44 24 08 00 00 00 00                        mov      [esp+34h+var_2C], 0
```

Figure 4.2: Example of asm file of Microsoft Malware dataset.

However, this disadvantage did not make me take the decision to continue the search for another dataset since the advantages it offered me were many and it met all the objectives that I had set myself in the search for a dataset. However, another dataset was used in addition to this one created, as written in the introduction to this chapter, by a thesis student at the Politecnico di Torino.

4.5 Image dataset

In addition to the malware binary datasets, in my research I found a grayscale representation malware dataset called MalImg. It contains images representing malware from 25 families, for a total of 9,435 samples. However, the malware executables have been converted to 32 x 32 images using the nearest neighbor interpolation from Kaggle website. In thee Vision ResearchLab website [37] there are the original converted images before compression [38].

To create this type of dataset, the files must be read by bytes, creating a byte vector. Subsequently each byte is interpreted as a black or white pixel. Finally the vector is clipped to create a 2D image. A summary of the binary to grayscale image conversion process can be seen in Figure 4.3.

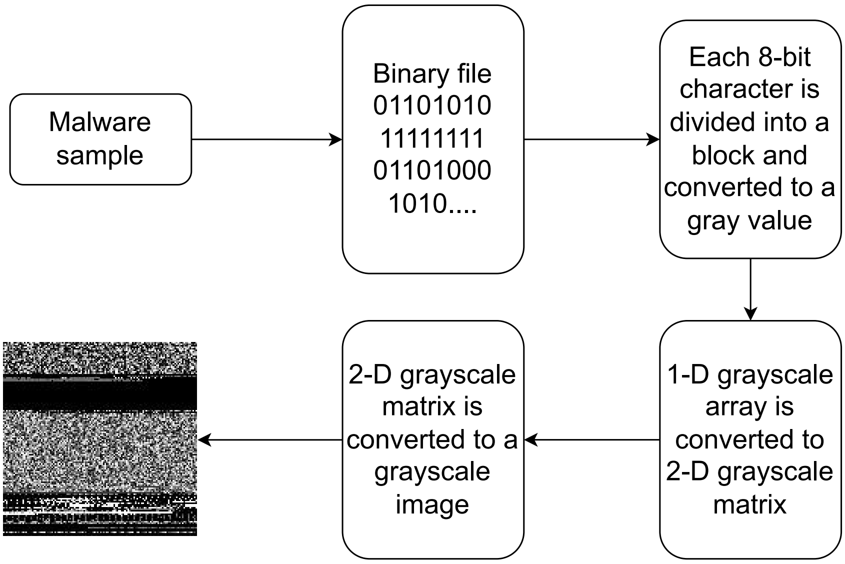

Malware sample
Binary file 01101010 11111111 01101000 1010....
Each 8-bit character is divided into a block and converted to a gray value
1-D grayscale array is converted to 2-D grayscale matrix
2-D grayscale matrix is converted to a grayscale image

Figure 4.3: Grayscale image conversion process.

4.6 DikeDataset

DikeDataset is a labeled dataset containing benign and malicious PE and OLE files [39]. Since the Microsoft dataset doesn't contain benign files and one of my goals is the identification of

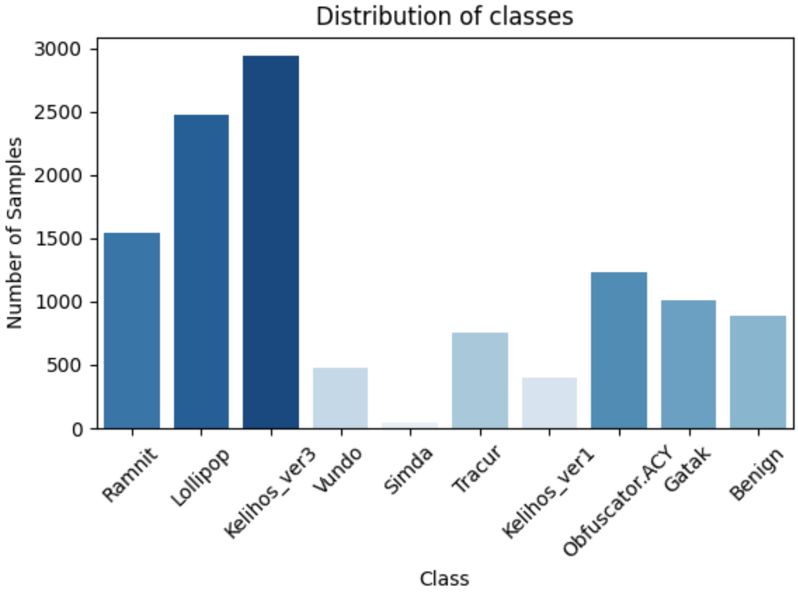


Figure 4.4: Distribution of classes.

malware through Machine Learning algorithms, I looked for a dataset containing the binaries and disassembled files of benign samples. The goal is to find a dataset containing a number of samples approximately equal to 1,000, in order to have a distribution similar to the number of malware samples in the different families. Unfortunately I was not able to find the disassembled files of the benevolent files but I managed to find some benign files together with my colleague Andrea Sindoni of the Politecnico di Torino, author of ST-WinMal described in the following section.

This dataset contains 982 benign PE files, which I add to the Microsoft Malware dataset to perform malware classification as well as identification. Following the union of these samples to the previous dataset, I obtained a data distribution as in Figure 4.4

4.7 ST-WinMal

This dataset contains malware targeting one precise operating system: Windows. Samples inside this dataset are selected from most common families on the internet. The author examined the most recent Kaspersky reports to understand which families are most common and dangerous [40]. He concluded that Trojans and ransomware are the most dangerous and common malware threats affecting Windows. From these two types of malware, it selected four families of Trojans and four families of ransomware. The malware sources used to create this dataset were varied, from malware collections such as VirusShare to datasets already created by other authors.

To be precise, the sources were:

- VirusShare
- MalwareBazaar
- DikeDataset

From the dataset I was able to receive three types of files for each malware:

- A file containing the binary, transformed into a file containing the byte sequence in hexadecimal format (like Microsoft's dataset for its challenge).

Trojan		Ransomware	
Family Name	*Number of Samples*	*Family Name*	*Number of Samples*
Ramnit	1,500	Virlock	800
Zbot	1,200	Stop	1,000
IceId	1,000	Magniber	808
Trickbot	800	Wannacry	1,533
Benign			961

Table 4.3: Malware families in ST-WinMal

- A file containing the entropy calculated for each section of the malware (entropy for .text, .data etc.)
- A file containing calls to API functions within the malware, extracted by static analysis. For this, the file contains a list of APIs that are called at least once.

The number of samples per family can be seen in Table 4.3

This dataset like all others has advantages and disadvantages. The dataset has a very balanced balance, there are no families that are too numerous than others. At the same time, reading the author's thesis, we can see that the malware is very recent, which means that the results are also more reliable and lead to a much more up-to-date study of malware. Of course, as with all others (apart from SOREL-20M), the number of samples is much smaller than in reality. SOREL in this respect remains the only dataset with a very high number of samples. This problem arises from the fact that malware is not so easy to find on the web and moreover that it belongs to the same families. In addition, malware naming is not a well defined process, there is not a common standard. If you search for malwares on the Internet, they might be catalogued under different family names among several collections of malwares.

This dataset was used because it was a great way to compare the performance of my system with families from different times. The malware belonging to the dataset created by Microsoft for its challenge was released in 2015 so it contains malware collected up to that year, this dataset on the other hand has much more recent malware.

4.8 Summary

In general, searching for the dataset to be used in this thesis was not an easy task. Each dataset has its disadvantages and advantages, and to date there is no dataset that is superior to the others. For instance, there are datasets that are numerous but do not provide the desired information, or that do not even contain malware executables. Others are rich in information but have problems with class imbalance and also only provide some information. Some have tried to create their own datasets with VirusShare, a promising but time-consuming way of obtaining executables.

The availability of malware is not very high and it is not only necessary to make an effort to obtain the executables, but also to find a way to extract features from them, as described in the chapter on the state of the art. To solve this problem, my colleague tried to create a dataset to be used both for malware analysis using static and dynamic methods and for identification and classification using machine learning tools. Certainly, however, there is still room for improvement, which will be addressed in the conclusion chapter.

Chapter 5

Proposed Method

The main purpose of this thesis is to try to find one good machine learning algorithm to use for malware identification and classification. To do this it is necessary to use different techniques (Naive Bayes, SVM, etc.) in order to have a complete comparison between them. All this has been considered necessary but not sufficient, in fact we must also try to compare the various techniques with different combinations of features. There is no evidence that determines that a certain algorithm does not work with all the malware features available, it is also possible that changing inputs will give better results.

There was therefore a very specific workflow aimed at understanding what was the best methodology for the task entrusted to me. The first step was to study the files present in the dataset to understand what information could be extracted. Once this was done, I tried to find an efficient way to extract these malware features and save them in files to be able to later use them as input to the algorithms. Thanks to the study of the state of the art, I was already aware of the most used algorithms and therefore the next step was to use the inputs described above in the various known machine learning algorithms. The experiments were initially performed with a single malware feature at a time as input. If a feature gave poor results, it was verified that there was no different way to extract that specific information, in order to improve the input of the machine learning algorithms. Once definitive results were obtained, comparisons were made between the different experiments. Finally, experiments were made with the various combinations of features. A summary of the workflow can be seen in Figure 5.1

In this chapter I will describe which algorithms I used for each of the phases previously described. To be precise, the first section will discuss the motivations that led me to choose which features to extract from the datasets at my disposal, while the second section will talk about the implementation details of my tool, going on to specify which was the working environment in which all the extractions and tests were carried out, the support libraries I used and the algorithms implemented to extract the features from the datasets. Finally, the last section will deal with the machine learning algorithms used to identify and classify malware.

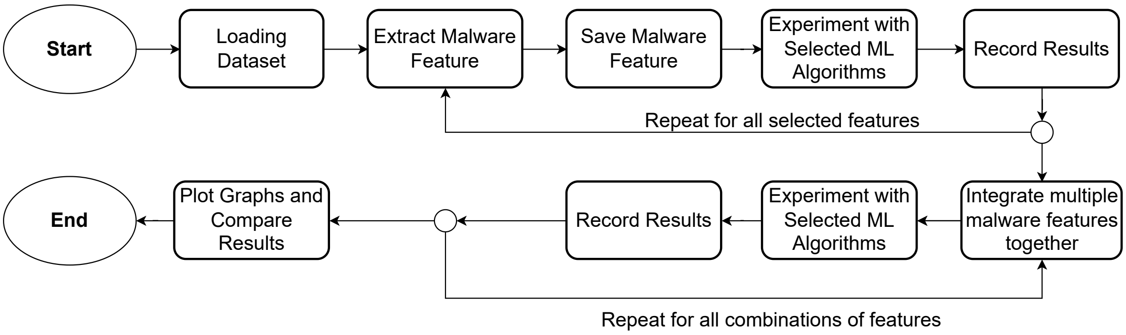

Start
Loading Dataset
Extract Malware Feature
Save Malware Feature
Experiment with Selected ML Algorithms
Record Results
Repeat for all selected features
Integrate multiple malware features together
Experiment with Selected ML Algorithms
Record Results
Plot Graphs and Compare Results
End
Repeat for all combinations of features

Figure 5.1: Workflow.

5.1 Extracted features: motivations behind the choices

To begin designing the tool to be used, the first step is to choose which information to extract from the files in the dataset. There is then to perform a preliminary analysis on the dataset in order to decide which features to extract to build the input of the machine learning algorithms.

Obviously, one first needs to find out whether it is possible to extract static features and/or dynamic features. Remember that static features involve analysing the program's executable with the possibility of extracting other files as disassembled files using tools such as IDA Pro, so they do not require anything to be run locally, but they do not allow us to get all the information if the author of the executable has used encryption or obfuscation techniques. Dynamic features, on the other hand, solve the problem of static features but involve running the malware inside a virtual machine called a sandbox, which means more work and a much longer analysis time. In addition, there are techniques to detect the use of a virtual machine.

I then carried out an analysis of my dataset, understanding which information could be extracted and which could not. If you had the malware executables available, it would be a good solution to try setting up a dynamic analysis tool so that it would create a virtual machine in which to execute the files in the dataset and monitor the operating system in order to be able to access, for instance, which API functions were called during the execution of the malware. Obviously in this case it will take a long time if the samples in the dataset are more than a few thousand files. If, like me, you have a dataset with which you can only extract static features, then you have to perform a process of selecting the features to be extracted. One must first try to hypothesise what information can lead to better results; studying the state of the art can be very helpful in this regard.

Static features can be obtained from the Microsoft Malware Classification Challenge dataset because there are non-executable text files available to the user. For each malware there are 2 files, the binary translated into hexadecimal characters and the corresponding disassembled file. In addition, the files were rendered harmless by removing the header from the binary, so that the user of the dataset runs no risk during the malware analysis process.

In the next subsections I will list the features I used and explain the reason behind my choice.

5.1.1 Binary file sizes

To start, I wanted to use a malware feature that was simple to extract and could already provide some indication of malware families: the size of the binaries. The idea behind the extraction of this feature is very simple: malware, as written in the introduction, are nowadays very numerous, one of the causes is the easy re-use of a given malware in a different sector thanks to simple changes in the code. Most malware belonging to one family is nothing more than code copied from its own kind, adapted for a different purpose. Thanks to this it is possible that two malware of the same family can have very similar dimensions.

5.1.2 Bytes n-grams

After extracting the size of the dataset's *.bytes* files I thought I'd try to get a feature that gave more information on a single malware. After studying the state of the art of this type of research I concluded that a good solution could be to extract n-grams of bytes. As already explained in chapter 2, an n-gram (sometimes also called Q-gram) is a contiguous sequence of n items from a given sample of text or speech, where each item can be a word, a letter, etc. In our case we choose to adopt a byte of the binary file as a single element.

Reflecting on the meaning of a file executable, it is merely the translation of assembly language into machine language, so extracting contiguous byte elements could be compared to extracting instructions one after the other. Of course, this statement is not entirely true as there is the possible use of obfuscation techniques to consider.

It should be remembered that byte n-grams and opcode n-grams are not poarithetic since assembly instructions can have different sizes. An example would be the following comparison:

- **83 EC 20** corresponds to the assembly operation **sub esp, 20h**
- **8B C8** corresponds to the assembly operation **mov ecx, eax**

This feature therefore brings advantages and disadvantages, the latter of which could be eliminated through the combination of other features to be given as input to the machine learning algorithm.

5.1.3 Entropy

File entropy measures the randomness of the data in a file and is used to determine whether a file contains hidden data or suspicious scripts. The scale of randomness is from 0, not random, to 8, totally random, such as an encrypted file [41]. The entropy value is larger when the system uncertainty greater. Compared with native code, packed and compressed segments tend to have larger entropy values, thus, entropy analysis can be used to quickly and efficiently identify packaged and encrypted samples [42].

Entropy can be calculated either over the whole file or for each section of the executable (.data, .text etc.). In my case both modes were tried, the first mode in one dataset, the other in the second dataset.

This technique can be seen as an alternative solution to the problems arising from the use of obfuscation techniques by malware authors. I say partial because in any case the result of an encryption or obfuscation algorithm is non-deterministic. I have found several papers that have nevertheless led to good results.

5.1.4 Opcode n-grams

As specified in the subsection describing n-gram byte extraction, there are differences between these 2 features. Opcodes do not coincide with a predefined byte sequence as the x8086 instruction set has assembly functions that have variable hexadecimal encoding, e.g. there are functions that are encoded to be 3 bytes long, others that are 6 bytes long. This made me take the decision to extract both features in order to try and see if there were any differences in performance and results. For opcode it was decided to consider only the assembly instruction, not the registers that were being modified by the individual functions. For example, if the complete operation is **mov ecx, eax** it was decided to extract only the string **mov**.

This decision was made to go against some obfuscation techniques (it should be remembered that one of the 9 families of malware is precisely the obfuscated malware family). There is in fact an obfuscation technique called register reassignment. It goes to change the registers used, going to replace the registers without changing the logic of the program. This would obviously affect our classifier in a negative way. The sequence of instructions in case this technique is used does not change since if an addition is to be done between two registers, the register reassignment will change the name of the operands but the function will remain the **add**. Of course in case techniques are used that create extra operations in the code that will never be executed, these will be taken into account since the analysis performed is static.

5.1.5 Opcode counter

The previous interpretation of the employment of opcodes in the study of malware is not the only one that can be used. Since it is possible to construct loops within the assembly code via jump functions, it is not certain that the sequence of n-grams extracted from the disassembled are the correct sequence of instructions during its execution. There are also many obfuscation techniques that go into inserting “useless” instructions for program execution but which are never actually executed because they are ignored through the use of jump functions.

One possible option is then to count the number of appearances of each individual assembly function, so the number of times the add function, sub function etc. appears. This figure is much less influential since it considers opcodes individually and not as sequences. Certainly, it does not completely solve the problem as a dynamic analysis of the opcodes used would. Obviously, such an analysis would take much more time.

5.1.6 API n-gram

As addressed in the chapter on the state of the art, calls to functions offered by the operating system can give us very important information on how a file works. How many files are opened, the number of times an attempt is made to allocate or deallocate memory, and so on, are all pieces of information that can tell us whether a particular file is part of a family or something else. It is precisely for this reason that we aimed to exploit this information provided by the dataset to extract such a feature.

Initially, it was not planned to use the n-grams of API as there was no such feature found in the state-of-the-art study, but I wanted to try to emulate a dynamic feature extraction system. The n-grams actually give a sense of dynamism as they collect a sequence of function calls (in my case 4 elements).

This feature is also very good for preventing obfuscation techniques from going against our malware identification and classification system. In fact, calls to API functions touch predefined memory addresses that are provided by the operating system. Even if a malware wanted to make an API call silently, it would somehow have to access a functionality external to it, i.e. the victim's operating system.

5.1.7 API check

Exactly as with opcodes, I decided to try to use a feature describing the number of times an API function is called. Also for this feature, I wondered whether the sequence of API function calls in the file was in the same order as in the execution of the malware. Here, too, the answer was uncertain as it is not certain that a subroutine was written at the beginning of the code section and then called from elsewhere.

Here the discussion does not deviate much from the concept of opcodes. The comparison of opcode n-gram and opcode counter is totally parallelizable to API n-gram and API counter.

5.2 Work environment and tools used in the implementation of the ML tool

This short section will introduce the working environment used to make the machine learning tool and the libraries used to derive the algorithms. In fact, it was not in the purpose of the thesis to create its own version of the algorithms used in machine learning identification and classification; instead, it was decided to employ algorithms that had already been implemented, had already been tested, and had obtained good results. In this way it was also possible to have a good comparison with the current state of the art.

5.2.1 Work environment: Legion

HPC@POLITO is a supercomputing initiative managed by DAUIN (Department of Automation and Informatics of the Politecnico di Torino) that provides high-performance computing resources and technical support for academic research and teaching activities [43]. It is not only addressed to research groups within the university but also to those outside the Politecnico di Torino. Of the three clusters made available by the HPC group, I used LEGION. It was designed to be modular; it is an InfiniBand cluster with the characteristics summarised in Table 5.1

All function tests and experiments were carried out on this cluster.

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

Table 5.1: Legion - Technical specifications

5.2.2 Support library used in the realisation of the ML tool: Scikit-learn

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 [44].

The library has been designed to tie in with the set of numeric and scientific packages centered around the NumPy and SciPy libraries. NumPy augments Python with a contiguous numeric array datatype and fast array computing primitives, while SciPy extends it further with common numerical operations, either by implementing these in Python/NumPy or by wrapping existing C/C++/Fortran implementations.

Other libraries were looked at before making the following choice, but scikit-learn seemed the easiest to use and achieved excellent results. Implementing their own algorithms was ruled out for reasons of time and knowledge. In fact, it was not even the main purpose of this thesis to write a machine learning algorithm from scratch, rather it was decided from the outset to use existing libraries.

The classes and functions belonging to this library, which were used during the realisation of the malware identification and classification tool, will be addressed in the following chapters in the sections in which they were utilised.

5.3 Malware feature extraction from Microsoft Malware Classification Challenge dataset

The first step is to extract the malware features to be used as input for our Machine Learning algorithms, finding a way to memorize them for all future experiments. Each sample of the dataset is represented by a vector of hand-crafted features, and together, all these vectors create a matrix $S \times F$ in which S represents the number of samples and F the number of features extracted.

Obviously, based on which malware features were extracted, the matrix changed size, for example in the case of the simple extraction of the size of the binary files this involved a one-column matrix (a vector per column), while in the case of n-grams of bytes we find many more columns, so much so that their dimensionality has to be reduced.

This section will deal with the methods by which the malware features were extracted from the malware samples, precisely from the bytes file containing the hexadecimal representation of the binary files and from the asm files containing the assembly code of the malware

To save all the matrices in which the malware features were loaded, the npz files were used. In fact, these files were created by the numpy library to be able to save several numpy arrays together in these archives. It is possible to read the pseudocode of the standard procedure (subsequently adapted for each feature) into Algorithm 1

Algorithm 1 Malware Feature Extraction

```
procedure MALWARE_FEATURE_EXTRACTION
    for all samples do
        $feature\_matrix$.append(extract_feature_function($sample$))
    end for
    $feature\_to\_save \leftarrow feature\_matrix$.to_np_array()
    save_npz($feature\_to\_save$)
end procedure
```

5.3.1 Binary file sizes

To extract this information, the use of the python os library was enough, thanks to the getsize function. This extraction was performed on the entire dataset, from both benign and malware files. Once the feature was extracted, it was possible to view the boxplot of the distribution of binary file size. The distribution can be seen in Figure 5.2.

From the graph it is possible to see that the hypothesis mentioned in section 5.1.1 is correct for some families while not for others. For example, if we look at the samples of Kelihos_ver1 (family number 7 in the graph) we can see that they almost all have a size between 5 and 6 MB. If instead we observe the Lollipop samples (family number 2) we notice that this rule is not respected, covering a larger range of dimensions.

The extraction of the following feature is very fast. In the working environment used by me described in section 5.1, it took 1.246 seconds. The results of the experiments carried out on this malware feature can be read in the next chapter, as well as all other attempts carried out with the other malware features.

5.3.2 Bytes 3-grams

The first step was to think about which n to choose, the possibilities were 2, 3 or 4. Respectively, matrices with 65,280, 16,581,120 or 4,195,023,360 columns would come out. Tests were carried out for all 3 possible choices and it was concluded that with 4 the time to extract the features and the required memory were too high. Subsequently it was tested which brought the best

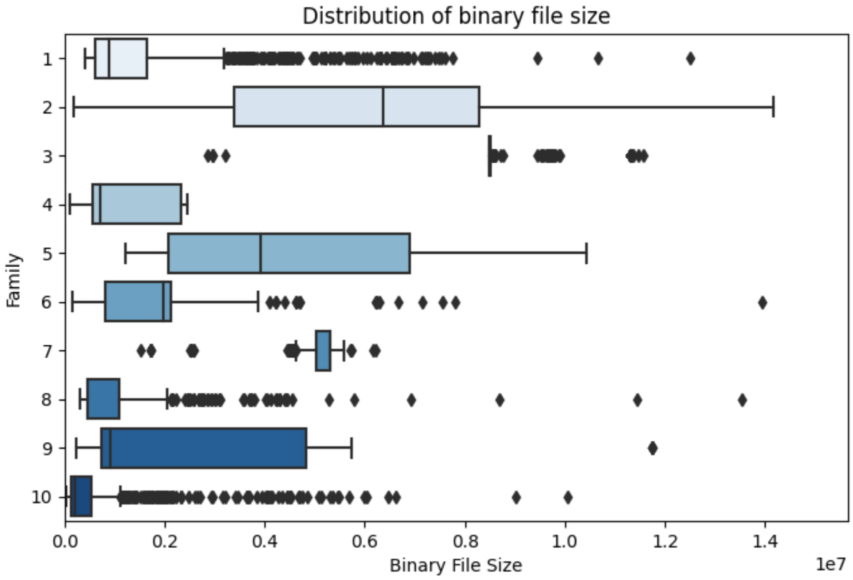


Figure 5.2: Distribution of binary file size.

Algorithm 2 Malware Feature Extraction

```
function EXTRACT_ENTROPY(id)
    $data \leftarrow$ read_file($id$)
    for byte in data do
        $possible$[int($byte$, 16)] += 1
    end for
    $data\_len \leftarrow$ len($data$)
    $entropy \leftarrow 0.0$
    for i in possible do
        if $possible[i] == 0$ then
            continue
        end if
        $p \leftarrow possible[i]/data\_len$
        $entropy$ -= $p \times log(p, 2)$
    end for
    return $entropy$
end function
```

results between grams of 2 or 3 elements. The results led to choose grams of 3 bytes. Being a feature that can only be obtained from the binaries, it was possible to extract from the complete dataset, i.e. from the malware files taken from the Microsoft dataset and from the benign files from DikeDataset. The time to extract this feature from the entire dataset (Micoroft Malware dataset plus DikeDataset) was 7 hours, 11 minutes and 55 seconds. The matrix obtained from this extraction had dimension $S \times N$ where $Srepresentsthesamplenumberand$N the number of different n-grams. In a cell [s, n] therefore it could be observed how many times the n-gram n appeared (example: 01 C3 48) in the sample s.

After carrying out the extraction it was possible to notice that the training phase of the algorithms required a very high calculation time. Precisely for this reason we wanted to apply a reduction in dimensionality, also in view of when the various matrices containing different features could have been merged. It was therefore decided to keep only 5,000 columns. This number was decided after various experiments discussed in the next chapter.

Columns that had n-grams appearing many times in the dataset were kept. To execute construct a vector as long as the columns of the matrix obtained and containing in each cell the sum of the corresponding column (index i of the vector contains the sum of the values in column i of the matrix). Then the indexes of the vector were sorted according to its content, only the first 5,000 were kept from the array of indexes obtained. Finally, only the columns having the index contained in the previous vector were traced.

5.3.3 Entropy

The last feature extracted only from *.bytes* files is entropy. File entropy can be calculated thanks to some libraries by giving the binary file to be analyzed as input, but it must be remembered that the files in the dataset were binary converted into hexadecimal. It was not possible for me to use these libraries but I had to search the internet for the formula used to calculate this value and then transform it into an executable algorithm. This procedure can be seen in pseudo code Algorithm 2

From the algorithm, one can see why the entropy file is a number between 0 and 8. To calculate this value, the occurrences of each individual byte in the file must first be counted, then the ratio of the occurrences of each individual byte to the size of the file is calculated. Finally, the entropy is the sum of all these ratios, each multiplied by the base two logarithm of itself. So if we had entropy of 0, it would mean that the file is composed of the same byte repeated for the length of the sample. If, on the other hand, we obtain 8, it would mean that the file is composed of an even number of all 256 possible bytes.

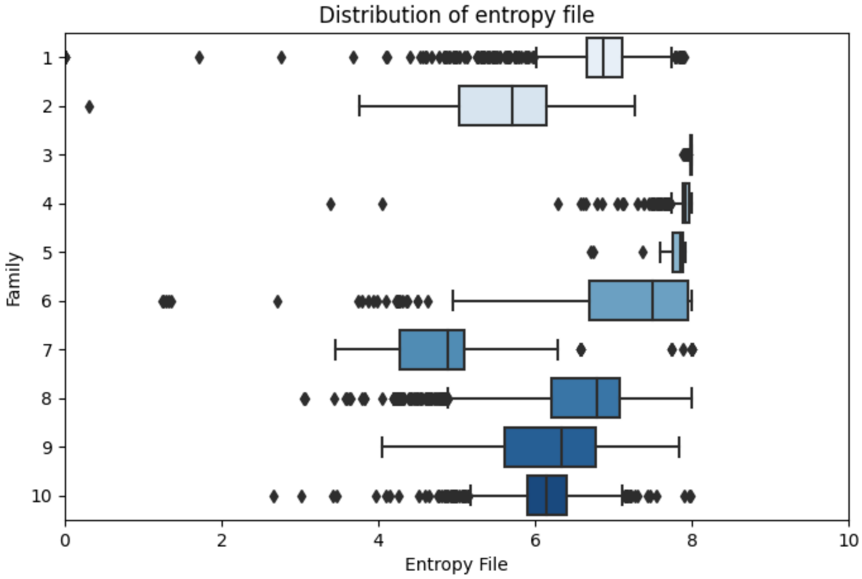


Figure 5.3: Distribution of entropy file.

Also for this malware feature, the boxplot. The boxplot can also be plotted for this malware feature. You can see it in Figure 5.3. In contrast to the size distribution of the binaries, we can see that the spaces are much smaller, based on the classes Kelihos_ver3, Vundo and Simda. Once I visualised the following graph, however, I could see that entropy cannot be used as a single feature to train a machine learning tool. It does not have a well-defined uniqueness for each family. If we see the samples of Lollipop and Gatak, we can deduce that they are files with a very similar entropy, which will disturb the classification work. However, this malware feature has been retained and experiments carried out.

The time taken to extract the entropy from the complete dataset was 6 hours, 16 minutes and 20 seconds. The results of these experiments can be read in the next chapter.

5.3.4 Opcode 4-grams

After extracting the previous features from the *.bytes* files in the dataset, the decision was made to read the information in the .asm files. Obviously, this excluded the use of the benign files. All features from the one described in this subsection and the next are features that can only be used in malware classification and not in identification. This applies to this specific dataset, if time permitted it would also have been possible to extract the disassembled files from the samples in DikeDataset. This note will be better addressed in the chapter on conclusions and future work.

A class from the scikit learn library [44] was used to extract the n-grams. This class allowed me to automate the process of extracting n-grams by giving as input only a list containing the sequence of assembly instructions in the file and specifying the length of each gram. This class proceeds to count all the occurrences of each individual n-gram returning in output the matrix we required in order to train and test the machine learning tool.

The time taken to extract the 4-grams opcodes was 2 hours, 16 minutes and 35 seconds. Dimensionality reduction was performed as for the bytes 3-gram since the matrix had an excessive number of columns (773,874), worsening the performance of our tool. The dimensionality reduction policy was the same, the 5,000 columns of the most counted n-grams in the various samples were kept. A trade off between performance and accuracy was therefore made here as well.

5.3.5 Opcode counter

The next step was to find all the opcodes that were present in the *.asm* files, and then create a $S \times N$ matrix where S is the number of samples and N is the number of opcodes found. The cell (s, n) will then contain the number of occurrences of that single opcode in the sample assembly code s.

To extract this information from the malware, a list was previously created containing all the opcodes present in the samples of the dataset. The correctness of this list was verified by checking the Microsoft [45]. In addition, what are called pseudo-instructions (db, dd etc.) were added. Pseudo-instructions are things which, though not real x86 machine instructions, are used in the instruction field anyway because that's the most convenient place to put them. These are simple assembly language instructions that do not have a direct machine language equivalent. The assembler translates each pseudo-instruction into one or more machine language instructions. The current pseudo-instructions are **DB**, **DW**, **DD**, **DQ**, **DT**, **DDQ**, **DO**, their uninitialized counterparts **RESB**, **RESW**, **RESD**, **RESQ**, **REST**, **RESDDQ**, and **RESO**, the **INCBIN** command, the **EQU** command, and the **TIMES** prefix.

The extracted matrix has dimension 10868 × 93. The time to extract this matrix was 7 hours, 39 minutes and 48 seconds. The results of experiments performed with this matrix can be found in the next chapter.

5.3.6 API 4-gram

In addition to the opcodes, the assembly code provided by Microsoft contains calls to the API functions of the operating system. Of course, the analysis provided by Microsoft is a static analysis, so API calls are inserted into the code, e.g. the following line can be found in the code of a malware:

- .text:1000122D 8B 3D 04 20 00 10 mov edi, ds:GetProcAddress

From rows like this, an end-of-line function extraction had to be performed. This extraction was carried out by means of a regular expression. After extracting these functions, it was necessary to create a list of them to give as input to the class already mentioned a few sections ago, CountVectorizer. It was useful for us to construct the n-grams of API calls.

More than 10,000 API function calls were found within the dataset, so the matrix obtained by extraction of the n-grams had a very large size, to be precise 10,868 rows and 402,972 columns. For this reason, it was decided to carry out dimensionality reduction in the same way as for the n-grams of bytes and opcodes. The computation time for extraction and dimensionality reduction was 1 hour, 27 minutes and 59 seconds. As with the other features, the results with the following feature can be found in the next chapter.

5.3.7 API check

After the previous feature extraction, I thought I would find all the API calls that were present in the .asm files and create an S × N matrix where S is the number of samples and N is the number of API function calls found. The cell (s, n) will contain 0 if the API function n is never called within sample s, 1 if it has been called at least once.

Before the actual extraction was carried out, a text file was created containing the syscalls in each line. This file was created to later create the columns of the matrix. Each API is in fact present once and only once in the text file. I did not use a hard-coded list in the code as for opcodes because we are talking about more than 10,000 APIs, this has a big impact on memory performance. Since the extraction and classification times were too long, it was decided to carry out dimensionality reduction. Again, the 5,000 columns with the highest sum of values within were kept. From the experiments, it was noted that accuracy did not change much while extraction and classification times improved greatly. The extraction time for this feature was much longer than the corresponding time for opcodes. In fact, it took 9 hours, 25 minutes and 59 seconds.

Feature name	Extraction Time	Dataset	Dimension	File used
Binary File Size	00:00:01.246	Malign + Benign Files	1	.bytes
Bytes 3-gram	07:11:55.393	Malign + Benign Files	16,777,216 → 5,000	.bytes
Entropy	06:16:20.259	Malign + Benign Files	1	.bytes
Opcode 4-grams	02:16:35.774	Only Malign Files	773,874 → 5,000	.asm
Opcode counter	07:39:48.787	Only Malign Files	93	.asm
API 4-gram	01:27:59.114	Only Malign Files	402,972 → 5,000	.asm
API check	10:41:22.843	Only Malign Files	13,516 ← 5,000	.asm

Table 5.2: Summary malware extraction

5.3.8 Summary

In this subsection, we can find summarised extraction times for each feature, divided by individual dataset. The times can be viewed in Table 5.2.

As was easy to predict, the file size had a very fast extraction time compared to all other features. Entropy even though it provides a very small matrix took a long time to extract, probably because calculating it over the entire file can take a lot of computational resources. In fact, many papers based on similar research calculated entropy on blocks of the binary, thus creating a sequence of values per sample that represented the entropy value at different points in the malware. Counting opcode recurrences was also a time-consuming operation. As far as n-grams are concerned, on the other hand, we find ourselves with higher performance. The scikit-learn Count Vetorizer class, made available to find the n-grams of elements in a sequence of text, bytes, opcodes or other, is very efficient, giving very good results. As far as bytes are concerned, the size of the array and thus the allocation of a large amount of memory probably played a role.

5.4 Malware feature extraction from ST-WinMal

Fewer features were extracted from the second dataset at my disposal than the previous one. The techniques used were the same: we tried to extract information from the available files and then save this data in matrices to be given as input to machine learning algorithms. With the following dataset, it was much easier to obtain the information as it had already been extracted by the author and made available in text files:

- files with the extension *.bytes* containing the binary (in hexadecimal format).
- files with the extension *.entropy* with the entropy calculated for each section of the malware (.text, .data etc.).
- files with the extension *.apis* containing the API functions called in the malware at least once.

Since files with the required information were provided, the features used were entropy, API function calls and n-grams of bytes. These three features had already been used with the previous dataset, so the same techniques were used. Since they have already been addressed, they will be briefly summarised in the following subsections.

5.4.1 Bytes 3-gram

The extraction of the n-gram of bytes was carried out with the same algorithm used with the previous dataset, only parts of the code had to be adapted to read the file differently. Here too, as with the dataset created by Microsoft, a dimensionality reduction was carried out. The number of columns was increased to 5,000, keeping the columns with the highest sum of values inside.

5.4.2 Entropy

The extraction of this feature was slightly different from the entropy of the files in the previous dataset. In this dataset, in fact, we have the entropy for each section of the disassembled file (section .text, .data etc.), which is why we can create a matrix where each row represents a sample and each column a section of the file. The matrix cell will then contain the entropy value of a section if present, otherwise a default value to indicate that a particular section is not present in the malware.

In malware, however, there were sometimes non-default sections. In fact, it is possible to create a customised section when creating a program and this created the possibility of making a choice: either take all sections into account or only the default ones. It was decided to go for the second option as it is possible for a section to be created only for one malware and not for the others. This avoids the possibility of overfitting, as the default sections are always used in executable programs.

To construct the matrix, I first searched the microsoft site [46] for a list of all Windows executable sections. The list was useful for us to construct the matrix with the entropy of the malware sections.

5.4.3 API check

The extraction of the presence of API function calls in malware was performed in a mirror-image manner to what was done with the previous dataset. First, the list of all API functions called by the samples in ST-WinMal dataset was constructed, which turned out to be more than 10,000, and then this list was used to go on to create the matrix to be used as input for the machine learning algorithms. In fact, each column represents a single API function; if there is a call to any API function in a sample, the cell in the relevant row and column is set to 1.

As with the other dataset, dimensionality reduction was performed since the matrix was found to be too large. The reduction technique was identical to the old dataset; in fact, the 5,000 columns with higher sum of values within were kept. Indeed, in the experiments it was noticed that the results did not vary too much after reducing the number of columns, while the classification speed improved quite a bit.

5.5 ML tools used for Malware Identification and Classification

This section will discuss the algorithms used to identify and classify malware. While studying the state of the art of this type of research, I tried to find out which algorithms were most commonly used. It must be emphasised that each algorithm then has slightly different implementations among the different libraries available. In my case, I used scikit-learn as already explained in section 5.2.

In my case, I used scikit-learn as already explained in section 5.2. This library offers many classifiers such as random forest, support vector machine, gradient boost etc. A typical use of the algorithms provided by scikit can be seen in Algorithm 3

As can be seen from the algorithm, the first step involves dividing the dataset into 2 parts, the training and test set. Scikit provides a special function so that the separation of the samples

Algorithm 3 Typical use of scikit-learn algorithms

```
procedure CLASSIFICATION_PROCEDURE(X, labels)
    $X\_train, X\_test, y\_train, y\_test \leftarrow$ train_test_split(X, labels)
    $clf \leftarrow$ Classifier()
    $clf$.fit(X_train, y_train)
    $clf\_prediction \leftarrow clf$.predict(X_test)
    $score \leftarrow$ calculate_score(y_test, clf_prediction)
end procedure
```

can also be randomised. Obviously, the ratios between the various families remain the same. In my case, it was decided to divide the dataset in the following way: 80% of the samples were used for the training set, and 20% for the test set. The training set in turn must be divided into two sets. During the training phase, it is indeed important to carry out training and evaluation in order to prevent overfitting. This phenomenon occurs when the model becomes very good at classifying the samples in the training set but fails to generalise and make accurate classification of the unseen data. This is the main reason why we end up with 3 sets: training set, validation set and test set.

A division into such percentages was then obtained:

- 64% **train set**
- 16% **validation test**
- 20% **test set**

In addition, cross-validation was performed, a technique used in machine learning, but in general in statistics, to divide the data set into k parts of equal numerosity, and at each step, the k part is used for the validation phase, while the remaining k-1 parts are used for training. It is in fact called k-fold cross-validation. This technique is used to reduce overfitting as much as possible, so that all samples are used at least once in the evaluation phase.

Below I list all the algorithms I have used, specifying mode of use and configuration.

Random Forest

While studying the state of the art, I found a lot of positive feedback on the use of this type of algorithm in the identification and classification of malware. It was therefore impossible for me not to use it in my research, as it was focused on comparing many algorithms with different features in order to compare their performance.

The Random Forest was set up with 100 trees (estimators), and no maximum depth was entered as the training and testing times did not need to be decreased. Since the ratio of the number of samples per household in the dataset is not representative of reality, the weight of the classes was left at 1 by default.

K-Nearest neighbors

The k-Nearest Neighbour is another machine learning algorithm widely used in machine learning identification and classification. It involves constructing the classification space during the training phase and inserting samples into this space. Then during the evaluation phase, the samples are classified according to which class is the most present in the k nearest neighbour.

In scikit-learn there is a special class, KNeighborsClassifier. In my case it was initialised with $k = 5$. I also tried using 3 and 7 as values for k, but they led to results with lower precision, so I decided to use 5.

Support Vector Machine

Along with the Random Forest and K-Nearest Neighbors, the Support Vector Machine algorithm was used as a classifier in many searches. This algorithm was created for binary classification, since as specified in Chapter 2, it creates a division in the dataset space and then labels the samples according to where they are inserted during the execution of the algorithm.

In the scikit-learn library, there is a class called SVC, i.e. Support Vector Classifier, specifically created to use this algorithm in the field of multi-classification. Multi-class support is managed through the one-vs-one scheme. This type of scheme involves a number of tool trainings equal to the number of classes in the dataset. At each training session, the algorithm attempts to create a division in the space between elements that belong to a class and those that do not.

The kernel function chosen was the radial basis function (RBF). It is a function for which the result depends solely on the distance between the function argument x and a fixed point c of the domain. For the type of problem addressed in this thesis, linear functions, sigmoid functions or polynomial functions did not perform well.

Gradient Boosting Classifier

Since the most cited algorithms in the state of the art have been used so far, I decided to adopt an algorithm that has not been commonly used. I therefore decided to adopt the Gradient Boosting classifier. This type of algorithm has not yet been addressed in this thesis, so I will first give a brief explanation of its general operation and then explain how I used it.

Gradient boosting is a popular machine learning algorithm in the field of classification, which has achieved excellent results. They are highly customizable to the particular needs of the application, like being learned with respect to different loss functions [47].

The loss function is the function used to map an event or values of one or more variables onto a real number intuitively representing some “cost” associated with the event. We usually try to optimise a model, so the goal most of the time is to reduce this function as much as possible. In the other pure machine learning algorithms, an attempt is made to achieve the same goal, but as we have seen, each time a different approach is tried.

Gradient boosting is an algorithm with an operation very similar to random forest. It creates a function that tries to best classify the samples in the dataset and then calculates the loss. It then creates another function that tries to improve on the defects of the first, aiming to decrease the loss. Several functions are then created that are nothing more than decision trees. When we have built enough trees to have reached our target then those trees are used in the predictive phase. The similarities with the random forest are many, but as we can see, the difference lies in the creation time of the trees. In gradient boosting, they are not created in parallel as they are in the random forest. We can therefore conclude that there is a trade off between accuracy and training time.

In my case, I applied the HistGradientBoostingClassifier class from scikit-learn. I did not use the GradientBostingClassifier class because the library itself for datasets with more than 10,000 samples recommends not using it because the training times are too high. The values inside are left the default ones, as for the other classes, in order to have a fairer comparison.

Chapter 6

Results

This chapter will show the results of experiments carried out for each individual feature and subsequently with different combinations that yielded promising results. The working environment was the same as described in the previous chapter (refer to the table 5.1). For each experiment, the same resources were required from the cluster, namely a single 8-core CPU (2.10 GHz) and 10 GB of RAM.

During the training phase, k-fold cross validation was used, a technique widely used in the statistical field, especially when trying to build a predictive model such as ours. The training data set was then divided into 5 subsets ($k = 5$), and the training process was performed 5 times, changing the training and evaluation subset at each stage. This technique is used to be able to understand whether the results obtained depend greatly on how you divide the dataset or not. 20% of the dataset was left for the testing phase while 80% of the dataset was used in the training and evaluation phase following the previously described technique. Statistics will be shown for each individual feature, including a comparison of all 4 algorithms used through histograms created through the use of the mathplotlib [48] and seaborn [49] libraries.

Before addressing the results of all experiments, it is good to define what metrics were used and what values are acceptable. It should be remembered, however, that the data should be examined along with the confounding matrices since they may lead to a poor understanding of the model under investigation.

The first metric is accuracy. It is a widely used metric for evaluating classification models and represents the part of the prediction that our model performed correctly. Formally, accuracy has the following definition:

$Accuracy = \frac{Number of correct predictions}{Total number of predictions}$.

In an unbalanced dataset, however, this measure is not optimal since in case we had a dataset consisting of 100 samples, 90 of class A and 10 of class B in case the model only predicted class A it would obtain an accuracy of 0.90 (90%). This model would be very fallible as you can read so it cannot be studied alone. To evaluate problems with unbalanced datasets there are 2 different metrics: precision and recall.

Precision seeks to answer the following question: What proportion of positive identifiers is actually correct? Precision in fact is defined as follows:

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

In our case this measure works reasonably well. Since true positives make up the bulk of the dataset (there are many more malware in the dataset than non-malicious files), it is not possible to get really low precision, at least if our model does not include many malware as benign files.

Instead, the recall answers the question: What proportion of actual positives has been correctly identified? Its mathematical definition is as follows:

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

These two metrics need to be studied together to understand whether a model works well or not, but there are still drawbacks here as well. Recall and precision in fact tend to be in tension, that is, when one improves the other gets worse. In the case of an excellent model, however, this is not the case.

Another metric used is the F1 score. Its definition is much more complex in that it is a harmonic mean, that is, the reciprocal of the arithmetic mean of the reciprocals. The mathematical definition is much simpler. It is as follows:

$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$

F1 score is very useful when accuracy, precision and recall individually are not enough (as in our case).It is very useful in 2 situations:

- When there are weight differences between false positives or false negatives. In our case, a malicious file predicted incorrectly is much more serious than a benevolent file predicted equally incorrectly.
- When you have an unbalanced dataset. In our case we totally re-enter.

Finally, the ROC curve was used. It is a function that shows the performance of a model at all thresholds of the classification. It plots two parameters, the percentage of true positives and the percentage of false positives. The area under this curve provides an aggregate measure of the performance of all possible classification thresholds. Generally, this metric is widely used in binary classification and less so in multi-class classification. In fact, as we shall see, this will also be the case in our case.

In the case of multi-classification, accuracy, precision and recall will have macro and weighted notation. Their meanings are as follows:

- Macro: each class's contribution equally to the final averaged metric
- Weighted: each classes' contribution to the average is weighted by its size

Micro notation also exists but has been unused since it considers each sample to be of equal weight but with an unbalanced dataset like ours this measure is not applicable. The weighted notation we can claim is the most suitable for our needs.

We will first analyze the results on the Microsoft malware classification challenge dataset and after that the results on ST-WinMal dataset.

6.1 Malware detection results with Microsoft Malware Classification Challenge Dataset

This section will discuss the results of the four algorithms with features related to the identification of the Microsoft challenge dataset(binary file size, bytes 3-gram and entropy). I will show and discuss some statistics such as accuracy, recall, precision and F1 score. Confusion matrix and related ROC curves will also be shown.

The models taken into account will be as specified in section 5.5 the Random Forest, K-Nearest Neighbours, SVM and Gradient Boosting.

6.1.1 Comments

As shown in Table 6.1, the two algorithms with the highest AUC-ROC are random forest and gradient boosting. In general, it can be seen that there is a clear gap between two groups of algorithms, the first one less performing consisting of K-NN and SVM, and the second one with

Algorithms	Feature			
	Binary Size	Bytes 3-gram	Entropy	All features
AUC-ROC				
RF	0,9073 ± 0,0151	**0.9997** ± 0,0003	0,7488 ± 0,0238	**0,9999 ± 0,0000**
K-NN	0,9041 ± 0,0199	0,9345 ± 0,0107	0,7472 ± 0,0327	0,9026 ± 0,0119
SVM	0.8928 ± 0,0130	0,8889 ± 0,0116	0,8517 ± 0,0158	0,9139 ± 0,0109
GB	**0,9328 ± 0,0122**	0,9997 ± 0,0004	**0,8625 ± 0,0166**	0,9999 ± 0,0001
Accuracy				
RF	**0,9820 ± 0,0021**	0,9951 ± 0,0008	0,8824 ± 0,0021	0,9962 ± 0,0007
K-NN	0,9749 ± 0,0029	0,9703 ± 0,0047	0,9093 ± 0,0044	0,9702 ± 0,0030
SVM	0,8094 ± 0,0135	0,7382 ± 0,0110	0,6822 ± 0,0377	0,7501 ± 0,0068
GB	0,9653 ± 0,0031	**0,9967 ± 0,0008**	**0,9165 ± 0,0027**	**0,9982 ± 0,0002**
Recall				
RF	**0,9980 ± 0,0007**	0,9997 ± 0,0006	0.9354 ± 0,0047	**1.0000 ± 0,0000**
K-NN	0,9977 ± 0,0005	0,9934 ± 0,0036	0.9738 ± 0,0033	0,9969 ± 0,0007
SVM	0,8079 ± 0,0146	0.7209 ± 0,0110	0.6622 ± 0,0430	0.7446 ± 0,0051
GB	0,9932 ± 0,0046	**0.9998 ± 0,0003**	**0.9946 ± 0,0033**	0,9998 ± 0,0003
Precision				
RF	**0,9826 ± 0,0021**	0,9950 ± 0,0014	0,9365 ± 0,0028	0,9959 ± 0,0008
K-NN	0,9756 ± 0,0031	0,9747 ± 0,0021	0,9307 ± 0,0018	0,9713 ± 0,0037
SVM	0,9814 ± 0,0015	0,9920 ± 0,0017	**0,9880 ± 0,0027**	0,9782 ± 0,0033
GB	0,9697 ± 0,0020	**0,9968 ± 0,0008**	0,9208 ± 0,0013	**0,9982 ± 0,0000**
F1 Score				
RF	**0,9902 ± 0,0011**	0,9973 ± 0,0005	0,9359 ± 0,0013	0,9979 ± 0,0004
K-NN	0,9865 ± 0,0016	0,9839 ± 0,0026	0,9517 ± 0,0024	0,9840 ± 0,0016
SVM	0,8862 ± 0,0090	0,8350 ± 0,0079	0,7924 ± 0,0297	0,8455 ± 0,0044
GB	0,9813 ± 0,0017	**0,9982 ± 0,0005**	**0,9563 ± 0,0015**	**0,9990 ± 0,0001**

Table 6.1: Results of the different models (RF: Random Forest, K-NN: K-Nearest Neighbors, SVM: Support Vector Machine, GB: Gradient Boosting) in the malware identification task with different features. Results were aggregated over 3 training runs. Best results for feature are shown in **bold**.

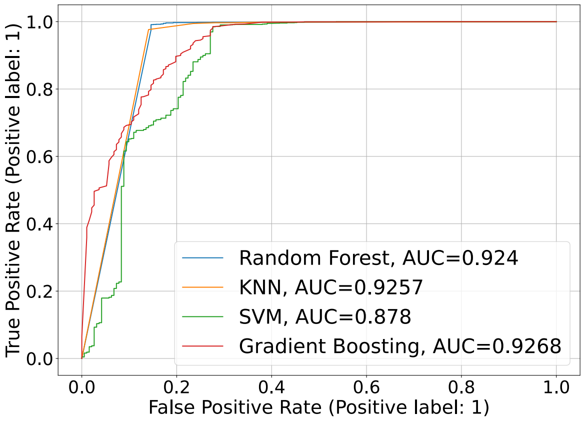


Figure 6.1: ROC curve and AUC-ROC for binary size.

higher results consisting of the two algorithms mentioned above. This figure can be read in parallel with the graphs shown in Figures 6.1, 6.2, 6.3 and 6.4, they in fact show the ROC curves created after the identification of files in the dataset with the four algorithms for each individual feature. One can immediately see that the ROC curves of the random forest and gradient boosting with the bytes 3-gram and with all features together are very close to the ideal function (a function touching the point (0, 1)). However instead through the study of the graphs we can see that

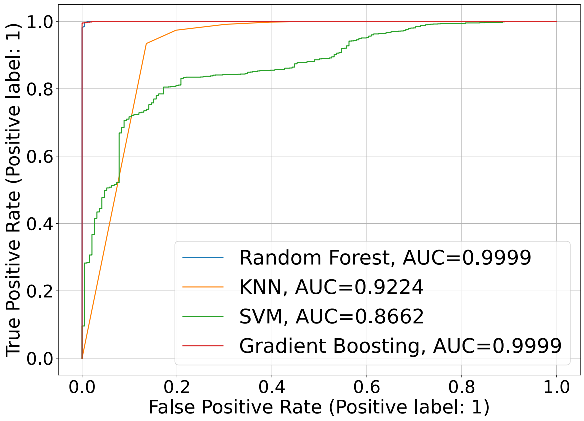


Figure 6.2: ROC curve and AUC-ROC for bytes 3-gram.

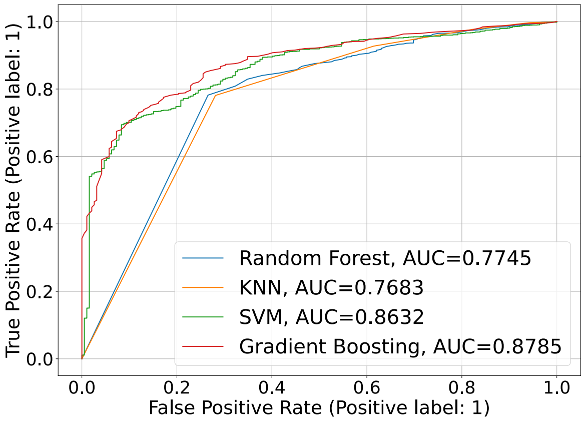


Figure 6.3: ROC curve and AUC-ROC for entropy.

the ROC curves of all four algorithms on entropy are very unpromising, indeed they provide the immediate given that entropy cannot be considered a reliable feature for malware detection. Obviously this phenomenon was expected since the entropy extracted from the files was calculated not on the small sections but on the whole sample taken under examination. To confirm that such a feature is not valid we must first wait to perform an analysis on the other dataset, where instead the entropy was extracted by section.

Table 6.1 also included all data regarding precision, recall, accuracy, and F1 score. These data should be observed together with the confusion matrices. They in fact show how some data could be misinterpreted if read without first seeing the results for each individual class (benign or malicious file).

Of course, there are also cases where the data get confirmation from what we observe from the confusion matrix. For example, if we look at SVM precision with entropy we would expect an excellent confusion matrix ($precision = \frac{TP}{TP+FP}$) instead we observe that the result is so high because few benign files were predicted as malware, but conversely many malicious files were predicted as benign (this would be the worst situation if we were in an antivirus environment).

Instead, there are cases in which the data obtain confirmation from the confusion matrices, and fortunately for us, these cases correspond with the best results. In fact, if we look at the random forest using all features, we observe how all the data (accuracy, recall, etc.) are all very promising (even the ROC curve in the image 6.4 has a very good trend), and this is confirmed by the matrix. Indeed, it shows that all malware were correctly identified (which is much more

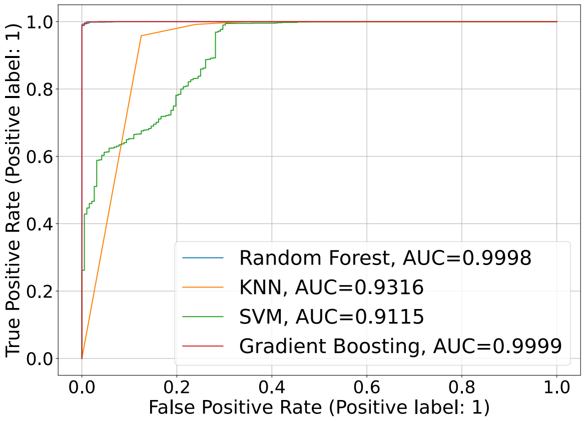
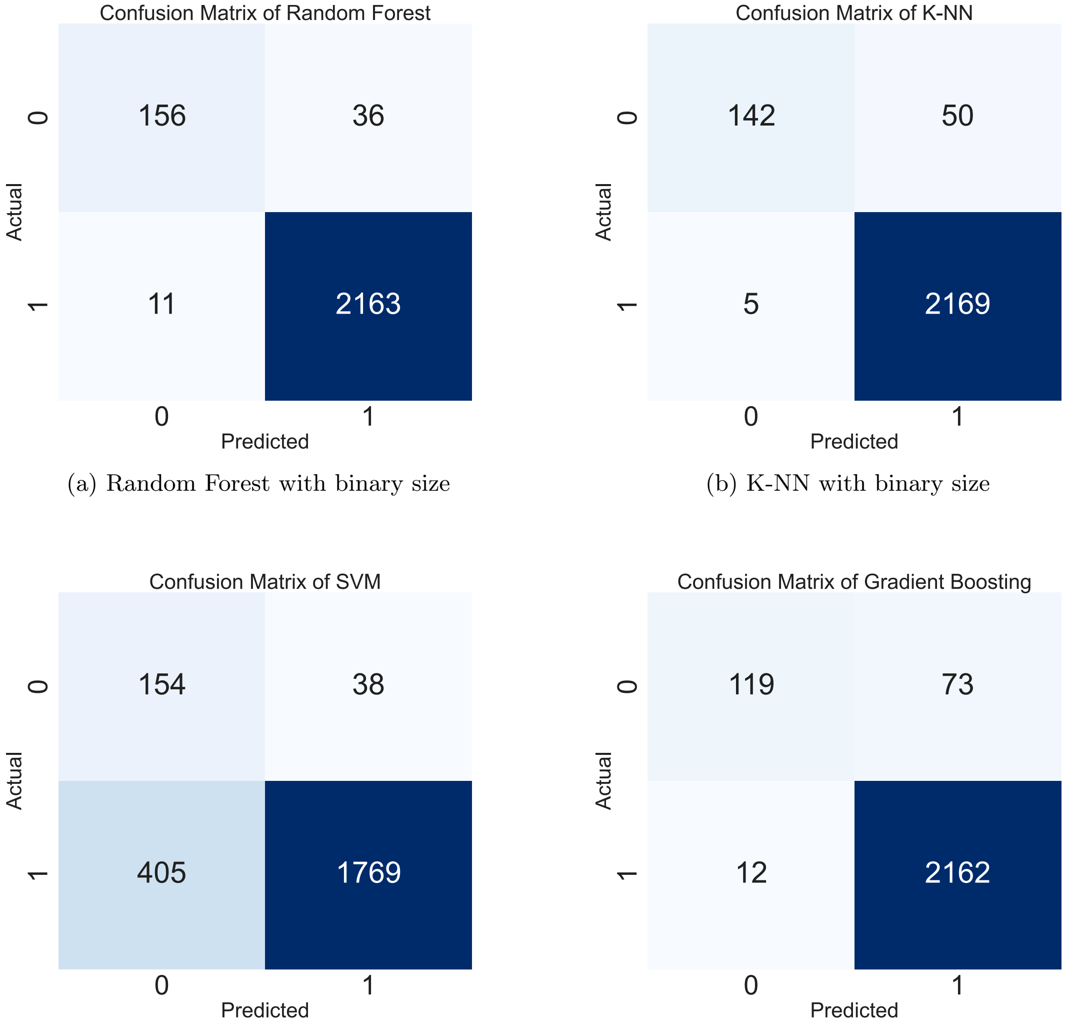


Figure 6.4: ROC curve and AUC-ROC for all feature.



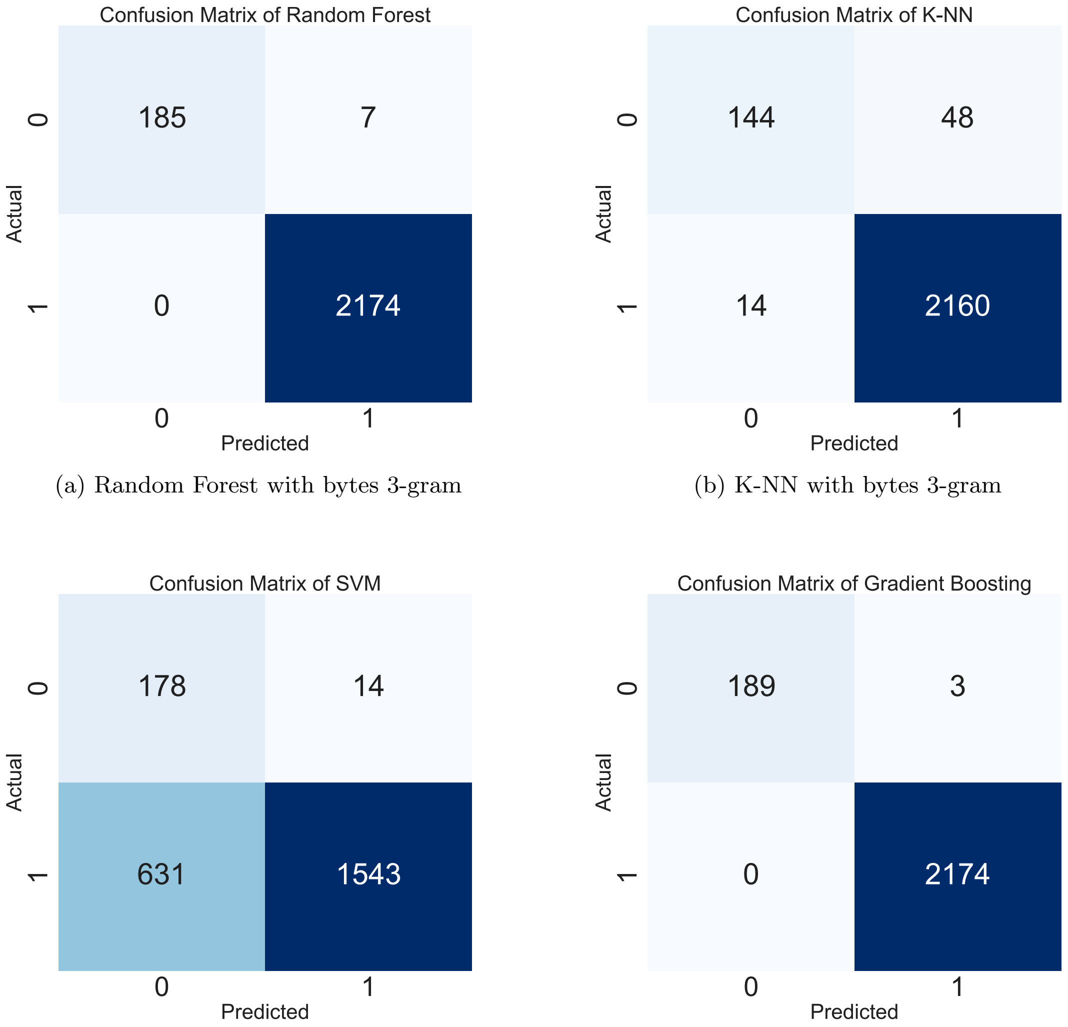
(a) Random Forest with binary size

(b) K-NN with binary size

(c) SVM with binary size

(d) Gradient boosting with binary size

Figure 6.5: Confusion Matrix of each algorithm with binary size(0: not malware, 1: malware). Confusion matrix from the last training.



(a) Random Forest with bytes 3-gram

(b) K-NN with bytes 3-gram

(c) SVM with bytes 3-gram

(d) Gradient boosting with bytes 3-gram

Figure 6.6: Confusion Matrix of each algorithm with bytes 3-gram feature (0: not malware, 1: malware). Confusion matrix from the last training.

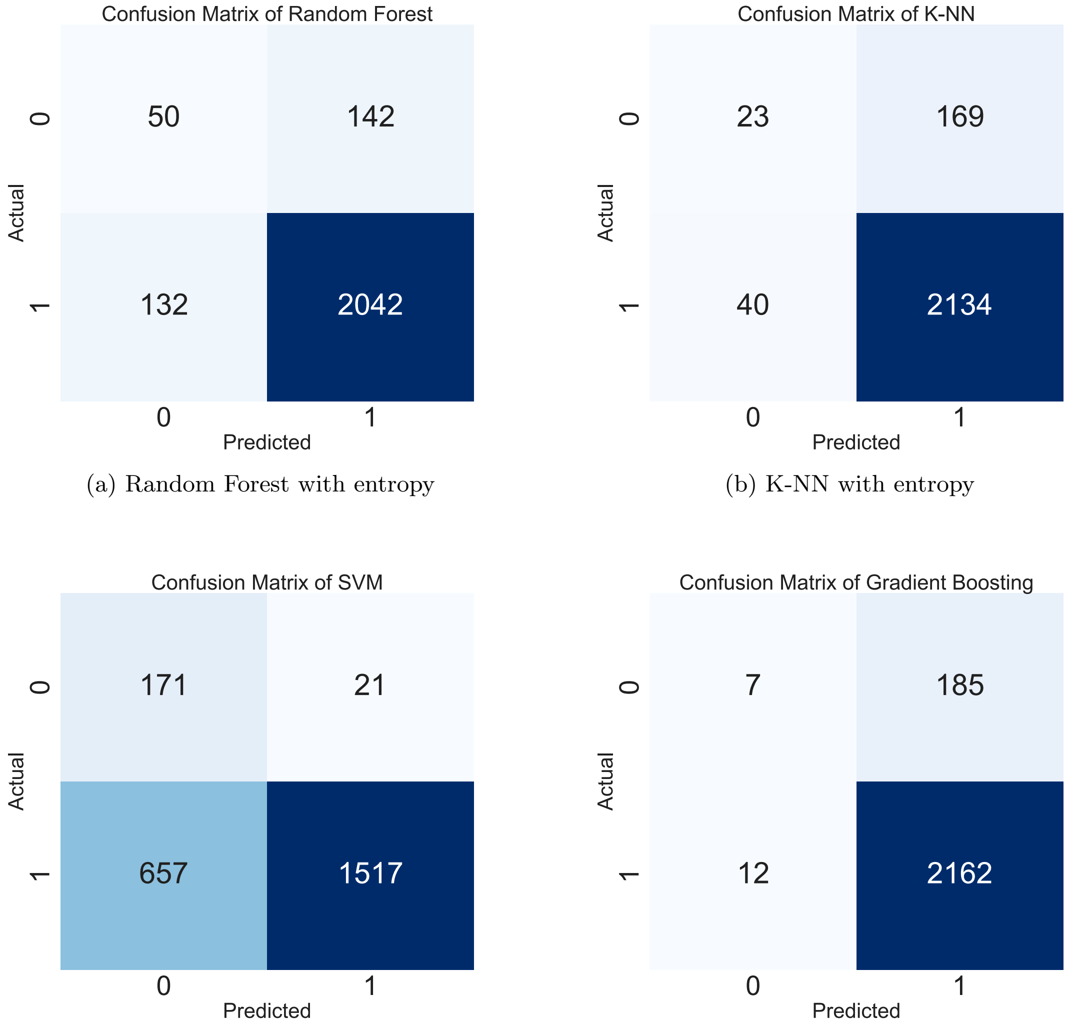
important than correctly predicting only all benign files) and only ten non-malware files were predicted as malicious.

We can therefore finally decree how random forest and gradient boosting are the best algorithm for this kind of work, while k-NN and SVM did not lead to noteworthy results. K-NN manages to keep up with the other 2 algorithms only with binary file cutting, in fact in that case it has both a good confusion matrix and a very promising ROC curve.

Looking at the graphs of the curves, I noticed how gradient boosting is generally more adaptive since it gets a better graph to the random forest in more cases. It does however take much longer to train so in case there are limitations on the training timings it is better to employ random forest, otherwise gradient boosting remains in my opinion the best solution to employ in this area.

6.2 Malware classification results with Microsoft Malware Classification Challenge Dataset

This section will show the results of classifying malware into families according to different algorithms. The results will be divided by feature extracted from the samples but since some did not



(a) Random Forest with entropy

(b) K-NN with entropy

(c) SVM with entropy

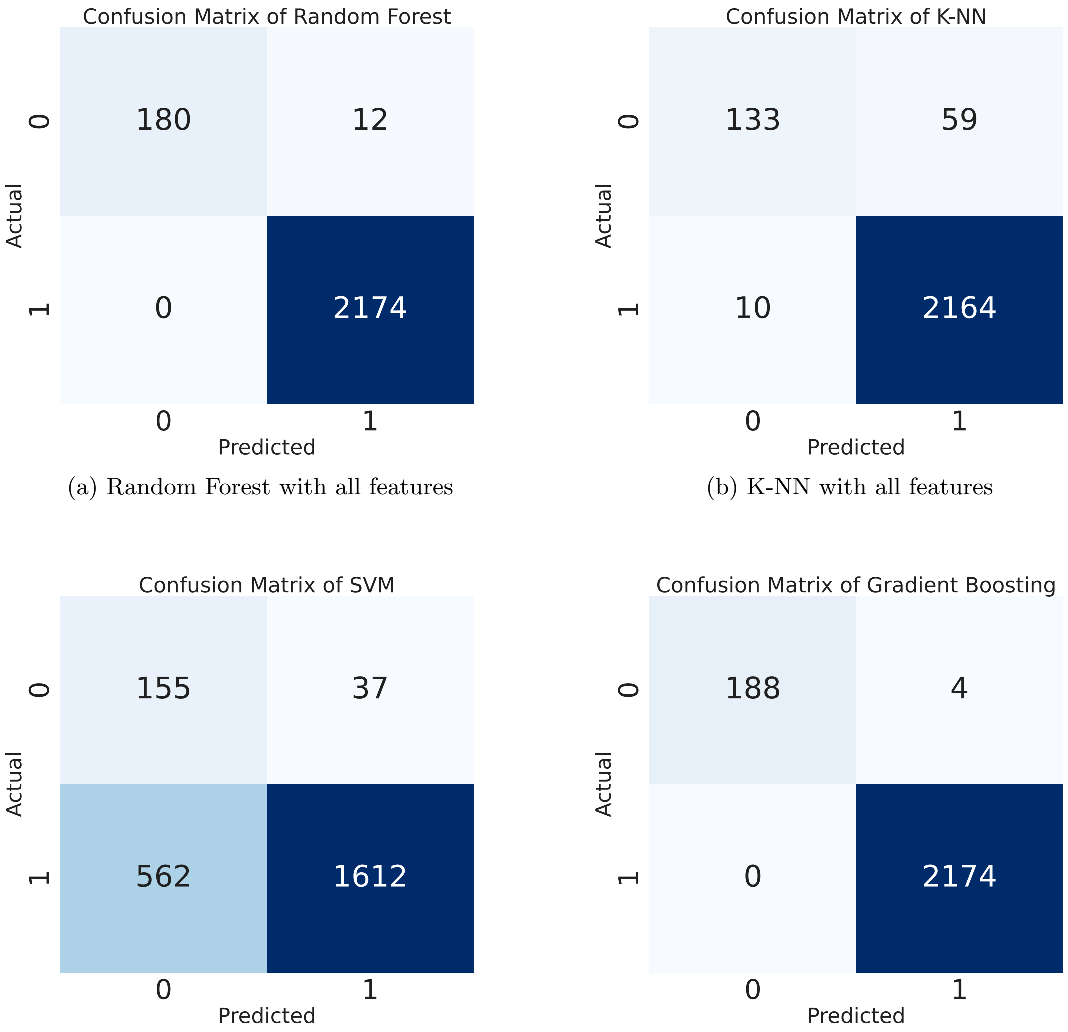
(d) Gradient boosting with entropy

Figure 6.7: Confusion Matrix of each algorithm with entropy feature(0: not malware, 1: malware). Confusion matrix from the last training.

lead to the desired results only the most significant features will be reported. Therefore, I had to determine which metric was the most appropriate to use as a filter to decide which features were acceptable and which were not. After some study on the subject and after reviewing my results and the confusion matrices produced, I decided to employ the weighted F1 score. It reported the truest results in the form of a number. Therefore, in Table 6.2 I report all the weighted F1 scores of all the features.

As you can see there were three features that led to very encouraging results, these are the bytes 3-gram, the opcode 4-gram, and the opcode counter. The other four, however, led to disappointing results. Regarding entropy we know that the extraction method was already not optimal, with the second dataset we will be able to confirm whether this hypothesis is true or whether it is the feature itself that is not appropriate for this work. The 4-gram of API functions had the worst result, so we can conclude that a static API function extraction that makes dynamic sense, i.e., inserts the concept of sequence, cannot work. On the other hand, API check did not bring the desired results. Through the second dataset, as with entropy, we will be able to find out whether it is this extraction was performed correctly or not.

Once these results were obtained, 3 more experiments were tried, namely merging the features extracted from the .bytes files, the features extracted from the .asm files, and finally merging the 3 most promising features to see if more features together could lead to better results.



(a) Random Forest with all features

(b) K-NN with all features

(c) SVM with all features

(d) Gradient boosting with all features

Figure 6.8: Confusion Matrix of each algorithm with all feature together (0: not malware, 1: malware). Confusion matrix from the last training.

F1-score weighted	RF	K-NN	SVM	GB
Binary File Size	**0.7582 ± 0.0015**	0.7372 ± 0.0046	0.4493 ± 0.0030	0.6882 ± 0.0070
Bytes 3-gram	0.9898 ± 0.0032	0.9142 ± 0.0082	0.6593 ± 0.0122	**0.9929 ± 0.0023**
Entropy	0.6374 ± 0.0199	**0.6851 ± 0.0147**	0.5357 ± 0.0172	0.6457 ± 0.0059
Opcode 4-gram	0.9904 ± 0.0014	0.9386 ± 0.0046	0.2802 ± 0.0064	**0.9944 ± 0.0013**
Opcode counter	0.9928 ± 0.0007	0.9640 ± 0.0069	0.3169 ± 0.0097	**0.9949 ± 0.0017**
API 4-gram	**0.5270 ± 0.0078**	0.4515 ± 0.1299	0.2631 ± 0.1122	0.3941 ± 0.0048
API check	0.7310 ± 0.0129	0.6571 ± 0.0288	0.7355 ± 0.0108	**0.7454 ± 0.0089**

Table 6.2: Mean and standard deviation of family classifier F1 score. The models are Random Forest (RF), K-Nearest Neighbour (K-NN), Support Vector Machine (SVM) and Gradient Boosting (GB). The results are from three different simulations, and the best result for each feature is shown in **bold**.

Below you can find all the tables divided by feature or combination of features with associated confusion matrices. In the confusion matrices for readability issues, numbers were inserted to identify the classes instead of their names. A legend of the families can be found in the table 6.3

Number	*Family*
0	Ramnit
1	Lollipop
2	Kelihos_ver3
3	Vundo
4	Simda
5	Tracur
6	Kelihos_ver1
7	Obfuscator.ACY
8	Gatak

Table 6.3: family legend

Bytes 3-gram	RF	K-NN	SVM	GB
Accuracy	0.9900 ± 0.0031	0.9127 ± 0.0086	0.6328 ± 0.0117	**0.9930 ± 0.0022**
Macro-Precision	0.9884 ± 0.0032	0.8026 ± 0.0291	0.5861 ± 0.0102	**0.9920 ± 0.0013**
Weighted-Precision	0.9901 ± 0.0030	0.9208 ± 0.0072	0.7769 ± 0.0165	**0.9930 ± 0.0021**
Marco-Recall	0.9464 ± 0.0171	0.8200 ± 0.0257	0.5723 ± 0.0389	**0.9671 ± 0.0156**
Weighted-Recall	0.9900 ± 0.0031	0.9127 ± 0.0086	0.6328 ± 0.0117	**0.9930 ± 0.0022**
Macro-F1 Score	0.9619 ± 0.0134	0.8085 ± 0.0272	0.4974 ± 0.0148	**0.9775 ± 0.0106**
Weighted-F1 Score	0.9898 ± 0.0032	0.9142 ± 0.0082	0.6593 ± 0.0122	**0.9929 ± 0.0023**
Micro-AUC ROC	0.9996 ± 0.0005	0.9809 ± 0.0027	0.9660 ± 0.0026	**0.9999 ± 0.0001**
Marco-AUC ROC	0.9968 ± 0.0050	0.9506 ± 0.0109	0.9330 ± 0.0108	**0.9998 ± 0.0002**
Weighted- AUC ROC	0.9996 ± 0.0005	0.9805 ± 0.0027	0.9585 ± 0.0032	**0.9999 ± 0.0001**

Table 6.4: Results of the different models (RF: Random Forest, K-NN: K-Nearest Neighbors, SVM: Support Vector Machine, GB: Gradient Boosting) in the malware classification task with bytes 3-gram. Results were aggregated over 3 training runs. Best results for feature are shown in **bold**.

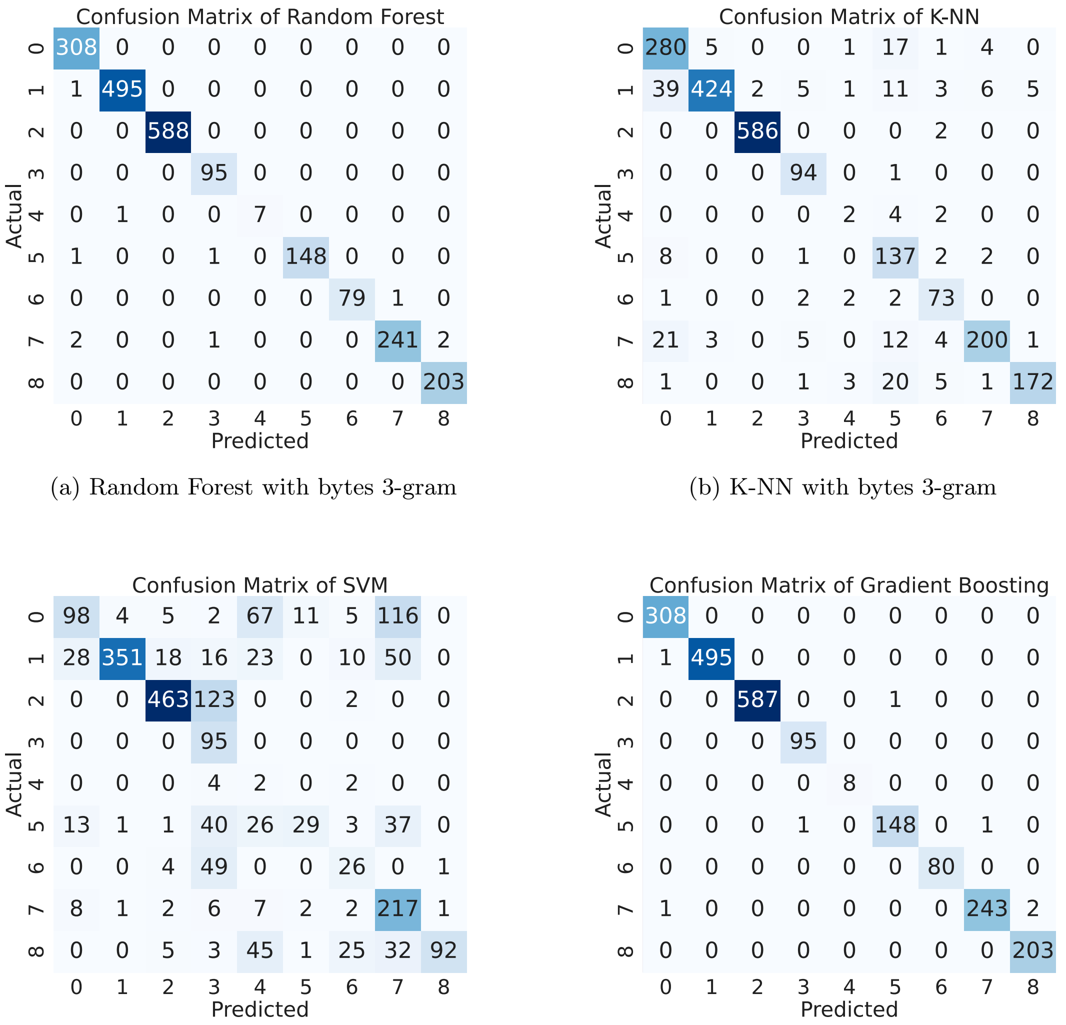
6.2.1 Comments Bytes 3-gram

From the table 6.4 it can be immediately understood that even for classification the two best algorithms are random forest and gradient boosting. These two algorithms often achieve 99% accuracy. The confusion matrices in Figure 6.9 confirm the reading of the results in the table.

An interesting note falls on the area under the ROC curve (AUC-ROC). It is consistently high even though the other metrics report very disappointing results. As we can deduce then, the theory expressed in the introduction of this chapter is not wrong at all. This metric is very useful in case we are talking about binary classification but when it comes to multi-class classification we end up with an unclear metric. F1 score, precision, and recall come to our aid by helping us better understand what the situation is.

From the confusion matrices, we can see how few errors are made by the two best models. To be precise, the random forest incorrectly classifies ten malware, while gradient boosting incorrectly classifies only seven. We can therefore point out that this feature is suitable for both the classification and identification of malware for this task. This is a very promising result as it is not very difficult to extract such a feature, the only difficulty being the time taken to create a matrix that can then be given as input to the selected ML model.

I conclude by pointing out the results of the SVM. This algorithm is in fact widely used for binary classification, and in fact brought acceptable results in the field of malware identification with the following feature. In the field of classification, however, we can see that even with this feature it still fails to bring acceptable results.



Confusion Matrix of Random Forest

	0	1	2	3	4	5	6	7	8
0	308	0	0	0	0	0	0	0	0
1	1	495	0	0	0	0	0	0	0
2	0	0	588	0	0	0	0	0	0
3	0	0	0	95	0	0	0	0	0
4	0	1	0	0	7	0	0	0	0
5	1	0	0	1	0	148	0	0	0
6	0	0	0	0	0	0	79	1	0
7	2	0	0	1	0	0	0	241	2
8	0	0	0	0	0	0	0	0	203

(a) Random Forest with bytes 3-gram

Confusion Matrix of K-NN

	0	1	2	3	4	5	6	7	8
0	280	5	0	0	1	17	1	4	0
1	39	424	2	5	1	11	3	6	5
2	0	0	586	0	0	0	2	0	0
3	0	0	0	94	0	1	0	0	0
4	0	0	0	0	2	4	2	0	0
5	8	0	0	1	0	137	2	2	0
6	1	0	0	2	2	2	73	0	0
7	21	3	0	5	0	12	4	200	1
8	1	0	0	1	3	20	5	1	172

(b) K-NN with bytes 3-gram

Confusion Matrix of SVM

	0	1	2	3	4	5	6	7	8
0	98	4	5	2	67	11	5	116	0
1	28	351	18	16	23	0	10	50	0
2	0	0	463	123	0	0	2	0	0
3	0	0	0	95	0	0	0	0	0
4	0	0	0	4	2	0	2	0	0
5	13	1	1	40	26	29	3	37	0
6	0	0	4	49	0	0	26	0	1
7	8	1	2	6	7	2	2	217	1
8	0	0	5	3	45	1	25	32	92

(c) SVM with bytes 3-gram

Confusion Matrix of Gradient Boosting

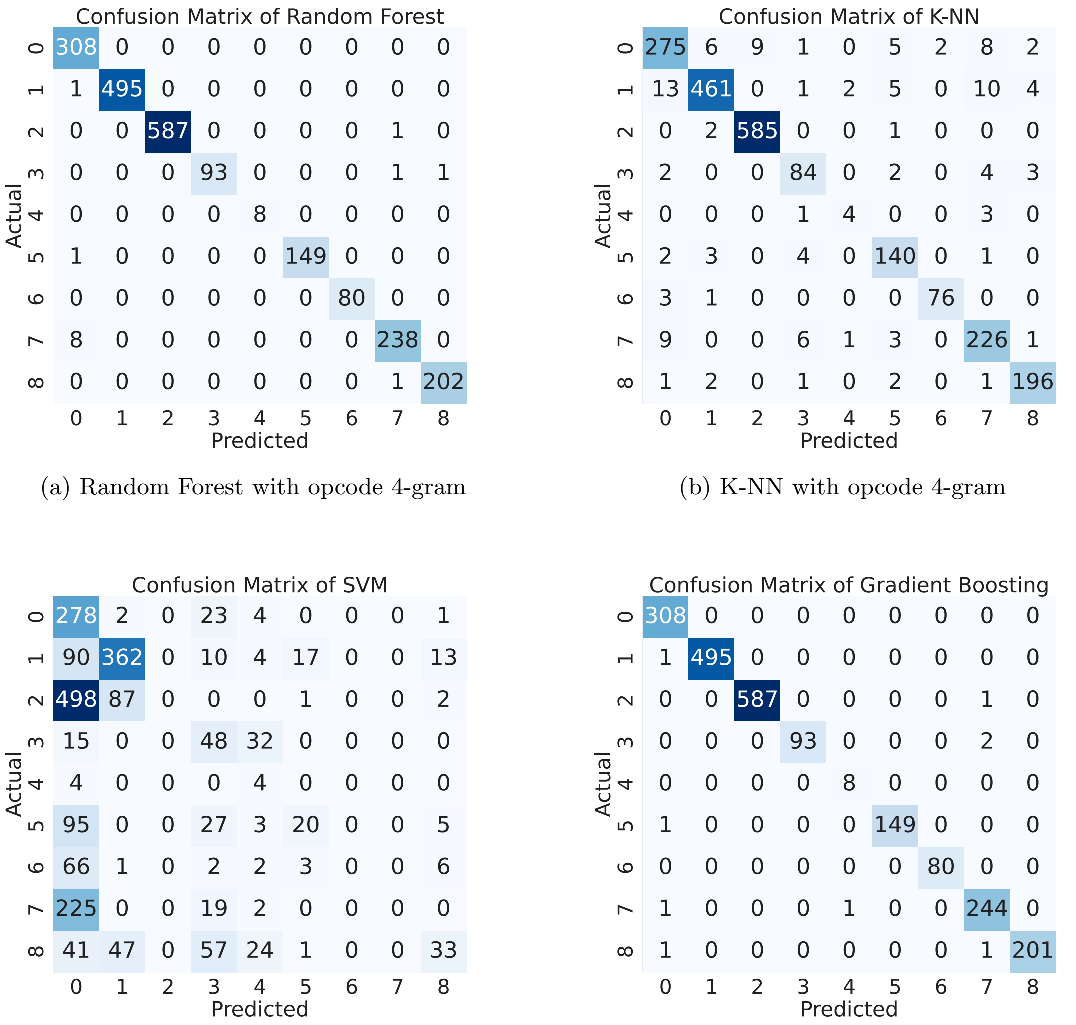
	0	1	2	3	4	5	6	7	8
0	308	0	0	0	0	0	0	0	0
1	1	495	0	0	0	0	0	0	0
2	0	0	587	0	0	1	0	0	0
3	0	0	0	95	0	0	0	0	0
4	0	0	0	0	8	0	0	0	0
5	0	0	0	1	0	148	0	1	0
6	0	0	0	0	0	0	80	0	0
7	1	0	0	0	0	0	0	243	2
8	0	0	0	0	0	0	0	0	203

(d) Gradient Boosting with bytes 3-gram

Figure 6.9: Confusion Matrix of each algorithm with bytes 3-gram. Confusion matrix from the last training.

Opcode 4-gram	RF	K-NN	SVM	GB
Accuracy	0.9905 ± 0.0014	0.9384 ± 0.0045	0.3579 ± 0.0108	$\mathbf{0.9945 \pm 0.0012}$
Macro-Precision	0.9903 ± 0.0018	0.8899 ± 0.0190	0.2655 ± 0.0034	$\mathbf{0.9941 \pm 0.0005}$
Weighted-Precision	0.9907 ± 0.0013	0.9400 ± 0.0048	0.3006 ± 0.0033	$\mathbf{0.9946 \pm 0.0012}$
Marco-Recall	0.9671 ± 0.0080	0.8981 ± 0.0283	0.3360 ± 0.0184	$\mathbf{0.9767 \pm 0.0079}$
Weighted-Recall	0.9905 ± 0.0014	0.9384 ± 0.0045	0.3579 ± 0.0108	$\mathbf{0.9945 \pm 0.0012}$
Macro-F1 Score	0.9772 ± 0.0050	0.8921 ± 0.0223	0.2376 ± 0.0037	$\mathbf{0.9844 \pm 0.0050}$
Weighted-F1 Score	0.9904 ± 0.0014	0.9386 ± 0.0046	0.2802 ± 0.0064	$\mathbf{0.9944 \pm 0.0013}$
Micro-AUC ROC	0.9994 ± 0.0002	0.9862 ± 0.0018	0.8970 ± 0.0011	$\mathbf{1.0000 \pm 0.0001}$
Marco-AUC ROC	0.9990 ± 0.0002	0.9733 ± 0.0099	0.8496 ± 0.0075	$\mathbf{0.9999 \pm 0.0000}$
Weighted- AUC ROC	0.9993 ± 0.0002	0.9857 ± 0.0019	0.8725 ± 0.0026	$\mathbf{0.9999 \pm 0.0000}$

Table 6.5: Results of the different models (RF: Random Forest, K-NN: K-Nearest Neighbors, SVM: Support Vector Machine, GB: Gradient Boosting) in the malware classification task with opcode 4-gram. Results were aggregated over 3 training runs. Best results for feature are shown in **bold**.



Confusion Matrix of Random Forest

	0	1	2	3	4	5	6	7	8
0	308	0	0	0	0	0	0	0	0
1	1	495	0	0	0	0	0	0	0
2	0	0	587	0	0	0	0	1	0
3	0	0	0	93	0	0	0	1	1
4	0	0	0	0	8	0	0	0	0
5	1	0	0	0	0	149	0	0	0
6	0	0	0	0	0	0	80	0	0
7	8	0	0	0	0	0	0	238	0
8	0	0	0	0	0	0	0	1	202

(a) Random Forest with opcode 4-gram

Confusion Matrix of K-NN

	0	1	2	3	4	5	6	7	8
0	275	6	9	1	0	5	2	8	2
1	13	461	0	1	2	5	0	10	4
2	0	2	585	0	0	1	0	0	0
3	2	0	0	84	0	2	0	4	3
4	0	0	0	1	4	0	0	3	0
5	2	3	0	4	0	140	0	1	0
6	3	1	0	0	0	0	76	0	0
7	9	0	0	6	1	3	0	226	1
8	1	2	0	1	0	2	0	1	196

(b) K-NN with opcode 4-gram

Confusion Matrix of SVM

	0	1	2	3	4	5	6	7	8
0	278	2	0	23	4	0	0	0	1
1	90	362	0	10	4	17	0	0	13
2	498	87	0	0	0	1	0	0	2
3	15	0	0	48	32	0	0	0	0
4	4	0	0	0	4	0	0	0	0
5	95	0	0	27	3	20	0	0	5
6	66	1	0	2	2	3	0	0	6
7	225	0	0	19	2	0	0	0	0
8	41	47	0	57	24	1	0	0	33

(c) SVM with opcode 4-gram

Confusion Matrix of Gradient Boosting

	0	1	2	3	4	5	6	7	8
0	308	0	0	0	0	0	0	0	0
1	1	495	0	0	0	0	0	0	0
2	0	0	587	0	0	0	0	1	0
3	0	0	0	93	0	0	0	2	0
4	0	0	0	0	8	0	0	0	0
5	1	0	0	0	0	149	0	0	0
6	0	0	0	0	0	0	80	0	0
7	1	0	0	0	1	0	0	244	0
8	1	0	0	0	0	0	0	1	201

(d) Gradient Boosting with opcode 4-gram

Figure 6.10: Confusion Matrix of each algorithm with opcode 4-gram. Confusion matrix from the last training.

6.2.2 Comments Opcode 4-gram

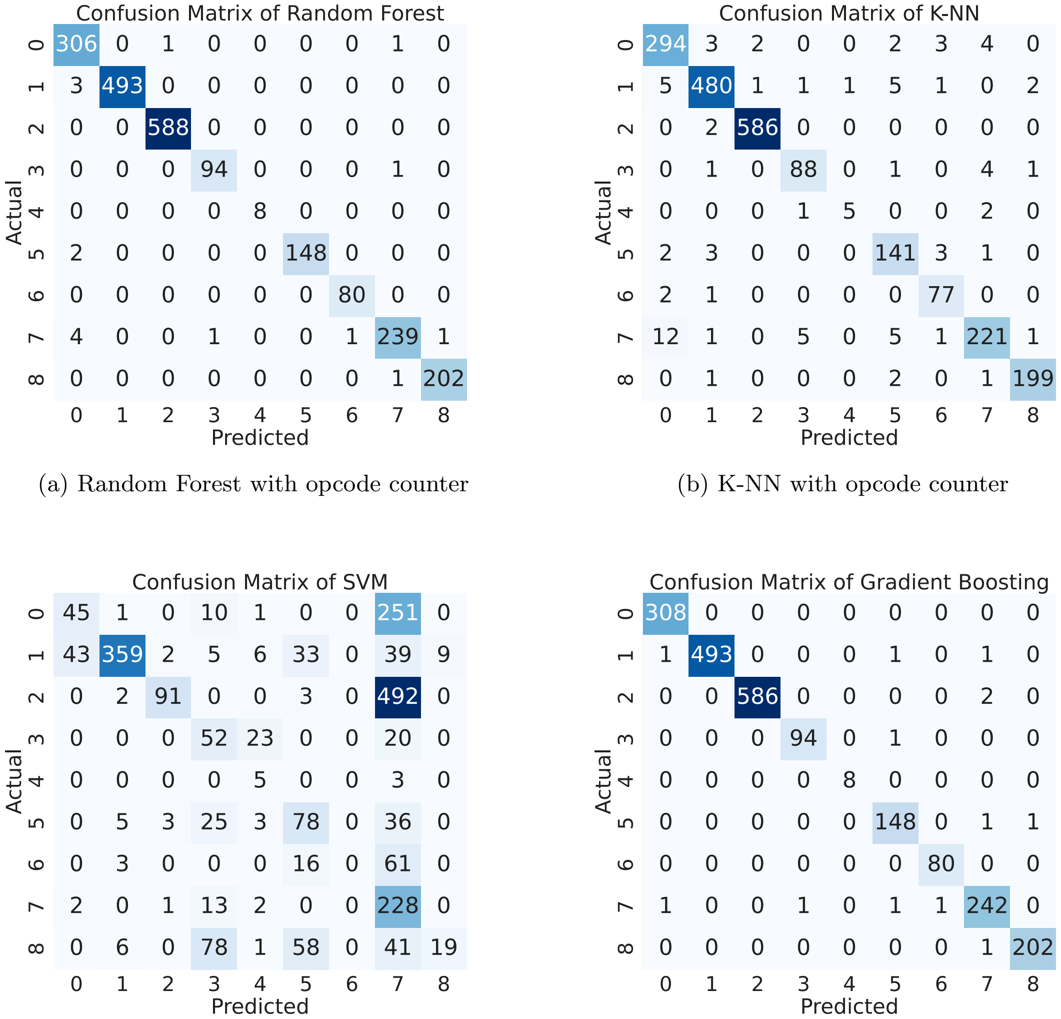
With the use of the opcode 4-gram, we can see that the situation remains almost unchanged. In fact, we find ourselves with the model using Gradient Boosting obtaining excellent results, the random forest following very closely behind, the K-NN obtaining discrete results without particularly excelling in anything, and the SVM obtaining poor results. The confusion matrices in Figure 6.10 confirm the data and also show that K-NN does not actually perform that badly. Obviously it cannot be compared with the two best algorithms, but it is still a good solution that with a few improvements could achieve the results of Random Forest and Gradient Boosting.

SVM in particular we can see that it is greatly influenced by the unbalance of the dataset but does not predict many samples in the largest class. This behaviour is very unexpected. Of course, all kernel functions were tried, but none yielded better results. With K-NN as well, several experiments were carried out with different values of K, but 5 was the number that brought the most results.

From the table, we can see that the weighted results are generally higher than the macro results. As I have already mentioned, for me the weighted results bring a better view of the situation being studied as we are dealing with a very unbalanced dataset and therefore the results must also be weighted between the various classes.

Opcode counter	RF	K-NN	SVM	GB
Accuracy	0.9928 ± 0.0007	0.9640 ± 0.0069	0.3704 ± 0.0146	**0.9949 ± 0.0017**
Macro-Precision	0.9926 ± 0.0018	0.9230 ± 0.0230	0.3561 ± 0.0141	**0.9940 ± 0.0022**
Weighted-Precision	0.9929 ± 0.0007	0.9645 ± 0.0065	0.4173 ± 0.0239	**0.9950 ± 0.0016**
Marco-Recall	0.9816 ± 0.0159	0.9257 ± 0.0151	0.3970 ± 0.0219	**0.9892 ± 0.0099**
Weighted-Recall	0.9928 ± 0.0007	0.9640 ± 0.0069	0.3704 ± 0.0146	**0.9949 ± 0.0017**
Macro-F1 Score	0.9863 ± 0.0092	0.9232 ± 0.0181	0.2827 ± 0.0007	**0.9914 ± 0.0063**
Weighted-F1 Score	0.9928 ± 0.0007	0.9640 ± 0.0069	0.3169 ± 0.0097	**0.9949 ± 0.0017**
Micro-AUC ROC	0.9998 ± 0.0002	0.9916 ± 0.0027	0.9275 ± 0.0032	**1.0000 ± 0.0001**
Marco-AUC ROC	0.9997 ± 0.0002	0.9819 ± 0.0055	0.9142 ± 0.0075	**0.9999 ± 0.0001**
Weighted- AUC ROC	0.9997 ± 0.0002	0.9914 ± 0.0027	0.9251 ± 0.0054	**0.9999 ± 0.0001**

Table 6.6: Results of the different models (RF: Random Forest, K-NN: K-Nearest Neighbors, SVM: Support Vector Machine, GB: Gradient Boosting) in the malware classification task with opcode counter. Results were aggregated over 3 training runs. Best results for feature are shown in **bold**.



Confusion Matrix of Random Forest

	0	1	2	3	4	5	6	7	8
0	306	0	1	0	0	0	0	1	0
1	3	493	0	0	0	0	0	0	0
2	0	0	588	0	0	0	0	0	0
3	0	0	0	94	0	0	0	1	0
4	0	0	0	0	8	0	0	0	0
5	2	0	0	0	0	148	0	0	0
6	0	0	0	0	0	0	80	0	0
7	4	0	0	1	0	0	1	239	1
8	0	0	0	0	0	0	0	1	202

(a) Random Forest with opcode counter

Confusion Matrix of K-NN

	0	1	2	3	4	5	6	7	8
0	294	3	2	0	0	2	3	4	0
1	5	480	1	1	1	5	1	0	2
2	0	2	586	0	0	0	0	0	0
3	0	1	0	88	0	1	0	4	1
4	0	0	0	1	5	0	0	2	0
5	2	3	0	0	0	141	3	1	0
6	2	1	0	0	0	0	77	0	0
7	12	1	0	5	0	5	1	221	1
8	0	1	0	0	0	2	0	1	199

(b) K-NN with opcode counter

Confusion Matrix of SVM

	0	1	2	3	4	5	6	7	8
0	45	1	0	10	1	0	0	251	0
1	43	359	2	5	6	33	0	39	9
2	0	2	91	0	0	3	0	492	0
3	0	0	0	52	23	0	0	20	0
4	0	0	0	0	5	0	0	3	0
5	0	5	3	25	3	78	0	36	0
6	0	3	0	0	0	16	0	61	0
7	2	0	1	13	2	0	0	228	0
8	0	6	0	78	1	58	0	41	19

(c) SVM with opcode counter

Confusion Matrix of Gradient Boosting

	0	1	2	3	4	5	6	7	8
0	308	0	0	0	0	0	0	0	0
1	1	493	0	0	0	1	0	1	0
2	0	0	586	0	0	0	0	2	0
3	0	0	0	94	0	1	0	0	0
4	0	0	0	0	8	0	0	0	0
5	0	0	0	0	0	148	0	1	1
6	0	0	0	0	0	0	80	0	0
7	1	0	0	1	0	1	1	242	0
8	0	0	0	0	0	0	0	1	202

(d) Gradient Boosting with opcode counter

Figure 6.11: Confusion Matrix of each algorithm with opcode counter. Confusion matrix from the last training.

Bytes features	RF	K-NN	SVM	GB
Accuracy	0.9939 ± 0.0033	0.8761 ± 0.0057	0.4928 ± 0.0046	$\mathbf{0.9960 \pm 0.0017}$
Macro-Precision	0.9931 ± 0.0051	0.7683 ± 0.0213	0.3056 ± 0.0059	$\mathbf{0.9950 \pm 0.0039}$
Weighted-Precision	0.9939 ± 0.0032	0.8756 ± 0.0059	0.5178 ± 0.0134	$\mathbf{0.9961 \pm 0.0017}$
Marco-Recall	0.9532 ± 0.0160	0.7722 ± 0.0139	0.4026 ± 0.0042	$\mathbf{0.9599 \pm 0.0088}$
Weighted-Recall	0.9939 ± 0.0033	0.8761 ± 0.0057	0.4928 ± 0.0046	$\mathbf{0.9960 \pm 0.0017}$
Macro-F1 Score	0.9678 ± 0.0139	0.7690 ± 0.0164	0.2803 ± 0.0020	$\mathbf{0.9736 \pm 0.0074}$
Weighted-F1 Score	0.9937 ± 0.0033	0.8754 ± 0.0058	0.4211 ± 0.0088	$\mathbf{0.9959 \pm 0.0018}$
Micro-AUC ROC	0.9998 ± 0.0004	0.9758 ± 0.0009	0.9155 ± 0.0021	$\mathbf{1.0000 \pm 0.0000}$
Marco-AUC ROC	0.9995 ± 0.0004	0.9241 ± 0.0008	0.8383 ± 0.0148	$\mathbf{1.0000 \pm 0.0001}$
Weighted- AUC ROC	0.9998 ± 0.0004	0.9728 ± 0.0010	0.8386 ± 0.0076	$\mathbf{1.0000 \pm 0.0000}$

Table 6.7: Results of the different models (RF: Random Forest, K-NN: K-Nearest Neighbors, SVM: Support Vector Machine, GB: Gradient Boosting) in the malware classification task with bytes features. Results were aggregated over 3 training runs. Best results for feature are shown in **bold**.

6.2.3 Comments Opcode counter

On this feature, I was not sure what results it would report as it had not been found during the state-of-the-art study of this research. My reasoning was that in an assembly code it is very likely to find a recurrence of certain functions over others depending on which malware family is being studied. In the study of static malware analysis, this type of procedure was carried out to create signatures of certain malware in signature-based identification systems, so I decided to study their behaviour with machine learning algorithms.

Far beyond my expectations, it led to excellent results, comparable to byte and opcode grams (which in the state of the art were the features that were used most along with API function calls. We end up with the same situation as in previous features, i.e. a very efficient Gradient Boosting and Random Forest, a good but not excellent K-NN and finally a poor SVM.

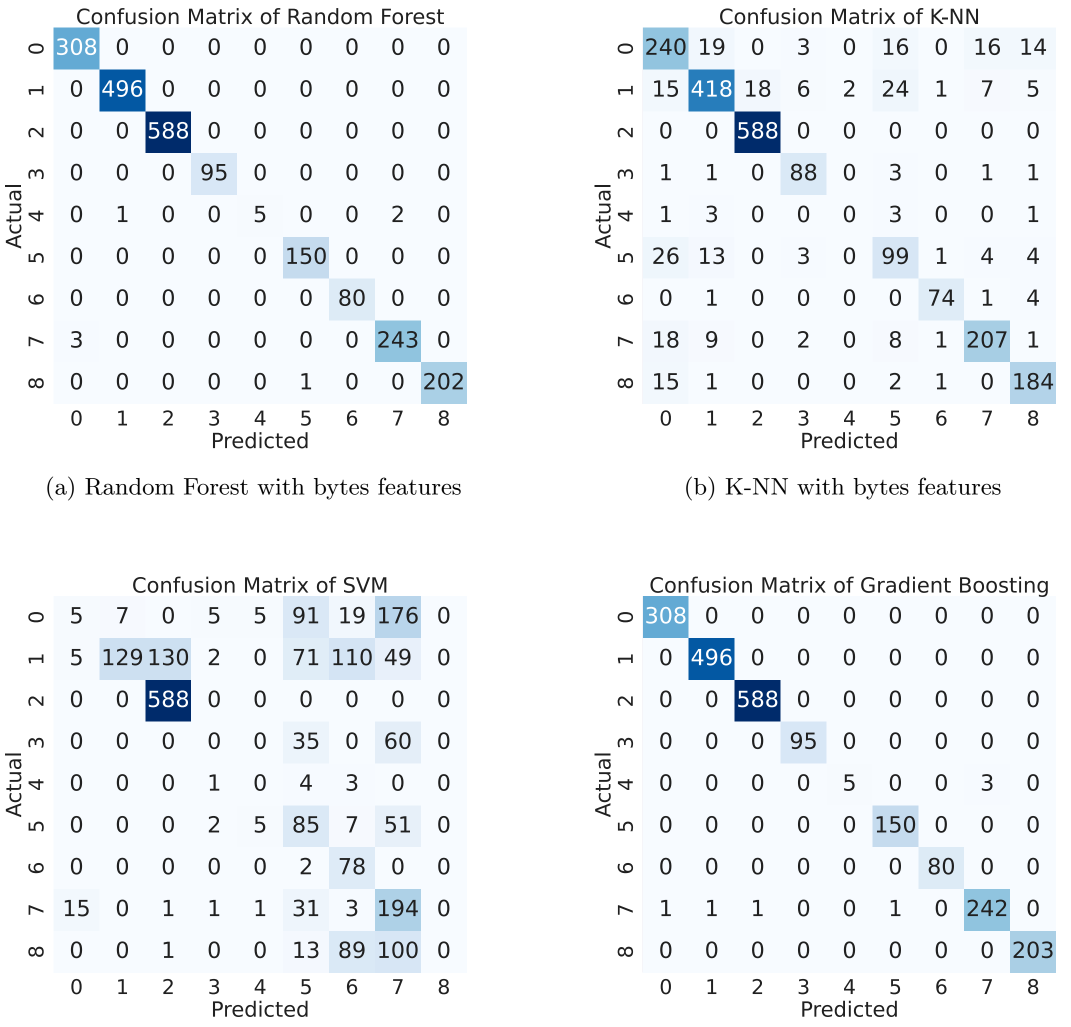
However, we can see from the confusion matrices (Figure 6.11) that the SVM is much more balanced in its decision-making than the opcode 4-gram. In fact, this feature is a better match for this algorithm, which is not entirely taken for granted.

As can be seen from Table 6.6, the results of the metrics on SVM did not lead to notice this marked improvement, this is because the results are always negative but this time the decisions are made following reasoning constructed during the training phase. In the previous situation, on the other hand, during training the algorithm learned very little besides the unbalanced distribution of the dataset.

6.2.4 Comments Bytes features

Bringing together the features extracted from the bytes files led, as one might have expected, to very good results. This statement comes from the fact that among the three features extracted from the bytes files were the bytes 3-gram, which as we know led to excellent results. Since the results were already excellent in themselves, it is difficult for there to be a marked improvement, and in fact the accuracy increased not by much for Random Forest and Gradient Boosting. If we compare table 6.4 with table 6.7, we can see that there are two things that stand out the most: the deterioration of K-NN and SVM.

The Random forest and Gradient Boosting algorithms learned more by having more information at their disposal, such behaviour was what all four of us expected. In fact, if we reflect, having in addition to grams of bytes additional information about the size of the binary file and the total entropy of the file, we should be better able to classify files. SVM and K-NN, on the other hand, do not reflect such behaviour. As to the reasons behind this phenomenon, I can only speculate, and my first thought was an excess of information for these two algorithms. Since they are algorithms that make decisions based on the positioning of samples in the classification space



Confusion Matrix of Random Forest

Actual \ Predicted	0	1	2	3	4	5	6	7	8
0	308	0	0	0	0	0	0	0	0
1	0	496	0	0	0	0	0	0	0
2	0	0	588	0	0	0	0	0	0
3	0	0	0	95	0	0	0	0	0
4	0	1	0	0	5	0	0	2	0
5	0	0	0	0	0	150	0	0	0
6	0	0	0	0	0	0	80	0	0
7	3	0	0	0	0	0	0	243	0
8	0	0	0	0	0	1	0	0	202

(a) Random Forest with bytes features

Confusion Matrix of K-NN

Actual \ Predicted	0	1	2	3	4	5	6	7	8
0	240	19	0	3	0	16	0	16	14
1	15	418	18	6	2	24	1	7	5
2	0	0	588	0	0	0	0	0	0
3	1	1	0	88	0	3	0	1	1
4	1	3	0	0	0	3	0	0	1
5	26	13	0	3	0	99	1	4	4
6	0	1	0	0	0	0	74	1	4
7	18	9	0	2	0	8	1	207	1
8	15	1	0	0	0	2	1	0	184

(b) K-NN with bytes features

Confusion Matrix of SVM

Actual \ Predicted	0	1	2	3	4	5	6	7	8
0	5	7	0	5	5	91	19	176	0
1	5	129	130	2	0	71	110	49	0
2	0	0	588	0	0	0	0	0	0
3	0	0	0	0	0	35	0	60	0
4	0	0	0	1	0	4	3	0	0
5	0	0	0	2	5	85	7	51	0
6	0	0	0	0	0	2	78	0	0
7	15	0	1	1	1	31	3	194	0
8	0	0	1	0	0	13	89	100	0

(c) SVM with bytes features

Confusion Matrix of Gradient Boosting

Actual \ Predicted	0	1	2	3	4	5	6	7	8
0	308	0	0	0	0	0	0	0	0
1	0	496	0	0	0	0	0	0	0
2	0	0	588	0	0	0	0	0	0
3	0	0	0	95	0	0	0	0	0
4	0	0	0	0	5	0	0	3	0
5	0	0	0	0	0	150	0	0	0
6	0	0	0	0	0	0	80	0	0
7	1	1	1	0	0	1	0	242	0
8	0	0	0	0	0	0	0	0	203

(d) Gradient Boosting with bytes features

Figure 6.12: Confusion Matrix of each algorithm with bytes features. Confusion matrix from the last training.

created by the algorithms, having an excess of dimensions can lead to having more difficulty in creating a clear separation between the various classes. Having said this, I continue to emphasise that if the entropy were extracted differently, as in the case of ST-WinMal dataset, we would obtain totally different results.

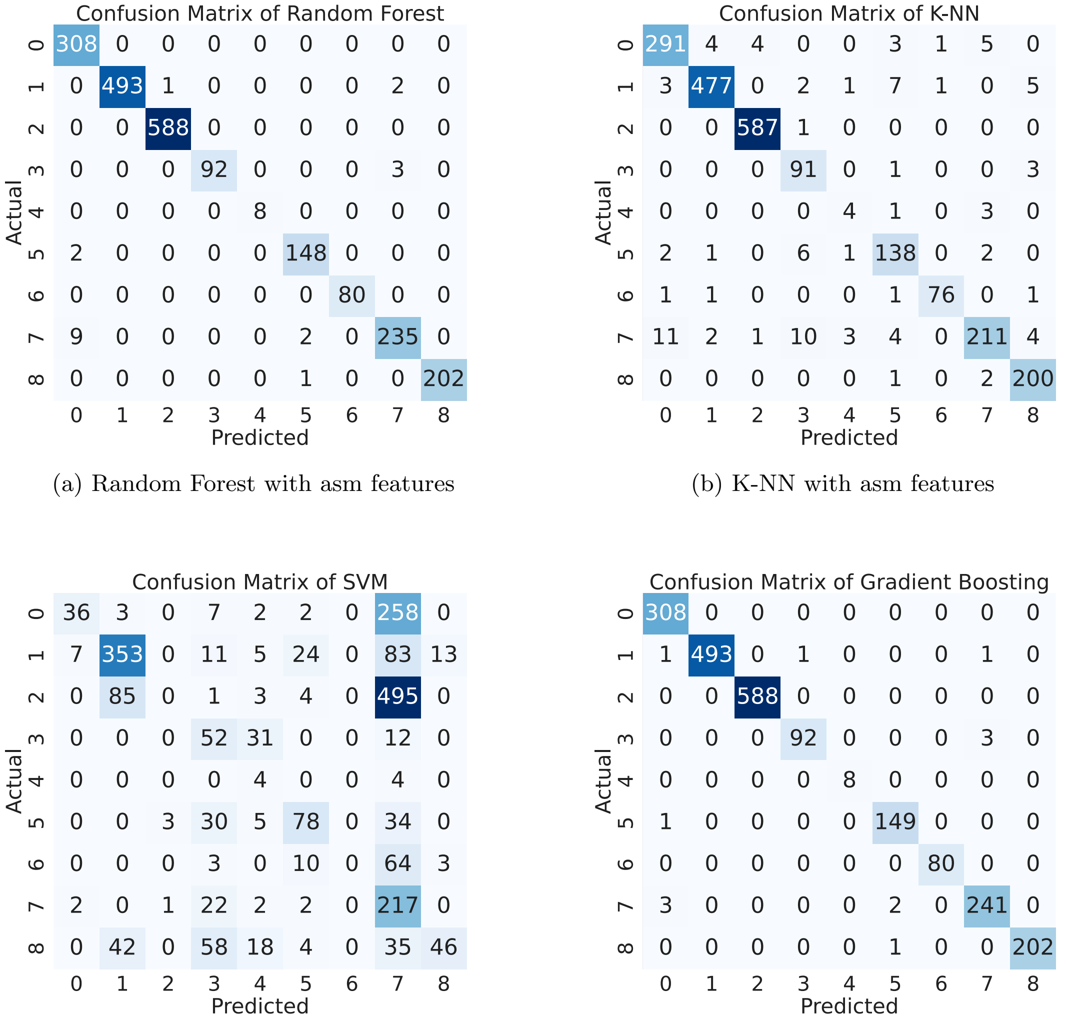
I conclude by stating that the study of the confusion matrices in figure 6.12 confirm the correlation of the data between the metrics and the actual classification of samples. The only particular note on the confusion matrices is the poor ability of the SVM to understand the unequal distribution of samples between the various classes in the dataset.

6.2.5 Comments Asm features

As can be seen from the table 6.8, the combination of the four features extracted from the .asm files leads to excellent results. The Random Forest and Gradient Boosting once again find themselves competing for the place of best algorithm for malware classification, while K-NN improves greatly on the previously reported results. SVM, on the other hand, still tends to have very low results in general. The ROC curve again shows us that it is not a good metric in the field of classification, in fact it easily reaches high numbers even with poor results.

Asm features	RF	K-NN	SVM	GB
Accuracy	0.9916 ± 0.0007	0.9546 ± 0.0030	0.3635 ± 0.0021	$\mathbf{0.9952 \pm 0.0014}$
Macro-Precision	$\mathbf{0.9914 \pm 0.0001}$	0.8944 ± 0.0135	0.3804 ± 0.0129	0.9900 ± 0.0084
Weighted-Precision	0.9917 ± 0.0008	0.9553 ± 0.0030	0.4297 ± 0.0253	$\mathbf{0.9953 \pm 0.0014}$
Marco-Recall	0.9820 ± 0.0146	0.8823 ± 0.0131	0.3859 ± 0.0094	$\mathbf{0.9854 \pm 0.0134}$
Weighted-Recall	0.9916 ± 0.0007	0.9546 ± 0.0030	0.3635 ± 0.0021	$\mathbf{0.9952 \pm 0.0014}$
Macro-F1 Score	0.9860 ± 0.0084	0.8832 ± 0.0045	0.2891 ± 0.0039	$\mathbf{0.9869 \pm 0.0057}$
Weighted-F1 Score	0.9916 ± 0.0007	0.9544 ± 0.0029	0.3138 ± 0.0018	$\mathbf{0.9952 \pm 0.0014}$
Micro-AUC ROC	0.9997 ± 0.0003	0.9870 ± 0.0011	0.9201 ± 0.0014	$\mathbf{1.0000 \pm 0.0001}$
Marco-AUC ROC	0.9996 ± 0.0003	0.9665 ± 0.0144	0.8880 ± 0.0009	$\mathbf{0.9999 \pm 0.0001}$
Weighted- AUC ROC	0.9996 ± 0.0003	0.9867 ± 0.0011	0.9060 ± 0.0008	$\mathbf{1.0000 \pm 0.0001}$

Table 6.8: Results of the different models (RF: Random Forest, K-NN: K-Nearest Neighbors, SVM: Support Vector Machine, GB: Gradient Boosting) in the malware classification task with asm features. Results were aggregated over 3 training runs. Best results for feature are shown in **bold**.



(a) Random Forest with asm features

(b) K-NN with asm features

(c) SVM with asm features

(d) Gradient Boosting with asm features

Figure 6.13: Confusion Matrix of each algorithm with asm features. Confusion matrix from the last training.

The motivation behind these excellent results is the inclusion of the opcode counter and the

3 Best features	RF	K-NN	SVM	GB
Accuracy	0.9962 ± 0.0014	0.9621 ± 0.0015	0.6357 ± 0.0112	**0.9979 ± 0.0006**
Macro-Precision	**0.9964 ± 0.0011**	0.8984 ± 0.0043	0.6396 ± 0.0149	0.9932 ± 0.0080
Weighted-Precision	0.9962 ± 0.0014	0.9626 ± 0.0016	0.7404 ± 0.0124	**0.9979 ± 0.0006**
Marco-Recall	0.9912 ± 0.0083	0.9056 ± 0.0042	0.5181 ± 0.0141	**0.9932 ± 0.0072**
Weighted-Recall	0.9962 ± 0.0014	0.9621 ± 0.0015	0.6357 ± 0.0223	**0.9979 ± 0.0006**
Macro-F1 Score	**0.9936 ± 0.0046**	0.9017 ± 0.0032	0.4620 ± 0.0032	0.9929 ± 0.0043
Weighted-F1 Score	0.9962 ± 0.0014	0.9621 ± 0.0015	0.6136 ± 0.0067	**0.9979 ± 0.0005**
Micro-AUC ROC	0.9999 ± 0.0001	0.9917 ± 0.0006	0.9426 ± 0.0045	**1.0000 ± 0.0000**
Marco-AUC ROC	0.9999 ± 0.0001	0.9796 ± 0.0101	0.8991 ± 0.0045	**1.0000 ± 0.0000**
Weighted- AUC ROC	0.9999 ± 0.0001	0.9915 ± 0.0006	0.9338 ± 0.0024	**1.0000 ± 0.0000**

Table 6.9: Results of the different models (RF: Random Forest, K-NN: K-Nearest Neighbors, SVM: Support Vector Machine, GB: Gradient Boosting) in the malware classification task with the three best features. Results were aggregated over 3 training runs. Best results for feature are shown in **bold**.

opcode 4-gram. Together they manage to overcome the classifier's difficulties in using the API check and the 4-gram of API, which alone brought poor results.

The confusion matrices in figure 6.13 confirm the correlation with the data. Here, the SVM heavily favours the Obfuscator.ACY class, probably because it cannot overcome the difficulties imposed by malware authors with obfuscation and/or encryption techniques.

6.2.6 Comments Best features

In this section, no feature or combination of them led to higher results. As was easy to imagine, SVM and K-NN have also improved, to be precise K-NN is a very good algorithm with this feature, not at the level of Random Forest or Gradient Boosting but it exceeds 95% which was an important threshold for me to reach. SVM still remains poor but manages to reach 60% score, not so trivial given previous experiments.

This time the algorithms also learn from the addition of the new information. Obviously already with the use of the individual features, very good results were obtained, but combining them made it possible to extract the maximum from all 4 models. Therefore, it can be concluded that this is the best possible combination if we were to generalise for any algorithm the input to be extracted from the dataset.

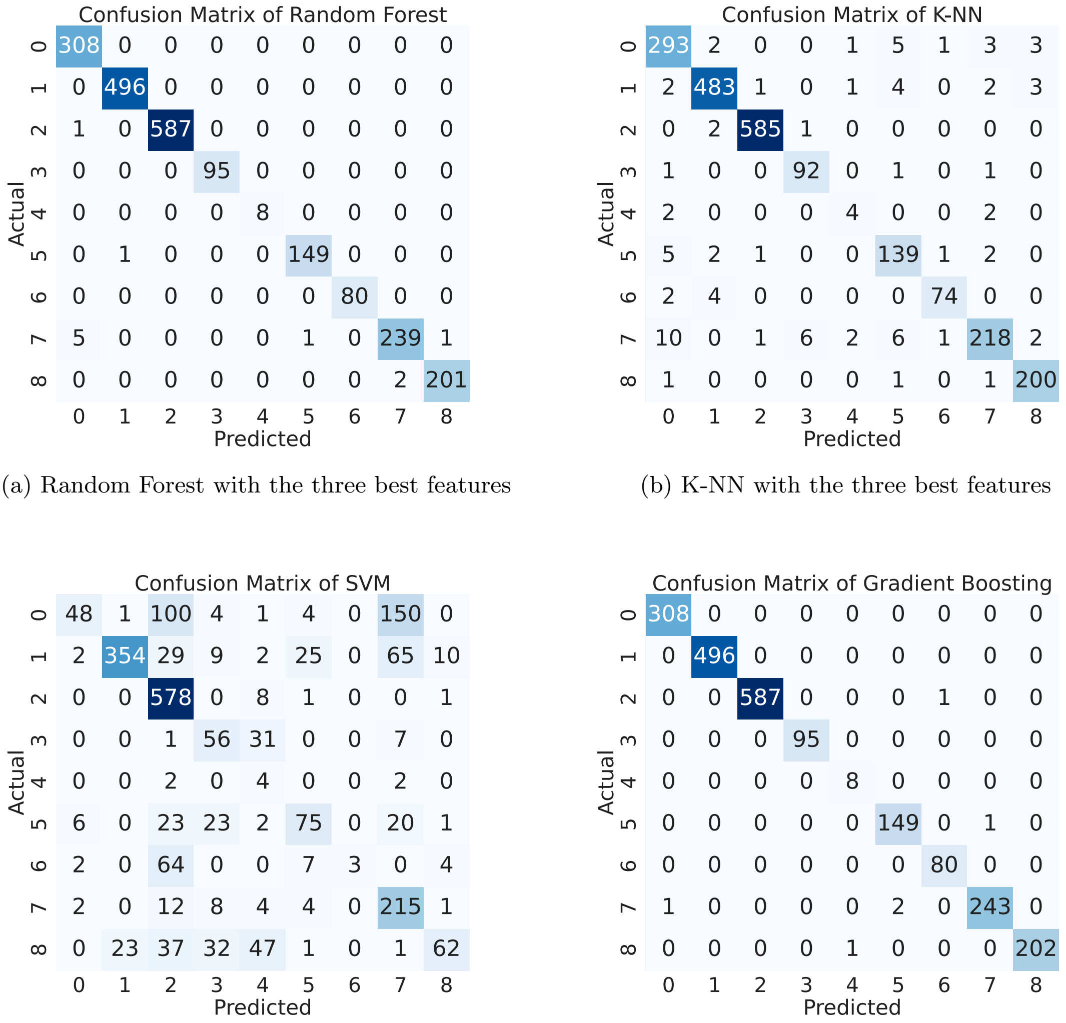
From the table 6.9 we can see that in some metrics the Random Forest manages to outperform (always by a little) Gradient Boosting. This event allows us to understand how Random Forest is still a very good algorithm to use for this task.

The confusion matrices in figure 6.14 confirm the goodness of this input, in fact the SVM also learns much more egregiously than with previous inputs. It succeeds first of all in not only predicting the usual 2/3 classes, but it succeeds in distributing its decisions better. It still fails to have noteworthy accuracy.

6.2.7 Summary

After the previous analysis, we can confirm that there are better algorithms and features for this type of task. In fact, we confirm that SVM is an imprecise algorithm with this dataset, which runs into many difficulties. With regard to the other algorithms, however, we note that they perform exceptionally well. K-NN even though it does not bring the results of the other algorithms with some features manages to behave quite adequately. I am convinced that with a little more work it can easily reach the level of Random Forest and Gradient Boosting.

About the latter two algorithms there is not much to say, they are in fact very precise, tend to have a high recall, which is very positive, and very good precision, another factor in their favour.



Confusion Matrix of Random Forest

Actual / Predicted	0	1	2	3	4	5	6	7	8
0	308	0	0	0	0	0	0	0	0
1	0	496	0	0	0	0	0	0	0
2	1	0	587	0	0	0	0	0	0
3	0	0	0	95	0	0	0	0	0
4	0	0	0	0	8	0	0	0	0
5	0	1	0	0	0	149	0	0	0
6	0	0	0	0	0	0	80	0	0
7	5	0	0	0	0	1	0	239	1
8	0	0	0	0	0	0	0	2	201

(a) Random Forest with the three best features

Confusion Matrix of K-NN

Actual / Predicted	0	1	2	3	4	5	6	7	8
0	293	2	0	0	1	5	1	3	3
1	2	483	1	0	1	4	0	2	3
2	0	2	585	1	0	0	0	0	0
3	1	0	0	92	0	1	0	1	0
4	2	0	0	0	4	0	0	2	0
5	5	2	1	0	0	139	1	2	0
6	2	4	0	0	0	0	74	0	0
7	10	0	1	6	2	6	1	218	2
8	1	0	0	0	0	1	0	1	200

(b) K-NN with the three best features

Confusion Matrix of SVM

Actual / Predicted	0	1	2	3	4	5	6	7	8
0	48	1	100	4	1	4	0	150	0
1	2	354	29	9	2	25	0	65	10
2	0	0	578	0	8	1	0	0	1
3	0	0	1	56	31	0	0	7	0
4	0	0	2	0	4	0	0	2	0
5	6	0	23	23	2	75	0	20	1
6	2	0	64	0	0	7	3	0	4
7	2	0	12	8	4	4	0	215	1
8	0	23	37	32	47	1	0	1	62

(c) SVM with the three best features

Confusion Matrix of Gradient Boosting

Actual / Predicted	0	1	2	3	4	5	6	7	8
0	308	0	0	0	0	0	0	0	0
1	0	496	0	0	0	0	0	0	0
2	0	0	587	0	0	0	1	0	0
3	0	0	0	95	0	0	0	0	0
4	0	0	0	0	8	0	0	0	0
5	0	0	0	0	0	149	0	1	0
6	0	0	0	0	0	0	80	0	0
7	1	0	0	0	0	2	0	243	0
8	0	0	0	0	1	0	0	0	202

(d) Gradient Boosting with the three best features

Figure 6.14: Confusion Matrix of each algorithm with the three best features. Confusion matrix from the last training.

The F1 score sums up very well how excellent these two algorithms are for this research. In summary, we can therefore say that opcode or byte grams are the best source of information that can be extracted from this dataset, along with the assembly function count within the disassembled file.

6.3 Malware detection results with ST-WinMal dataset

From here on, the results of the experiments performed with ST-WinMal dataset will be addressed. Before starting to present the results, it is worth pointing out the differences with the first dataset. Here we are testing a dataset created from scratch by collecting several samples from different sources. We therefore have much more heterogeneity of samples. Moreover, ST-WinMal dataset is much more balanced than the official Microsoft dataset. Here each class also has malware inside that employs obfuscation techniques, so it is easier for the malware detector or classifier to learn more about it during the training phase.

The dataset consists of executables, but because we use features as input that were not extractable for some samples, the number of malware is slightly reduced. In the table 6.10 we can

Trojan		Ransomware	
Family Name	*Number of Samples*	*Family Name*	*Number of Samples*
Ramnit	1,499	Virlock	744
Zbot	1,159	Stop	1,000
IceId	860	Magniber	804
Trickbot	725	Wannacry	1,444
Benign		955	

Table 6.10: Malware families in ST-WinMal dataset combined

Algorithms	Feature			
	Bytes 3-gram	Entropy	API check	All features
AUC-ROC				
RF	0.9984 ± 0.0002	0.9931 ± 0.0012	**0.9921 ± 0.0014**	0.9978 ± 0.0004
K-NN	0.9229 ± 0.0004	0.9725 ± 0.0057	0.9418 ± 0.0036	0.9318 ± 0.0120
SVM	0.9853 ± 0.0025	0.9726 ± 0.0121	0.9693 ± 0.0271	0.9890 ± 0.0015
GB	**0.9992 ± 0.0004**	**0.9950 ± 0.0014**	0.9893 ± 0.0022	**0.9989 ± 0.0002**
Accuracy				
RF	0.9873 ± 0.0019	0.9839 ± 0.0026	**0.9819 ± 0.0025**	0.9888 ± 0.0091
K-NN	0.9554 ± 0.0029	0.9783 ± 0.0038	0.8943 ± 0.0036	0.9626 ± 0.0051
SVM	0.9545 ± 0.0016	0.9708 ± 0.0039	0.9616 ± 0.0049	0.9714 ± 0.0014
GB	**0.9949 ± 0.0003**	**0.9850 ± 0.0031**	0.9788 ± 0.0034	**0.9940 ± 0.0011**
Recall				
RF	0.9968 ± 0.0016	**0.9966 ± 0.0010**	0.9901 ± 0.0017	0.9964 ± 0.0016
K-NN	0.9866 ± 0.0032	0.9925 ± 0.0046	0.9014 ± 0.0052	0.9897 ± 0.0016
SVM	0.9980 ± 0.0020	0.9897 ± 0.0012	**0.9933 ± 0.0043**	0.9968 ± 0.0013
GB	**0.9978 ± 0.0009**	0.9961 ± 0.0013	0.9877 ± 0.0043	**0.9976 ± 0.0016**
Precision				
RF	0.9891 ± 0.0016	0.9856 ± 0.0036	**0.9897 ± 0.0010**	0.9911 ± 0.0018
K-NN	0.9644 ± 0.0001	0.9834 ± 0.0038	0.9790 ± 0.0095	0.9693 ± 0.0042
SVM	0.9534 ± 0.0028	0.9780 ± 0.0047	0.9649 ± 0.0085	0.9720 ± 0.0017
GB	**0.9966 ± 0.0007**	**0.9872 ± 0.0029**	0.9887 ± 0.0018	**0.9958 ± 0.0006**
F1 Score				
RF	0.9929 ± 0.0011	0.9910 ± 0.0014	**0.9899 ± 0.0014**	0.9937 ± 0.0017
K-NN	0.9754 ± 0.0016	0.9879 ± 0.0021	0.9386 ± 0.0017	0.9318 ± 0.0120
SVM	0.9752 ± 0.0009	0.9838 ± 0.0021	0.9789 ± 0.0026	0.9890 ± 0.0015
GB	**0.9972 ± 0.0002**	**0.9916 ± 0.0017**	0.9882 ± 0.0019	**0.9989 ± 0.0002**

Table 6.11: Results of the different models (RF: Random Forest, K-NN: K-Nearest Neighbors, SVM: Support Vector Machine, GB: Gradient Boosting) in the malware identification task with different features. Results were aggregated over 3 training runs. Best results for feature are shown in **bold**.

see the distribution of the dataset composed of all features (binary file, file containing the API and file containing the entropy divided by section)

The metrics used will be the same as those used with the Microsoft dataset, their description can be found at the beginning of this chapter.

6.3.1 Comments

The table 6.11 shows all results obtained with the ST-WinMal dataset, showing the area under the ROC curve, accuracy, recall, precision and F1 score. As already mentioned, the ROC curve is an excellent metric to analyse in the field of binary classification, which is why, as in the last dataset, they have all been reported in figures 6.15, 6.16, 6.17 and 6.18. By changing the dataset, the results changed a lot with regard to entropy. In fact we can confirm my hypothesis already

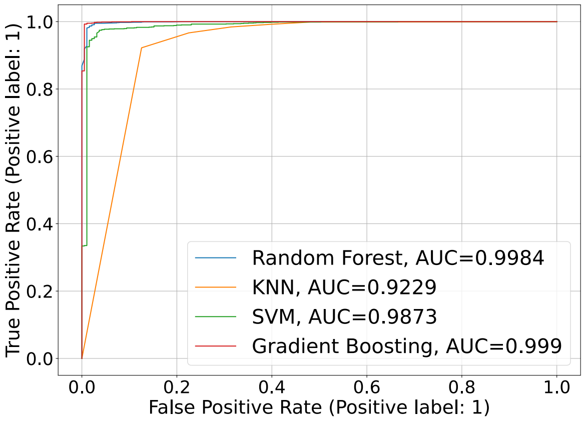


Figure 6.15: ROC curve and AUC-ROC for bytes 3-gram.

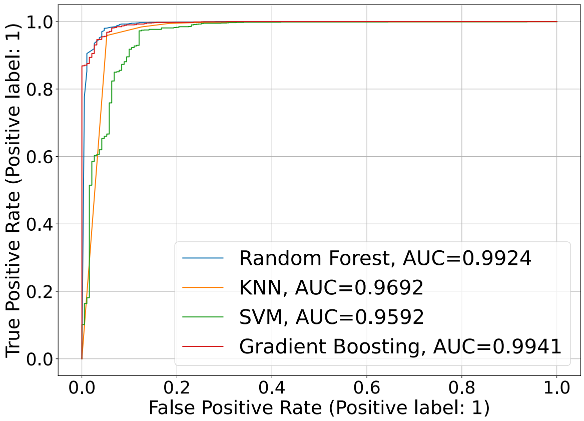


Figure 6.16: ROC curve and AUC-ROC for entropy.

explained several times above, namely the importance of how they are extracted and features. We can see that the entropy extracted over the whole file did not lead to exciting results with the last dataset, whereas now with ST-WinMal we have achieved excellent results. In terms of the feature in common between the two datasets, we can see that the bytes 3-gram had very little difference from the last dataset. Precision dropped slightly but not so much as to assume a problem in feature extraction (order 10^{-3}).

With the introduction of the use of APIs to identify malware, very promising results were achieved. It is the feature with the lowest but still very high results with some algorithms. The Random Forest, which achieves the best results with this feature, achieves an AUC-ROC and recall of over 0.99. Indeed, recall is a very important metric for us as the higher it is, the more likely it is that our model does not predict many malware as benign files (a worse situation than predicting benign files as malware).

As far as accuracy is concerned, the model that brought the highest results was gradient boosting using bytes gram, confirming that they are the most suitable feature for this type of task. Joining the features together, it can be seen that the metrics receive a slight lowering compared to the use of grams of bytes. This phenomenon had already occurred with the last dataset, demonstrating how putting in too much information does not necessarily lead to better results.

As with the last dataset, confusion matrices were also reported here. The best results can be seen in the matrices 6.6d and 6.8d where the two error classes have less than ten predictions each.

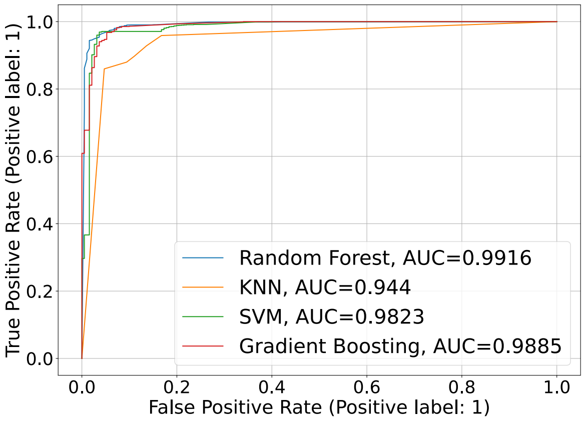


Figure 6.17: ROC curve and AUC-ROC for API check.

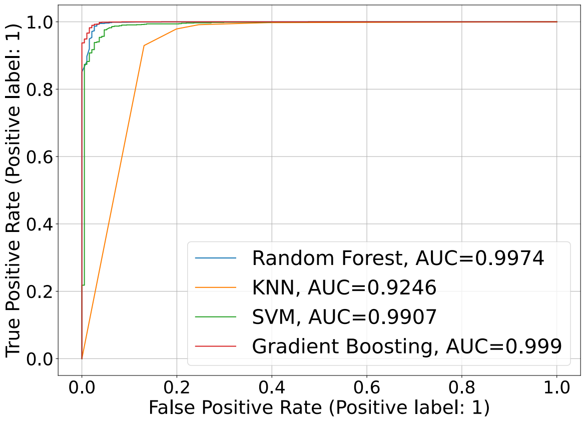


Figure 6.18: ROC curve and AUC-ROC for all feature.

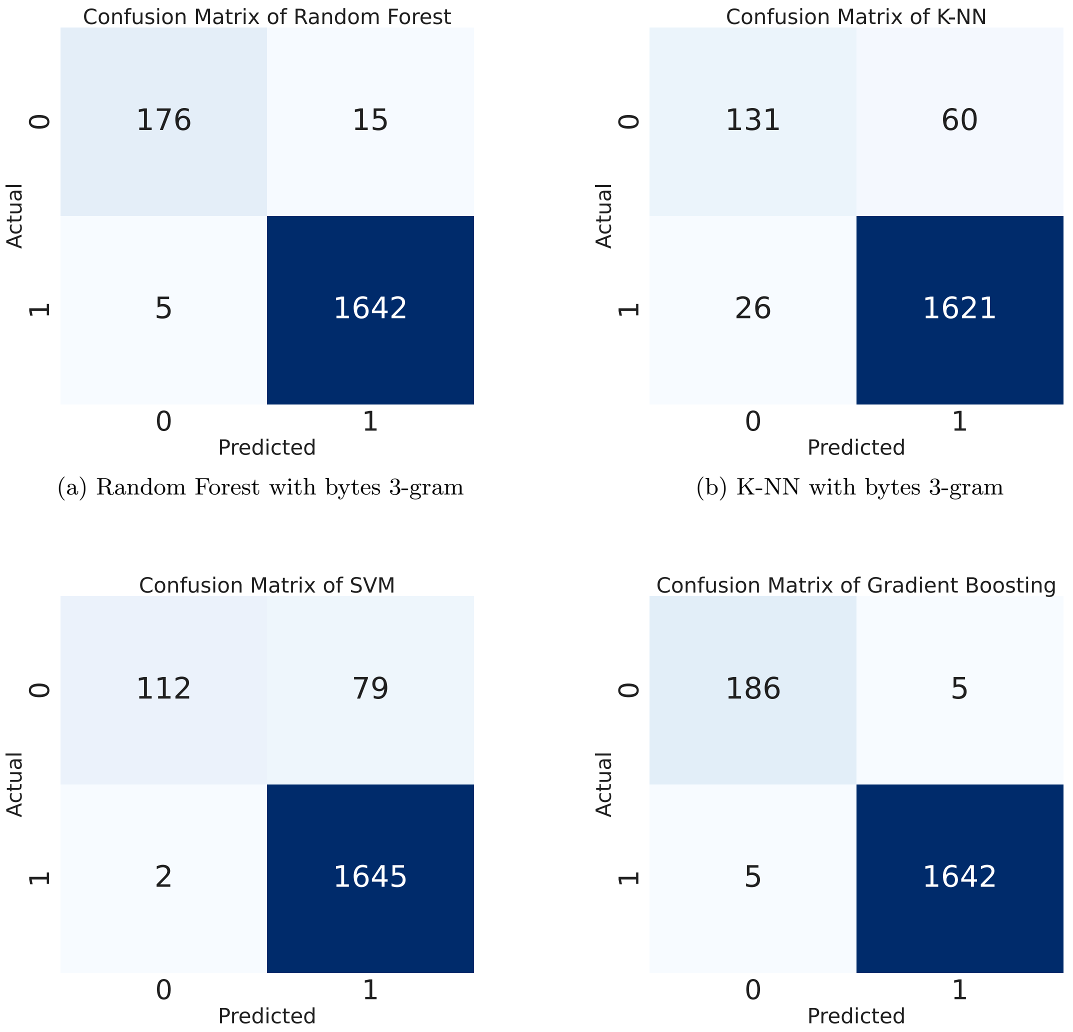
In general, all the matrices performed very well, as already shown in the table, only the only one we can define as a failure is in figure 6.21b. This matrix reports that 170 malware were predicted as benign files, a very poor result that leads us to exclude this model as a possible choice for the work.

We can finally conclude how Random Forest and Gradient boosting remain the two best algorithms even with this dataset, SVM on the other hand becoming a much more desirable model for this type of task, and K-NN taking the role of the worst algorithm. So it can be argued that the choice of dataset is also very important in order to understand which algorithm to choose. In fact, SVM previously seemed an option to be immediately discarded a priori, whereas with the use of the ST-WinMal dataset, it achieved very encouraging results.

6.4 Malware classification results with ST-WinMal dataset

This section will show the results achieved with the three available features together with the use of all three integrated into one input. In contrast to section 6.2, all results will be shown as they achieved very similar results and there are only four different characteristics to the seven extracted from the previous dataset.

As specified in the section with the experiments on malware classification with Microsoft's dataset, the best metric for comparison is the F1 score, it best specifies whether the model works



(a) Random Forest with bytes 3-gram

(b) K-NN with bytes 3-gram

(c) SVM with bytes 3-gram

(d) Gradient boosting with bytes 3-gram

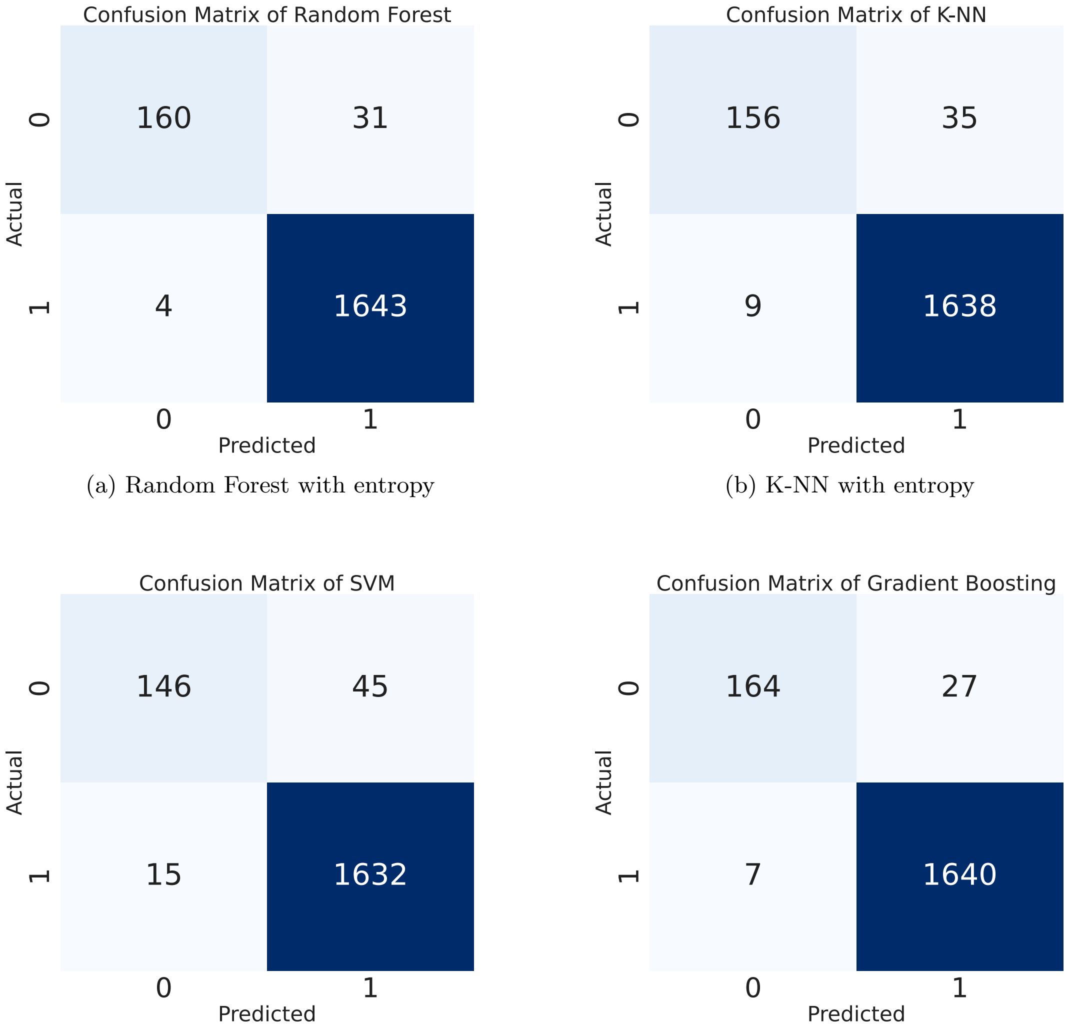
Figure 6.19: Confusion Matrix of each algorithm with bytes 3-gram feature (0: not malware, 1: malware). Confusion matrix from the last training.

Number	*Family*
0	Ramnit
1	Zbot
2	IcedId
3	Trickbot
4	Virlock
5	Stop
6	Magniber
7	Wannacry

Table 6.12: family legend

correctly or not, as it summarises accuracy and recall in a single metric.

In addition to the numerical results, the confusion matrices have been reported in order to confirm that the data shown in the tables have a practical match. In the confusion matrices for readability issues, numbers were inserted to identify the classes instead of their names. A legend of the families can be found in the table 6.12.



(a) Random Forest with entropy

(b) K-NN with entropy

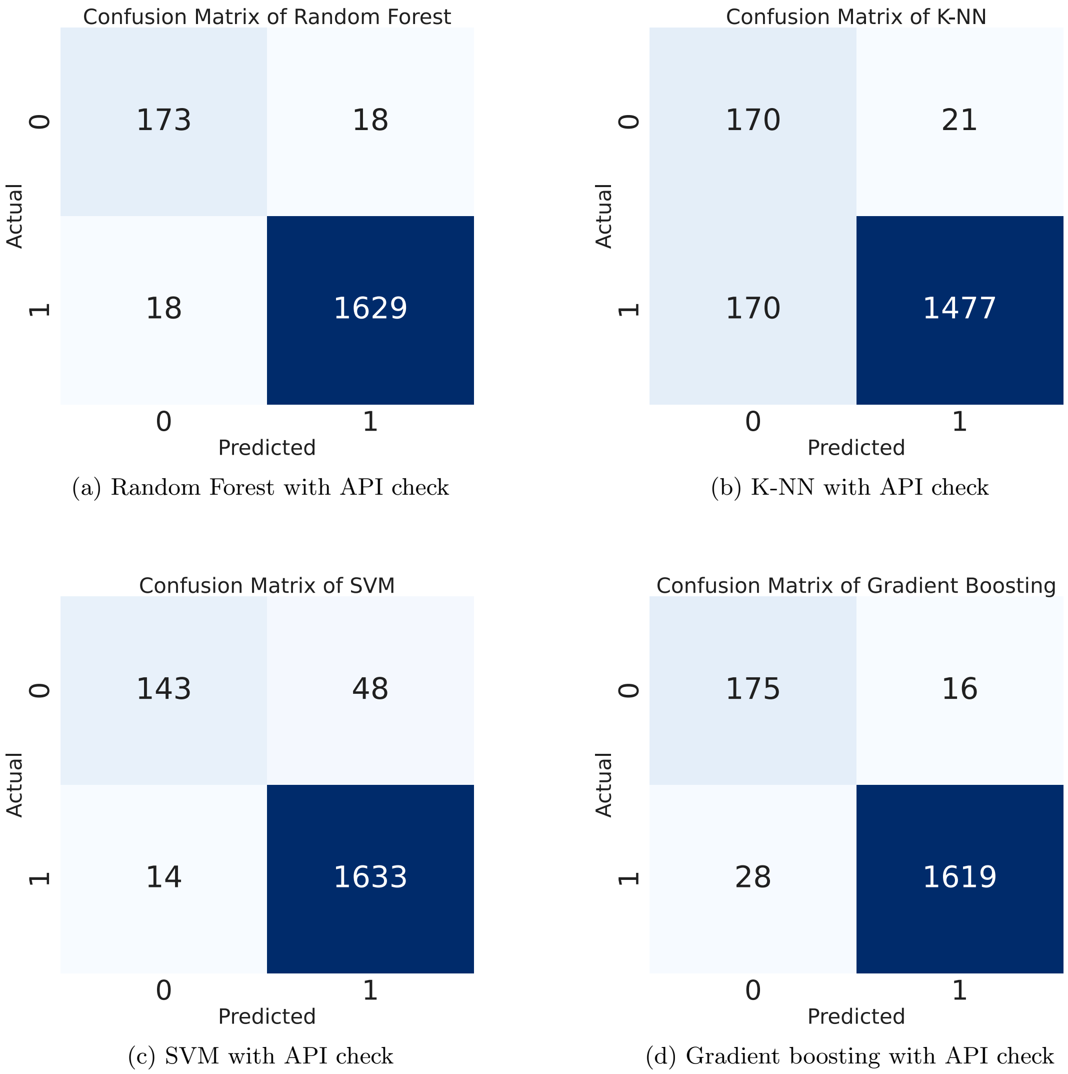
(c) SVM with entropy

(d) Gradient boosting with entropy

Figure 6.20: Confusion Matrix of each algorithm with entropy feature (0: not malware, 1: malware). Confusion matrix from the last training.

Bytes 3-gram	RF	K-NN	SVM	GB
Accuracy	0.9769 ± 0.0012	0.8676 ± 0.0016	0.7282 ± 0.0129	$\mathbf{0.9850} \pm \mathbf{0.0017}$
Macro-Precision	0.9768 ± 0.0026	0.8764 ± 0.0041	0.7406 ± 0.0068	$\mathbf{0.9837} \pm \mathbf{0.0025}$
Weighted-Precision	0.9773 ± 0.0012	0.8715 ± 0.0003	0.7651 ± 0.0062	$\mathbf{0.9852} \pm \mathbf{0.0017}$
Marco-Recall	0.9760 ± 0.0006	0.8658 ± 0.0019	0.6833 ± 0.0130	$\mathbf{0.9842} \pm \mathbf{0.0025}$
Weighted-Recall	0.9769 ± 0.0012	0.8677 ± 0.0016	0.7282 ± 0.0129	$\mathbf{0.9850} \pm \mathbf{0.0017}$
Macro-F1 Score	0.9762 ± 0.0016	0.8688 ± 0.0010	0.6666 ± 0.0146	$\mathbf{0.9839} \pm \mathbf{0.0025}$
Weighted-F1 Score	0.9769 ± 0.0012	0.8670 ± 0.0015	0.7044 ± 0.0138	$\mathbf{0.9850} \pm \mathbf{0.0018}$
Micro-AUC ROC	0.9991 ± 0.0004	0.9672 ± 0.0038	0.9876 ± 0.0004	$\mathbf{0.9997} \pm \mathbf{0.0002}$
Marco-AUC ROC	0.9991 ± 0.0002	0.9647 ± 0.0029	0.9835 ± 0.0008	$\mathbf{0.9996} \pm \mathbf{0.0001}$
Weighted- AUC ROC	0.9990 ± 0.0003	0.9679 ± 0.0039	0.9838 ± 0.0005	$\mathbf{0.9996} \pm \mathbf{0.0002}$

Table 6.13: Results of the different models (RF: Random Forest, K-NN: K-Nearest Neighbors, SVM: Support Vector Machine, GB: Gradient Boosting) in the malware classification task with bytes 3-gram. Results were aggregated over 3 training runs. Best results for feature are shown in **bold**.



(a) Random Forest with API check

(b) K-NN with API check

(c) SVM with API check

(d) Gradient boosting with API check

Figure 6.21: Confusion Matrix of each algorithm with API check feature (0: not malware, 1: malware). Confusion matrix from the last training.

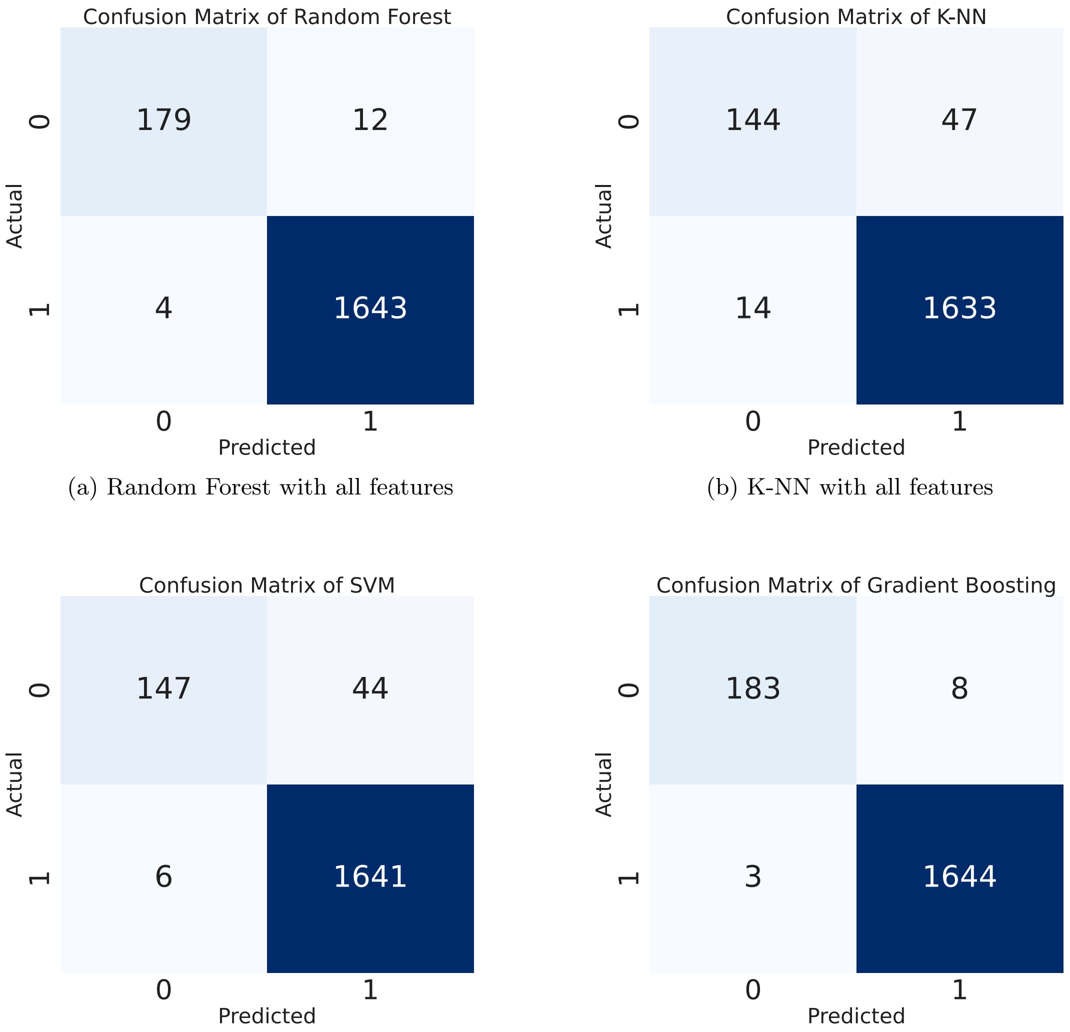
6.4.1 Comments Bytes 3-gram

The first feature under analysis is the bytes 3-gram. They reported the best results in the field of identification, and proved to be the best feature to use in this task as well. The reported data (Table 6.13 show a clear superiority of the Gradient Boosting algorithm, which achieves almost 99 per cent accuracy, ranking all eight classes of the dataset very well, without having any imbalance problems.

Another noteworthy algorithm is the Random Forest, which achieves excellent results in this area as well. We can then reconfirm the same situation seen with the Microsoft dataset. In fact even here the K-NN and SVM fail to bring noteworthy results, the K-NN actually gets worse than the experiments done with the previous dataset, while the SVM improves a lot but still fails to achieve results comparable to the K-NN.

Here again we can see that the area under the ROC curve is not a very reliable metric. It achieves excellent numbers even with the two algorithms that have just been described as not accurate enough for this type of research, going to not give a clear representation of the situation shown by the other metrics.

The confusion matrices in Figure 6.23 graphically confirm the results described in Table 6.13.



(a) Random Forest with all features

(b) K-NN with all features

(c) SVM with all features

(d) Gradient boosting with all features

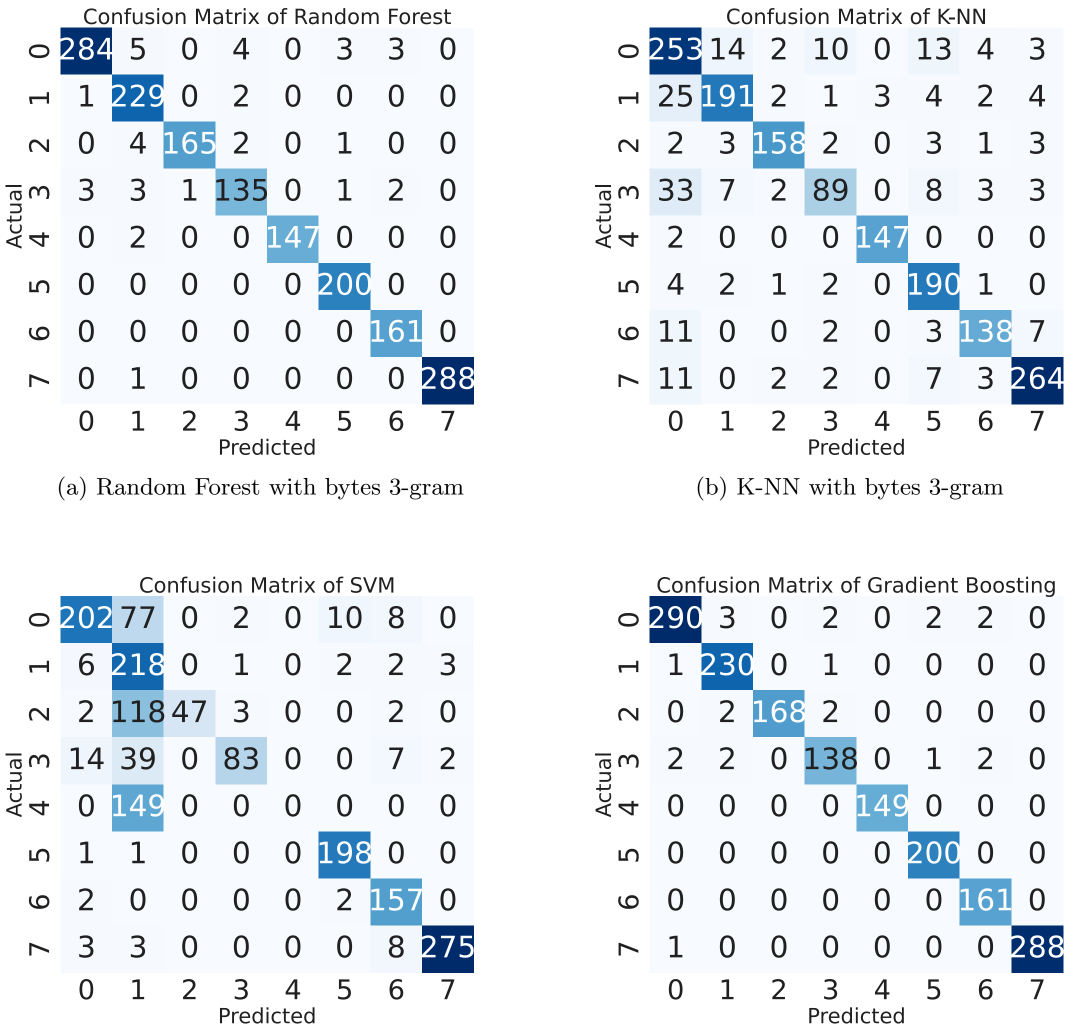
Figure 6.22: Confusion Matrix of each algorithm with all features (0: not malware, 1: malware). Confusion matrix from the last training.

Worth noting are the matrices 6.23a and 6.23d which show that Random Forest and Gradient Boosting are by far the best algorithms to use in this research. In contrast, the SVM (figure 6.23c) again shows a problem in identifying all eight classes in a balanced way. The Zbot family has too high a prediction rate compared to all the other classes. This in fact is the source of many classification problems for this algorithm, which is therefore not suitable for this type of work.

6.4.2 Comments Entropy

To analyze the results with entropy, it is very useful to compare the results obtained with the preceding dataset and those obtained with the ST-WinMal dataset. Since the results obtained with the preceding dataset were not outstanding a table corresponding to the table 6.14 is not present in this thesis but it is possible to compare the F1 scores. They are in fact given in the table 6.2.

It is immediately noticeable that the situation has definitely improved; in fact, f1 scores above 0.9 are now obtained with 3 algorithms. The SVM still reports some problems but it can be confirmed how with this algorithm it is so far the best feature to input. This phenomenon allows us to understand what the general problem is that this algorithm had not yet worked well with



Actual \ Predicted	0	1	2	3	4	5	6	7
0	284	5	0	4	0	3	3	0
1	1	229	0	2	0	0	0	0
2	0	4	165	2	0	1	0	0
3	3	3	1	135	0	1	2	0
4	0	2	0	0	147	0	0	0
5	0	0	0	0	0	200	0	0
6	0	0	0	0	0	0	161	0
7	0	1	0	0	0	0	0	288

(a) Random Forest with bytes 3-gram

Actual \ Predicted	0	1	2	3	4	5	6	7
0	253	14	2	10	0	13	4	3
1	25	191	2	1	3	4	2	4
2	2	3	158	2	0	3	1	3
3	33	7	2	89	0	8	3	3
4	2	0	0	0	147	0	0	0
5	4	2	1	2	0	190	1	0
6	11	0	0	2	0	3	138	7
7	11	0	2	2	0	7	3	264

(b) K-NN with bytes 3-gram

Actual \ Predicted	0	1	2	3	4	5	6	7
0	202	77	0	2	0	10	8	0
1	6	218	0	1	0	2	2	3
2	2	118	47	3	0	0	2	0
3	14	39	0	83	0	0	7	2
4	0	149	0	0	0	0	0	0
5	1	1	0	0	0	198	0	0
6	2	0	0	0	0	2	157	0
7	3	3	0	0	0	0	8	275

(c) SVM with bytes 3-gram

Actual \ Predicted	0	1	2	3	4	5	6	7
0	290	3	0	2	0	2	2	0
1	1	230	0	1	0	0	0	0
2	0	2	168	2	0	0	0	0
3	2	2	0	138	0	1	2	0
4	0	0	0	0	149	0	0	0
5	0	0	0	0	0	200	0	0
6	0	0	0	0	0	0	161	0
7	1	0	0	0	0	0	0	288

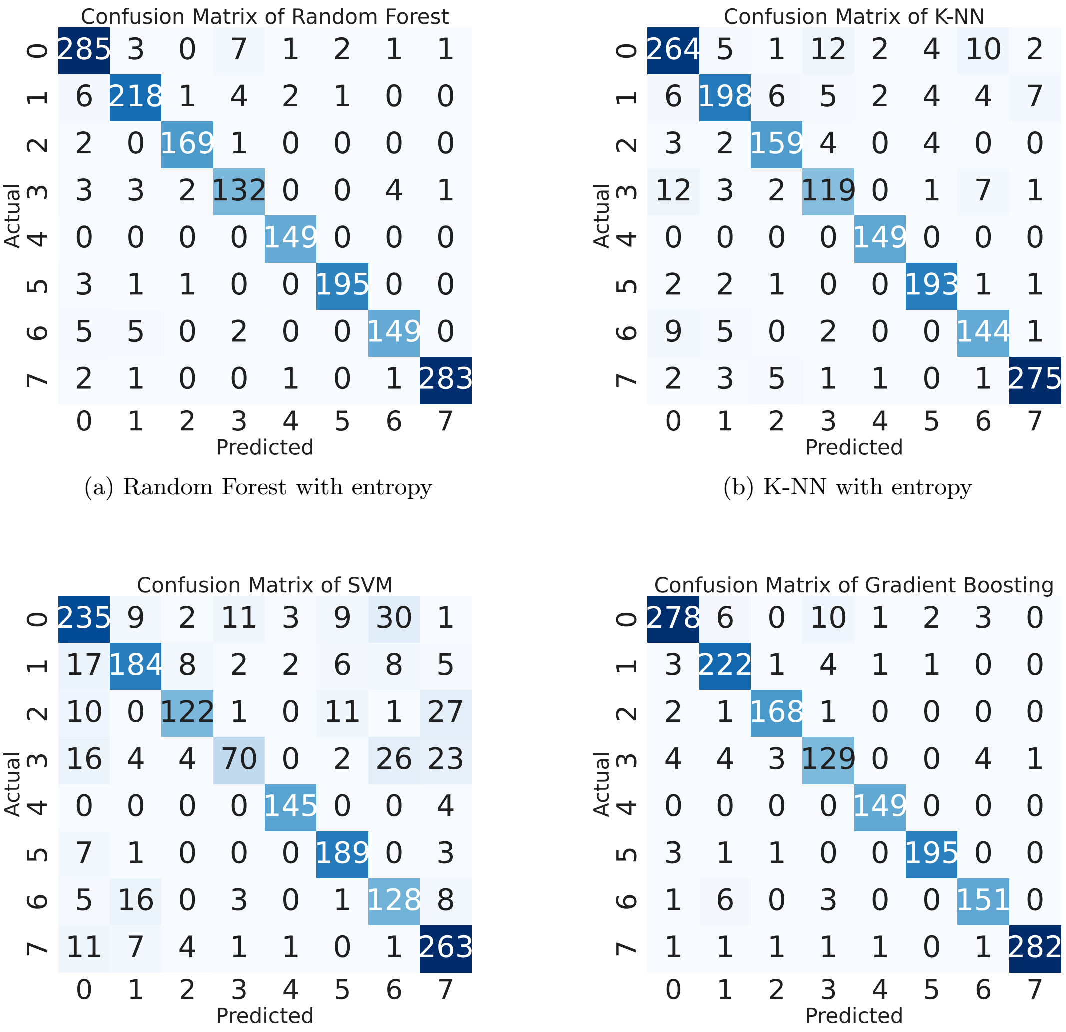
(d) Gradient boosting with bytes 3-gram

Figure 6.23: Confusion Matrix of each algorithm with bytes 3-gram feature (0: not malware, 1: malware). Confusion matrix from the last training.

Entropy	RF	K-NN	SVM	GB
Accuracy	$\mathbf{0.9534 \pm 0.0055}$	0.9142 ± 0.0034	0.8178 ± 0.0189	0.9514 ± 0.0046
Macro-Precision	$\mathbf{0.9532 \pm 0.0060}$	0.9110 ± 0.0034	0.8338 ± 0.0151	0.9492 ± 0.0040
Weighted-Precision	$\mathbf{0.9538 \pm 0.0053}$	0.9138 ± 0.0034	0.8278 ± 0.0143	0.9521 ± 0.0045
Marco-Recall	$\mathbf{0.9528 \pm 0.0049}$	0.9137 ± 0.0035	0.8108 ± 0.0138	0.9513 ± 0.0051
Weighted-Recall	$\mathbf{0.9534 \pm 0.0055}$	0.9142 ± 0.0034	0.8235 ± 0.0142	0.9514 ± 0.0046
Macro-F1 Score	$\mathbf{0.9528 \pm 0.0055}$	0.9121 ± 0.0034	0.8150 ± 0.0149	0.9501 ± 0.0045
Weighted-F1 Score	$\mathbf{0.9534 \pm 0.0056}$	0.9137 ± 0.0034	0.8195 ± 0.0145	0.9516 ± 0.0046
Micro-AUC ROC	0.9962 ± 0.0004	0.9797 ± 0.0010	0.9726 ± 0.0032	$\mathbf{0.9977 \pm 0.0003}$
Marco-AUC ROC	0.9956 ± 0.0001	0.9798 ± 0.0017	0.9652 ± 0.0027	$\mathbf{0.9971 \pm 0.0003}$
Weighted- AUC ROC	0.9955 ± 0.0002	0.9788 ± 0.0011	0.9667 ± 0.0027	$\mathbf{0.9971 \pm 0.0002}$

Table 6.14: Results of the different models (RF: Random Forest, K-NN: K-Nearest Neighbors, SVM: Support Vector Machine, GB: Gradient Boosting) in the malware classification task with entropy. Results were aggregated over 3 training runs. Best results for feature are shown in **bold**.

any kind of input. That is, the number of dimensions of the input data. Grams of bytes for



Confusion Matrix of Random Forest

Actual	0	1	2	3	4	5	6	7
0	285	3	0	7	1	2	1	1
1	6	218	1	4	2	1	0	0
2	2	0	169	1	0	0	0	0
3	3	3	2	132	0	0	4	1
4	0	0	0	0	149	0	0	0
5	3	1	1	0	0	195	0	0
6	5	5	0	2	0	0	149	0
7	2	1	0	0	1	0	1	283

Predicted

(a) Random Forest with entropy

Confusion Matrix of K-NN

Actual	0	1	2	3	4	5	6	7
0	264	5	1	12	2	4	10	2
1	6	198	6	5	2	4	4	7
2	3	2	159	4	0	4	0	0
3	12	3	2	119	0	1	7	1
4	0	0	0	0	149	0	0	0
5	2	2	1	0	0	193	1	1
6	9	5	0	2	0	0	144	1
7	2	3	5	1	1	0	1	275

Predicted

(b) K-NN with entropy

Confusion Matrix of SVM

Actual	0	1	2	3	4	5	6	7
0	235	9	2	11	3	9	30	1
1	17	184	8	2	2	6	8	5
2	10	0	122	1	0	11	1	27
3	16	4	4	70	0	2	26	23
4	0	0	0	0	145	0	0	4
5	7	1	0	0	0	189	0	3
6	5	16	0	3	0	1	128	8
7	11	7	4	1	1	0	1	263

Predicted

(c) SVM with entropy

Confusion Matrix of Gradient Boosting

Actual	0	1	2	3	4	5	6	7
0	278	6	0	10	1	2	3	0
1	3	222	1	4	1	1	0	0
2	2	1	168	1	0	0	0	0
3	4	4	3	129	0	0	4	1
4	0	0	0	0	149	0	0	0
5	3	1	1	0	0	195	0	0
6	1	6	0	3	0	0	151	0
7	1	1	1	1	1	0	1	282

Predicted

(d) Gradient boosting with entropy

Figure 6.24: Confusion Matrix of each algorithm with entropy feature (0: not malware, 1: malware). Confusion matrix from the last training.

example are 5000 different pieces of information that each become a different dimension in the space created for malware classification. So it is very difficult for excellent results to be achieved with so many dimensions. On the other hand, with the previous dataset, with the size of the binaries and the entropy, poor results had been achieved since it was very difficult that with only one dimension some noteworthy results could be achieved. By using the entropy extracted for each individual section of a malware instead, noteworthy results can finally be achieved. In fact, the number of dimensions is now only 24, which makes it easier for the SVM to be able to better fit the samples in the dataset into space.

For the other algorithms we get the usual situation, in fact Random Forest and Gradient Boosting are the two models that perform best, but without reaching perfect results, while K-NN still manages to achieve an F1 score of 0.9.

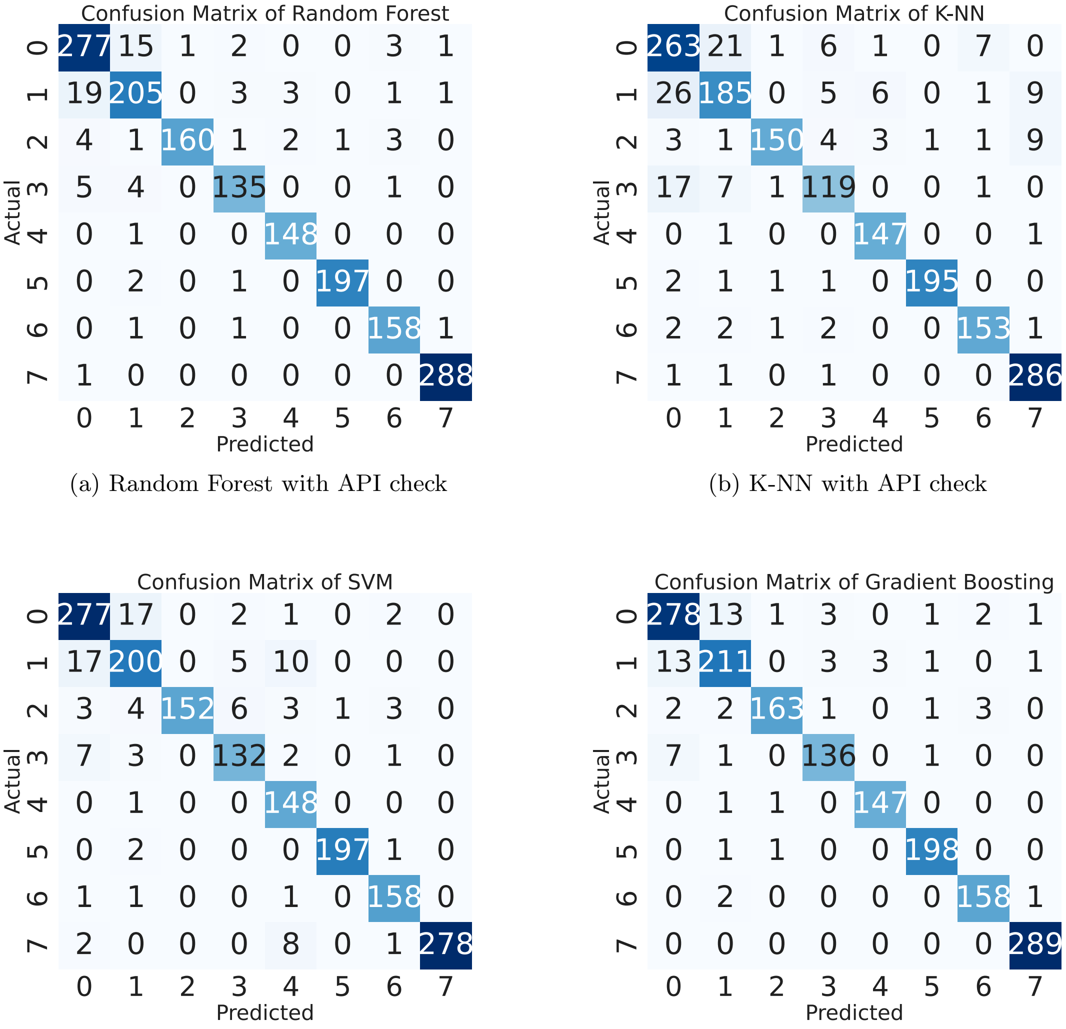
The confusion matrices in Figure 6.24 confirm the above, especially on the SVM.

6.4.3 Comments API check

As mentioned in the previous section, it is also worthwhile here to make a comparison between the F1 scores obtained with the same feature in the previous dataset (in table 6.2) and the F1

API Check	RF	K-NN	SVM	GB
Accuracy	0.9573 ± 0.0047	0.9146 ± 0.0046	0.9370 ± 0.0073	$\mathbf{0.9605 \pm 0.0021}$
Macro-Precision	0.9606 ± 0.0049	0.9212 ± 0.0058	0.9374 ± 0.0066	$\mathbf{0.9619 \pm 0.0005}$
Weighted-Precision	0.9578 ± 0.0048	0.9165 ± 0.0056	0.9404 ± 0.0060	$\mathbf{0.9606 \pm 0.0021}$
Marco-Recall	0.9577 ± 0.0039	0.9119 ± 0.0038	0.9378 ± 0.0086	$\mathbf{0.9623 \pm 0.0034}$
Weighted-Recall	0.9573 ± 0.0047	0.9146 ± 0.0046	0.9370 ± 0.0073	$\mathbf{0.9605 \pm 0.0021}$
Macro-F1 Score	0.9588 ± 0.0044	0.9152 ± 0.0034	0.9359 ± 0.0084	$\mathbf{0.9621 \pm 0.0019}$
Weighted-F1 Score	0.9573 ± 0.0047	0.9141 ± 0.0048	0.9373 ± 0.0073	$\mathbf{0.9605 \pm 0.0021}$
Micro-AUC ROC	0.9966 ± 0.0011	0.9817 ± 0.0028	0.9961 ± 0.0010	$\mathbf{0.9982 \pm 0.0005}$
Marco-AUC ROC	0.9957 ± 0.0013	0.9813 ± 0.0024	0.9944 ± 0.0016	$\mathbf{0.9976 \pm 0.0003}$
Weighted- AUC ROC	0.9958 ± 0.0012	0.9808 ± 0.0031	0.9945 ± 0.0013	$\mathbf{0.9975 \pm 0.0003}$

Table 6.15: Results of the different models (RF: Random Forest, K-NN: K-Nearest Neighbors, SVM: Support Vector Machine, GB: Gradient Boosting) in the malware classification task with API check. Results were aggregated over 3 training runs. Best results for feature are shown in **bold**.



(a) Random Forest with API check

(b) K-NN with API check

(c) SVM with API check

(d) Gradient boosting with API check

Figure 6.25: Confusion Matrix of each algorithm with API check feature (0: not malware, 1: malware). Confusion matrix from the last training.

scores obtained with ST-WinMal (table 6.15). It can be seen that the situation has improved

All features	RF	K-NN	SVM	GB
Accuracy	0.9759 ± 0.0050	0.8644 ± 0.0082	0.9178 ± 0.0076	**0.9873 ± 0.0016**
Macro-Precision	0.9766 ± 0.0055	0.8739 ± 0.0073	0.9309 ± 0.0067	**0.9866 ± 0.0022**
Weighted-Precision	0.9762 ± 0.0047	0.8656 ± 0.0080	0.9248 ± 0.0068	**0.9873 ± 0.0015**
Marco-Recall	0.9754 ± 0.0056	0.8625 ± 0.0092	0.9162 ± 0.0090	**0.9868 ± 0.0021**
Weighted-Recall	0.9759 ± 0.0050	0.8644 ± 0.0082	0.9178 ± 0.0076	**0.9873 ± 0.0016**
Macro-F1 Score	0.9759 ± 0.0057	0.8666 ± 0.0085	0.9212 ± 0.0080	**0.9866 ± 0.0021**
Weighted-F1 Score	0.9760 ± 0.0049	0.8636 ± 0.0083	0.9187 ± 0.0073	**0.9873 ± 0.0016**
Micro-AUC ROC	0.9987 ± 0.0007	0.9658 ± 0.0072	0.9959 ± 0.0010	**0.9996 ± 0.0003**
Marco-AUC ROC	0.9984 ± 0.0006	0.9629 ± 0.0069	0.9951 ± 0.0010	**0.9994 ± 0.0004**
Weighted- AUC ROC	0.9984 ± 0.0006	0.9629 ± 0.0072	0.9947 ± 0.0010	**0.9995 ± 0.0003**

Table 6.16: Results of the different models (RF: Random Forest, K-NN: K-Nearest Neighbors, SVM: Support Vector Machine, GB: Gradient Boosting) in the malware classification task with all features. Results were aggregated over 3 training runs. Best results for feature are shown in **bold**.

a lot here as well, in fact, all algorithms exceed 0.9 as F1 score while in the previous dataset they never exceeded 0.7. Definitely the motivation is the extraction method. In the Microsoft dataset, the APIs were extracted from the disassembled when called via the call function, while in the ST-WinMal dataset, the APIs were extracted from the executable header. The headers were removed from the malware in the Microsoft dataset to render the malicious files harmless. For this reason, it can be ascertained that in addition to evaluating dataset, input, and algorithm, it would also be appropriate to evaluate the extraction methods. Entropy and API are the clear demonstration of the above.

Gradient Boosting and Random Forest achieve the best results, but unlike before, K-NN and SVM are not that far off, and in fact can be considered good algorithms to use for classifying malware with the following characteristic.

Confusion matrices confirm the above, showing the goodness of this feature in the field of malware classification.

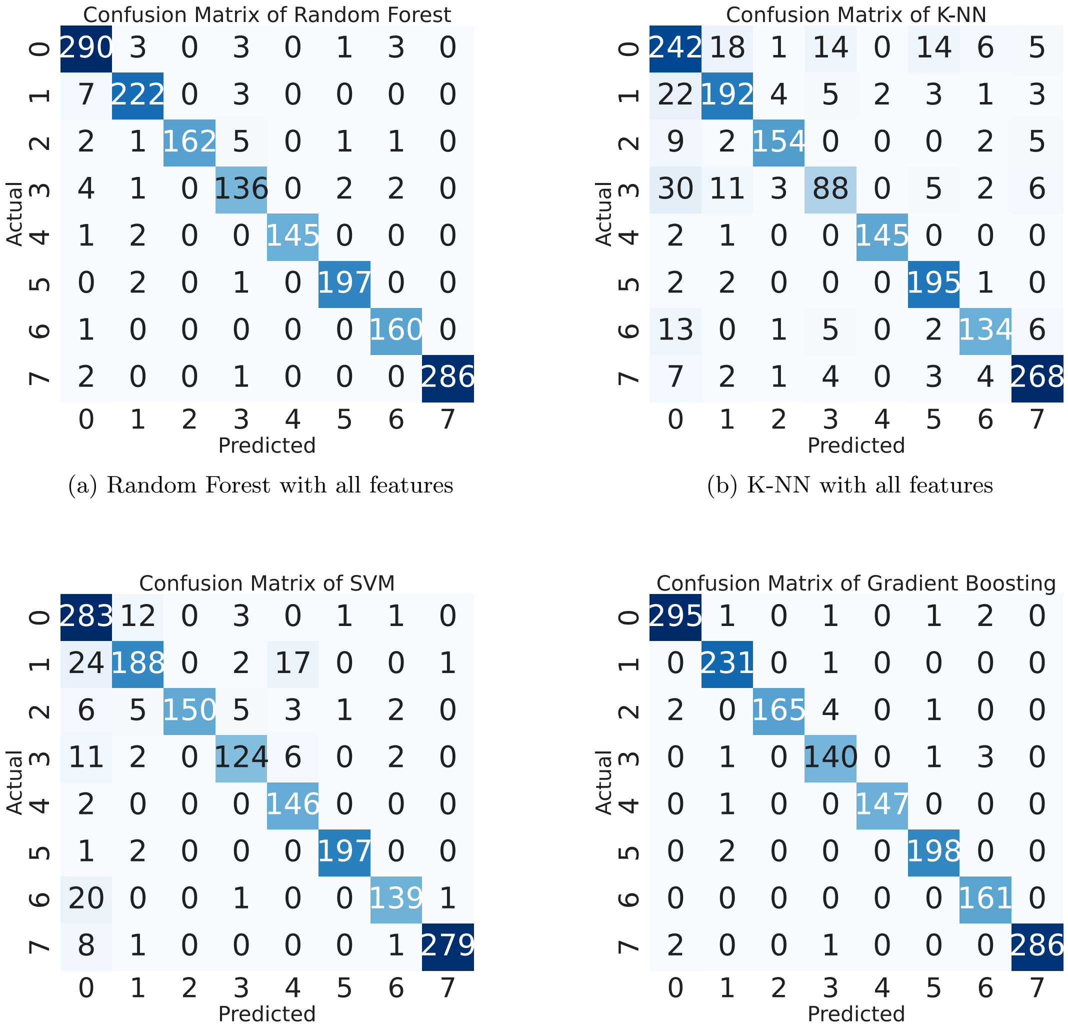
6.4.4 Comments All Features

Experiments with all the characteristics used as input together did not lead to the desired results. SVM is the only algorithm worth mentioning, in fact it achieved the same results as with entropy but using all other dimensions. In the table 6.16, it is clear that Gradient Boosting is again the best algorithm, along with Random Forest. Confusion matrices (in Figure 6.26) can confirm the goodness of this experiment but also show how no data excel and reach a 99% accuracy, a target that was thought to be easily achieved. The reasons for this have not been found, so it can only be assumed that the dataset cannot lead to the same results as the Microsoft dataset.

One might also think that it is also due to the families chosen, in this case we are in fact only with two classes of malware, namely Trojans and Ransomware, whereas Microsoft had no less than five different classes: Worm, Adware, Backdoor, Trojan and TrojanDownloader. Even if there are several families in a class, it is very likely that there are many more common features than with different classes, as the purpose is much more similar between different families of the same class than between different families of different classes.

6.4.5 Summary

With this analysis, one can confirm what has already been said to exist with the previous dataset, namely that there are better algorithms and features for this type of task, but not as clearly as seen with the Microsoft dataset. SVM and K-NN do not carry the results of the other algorithms with almost all features, they manage to perform quite adequately. I am convinced that with a little more work it can easily reach the level of Random Forest and Gradient Boosting.



Confusion Matrix of Random Forest

Actual \ Predicted	0	1	2	3	4	5	6	7
0	290	3	0	3	0	1	3	0
1	7	222	0	3	0	0	0	0
2	2	1	162	5	0	1	1	0
3	4	1	0	136	0	2	2	0
4	1	2	0	0	145	0	0	0
5	0	2	0	1	0	197	0	0
6	1	0	0	0	0	0	160	0
7	2	0	0	1	0	0	0	286

(a) Random Forest with all features

Confusion Matrix of K-NN

Actual \ Predicted	0	1	2	3	4	5	6	7
0	242	18	1	14	0	14	6	5
1	22	192	4	5	2	3	1	3
2	9	2	154	0	0	0	2	5
3	30	11	3	88	0	5	2	6
4	2	1	0	0	145	0	0	0
5	2	2	0	0	0	195	1	0
6	13	0	1	5	0	2	134	6
7	7	2	1	4	0	3	4	268

(b) K-NN with all features

Confusion Matrix of SVM

Actual \ Predicted	0	1	2	3	4	5	6	7
0	283	12	0	3	0	1	1	0
1	24	188	0	2	17	0	0	1
2	6	5	150	5	3	1	2	0
3	11	2	0	124	6	0	2	0
4	2	0	0	0	146	0	0	0
5	1	2	0	0	0	197	0	0
6	20	0	0	1	0	0	139	1
7	8	1	0	0	0	0	1	279

(c) SVM with all features

Confusion Matrix of Gradient Boosting

Actual \ Predicted	0	1	2	3	4	5	6	7
0	295	1	0	1	0	1	2	0
1	0	231	0	1	0	0	0	0
2	2	0	165	4	0	1	0	0
3	0	1	0	140	0	1	3	0
4	0	1	0	0	147	0	0	0
5	0	2	0	0	0	198	0	0
6	0	0	0	0	0	0	161	0
7	2	0	0	1	0	0	0	286

(d) Gradient boosting with all features

Figure 6.26: Confusion Matrix of each algorithm with all features (0: not malware, 1: malware). Confusion matrix from the last training.

On the latter two algorithms there is not much to say, they are in fact very precise, they tend to have a high recall, which is very good, and a very good precision, another factor in their favour.

The F1 score sums up very well the excellence of these two algorithms for this research. To summarise, we can say that all the available features led to very good results with Random Forest and Gradient Boosting, and fair results with the remaining 2 algorithms.

6.5 Summary

After this careful analysis of all the results obtained, it is good to summarise the concepts learned. First of all, we have seen how changing the methods of feature extraction can have a very important influence on the results that can be obtained with the classification of samples (whether binary or multi-class). The entropy and presence control of API function calls are tangible proof of this. The results obtained with the ST-WinMal dataset lead us to conclude that they are both two excellent solutions to propose for this type of work. On the contrary, with the tests carried out on the first dataset, this statement is no longer valid. For any future research, it is therefore important to check how the data was extracted in the dataset and how this may affect all the experiments carried out during the final part of the work.

Algorithm	*Feature(s)*	*Weighted-F1 Score*
Microsoft Malware Classification Challenge Dataset		
Gradient Boosting	All features	0.9979
Random Forest	All features	0.9962
Gradient Boosting	Bytes features	0.9959
ST-WinMal		
Gradient Boosting	All features	0.9873
Gradient Boosting	Bytes 3-gram	0.9850
Random Forest	Bytes 3-gram	0.9769

Table 6.17: Summary best results

A second concept learnt is the confirmation of the success of the grams of bytes and opcodes. As already mentioned during the study of the state of the art of this subject, this feature that can be extracted from binary and disassembled files has wide success in the field of malware identification, both with machine learning techniques and without. Even in the presence of malware with obfuscation and/or encryption techniques, we find ourselves with a high success rate, which is why it is good to keep these features, perhaps supplementing them with other extractable information from malware that has little size. The motivation behind this success is probably their simplicity. Even if obfuscation techniques are used, it is impossible for the entire code to be rendered unreadable, in which case there would still be the possibility of dynamic feature extraction (slower but more effective). If the code contains some 'real' parts of the malware, a classifier will be able to realise that that part is more relevant than others in order to be able to go and create a general rule. Obviously, having a large number of samples helps to obtain excellent results.

Finally, the last very important notion learnt during this analysis is the ability of the Random Forest and Gradient Boosting algorithms to adapt to each feature and each dataset. They turned out to be the best algorithms, without any kind of doubt. They are certainly the algorithms that also have the highest growth rate and can achieve an accuracy close to 100 per cent. The other two paths, namely K-NN and SVM, led to decent results with some features but bad with others. The table 6.17 summarises which were the three best patterns found in the field of malware classificationfor each dataset.

Limitations of static API extraction

The API never managed to bring the desired results. In the background, this feature was described as very positive in the field of malware identification and classification, but after reading the results of this analysis, it may seem that this is not true. Actually, the reasons are very simple, in fact we are dealing with the phenomenon of name mangling. It is a technique used to solve various problems caused by the need to uniquely identify different programming entities in modern programming languages. Compilers in fact need to have their own methods to identify these entities. An example is the overloading of two C++ functions that can be called `void fun(int)` and `void fun(double)` For the programmer, these functions have the same name, but for the compiler, they must have two different identifiers. Each compiler has different ways, for example GCC 8 would create two identifiers such as `_Z3funi` and `_Z3fund` where the final i and d would stand for int and double. For this reason, perhaps two very similar malware written in two different languages would produce totally different results. One solution would be to match the names given by the compiler to those given by the programmer, but this is almost impossible, if not very difficult to do for every compiler in existence. Moreover, some compilers have never provided details of their inner workings.

Reason for SVM's failure

To better understand why the SVM sometimes misclassified malware, one must study the various families in the dataset in more detail. Below is a description of the malware families in Microsoft's dataset:

- The **Ramnit** family has numerous variants, which can be classified individually as trojans, viruses or worms. Microsoft's dataset contains the category of worms. This family of malware steals sensitive information, such as usernames and passwords of bank accounts. It can also give a malicious hacker remote access, so that he can control the victim's PC and stop the security software installed on the affected device [50].
- Malware belonging to the **Lollipop** family displays advertisements when the victim surfs the web. These programs redirect search engine results, monitor what is happening on the PC and send the collected information to the attacker. This malware can be downloaded from the web or bounded with some third-party software installation programs [51].
- **Kelihos** is a family of malware that distributes spam e-mail messages that may contain web links to installers of itself. It may also contain executables that can control the victim's PC to change configuration data and download other arbitrary files [52]. Version 3 added the possibility of creating botnets with the computers of victims affected by the malware.
- The **Vundo** family is part of the Trojan category, which allows the attacker to perform any number of actions of his choice on the victim's PC [53].
- Malware of the **Simda** family are Trojans designed to steal users' passwords, in order to give a malicious hacker backdoor access, so that he can control your PC and perform malicious actions [54].
- **Tracur** is a family of trojans that can redirect your web searches. The attacker can thus make money by placing fraudulent advertisements [55].
- Malware of the **Obfuscator.ACY** family are obfuscated, i.e. they try to hide their purpose from the antivirus so that it cannot identify them. The malware itself can have quite different purposes, depending on the attacker's wishes [56].
- **Gatak**: This trojan gathers information about your PC and sends it to a hacker. It can arrive on your PC as part of a key generator application, or by appearing to be an update for a legitimate application[57].

After this brief description of each family, one can better understand some of the situations we encountered earlier. Looking at the SVM confusion matrix when all the features extracted from the *.asm* files were used (Figure 6.13c), we can understand why many malware fall into the category of obfuscated malware. As they are a mixture of many malware with different behaviours, it is very easy for other malware to be confused for this category. There is therefore an explanation behind some of the SVM failures. Another similar situation is found in the SVM used with opcode counters. Certainly the assembly functions will contain similar functions, in the same order, as they will be the micro-functions of the same API called by similar malware or for the same purpose.

If we were to analyse the ST-WinMal dataset, I am sure we would come into the same case. Certainly less than the Microsoft dataset given the excellent results anyway. This can therefore be considered the main reason for the failure behind SVM.

Chapter 7

Conclusions

The aim of this thesis was to create a tool that could identify and classify malware using machine learning algorithms. In the case of malware identification, the aim was to have a high recall and low false-positive rate, while for classification, we determined that the F1 score was the best metric to take into account and that it managed to be as high as possible. The aim was to not only identify malware from a single dataset, but also to be able to recognise different families and thus be usable with multiple datasets.

To achieve this, an attempt was made to use four different algorithms, namely Random Forest, K-Nearest Neighbours, Support Vector Machine and Gradient Boosting, with different characteristics extracted from malware. The first step was therefore to decide what information could be extracted from the selected datasets, the Microsoft Malware Classification Challenge dataset and St-WinMal, and then to create a tool that was able to automate the extraction for a large number of malware. After that, the second step was to test the aforementioned algorithms to see which ones worked best with the different features available in order to understand which tools had the most potential in this field.

In the results shown in Chapter 6, it can be seen that in the field of identification, the algorithms tend to perform very well, effectively understanding which samples were malware and which were not, and more than correctly labelling as many samples as possible. The Random Forest and Gradient Boosting were confirmed as the two algorithms with the highest accuracy, precision and recall, the former algorithm being confirmed as the fastest of the two in the training phase, slightly lacking in precision.During the classification tests, on the other hand, the difference between the algorithms was more pronounced, with an SVM unable to replicate the results obtained during the malware identification tests and a K-NN with discrete results. The Random Forest and Gradient Boosting also achieved excellent results in this field, reaching high precision especially with the gram of bytes and opcode, and with the assembly function counter.

Compared to traditional malware analysis techniques, machine learning provides a fast and accurate classifier. It does not need to understand the code and if placed in a context where information is extracted in an automated manner, it can perform training much faster. This approach cannot give an insight into the detailed behaviour of malware, as it does not output any kind of additional information other than the label of the malware family to which the sample belongs. To perform a deeper analysis, more features are required together, as shown in Chapter 6.

7.1 Future Works

The models described and tested in this thesis have the potential to become very useful in the field of computer security in the near future. They can be used to identify and classify malware in an infected system, without envying current systems. The current implementation of the models with Random Forest and Gradient Boosting achieved excellent results in the field of malware detection and malware classification by means of static features, also considering the number of samples available on which they were trained.

However, there are also limitations, especially with the use of certain features with which all 4 models failed to learn enough. As specified in section 6.5, there are problems with the extraction of APIs due to the Name Mangling phenomenon. Furthermore, it is not guaranteed that malware can be correctly classified among the various families, because as described in the same section above, it is possible that several families with the same purpose use the same features in order to achieve the intended goal. Listed below are suggestions that can be taken against them to overcome the limitations described above:

- The problem arising from Name Mangling arises from the existence of a large number of C/C++ compilers, leading to many variants of the same functions, which perhaps only change the type of parameters or the type of return value. In reality, however, the compilers used are not so many as some have been developed but never used on a large scale. An attempt could therefore be made to develop a tool that would be able to rename all the APIs created with Name Mangling according to a convention decided a priori.
- The datasets used were very useful in the development of this thesis, but unfortunately they have one major limitation, namely the number of samples is not large enough. In order to be able to carry out more precise tests, it is necessary to increase the number of samples, while still being able to keep the samples balanced against each other as in the ST-WinMal dataset.
- The technique used to perform dimensionality reduction is an improvable one, in fact there are techniques created specifically for this purpose, such as PCA (Principal Component Analysis) and LDA (Linear Discriminant Analysis). These techniques could open up many avenues for new dimensionality reduction methods. Other customised techniques can be created by going deeper into the individual malware families in the dataset in order to understand which APIs are the most significant, opcode grams, etc.
- Another suggestion would be to study a method for dynamic extraction of malware, or to find a tool that can be automated through some programming language, to try to create a machine learning system that employs the models realised in this thesis, but uses dynamically extracted features.
- Finally, a somewhat heavier but potentially interesting piece of work would be to adapt the code to work with malware datasets that target operating systems other than Windows, such as Linux or Android.

Appendix A

User Manual

The program is independent of the operating system on which it runs, you need to have python 3.10 and the following dependencies installed:

- NumPy 1.24.2
- SciPy 1.10.0
- scikit-learn 1.2.2
- matplotlib 3.7.0
- seaborn 0.12.2
- pandas 1.5.3

To install all dependencies, use the command **pip install <library>**. All libraries have their own official page with documentation included; should you need more libraries, they will automatically be downloaded using the command described above. Packages are also available on conda.

A.1 Instructions

The matrices with the extracted features have already been created and placed in the *.npz* files in the pre-extract folder. To run the program, clone the project from the following link `https://github.com/saci98/ML_Malware_Classification` with the following command:

git clone https://github.com/saci98/ML_Malware_Classification.git

To run the program, the following command must be executed from the project folder:

python main.py [-h] --dataset {benign, malign, ST-WinMal-benign, ST-WinMal-malign} --features {binary_size, 3_gram_bytes, 4_gram_opcode, 4_gram_API, entropy,opcode_counter, API_check} [{binary_size, 3_gram_bytes, 4_gram_opcode, 4_gram_API, entropy,opcode_counter, API_check} ...]

options:

- -h, --help show this help message and exit
- --dataset {benign, malign, ST-WinMal-benign, ST-WinMal-malign} benign for microsoft dataset with benign file or malign for microsoft dataset without benign file, ST-WinMal-benign for second dataset with benign file or ST-WinMal-malign for second dataset with malign file

- --features {binary_size, 3_gram_bytes, 4_gram_opcode, 4_gram_API, entropy, opcode_counter, API_check} [{binary_size, 3_gram_bytes, 4_gram_opcode, 4_gram_API, entropy, opcode_counter, API_check} ...] insert feature(s) to use

Execution time varies from feature to feature, it is recommended to run on a programming environment with high performance capabilities. To make the execution lighter, individual algorithms can be tried out; to do this, the following instructions on main.py must be followed:

- If you only wish to use Random Forest you must comment on lines 382-398 and lines 407-415.
- If you only want to use K-NN you must comment on lines 378-380, lines 386-398, lines 404-406 and lines 410-415.
- If you only wish to use SVM you must comment on lines 378-384, lines 393-398, lines 404-409 and lines 413-415.
- If you only wish to use Gradient Boosting, comment on lines 378-391 and lines 404-412.

The output will show the numerical results described in Chapter 6 and also save the confusion matrices in the results folder. The file will be saved with the following name:

Confusion Matrix of <algorithm>_<dataset>_<features>.svg

ROC_Curve_<dataset>_<features>.svg

The execution time without feature extraction can vary from seconds up to hours depending on the selected feature, in case extraction is performed this time could be longer than a day.

Appendix B

Programmer Manual

The code consists of a main and several files that provide the functions needed to extract features and model them for training. In addition, there are several folders with different functions:

- **data**: Folder where there are datasets (not reported on GitHub) and csvs to allow sample-label association.
- **pre-extract**: Contains pre-extracted arrays in files *.npz*
- **results**: Contains images of confusion matrices and ROC curves.
- **Utils**: It contains all the files *.py* created to support the main.

In the Utils folder there are several files with different functions. The functions have been divided into the files in the following way:

- *CSVReader.py*: It contains functions for reading and saving csv files.
- *Extract_feature.py*: It contains functions that work with the dataset to extract selected features from it.
- *Integrate_features.py*: It contains the functions for integrating features with each other to allow several features to be used together.
- *ReadFile.py*: It contains the functions for reading the files in the dataset.

There are several functions in each file, all adequately commented so that they can be used for future work. Below are the functions divided by file and their description.

CSVReader.py

Class CSVReader: Manage the saving and loading of a *.csv* file.

- __init__(self, name_file):
  create object CSVReader
  :param name_file: Name of file CSV.
- save(self, data):
  Save a DataFrame as CSV file
  :param data: Dataframe to save in file CSV.
- load(self):
  Load into a DataFrame the contents of a CSV
  :return: DataFrame with the contents of a CSV

Extract_features.py

- extract_binary_sizes(id):
  Extracts the size of a binary file passed the id
  :param id: binary file id
  :return: binary file size.

- extract_ngrams(ids, seq_reader, n):
  Extracts the n-grams of a specific object, which may be bytes, opcodes or APIs
  :param ids: id of files from which to extract n-grams
  :param seq_reader: function to be used to extract the sequence of the specified object (bytes, opcode or API)
  :param n: gram size
  :return: The CountVectorizer object with a matrix in which the occurrences of n-grams and found n-grams are counted.

- extract_bytes_sequence(id):
  Extracts the sequence of bytes from a specified file
  :param id: file id
  :return: Sequence of file bytes.

- extract_opcode_sequence(id):
  Extracts the sequence of opcode from a specified file
  :param id: file id
  :return: Sequence of file opcodes.

- count_asm_opcode(id):
  Counts the occurrences of each assembly function within a specified asm file
  :param id: file id
  :return: array with the occurrences of each opcode.

- extract_syscall(id):
  Inserts in the list of all syscalls those found in a specified file
  :param id: file id.

- save_syscall():
  Saves the syscalls entered up to that point in the global list within the syscall.txt file

- load_syscall():
  Load the syscalls into the syscall.txt file in the global list.

- check_syscall(id):
  Checks for the presence of syscalls within a specified asm file
  :param id: file id
  :return: Array with a 0 in the columns of syscalls not found and 1 in those found.

- extract_syscall_sequence(id):
  Extracts the syscall sequence within an asm file
  :param id: file id
  :return: syscall sequence of the file.

- extract_entropy(id): Calculates the entropy of a binary file
  :param id: file id
  :return: file entropy.

- extract_apis(id):
  Extracts APIs from a sample of the ST-WinMal dataset
  :param id: file id.

- save_apis():
  Save the API in the global list within the apis.txt file.

- load_apis():
  Load the API within the apis.txt file into the global list.
- check_apis(id):
  Checks for the presence of apis within a sample of the ST-WinMal dataset
  :param id: file id
  :return: Array with a 0 in the columns of API not found and 1 in those found.
- extract_entropy_section(id):
  Extracts entropy from the .entropy files of the ST-WinMal dataset
  :param id: file id
  :return: Array with entropy for each section.
- extract_bytes_microsoft(id):
  Extracts the byte sequence from a sample of the ST-WinMal dataset
  :param id: file id
  :return: Sequence of bytes.

Integrate_features.py

- integrated_features(features, dataset):
  Integrates several features together
  :param features: Deature to be integrated
  :param dataset: Dataset used
  :return: DataFrame with several features together.

ReadFile.py

class ReadFile: Read different kinds of content from a sample file.

- bytes_reader(id):
  Reads the bytes within a Microsoft dataset .bytes file
  :param id: file id
  :return: Byte sequence.
- asm(id):
  Returns the entire .asm file in string format
  :param id: file id
  :return: .asm files in string format.
- asm_lines(id):
  It reads an .asm file from the Microsoft dataset and returns the contents in the form of a list of strings, each containing a line from the asm file
  :param id: file id
  :return: List with lines in the file.
- apis(id):
  Reads the api functions present in a sample of the ST-WinMal dataset
  :param id: file id
  :return: List of api functions found.
- entropy(id):
  Returns a list of containing each element the name of the section and its entropy
  :param id: file id
  :return: Entropy per section.
- bytes_microsoft(id):
  Reads the bytes within a ST-WinMal dataset .bytes file
  :param id: file id
  :return: Byte sequence.

main.py

It contains the main body of code that creates the tool for identifying and classifying malware. The code contains an initial part in which the dataset is loaded, followed by feature extraction and finally the tests to obtain the results.

Bibliography

[1] Statista - Number of internet and social media users worldwide as of January 2023, https://www.statista.com/statistics/617136/digital-population-worldwide/

[2] Cisco definition of malware, https://www.cisco.com/c/en/us/products/security/advanced-malware-protection/what-is-malware.html

[3] A. Ray and A. Nath, "Introduction to Malware and Malware Analysis: A brief overview", International Journal of Advance Research in Computer Science and Management Studies, vol. 4, October 2016, pp. 22–30

[4] What is malware? AVG, https://www.avg.com/en/signal/what-is-malware

[5] R. Sharp, "An introduction to malware", Spring, 2017

[6] A. P. Namanya, A. Cullen, I. U. Awan, and J. P. Disso, "The world of malware: An overview", 2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud), Barcelona (Spain), August 6-8, 2018, pp. 420–427, DOI 10.1109/FiCloud.2018.00067

[7] N. Bhojani, "Malware analysis", Ethical Hacking, Ahmedabad (India), October, 2014, pp. 1–5, DOI 10.13140/2.1.4750.6889

[8] S. Sibi Chakkaravarthy, D. Sangeetha, and V. Vaidehi, "A Survey on malware analysis and mitigation techniques", Computer Science Review, vol. 32, May 2019, pp. 1–23, DOI 10.1016/j.cosrev.2019.01.002

[9] E. Gandotra, D. Bansal, and S. Sofat, "Malware Analysis and Classification: A Survey", Journal of Information Security, vol. 5, January 2014, pp. 56–64, DOI 10.4236/jis.2014.52006

[10] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools", ACM computing surveys (CSUR), vol. 44, February 2008, pp. 1–42, DOI 10.1145/2089125.2089126

[11] J. Singh and J. Singh, "Challenges of Malware Analysis: Obfuscation Techniques", International Journal of Information Security Science, vol. 7, September 2018, pp. 100–110

[12] Y. Gao, Z. Lu, and Y. Luo, "Survey on malware anti-analysis", Fifth International Conference on Intelligent Control and Information Processing, Dalian (China), August 18-20, 2014, pp. 270–275, DOI 10.1109/ICICIP.2014.7010353

[13] I. You and K. Yim, "Malware obfuscation techniques: A brief survey", 2010 International conference on broadband, wireless computing, communication and applications, Fukuoka (Japan), November 04-06, 2010, pp. 297–300, DOI 10.1109/BWCCA.2010.85

[14] A. Sharma and S. K. Sahay, "Evolution and Detection of Polymorphic and Metamorphic Malwares: A Survey", International Journal of Computer Applications, vol. 90, March 2014, pp. 7–11, DOI 10.5120/15544-4098

[15] 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

[16] K. Alzarooni, "Malware variant detection". PhD thesis, UCL (University College London), 2012

[17] P. Vinod, R. Jaipur, V. Laxmi, and M. Gaur, "Survey on malware detection methods", Proceedings of the 3rd Hackers' Workshop on computer and internet security (IITKHACK'09), Kanpur (India), March 17-19, 2009, pp. 74–79

[18] W. Wong and M. Stamp, "Hunting for Metamorphic Engines", Journal in Computer Virology, vol. 2, November 2006, pp. 211–229, DOI 10.1007/s11416-006-0028-7

[19] M. Sikorski and A. Honig, "Practical malware analysis: The hands-on guide to dissecting malicious software", No Starch Press, 2012

[20] A. Pektaş and T. Acarman, "A Dynamic Malware Analyzer Against Virtual Machine Aware Malicious Software", Security and Communication Networks, vol. 7, December 2014, pp. 2245–2257, DOI 10.1002/sec.931

[21] ScoopyNG, The VMware detection tool, `https://www.trapkit.de/tools/scoopyng/index.html`

[22] T. Apostolopoulos, V. Katos, K.-K. R. Choo, and C. Patsakis, "Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks", Future Generation Computer Systems, vol. 116, March 2021, pp. 393–405, DOI 10.1016/j.future.2020.11.004

[23] M. Goyal and R. Kumar, "A Survey on Malware Classification Using Machine Learning and Deep Learning", International Journal of Computer Networks and Applications, vol. 8, December 2021, pp. 758–775, DOI 10.22247/ijcna/2021/210724

[24] 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, March 2020, p. 102526, DOI 10.1016/j.jnca.2019.102526

[25] Software Engineering — Control Flaw Graph (CFG), `https://www.geeksforgeeks.org/software-engineering-control-flow-graph-cfg/`

[26] A. Abusitta, M. Q. Li, and B. C. Fung, "Malware classification and composition analysis: A survey of recent developments", Journal of Information Security and Applications, vol. 59, June 2021, p. 102828, DOI 10.1016/j.jisa.2021.102828

[27] Ö. A. Aslan and R. Samet, "A Comprehensive Review on Malware Detection Approaches", IEEE Access, vol. 8, January 2020, pp. 6249–6271, DOI 10.1109/ACCESS.2019.2963724

[28] Y. Supriya, G. Kumar, D. Sowjanya, D. Yadav, and D. L. Kameshwari, "Malware detection techniques: A survey", 2020 Sixth International Conference on Parallel, Distributed and Grid Computing (PDGC), Waknaghat (India), November 06-08, 2020, pp. 25–30, DOI 10.1109/PDGC50313.2020.9315764

[29] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh, "A survey on heuristic malware detection techniques", The 5th Conference on Information and Knowledge Technology, May 28-30, 2013, pp. 113–120, DOI 10.1109/IKT.2013.6620049

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

[31] VirusShare about, `https://virusshare.com/about`

[32] J. Singh and J. Singh, "A survey on machine learning-based malware detection in executable files", Journal of Systems Architecture, vol. 112, January 2021, p. 101861, DOI 10.1016/j.sysarc.2020.101861

[33] R. Harang and E. M. Rudd, "Sorel-20m: A large scale benchmark dataset for malicious pe detection", 2020, DOI 10.48550/ARXIV.2012.07634

[34] VirusTotal intelligence, `https://www.virustotal.com/gui/intelligence-overview`

[35] IDA disassembler, `https://www.hex-rays.com/ida-pro/ida-disassembler/`

[36] Microsoft malware dataset download page, `https://www.kaggle.com/competitions/malware-classification/data`

[37] MalImg dataset download page, `https://vision.ece.ucsb.edu/research/signal-processing-malware-analysis`

[38] W.-C. Lin and Y.-R. Yeh, "Efficient Malware Classification by Binary Sequences with One-Dimensional Convolutional Neural Networks", Mathematics, vol. 10, February 2022, p. 608, DOI 10.3390/math10040608

[39] DikeDataset page, `https://github.com/iosifache/DikeDataset`

[40] A. Sindoni, "Toward a methodology for malware analysis and characterization for machine learning application", Master's thesis, Politecnico di Torino, 2023

[41] Definition of file entropy, IBM, `https://www.ibm.com/docs/en/qsip/7.4?topic=content-analyzing-files-embedded-malicious-activity`

[42] G. Xiao, J. Li, Y. Chen, and K. Li, "MalFCS: An effective malware classification framework with automated feature extraction based on deep convolutional neural networks", Journal of Parallel and Distributed Computing, vol. 141, July 2020, pp. 49–58, DOI 10.1016/j.jpdc.2020.03.012

[43] Computing resources provided by hpc@polito, `http://www.hpc.polito.it`

[44] 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, November 2011, pp. 2825–2830, DOI 10.5555/1953048.2078195

[45] Microsoft x86 instructions page, https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/x86-instructions

[46] PE Format Files, https://learn.microsoft.com/en-us/windows/win32/debug/pe-format

[47] A. Natekin and A. Knoll, “Gradient boosting machines, a tutorial”, Frontiers in neurorobotics, vol. 7, December 2013, pp. 21–21, DOI 10.3389/fnbot.2013.00021

[48] Matplotlib page, https://matplotlib.org/

[49] Seaborn page, https://seaborn.pydata.org/

[50] Win32/Ramnit, https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?name=win32%2Framnit

[51] Adware:Win32/Lollipop, https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Adware:Win32/Lollipop#:~:text=This%20adware%20program%20shows%20ads,your%20PC%20to%20a%20hacker.

[52] Backdoor:Win32/Kelihos.A, https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Backdoor%3AWin32%2FKelihos.A

[53] Trojan:Win32/Vundo.SA, https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Trojan:Win32/Vundo.SA

[54] Win32/Simda, https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Win32/Simda

[55] Win32/Tracur, https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?name=win32%2Ftracur

[56] VirTool:Win32/Obfuscator.ACY, https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=VirTool:Win32/Obfuscator.ACY

[57] Win32/Gatak, https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?name=win32/gatak