Journal of Network and Computer Applications 153 (2020) 102526

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Review

The rise of machine learning for detection and classification of malware: Research developments, trends and challenges

Daniel Gibert *, Carles Mateu, Jordi Planes

University of Lleida, Jaume II, 69, Lleida, Spain

ARTICLE INFO

Keywords:
Malware detection
Feature engineering
Machine learning
Deep learning
Multimodal learning

ABSTRACT

The struggle between security analysts and malware developers is a never-ending battle with the complexity of malware changing as quickly as innovation grows. Current state-of-the-art research focus on the development and application of machine learning techniques for malware detection due to its ability to keep pace with malware evolution. This survey aims at providing a systematic and detailed overview of machine learning techniques for malware detection and in particular, deep learning techniques. The main contributions of the paper are: (1) it provides a complete description of the methods and features in a traditional machine learning workflow for malware detection and classification, (2) it explores the challenges and limitations of traditional machine learning and (3) it analyzes recent trends and developments in the field with special emphasis on deep learning approaches. Furthermore, (4) it presents the research issues and unsolved challenges of the state-of-the-art techniques and (5) it discusses the new directions of research. The survey helps researchers to have an understanding of the malware detection field and of the new developments and directions of research explored by the scientific community to tackle the problem.

1. Introduction

A brief look at the history of malicious software reminds us that the presence of malware threats has been with us since the dawn of computing. The earliest documented virus appeared during the 1970s. It was known as the Creeper Worm and was an experimental self-replicating program that copied itself to remote systems and displayed the message: "I'm the creeper, catch me if you can". Later, in the early 80s, appeared Elk Cloner, a boot-sector virus that targeted Apply II computers. From these simple beginnings, a massive industry was born and, since then, the fight against malware has never stopped. By the looks of it, this fight turned out to be a never-ending and cyclical arms race: as security analysts and researchers improve their defenses, malware developers continue to innovate, find new infection vectors and enhance their obfuscation techniques. Malware threats continue to expand vertically (i.e. numbers and volumes) and horizontally (i.e. types and functionality) due to the opportunities provided by technological advances. Internet, social networks, smartphones, IoT devices and so on, make it possible for the creation of smart and sophisticated malware. In recent years, ransomware and cryptomining malware emerged as the most prolific types, with Cerber and Locky holding computers all over the globe for ransom while Cryptoloot used the victim's computing power to mine for crypto without their knowledge. Even though malware targeting computer systems still predominates in the ecosystem, mobile and IoT malware is on the rise. According to Symantec (Corporation, 2018), mobile malware variants increased by 54% in 2017 while IoT attacks had a 600% increase, with the Mirai botnet and its variants serving as the vehicle for some of the most potent DDoS attacks in history (Kolias et al., 2017).

To keep up with malware, security analysts and researchers need to constantly improve their cyber-defenses. One essential element is endpoint protection. Endpoint protection provides a suite of security programs including, but not limited to, firewall, URL filtering, email protection, anti-spam and sandboxing. Specifically, anti-malware software provides the last layer of defense. AV engines are responsible for preventing, detecting and removing malicious software installed on the endpoint device. Traditionally, AV solutions relied on signature-based and heuristic-based methods. A signature is an algorithm or hash that uniquely identifies a specific malware while heuristics are a set of rules determined by experts after analyzing the behavior of malware. However, both approaches require the malware to be analyzed prior to the definition of these rules and heuristics. The goal of malware anal-

* Corresponding author.
E-mail addresses: daniel.gibert@diei.udl.cat (D. Gibert), carlesm@diei.udl.cat (C. Mateu), jplanes@diei.udl.cat (J. Planes).

https://doi.org/10.1016/j.jnca.2019.102526
Received 1 July 2019; Received in revised form 29 November 2019; Accepted 26 December 2019
Available online 2 January 2020

ysis is to provide information about the characteristics, purpose and behavior of a given piece of software. There are two types of analysis: (1) static analysis and (2) dynamic analysis. On the one hand, static analysis involves examining an executable without execution. On the other hand, dynamic analysis involves examining the behavior of the executable by running it. Both types of analysis have their advantages and limitations and they complement each other. Static analysis is faster but, if malware is successfully concealed using code obfuscation techniques, it could evade detection. Contrarily, code obfuscation techniques and polymorphic malware hardly evades dynamic analysis as it monitors and analyzes the runtime execution of a program. Nevertheless, traditional malware detection and malware analysis are unable to keep pace with new attacks and variants. Organizations are facing the daunting challenge of dealing with millions of attacks a day. In addition, organizations are also experiencing a shortage of cybersecurity skills and talent (on Cybersecurity for the 44th Presidency et al., 2010). The identified issues present a unique opportunity for machine learning to significantly impact and change the cybersecurity landscape due to its ability to handle large volumes of data (Fraley et al., 2017).

During the last decade, machine learning has triggered a radical shift in many sectors, including cybersecurity. There is a general belief among cybersecurity experts that AI-powered antimalware tools will help detect modern malware attacks and improve scanning engines. Evidence of this belief is the number of studies published in the last few years on malware detection techniques that leverage machine learning. According to Google Scholar,[1] the number of research papers published in 2018 is 7720, a 95% increase with respect to 2015 and a 476% increase with respect to 2010. This increase in the number of studies is the result of various factors, including, but not limited to, the increase in public labeled feeds of malware, the increase in computational power as the same time as its reduction in price, and the evolution of the machine learning field, which achieved breakthrough success on a wide range of tasks such as computer vision and speech recognition. Traditional machine learning approaches can be categorized into two primary groups, static and dynamic approaches, depending on the type of analysis. The main difference between them is that static approaches extract features from the static analysis of malware, while dynamic approaches extract features from the dynamic analysis. A third group, defined as hybrid approaches, might be considered. Hybrid approaches combine aspects of both static and dynamic analysis. Furthermore, neural networks have outshone in learning features from raw inputs in various fields. Recent trends in machine learning for cybersecurity are replicating the success of neural networks in the malware domain. For instance, Raff et al. (2018a) and Krčál et al. (2018) proposed building a convolutional neural network to determine the maliciousness of PE executables from the raw bytes of the file itself. The motivation behind neural network approaches is to build detection systems that do not rely on the experts' knowledge of the domain to define discriminative features.

Given the growing impact of AI-powered tools to detect malware, a new literature review is needed considering the recent research studies and exploring the details of traditional static and dynamic approaches. There is some research discussing malware detection methods but we consider it is incomplete. (The reader is referred to Section 2). To complement the papers surveyed and mitigate some flaws in the literature, this paper presents a systematic review on traditional and state-of-the-art machine-learning-powered techniques for malware detection and classification, with special emphasis on the type of information (features) extracted from Portable Executable files. This paper provides the basic background in malware analysis, and a brief description of the process and tools to dissect malware. For a more complete description we refer the reader to Ligh et al. (2010); Sikorski and Honig (2012); Monnappa (2018). This review is intended to support security analysts, who may be interested in applying machine learning to automate part of the malware analysis process, to have a general understanding of the methods currently in use and of the new trends. This paper categorizes methods in three main groups: (i) static methods, (2) dynamic methods and (3) hybrid methods. Furthermore, it provides a detailed description of neural-based methods for detecting and classifying malware, categorized according to how the input is preprocessed before feeding the neural network, as well as a brief description of multimodal learning approaches. This paper closes by discussing the research issues and challenges faced by researchers in the field, including the availability of open and public benchmarks to evaluate the performance of methods, the problem of concept drift in the malware domain, incremental learning, adversarial learning, and the problem of class imbalance.

This survey is organized as follows. Section 2 provides a summary of the surveyed research in the literature. Section 3 describes the background of malware analysis. Section 4 provides a systematic description of static and dynamic methods for malware detection and outlines the most discriminant features for the task at hand. Section 5 presents a detailed overview of the neural-based methods. Section 6 introduces the multimodal and hybrid approaches. In Section 7, a comprehensive analysis of new challenges and the issues of malware detection are discussed. Finally, Section 8 summarizes the concluding remarks of this survey.

2. Related work

This section provides a summary of the surveyed research in the literature and discusses some of its defects. Table 1 sums up the main contributions of the surveys in the literature. We follow by presenting a brief description for each survey, and their flaws that we try to mitigate in our work.

Shabtai et al. (2009) provide a taxonomy for malware detection using machine learning algorithms by reporting some feature types and feature selection techniques used in the literature. They mainly focus on the feature selection techniques (Gain ratio, Fisher score, document frequency, and hierarchical feature selection) and classification algorithms (Artificial Neural Networks, Bayesian Networks, Naïve Bayes, K-Nearest Neighbor, etc). In addition, they review how ensemble algorithms can be used to combine a set of classifiers. Bazrafshan et al. (2013) identify three main methods for detecting malicious software: (1) signature-based methods, (2) heuristic-based methods and behavior-based methods. In addition, they investigate some features for malware detection and discuss concealment techniques used by malware to evade detection. Nonetheless, the aforementioned research does not consider either dynamic or hybrid approaches. Souri et al. (2018) present a survey of malware detection approaches divided into two categories: (1) signature-based methods and (2) behavior-based methods. However, the survey does not provide either a review of the most recent deep learning approaches or a taxonomy of the types of features used in data mining techniques for malware detection and classification. Ucci et al. (2019) categorize methods according to: (i) what is the target task they try to solve, (ii) what are the feature types extracted from Portable Executable files (PEs), and (iii) what machine learning algorithms they use. Although the survey provides a complete description of the feature taxonomy, it does not outline new research trends, especially deep learning and multimodal approaches. Ye et al. (2017) cover traditional machine learning approaches for malware detection, that consists of feature extraction, feature selection and classification steps. However, important features such as the entropy or structural entropy of a file, and some dynamic features such as network activity, opcode and API traces, are missing. In addition, deep learning methods or multimodal approaches for malware detection, which have been hot topics for the last few years, are not covered. Lastly, Razak et al. (2016) provide a bibliometric analysis of malware. It analyzes the publications by country, institution or authors related to malware. Nonetheless, the paper does not provide a description of the features employed by malware

[1] https://scholar.google.es/.

Table 1
List of contributions by the surveyed papers. A ✓tick denotes the information that a survey tries to cover but it does not necessarily provides a thoroughly description of the topic.

Paper	Feature Taxonomy	Static Methods	Dynamic Methods	Hybrid Methods	Multimodal Learning Methods	Deep Learning Methods	Issues and Challenges
Shabtai et al. (2009)	✓	✓	×	×	×	×	×
Bazrafshan et al. (2013)	✓	✓	×	×	×	×	×
Souri et al. (2018)	×	✓	✓	×	×	×	×
Ucci et al. (2019)	✓	✓	✓	×	×	×	✓
Ye et al. (2017)	✓	✓	✓	✓	×	×	✓
Razak et al. (2016)	✓	✓	✓	×	×	×	✓
The present survey	✓	✓	✓	✓	✓	✓	✓

detectors and does not consider the state-of-the-art in the field.

Given the aforementioned limitations in the surveyed papers, the present research presents a systematic review on traditional and state-of-the-art machine learning techniques for malware detection and classification. This paper categorizes traditional methods into two groups: (1) static methods and (2) dynamic methods, categorizing the methods by the type of information or features extracted from Portable Executable files. It extends the surveyed papers by exploring various ways of combining different modalities or types of information, and analyzes state-of-the-art deep learning approaches, which are grouped according to the nature of the raw data fed into the systems. The paper closes with a discussion of the research issues and challenges faced by researches including, but not limited to, the problem of concept drift, adversarial learning and the problem of class imbalance.

3. Background

This section presents an overview of the types of analysis, techniques and tools for dissecting malware targeting the Windows operating system, by far the most used OS worldwide. First, we describe the Portable Executable file format. Then, we provide a description of the fundamental approaches for malware analysis and we give a list of the most common tools utilized for the examination of malicious software. Lastly, we introduce the taxonomy of malware and a brief overview of its evolution.

3.1. The Portable Executable file format

The Portable Executable (PE) format is a file format for executables, object code, DLLs, FON Font files and others used in 32-bit and 64-bit versions of the Windows operating system. The PE32 format stands for Portable Executables of 32-bit while PE32 + stands for Portable Executables of 64-bit format.

Portable Executables encapsulate the information necessary for a Windows operating system to manage the executable code. This includes dynamic library references for linking, API export and import tables, resource management data and threat-local storage data. A PE file consists of a number of headers and sections that tell the dynamic linker how to map the file into memory. See Fig. 1. The PE Header contains information about the executable such as the number of sections, the size of the "PE Optional Header", characteristics of the file, etc.[2] It also contains the import address table (IAT), which is a lookup table used by the application when calling a function in a different module. In addition, a Portable Executable file has various sections that contain the code and data of the executable including, but not limited to, the following:

- The. data section. This section is used to declare initialized data or constants that do not change at runtime.

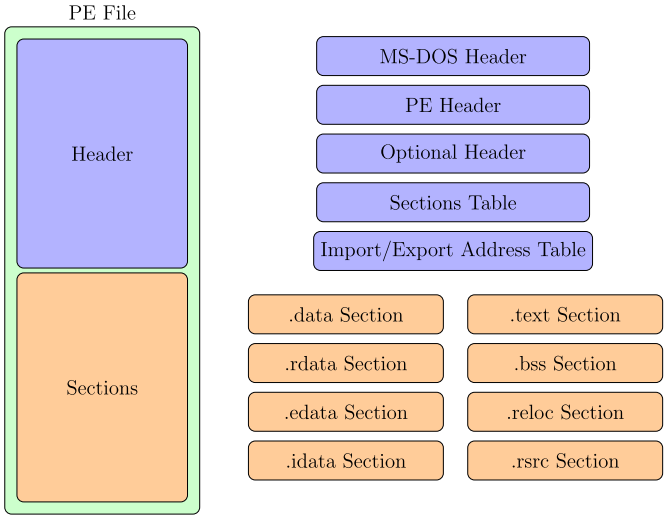

Fig. 1. Portable executable file format.

- The. bss section. This section is used for declaring variables and contains uninitialized data.
- The. text section. This section keeps the actual code of the program.
- The. rsrc section. This section contains all the resources of the program.
- The. rdata section. This section holds the debug directory which stores the type, size and location of various types of debug information stored in the file.
- The. idata section. This section contains information about functions and data that the program imports from DLLs.
- The. edata section. This section contains the list of the functions and data that the PE file exports for other programs.
- The. reloc section. This section holds a table of base relocations. A base relocation is an adjustment to an instruction or initialized variable value that is needed if the loader could not load the file where the linker assumed it would.

More information on the PE file format can be found in the documentation provided by Microsoft.[3]

3.2. Taxonomy of malware

Malicious software, also known as malware, is any kind of software that is specifically designed to disrupt, damage or gain unauthorized access to a computer system or network. Depending on the purposes and proliferation systems, malware can be divided into various, not mutually exclusive categories.

[2] https://en.wikibooks.org/wiki/X86_Disassembly/Windows_Executable_Files#PE_Header.

[3] https://docs.microsoft.com/en-_us/windows/win32/debug/pe-_format.

- Adware. Malware designed to automatically generate online advertisements. This type of malware generates revenue for its developer by displaying advertisements on the user interface or the screen.
- Backdoor. Computer software that is designed to bypass a system's security mechanism and install itself on a computer to allow the attacker to access it.
- Bot. Software created to automatically perform specific operations such as DDoS attacks or distribute other malware. Bots are part of a botnet, a network of interconnected devices, which are controlled using command and control (C&C) software.
- Downloader. A downloader program's purpose is to download and install additional malicious programs.
- Launcher. A launcher is a computer program designed to stealthily launch other malicious programs.
- Ransomware. Malicious software that restricts user access to the computer system by encrypting the files or locking down the system while demanding a ransom for its release.
- Rootkit. Malware designed to conceal the existence of other malicious programs.
- Spyware. Computer software that spies and collects sensitive information without permission from a victim's computer. Examples include key-loggers, password gravers and sniffers.
- Trojan. A Trojan is a type of malicious software that disguises itself as legitimate software to trick users into downloading and installing malware on their systems.
- Virus. Malicious software that can propagate itself from device to device.
- Worm. A type of virus that exploits vulnerabilities of the operating system to spread. The major difference between worms and viruses is the ability of worms to independently self-replicate and spread while viruses depend on human activity.

3.3. Malware analysis

The process of dissecting malware to understand how it works, determine its functionality, origin and potential impact is called malware analysis. With the millions of new malicious programs in the wild, and the mutated versions of previously detected programs, total malware encountered by security analysts has been growing over the past years.[4] Consequently, malware analysis is critical to any business and infrastructure that responds to security incidents.

There are two fundamental approaches to malware analysis: (1) static analysis and (2) dynamic analysis. On the one hand, static analysis involves examining the malware without running it. On the other hand, dynamic analysis involves running the malware. An in-depth description of both approaches is provided in Sections 3.3.1 and 3.3.2.

3.3.1. Static analysis

Static analysis consists of examining the code or structure of the executable file without executing it. This kind of analysis can confirm whether a file is malicious, provide information about is functionality and can also be used to produce a simple set of signatures. For instance, the most common method used to uniquely identify a malicious program is hashing. That is, a hashing program produces a unique hash, a sort of fingerprint, that identifies the program. The two most popular hash functions are the Message-Digest Algorithm 5 (MD5) and the Secure Hash Algorithm 1 (SHA-1). The most common static analysis approaches are:

- Finding sequences of characters or strings. Searching through the strings of a program is the simplest way to obtain hints about its functionality. Strings extracted from the binary can contain references to filepaths of files modified or accessed by the executable, URLs to which the program accesses, domain names, IP addresses, attack commands, names of Windows dynamic link libraries (DLLs) loaded, registry keys, and so on. The utility tool *Strings*[5] can be used to search ASCII or Unicode strings ignoring context and formatting in an executable.
- Gathering the linked libraries and functions of an executable, as well as the metadata about the file included in the headers. These data provide information about code libraries and functionalities common to many programs, that programmers link so that they do not need to re-implement a certain functionality. The names of this Windows functions can give us an idea of what the executable does. The utility *Dependency Walker*[6] is a free program for Microsoft Windows used to list the imported and exported functions of a PE file.
- Analyze PE file headers and sections. The PE file headers provide more information than just imports. They contain metadata about the file itself, such as the actual sections of the file. One way to retrieve this information is with the *PEView* tool.[7]
- Searching for packed/encrypted code. Malware writers usually use packing and encryption to make their files more difficult to analyze. Software programs that have been packed or encrypted usually contain very few strings and higher entropy compared to legitimate programs. One way to detect packed files is with the *PEiD* program[8]
- Disassembling the program, i.e. translating machine code into assembly language. This reverse-engineering process loads the executable into a disassembler to discover what the program does. The most relevant software programs for disassembling PE executables are *IDA Pro*,[9] *Radare2*[10] and *Ghidra*.[11]

3.3.2. Dynamic analysis

Dynamic analysis involves executing the program and monitoring its behavior on the system. This is typically performed when static analysis has reached a dead end, either due to obfuscation or on having exhausted the available static analysis techniques. Unlike static analysis, it traces the real actions executed by the program. However, the analysis must be run in a safe environment to not expose the system to unnecessary risks, where the system is both the machine running the analysis tool and the rest of the machines on the network. To this end, dedicated physical or virtual machines are set up.

Physical machines must be set up on air-gapped networks, that is isolated networks where machines are disconnected from the Internet or any other network, to prevent malware from spreading. The main downside of physical machines is this scenario with no Internet connection, as many malicious programs depend on Internet connection for updates, command and control and other features.

The second option is to set up virtual machines to perform dynamic analysis. A virtual machine emulates a computer system and provides the functionality of a physical computer. The OS running in the virtual machine is kept isolated from the host OS and thus, malware running on a virtual machine cannot harm the host OS. *VMware Workstation*[12] and *Oracle VM VirtualBox*[13] are some of the virtual machine solutions available to analysts. In addition, there are several all-in-one software products based on sandbox technology that can be used to perform basic dynamic analysis. The most well-known is the *Cuckoo Sandbox*,[14] an

[4] https://www.av-_test.org/en/statistics/malware/.

[5] https://docs.microsoft.com/en-_us/sysinternals/downloads/strings.
[6] http://www.dependencywalker.com/.
[7] http://wjradburn.com/software/PEview.zip.
[8] https://peid.waxoo.com/.
[9] https://www.hex-_rays.com/products/ida/.
[10] https://rada.re/r/.
[11] https://github.com/NationalSecurityAgency/ghidra.
[12] https://www.vmware.com/products/workstation-_pro.html.
[13] https://www.virtualbox.org/.
[14] https://cuckoosandbox.org/.

open source automated malware analysis system. This modular sandbox provides capabilities to trace API calls, analyze network traffic or perform memory analysis. Alternatively, there is a wide list of utilities for dynamically analyze malware and perform advanced and specific monitoring of some functionalities. *Process Monitor*,[15] or procmon, is a tool for Windows that monitors certain registry, file system, network, process and thread activity. *Process Explorer*[16] show the information about which handles and DLL processes are opened or loaded into the operating system. *Regshot*[17] is a registry compare utility that allows snapshots of registries to be taken and compared. *NetCat*[18] is a networking utility that can be used to monitor data transmission over a network. *Wireshark*[19] is an open source sniffer that allows packets to be captured and network traffic to be intercepted and logged. Another indispensable software utility are debuggers. A debugger is used to examine the execution of another program. They provide a dynamic view of a program as it runs. The primary debugger of choice for malware analysts is *OllyDbg*[20] , an ×86 debugger that is free and has many plugins to extend its capabilities.

The risks of using virtualization and sandboxing for malware analysis is that some malware can detect when it is running in a virtual machine or a sandbox and subsequently, they will execute differently than when in a physical machine to make the job of malware analysts harder. In addition, even if you take all possible precautions, some risk is always present when analyzing malware. From time to time, vulnerabilities have been found in the virtualization tools that allow an attacker to exploit some of its features such as the share folders feature.

3.4. Malware evolution

The diversity, sophistication and availability of malicious software pose enormous challenges for securing networks and computer systems from attacks. Malware is constantly evolving and forces security analysts and researchers to keep pace by improving their cyberdefenses. The proliferation of malware increased due to the use of polymorphic and metamorphic techniques used to evade detection and hide its true purpose. Polymorphic malware uses a polymorphic engine to mutate the code while keeping the original functionality intact. Packing and encryption are the two most common ways to hide code. Packers hide the real code of a program through one or more layers of compression. Then, at runtime the unpacking routines restore the original code in memory and execute it. Crypters encrypt and manipulate malware or part of its code, to make it harder for researchers to analyze the program. A crypter contains a stub used to encrypt and decrypt malicious code. Metamorphic malware rewrites its code to an equivalent whenever it is propagated. Malware authors may use multiple transformation techniques including, but not limited to, register renaming, code permutation, code expansion, code shrinking and garbage code insertion. The combination of the aforementioned techniques resulted in rapidly growing malware volumes, making forensic investigations of malware cases time-consuming, costly and more difficult.

Traditional antivirus solutions that relied on signature-based and heuristic/behavioral methods present some problems. A signature is a unique feature or set of features that uniquely distinguishes an executable, like a fingerprint. However, signature-based methods are unable to detect unknown malware variants. To tackle these challenges, security analysts proposed behavior-based detection, which analyzes the file's characteristics and behavior to determine if it is indeed malware, though the scanning and analysis can take some time. To overcome the prior pitfalls of traditional antivirus engines and keep pace with new attacks and variants, researchers started adopting machine learning to complement their solutions, as machine learning is well suited for processing large volumes of data.

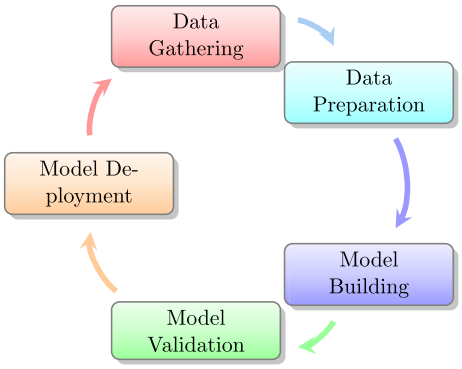

Fig. 2. Machine learning workflow.

4. Traditional machine learning approaches

Over the past decade there has been an increase in the research and deployment of machine learning solutions to tackle the tasks of malware detection and classification. The success and consolidation of machine learning approaches would not have been possible without the confluence of three recent developments:

1. The first development is the increase in labeled feeds of malware meaning that, for the first time, labeled malware is available not only to the security community but also to the research community. The size of these feeds ranges from limited high-quality samples, like the ones provided by Microsoft (Ronen et al., 2018) for the Big Data Innovators Gathering Anti-Malware Prediction Challenge, to huge volumes of malware, such as theZoo (Yuval Nativ, 2015) or VirusShare (2011).
2. The second development is that computational power has increased rapidly and at the same time has become cheaper and closer to the budget of most 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.
3. 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.

In machine learning, a workflow is an iterative process that involves gathering available data, cleaning and preparing the data, building models, validating and deploying into production. See Fig. 2. Instead of dealing with raw malware, the data preparation process of traditional machine learning approaches involves preprocessing the executable to extract a set of features that provide an abstract view of the software. Afterwards the features are used to train a model to solve the task at hand. Because of the variety of malware functionalities, it is important not only to detect malicious software, but also to differentiate between different kinds of malware in order to provide a better understanding of their capabilities. The main difference between machine learning solutions for detection or classification of malware is the output returned by the system implemented. On the one hand, a malware detection system outputs a single value $y = f(x)$, in the range from 0 to 1, which indicates the maliciousness of the executable. On the other hand, a classification system outputs the probability of a given executable belonging to each output class or family, $y \in \mathbb{R}^N$, where N indicates the number of different families.

[15] https://docs.microsoft.com/en-_us/sysinternals/downloads/procmon.
[16] https://docs.microsoft.com/en-_us/sysinternals/downloads/process-_explorer.
[17] https://sourceforge.net/p/regshot/wiki/Home/.
[18] http://netcat.sourceforge.net/.
[19] https://www.wireshark.org/.
[20] http://www.ollydbg.de/.

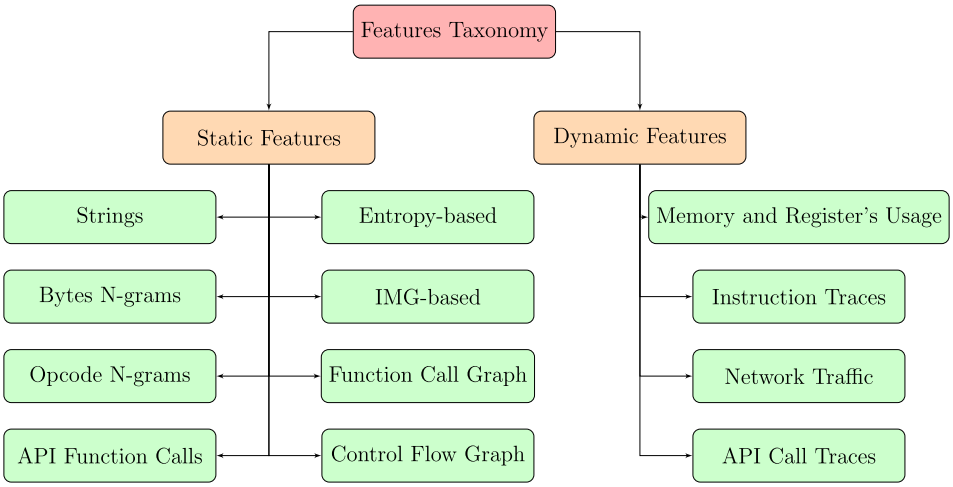

Fig. 3. Taxonomy of features used by traditional M.L. approaches.

A taxonomy of the features is provided in Fig. 3. Accordingly, the types of features can be divided into two groups, like the types of malware analysis approaches: (1) static features and (2) dynamic features. Each feature type individually below.

4.1. Static features

Static features are extracted from a piece of program without involving its execution. In Windows Portable Executable files, static features are basically derived from two sources of information, the binary content of the executable or the assembly language source file obtained after decompiling and disassembling the binary executable. On the other hand, in Android applications these features are extracted by disassembling the APK. To extract the assembly language source code of some given software you can use the disassembler tool of your choice. For Windows, you might use IDA Pro or Radare2. Tables 2 and 3 provide a summary of the static methods reviewed. Below you will find a description of each static feature type presented in Fig. 3.

4.1.1. String analysis

String analysis refers to the extraction of every printable string within an executable or program. A string refers to a sequence of characters. Searching for strings is the simplest way to obtain clues about the functionality of a program. The information that may be found in these strings can be, for instance, URLs that the program connects to, file locations or filepaths of files accessed/modified by the program, names of the menus of the application, etc. The utility called "Strings" can be used to search an executable for ASCII and Unicode strings, ignoring context and formatting.

Although there are studies using string analysis to detect malware (Konopisky, 2018; Lee et al., 2011), string analysis is commonly employed together with other static or dynamic techniques to reduce its pitfalls. (Ye et al., 2008a). developed a malware detection system based on interpretable strings extracted from both API execution calls and semantic strings reflecting an attacker's intent and goal. The system was composed of a parser to extract interpretable strings for each PE file and a SVM ensemble with bagging to construct the detector. The performance of the system was evaluated on a dataset collected by Kingsoft anti-virus lab.

4.1.2. Bytes and opcode N-Grams

The most common type of features for malware detection and classification is n-grams. An n-gram is a contiguous sequence of n items from a given sequence of text. N-grams can be extracted from the bytes sequences representing the malware's binary content and from the assembly language source code. By treating a file as a sequence of bytes, byte n-grams are extracted by looking at the unique combination of every n consecutive bytes as an individual feature. On the other hand, the sequence of assembly language instructions can also be extracted from the assembly language source code. In this case, only the mnemonic of the instruction, i.e. "ADD", "MUL", "PUSH", etc., is retained. Thus, opcode or mnemonic n-grams refer to the unique combination of every n consecutive opcodes as an individual feature.

Moskovitch et al. (2008) presented a method for classifying malware based on text categorization techniques. First, they extracted all n-grams from the training data, with n ranging from 3 to 6. Second, they selected the top 5500 features according to their Document Frequency (DF) score, to which the Fisher Score feature selection technique was later applied. Afterwards, using the resulting features as input they trained various algorithms such as an Artificial Neural Network (ANN), a Support Vector Machine (SVM), Naïve Bayes (NB) and Decision Trees (DT).

Jain and Meena (2011) proposed a method to extract bytes n-gram features, with n ranging from 1 to 8, from known malicious samples to assist in classification of unknown executables. As the number of unique n-grams is extremely large, they used a technique called class-wise document frequency to reduce the feature space. Finally, different N-gram models were prepared using various classifiers like Naïve Bayes, Instance-based Learner, Decision Trees, Adaboost and Random Forests.

Fuyong et al. (2017) proposed a method that calculates the information gain of each bytes n-gram in the training samples and selected K n-grams with the maximum information gain as features. Afterwards, they calculated the averages of each attribute of the feature vectors from the malware and benign samples separately. Lastly, a new piece of software was assigned to one of the two categories according to the similarity between the feature vector of the unknown sample and the average vectors of the two categories.

Santos et al. (2013) proposed a technique for malware detection based on the frequency of appearance of opcode sequences and its relevance. Each program was represented as a vector of features where each feature corresponds to a distinct 1-g or 2-g. To reduce the number of 2-g features, they applied Information Gain to select the top 1000 features. Their approach was validated on 17000 malicious and 1000 benign programs, and results show that the higher accuracy was achieved by a Support Vector Machine classifier with Pearson VII as kernel.

Shabtai et al. (2012) proposed a framework for detecting malware based on opcode n-gram features with n ranging from 1 to 6. They performed a wide set of experiments to: (1) identify the best term representation, whether it is the Term Frequency (TF) or Term Frequency-Inverse Document Frequency, (2) determine the n-gram size, (3) find

Table 2
A side-by-side comparison of the algorithms and feature types of the reviewed static-based methods. Algorithms: Support Vector Machine (SVM), Random Forests (RF), Inference Trees (IT), Recursive Bipartition (RB), Naive Bayes (NB), Artificial Neural Networks (ANN), Decision Trees (DT), Instance-based Learner (IL), K-Nearest Neighbor (K-NN), Logistic Regression (LR), Gradient Boosting (GB), Sequential Minimal Optimization (SMO), Decision Stump (DS), Random Tree (RT), Voted Perceptron (VT).

Paper	Feature Type	Feature Selection, Reduction	Classification Algorithm
Ye et al. (2008a)	Strings	–	SVM ensemble with bagging
Moskovitch et al. (2008)	3, 4, 5, 6-g (bytes)	Fisher Score	ANN, SVM, NB, DT
Jain and Meena (2011)	n-grams (bytes)	Classwise Document Frequency	NB, IL, DT, AdaBoost, RF
Fuyong et al. (2017)	3-g (bytes)	Information Gain	1-NN
Santos et al. (2013)	1,2-g (opcodes)	Information Gain	SVM with Pearson's VII Kernel
Shabtai et al. (2012)	1,2,3,4,5,6-g (opcodes	Term Frequency (TF) and TF-Inverse Document Frequency (TF-IDF)	DT, ANN, LR, RF, BDT; NB, BNB
Hu et al. (2013)	n-grams (opcodes)	hashing trick	agglomerative hierarchical clustering
Yuxin et al. (2019)	n-grams (opcodes)	–	DBN, SVM, K-NN, DT
Sami et al. (2010)	API calls	Fisher Score + Clospan Algorithm	RF
Ye et al. (2008b)	API calls	–	Rule-based Classification System
Ahmadi et al. (2016)	API calls	–	GB
Sorokin and Jun (2011)	Structural Entropy	Discrete Wavelet Transform	Sequence Similarity + Wagner-Fischer Dynamic Programming
Baysa et al. (2013)	Structural Entropy	Discrite Wavelet Transform	Sequence Similarity + Levenhstein distance
Wojnowicz et al. (2016)	Structural Entropy	Haar Discrete Wavelet Transform	Suspiciously Structured Entropic Change Score (SSECS) + LR
Gibert et al. (2018b)	Structural Entropy	Haar Discrete Wavelet Transform	K-NN + Levenhstein distance
Nataraj et al. (2011)	Gray Scale IMG	GIST features	K-NN
Ahmadi et al. (2016)	Gray Scale IMG	Haralick & Local Binary Pattern features	GB
Kancherla et al. (2013)	Gray Scale IMG	Intensity-based, Wavelete-based and Gabor-based features	SVM
Kinable et al. (2011)	Function Call Graph	–	DBSCAN, K-medoids
Hassen and Chan (2017)	Function Call Graph	In-house vector representation algorithm	RF, meta-classifier
Eskandari and Hashemi (2011)	CFG	–	RF, SMO, DS, K-Star, NB, RT
Faruki et al. (2012)	CFG	–	RF, SMO, J-48 DT, NB, VP

Table 3
A side-by-side comparison of the dataset characteristics of the reviewed of the static methods.

Paper	Source	Total Size	Task
Ye et al. (2008a)	Kingsoft lab	39838	Detection
Moskovitch et al. (2008)	VXHeavens, Windows XP	30423	Detection
Jain and Meena (2011)	VXHeavens, Windows XP	2138	Detection
Fuyong et al. (2017)	Open Malware Benchmark, Windows XP, Windows 8	2540	Detection
Santos et al. (2013)	VXHeavens, Windows OS	18000	Detection
Shabtai et al. (2012)	VXHeavens, Windows XP	30423	Detection
Hu et al. (2013)	VXHeavens	132234	Classification
Yuxin et al. (2019)	–	9200	Detection
Sami et al. (2010)	–	34820	Detection
Ye et al. (2008b)	Kingsoft Corporation	29580	Detection
Ahmadi et al. (2016)	Microsoft Malware Classification Challenge	21741	Classification
Sorokin and Jun (2011)	–	–	–
Baysa et al. (2013)	–	–	–
Wojnowicz et al. (2016)	Cylance repository	699121	Detection
Gibert et al. (2018b)	Microsoft Malware Classification Challenge	21741	Classification
Nataraj et al. (2011)	–	9458	Classification
Ahmadi et al. (2016)	Microsoft Malware Classification Challenge	21741	Classification
Kancherla et al. (2013)	Offensive Computing, Windows XP, Windows Vista, Windows 7, Windows NP	27000	Detection
Kinable et al. (2011)	–	1919	Classification
Hassen and Chan (2017)	Microsoft Malware Classification Challenge	21741,	Classification
Eskandari and Hashemi (2011)	APA malware research center	4445	Detection
Faruki et al. (2012)	–	6234	Detection

the optimal K top n-grams and feature selection method, and (4) evaluate the performance of various machine learning algorithms.

Hu et al. (2013) presented MutantX-S, a clustering approach based on opcode N-gram features extracted from the assembly language source code of malware obtained after a disassemble process. MutantX-S improves the scalability on handling very large numbers of malware with high-dimensional features by applying a hashing trick and a close-to-linear clustering algorithm. Instead of working on the large volumes of data, the algorithm performed agglomerative hierarchical clustering only on prototypes.

Alternatively, Yuxin et al., (2019) used a Deep Belief Network (DBN) as an autoencoder to reduce the dimensions of the input feature vectors. As a result, after learning is completed, the last hidden layer of the DBN outputs a new representation or encoding of the N-gram vectors passed as input. By training the DBN with unlabeled data, their classification accuracy outperformed that of the K-Nearest Neighbor, Support Vector Machines and Decision Tree algorithms.

Despite their success at detecting malware, n-gram approaches have some issues that is worth mentioning. First, it is impractical and computationally prohibitive to exhaustively enumerate all n-grams. Estimating model parameters when the number of features is larger than the number of samples might lead to the curse of dimensionality. As a result, feature selection and reduction techniques must be employed. Second, researchers (Raff et al., 2018b) have concluded that byte n-grams appear to be learning mostly from string content in an executable, in particular items from the PE header. As there are millions of potential n-grams (for a larger n), feature selection techniques tend to select as features those that occur frequently enough. This encourages the selection of low entropy features consisting mostly of strings and padding. Third, regardless of what n-grams are learned, we must obtain an exact match when classifying a new sample. Consequently, any minor change will make the feature not occur and, thus, not impact our model. Thus, this lack of generalization is a potential source of over-fitting.

4.1.3. API function calls

Application Programming Interfaces (API) and their function calls are regarded as very discriminative features. Literature has shown that API functions invocation might be used to model the program's behavior. Essentially, API functions and system calls are related to services provided by the operating systems such as networking, security, file management, and so on. 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.

Sami et al. (2010) proposed a three-step framework to classify PE files based on API calls usage. First, they analyzed the Portable Executable files and extracted the list of imported API calls. Second, they reduced the feature vector using the Clospan algorithm (Yan et al., 2003). Lastly, the subset of features was used to learn a model using Random Forest.

Ye et al. (2008b) proposed a rule-based system for malware classification. The system consists of three major components: (1) the PE parser, (2) the OOA (Objective-Oriented Association) rule generator and (3) the malware detection module. The PE parser is responsible for parsing the executable and extracting the static execution calls of the corresponding API functions. Then, these calls are used as signatures of the PE files and stored in a signature database. Afterwards, an OOA algorithm is applied to generate class association rules which are stored in the rule database. Lastly, the feature calls and the rules are passed to the malware detection module to determine whether a file is benign or malicious.

Ahmadi et al. (2016) used the frequency of a subset of 794 API function calls, extracted from an analysis on almost 500 K malware samples, to build a multimodal system to classify malware into families. A complete description of their research is provided in Section 6

4.1.4. Entropy

Malware authors often employ a variety of obfuscation techniques to hide the malicious purpose of the executables. The two most commonly used are compression and encryption, which are used to conceal malicious segments from static analysis. 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. To this end, entropy analysis has been employed because files with segments of code that have been compressed or encrypted tend to have higher entropy than native code. 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. This behavior can be observed in a "padded" chunk of code. On the contrary, a high entropy value would indicate that the chunk consists entirely of distinct values. For instance, Lyda et al. (2007) analyzed a corpus of files consisting of plain text files, native, compressed and encrypted executables, and observed that the average entropy of the executables was 5.09, 6.80 and 7.17, respectively.

As a result, previous research has used a high mean entropy to detect the presence of encryption and compression. However, when the malicious code is concealed in a sophisticated manner it might be hard to detect through such simple entropy statistics. A common approach to reduce the entropy of a file is to pad "nop" instructions. Nevertheless, files with encrypted, compressed, native or padded segments tend to have distinct and unique entropy levels. Thus, researchers (Sorokin and Jun, 2011) started analyzing what is known as the structural entropy of a file, the representation of the malware's byte sequence as a stream of entropy values, where each value indicates the amount of entropy over a small chunk of code in a specific location (see Fig. 4). In particular, Sorokin and Jun (2011) compared the similarity between the structural entropy of an unknown file with that of the training dataset to detect malware.

Baysa et al. (2013) extended the previous work to detect metamorphic malware. They applied wavelet analysis to determine the areas where there are significant changes in the entropy values. Afterwards, they compared the similarity between two files using the Levenshtein distance. Therefore, given an unknown piece of software, it would be classified as the class corresponding to the most similar sample in the training set.

Wojnowicz et al. (2016) developed a method to automatically quantify the extent to which variations in a file's structural entropy make it suspicious. This score is calculated through a two-step process: (1) they computed the wavelet-based energy spectrum of the executable's structural entropy; and (2) they fit various logistic regression models over j-th resolution levels to produce a set of beta coefficients to weight the strength of each resolution energy on the file's probability of being malicious.

4.1.5. Malware representation as a gray scale image

An interesting approach for malware visualization was first introduced by

Nataraj et al. (2011) who visualized the malware's binary content as a gray scale image. This is achieved by interpreting every byte as one pixel in an image, where values range from 0 to 255 (0:black, 255:white). Afterwards, the resulting array is reorganized as a 2-D array.

Fig. 5 presents samples from two malware families represented as gray scale images. You can observe that the image representation of samples of a given family is quite similar while distinct from that belonging to a different family. This visual similarity is the result of reusing code to create new binaries. Thus, if old samples are re-used to implement new binaries, the resulting ones would be similar. In most cases, by representing an executable as a gray scale image it would be possible to detect small variations between samples belonging to the same family.

This visual similarity has been exploited by various authors for detecting and classifying malware. In particular, Nataraj et al. (2011) extracted GIST features from the gray scale representation of malware's binary content. Finally, a new executable is classified under one family or another using the K-Nearest Neighbor algorithm (K-NN) with the Euclidean distance as metric. Ahmadi et al. (2016) extracted Haralick and Local Binary Pattern features for classifying malware using boosting tree classifiers.

Kancherla et al. (2013) extracted three sets of features: (1) Intensity-

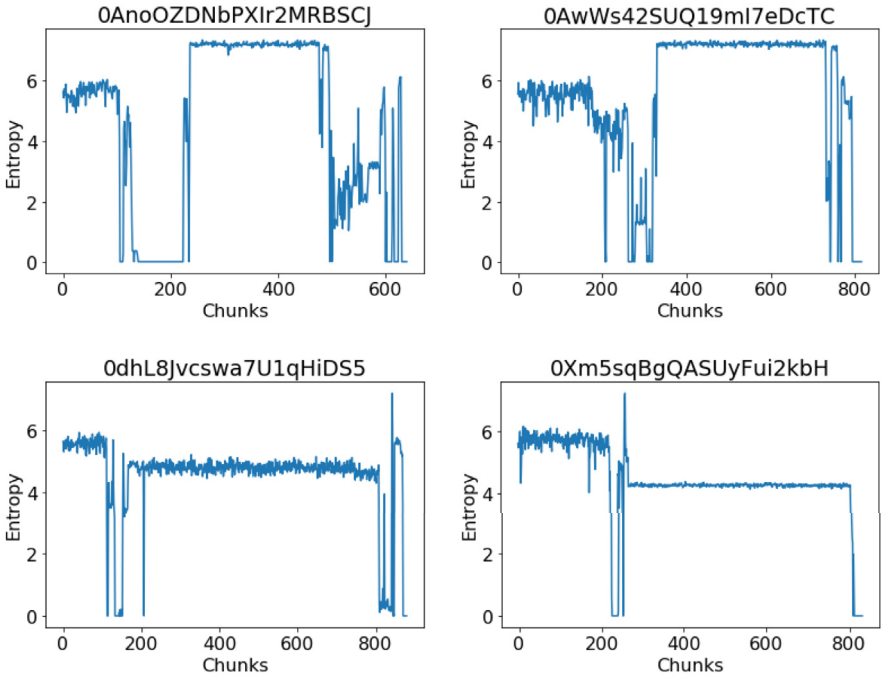

Fig. 4. Structural entropy representation of samples belonging to the Ramnit and Gatak families.

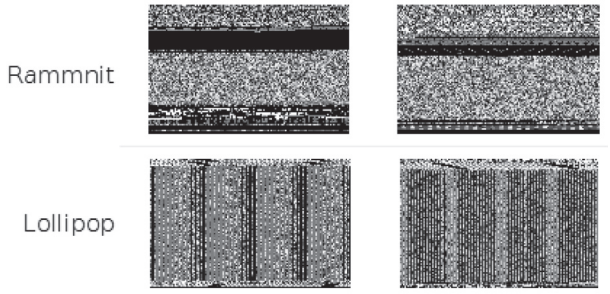

Fig. 5. Gray scale representation of the binary content of malware samples belonging to the Ramnit and the Lollipop families.

based, (2) Wavelet-based and (3) Gabor-based features. In particular, they extracted the average intensity, variance, mode, skewness, kurtosis and the number of pixels with intensity value 0 and 255. Regarding the wavelet-based features, they applied level 3 wavelet decomposition using the Daubechies wavelet, also known as db4, and they obtained one set of approximate coefficients and three sets of detailed coefficients. The features extracted from each of these coefficients were the mean, variance, maximum and minimum values. Finally, to extract the Gabor-based features they applied the Gabor filter (convolution of an input with Gabor function) to the image. The performance of support vector machines as learning algorithm was evaluated on a dataset of 15000 malicious and 12000 benign samples where 70% were used for training and 30% for testing.

The gray scale image representation of software has some drawbacks directly related to how images are generated. Primarily, binaries are not 2-D images and by transforming them as such you introduce unnecessary priors. First, 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. Second, it imposes non-existing spatial correlations between pixels in different rows, which might not be true.

Additionally, like the majority of static features, it suffers from code obfuscation techniques. In particular, techniques like encryption and compression might completely change the bytes structure of a binary program and, thus, methods based on this kind of representation would fail to correctly classify its class. This can be observed in the gray scale representation of samples belonging to the Autorun. K and Yuner. A families from the MalImg dataset (Nataraj et al., 2011), which are almost equal due to both having being compressed with the UPX packer.

4.1.6. 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. A vertex is represented by either one of the following two types of functions:

1. Local functions, implemented by the programmer to perform specific tasks.
2. External functions: provided by the O.S. or system and external libraries.

One particularity of the graph is that only local functions can invoke external functions, not the other way around. Function call graphs are generated from the static analysis of the disassembly file. To extract the FCG of Windows PE executables, IDA Pro or Radare2 can be used.

Kinable et al. (2011) presented an approach to cluster malware based on the structural similarities between function call graphs. They investigated the performance of the k-medoids and Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithms for malware clusterization. The comparison among call graphs was computed with pairwise graph similarity scores via graph matching. The experiments were performed on a dataset comprising 194,675 samples from 1050 different malware families.

Hassen and Chan (2017) proposed a method to extract a vector representation of the function call graph based on function clustering. The first module of the system extracts the FCG and labels the vertices with external functions with the function names. The original names of the

Table 4
A side-by-side comparison of the algorithms and input data of the reviewed dynamic-based methods. Algorithms: Logistic Regression (LR), Decision Tree (DT), Random Forest (RF), Support Vector Machine (SVM).

Paper	Input	Classification Algorithm	Feature Selection, Reduction Techniques
Ghiasi et al. (2012)	Register's Usage	Matching based on Registers Values Set Analysis	–
Ghiasi et al. (2015)	Register's Usage	Matching based on Jaccard's similarity distance on dynamic VSA representations	Prototype Extraction
Carlin et al. (2017a)	Instruction Traces	RF, Hidden Markov models	Opcode counts
O'kane et al. (2016)	Instruction Traces	SVM	PCA
Anderson et al. (2011)	Instruction Traces	SVM	–
Storlie et al. (2014)	Instruction Traces	flexible spline logistic regression	–
Bekerman et al. (2015)	Network Traffic	Naïve Bayes, J48 DT, RF	Correlation Feature Selection Algorithm
Zhao et al. (2015)	DNS and Network Traffic	Reputation Engine	–
Kheir (2013)	Network Traffic (HTTP traffic)	High-level clustering, fine clustering and Incremental K-means clustering	.
Boukhtouta et al. (2016)	Network Traffic	Boosted J48, J48, NB, Boosted NB, SVM, HMMs	–
Perdisci and Wenke Lee (2015)	HTTPs Traffic	Coarse-grain Clustering, Fine-grain Clustering, Cluster-merging	–
Galal et al. (2016)	API Call Traces	DT, RF, SVM	Hand-crafted Heuristics
Ding et al. (2013)	API Call Traces	Object Oriented Association Mining	Document Frequency, Information Gain
Salehi et al. (2017)	API Call Traces	RF, J48 DT, Bayesian Logistics Regression, Sequential Minimal Optimization	Fisher Score, SVM based on Recursive Feature Elimination
Rieck et al. (2011)	API Call Traces	Hierarchical Clustering	–
Uppal et al. (2014)	API Call Traces	NB, RF, DT, SVM	odds ratio

internal functions are not preserved and instead, each internal function is represented as the sequence of instructions that the function implements. Afterwards, the resulting FCG is passed to the next module to cluster the local functions and relabel them with their cluster-id. Finally, the graph representation is converted into a feature vector using function clustering based on the Minhash signatures of the functions.

4.1.7. Control Flow Graph

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

Eskandari and Hashemi (2011) presented an approach to detect metamorphic malware through their Control Flow Graphs. The system consists of three components. First, the PE file is disassembled. Second, a preprocessing algorithm is applied to assembly files to generate a CFG including the API calls. Then, the resulting sparse graph is converted to a vector representation. Third, the system labels the CFG using a classification algorithm. The performance of the system was evaluated on 2140 and 2305 benign and malicious PE executables, respectively, and the best results were achieved by a Random Forest classifier, with 97% accuracy.

Faruki et al. (2012) proposed an approach to generate API calls n-grams to detect malicious code from malware's CFG. In their work, the abstraction of the executable is represented by the API calls made. These API calls are later converted into feature vectors using n-gram analysis with n ranging from 1 to 4. Afterwards, classification is performed with various algorithms, including Random Forest, Sequential Mining Optimization, J-48 Decision Tree, Naïve Bayes and Voted Perceptron. The best results were achieved by the Random Forest classifier with the API 4-g feature vector as input.

4.2. Dynamic features

Dynamic features are those extracted from the execution of malware at runtime. Dynamic analysis involves monitoring malware (and observing the real sequence of instructions executed or the sequence of API functions triggered) as it runs or examining the system after the malware has executed. It reveals process creation, file and registry manipulation and modifications of memory values, registers and variables. Tables 4 and 5 compile the dynamic approaches reviewed. A description of the most common information and features extracted through dynamic analysis is provided below. Approaches are grouped in four groups depending on their input data. Section 4.2.1 presents methods that extract features from malware's memory, registers and CPU usage. Section 4.2.2 includes approaches that extract features from the runtime traces of executables. Section 4.2.3 summarizes methods that extract features from the network activity of malware. Finally, Section 4.2.4 presents methods that process the API call traces of malware.

4.2.1. Memory and Register's usage

The behavior of a computer program can be represented by the values of the memory contents at runtime. In other words, values stored in different registers while a computer program is running can distinguish benign from malicious programs.

Ghiasi et al. (2012) proposed a method based on similarities of malware behaviors. First, they monitored the runtime behavior of malware and stored the register values for each API call hooked, before and after the API was invoked. Subsequently, they traced the distribution and changes of register values and created a vector for each of the values of the EAX, EBX, EDX, EDI, ESI and EBP registers. In the matching phase, a similarity score was computed between a new file and the whole training files. Then, the new file was set to the label of the file in the training set that had the highest similarity score.

Ghiasi et al. (2015) proposed a method to find similarities of runtime behaviors based on the assumption that binary behaviors affect register values differently. In their work, the runtime behavior is recorded and some API calls from common DLLs are hooked. The system analyzes memory contents and register values to build a similarity score between two files. When a new file is entered into the system, the highest similarity score between this file and the prototypes is calculated. Prototypes are small sets of files that are representative samples of the whole dataset, which provide an acceptable approximation in pair-wise distance analysis. Afterwards, the two files are similar if they achieve the minimum threshold of similarity.

Table 5
A side-by-side comparison of the dataset characteristics of the reviewed dynamic methods.

Paper	Source	Total Size	Task
Ghiasi et al. (2012)	–	1211	Detection
Ghiasi et al. (2015)	–	1240	Detection
Carlin et al. (2017a)	VirusShare	1000000	Detection
O'kane et al. (2016)	–	750	Detection
Anderson et al. (2011)	Windows XP	2230	Detection
Storlie et al. (2014)	Offensive Computing repository	21988	Detection
Bekerman et al. (2015)	Vering, Emerging Threats	50720 (records)	Classification
Zhao et al. (2015)	Alexa's TOP 1000 sites	4000000 (records)	Detection
Kheir (2013)	AV Company	100000	Detection
Boukhtouta et al. (2016)	–	–	Detection, Classification
Perdisci and Wenke Lee (2015)	–	–	Detection, Classification
Galal et al. (2016)	VirusSign, Windows7	4000	Detection
Ding et al. (2013)	–	8170	Detection
Salehi et al. (2017) Windows XP	Sami et al. (2010)	4368	Detection
Rieck et al. (2011)	Sunblet Software	33698	Classification
Uppal et al. (2014)	VXHeavens	270	Detection

4.2.2. Instruction traces

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

Carlin et al. (2017a) presented an approach that performs dynamic analysis on virtual machines to extract program runtime traces from both benign and malicious executables. They analyzed the sequence of opcodes executed to detect malware by testing two algorithms: (1) a Random Forest classifier to classify all count-based data and (2) a Hidden Markov model to classify data based on temporal relations in the opcode sequences. Carlin etal., 2017b, instead of building a classification system based on opcode counts, performed n-gram analysis, where $n = 1 \ldots 3$, to enhance the feature set. Their approach detected malware with 99.01% accuracy using sequences of up to 32 K opcodes.

O'kane et al. (2016) analyzed malicious runtime traces to determine (1) the optimal set of opcodes necessary to build a robust indicator of maliciousness in software, and to determine (2) the optimal duration of the program's execution to accurately classify benign and malicious software. The proposed approach used a Support Vector Machine on the opcode density histograms extracted during the program's execution to detect malware.

Anderson et al. (2011) introduced a malware detection method based on the analysis of graphs constructed using the instruction traces collected from the execution of the target executable. These graphs represent Markov chains, where the vertices are the instructions and the transition probabilities were estimated by the data contained in the trace. A combination of graph kernels, including the Gaussian kernel and the spectral kernel, was used to calculate the similarity matrix between the instruction trace graphs. Finally, the resulting similarity matrix is fed to a support vector machine to perform classification.

Storlie et al. (2014) presented a malware detection system based on the analysis of dynamically collected instruction traces. Instruction traces were collected from the execution of malware in a sandbox environment, a modified version of the Ether malware analysis framework (Dinaburg et al., 2008). Each instruction trace was represented with a Markov chain structure in which each transition matrix P has rows modeled as Dirichlet vector. Afterwards, the maliciousness of the program was determined using a flexible spline logistic regression model.

4.2.3. Network traffic

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

Bekerman et al. (2015) presented a system for detecting malware by analyzing network traffic. In their work, they extracted 972 behavioral features from analyzing the network traffic on the Internet, Transport and Application layers. Afterwards, a subset of the features was selected using the Correlation Feature Selection Algorithm (Hall, 1999). Then, the resulting features were used to test three different classification algorithms, including Naïve Bayes, Decision Tree (J48) and Random Forest.

Zhao et al. (2015) proposed a system to detect APT malware infections based on both malicious DNS and traffic analysis. The system consists of two main components: (1) the malicious DNS detector and (2) the network traffic analyzer. On the one hand, the malicious DNS detector extracts 14 features indicative of APT malware and C&C domains. On the other hand, the network traffic analyzer combines a signature-based system and an anomalous-based system which detect infections based on the accuracy of the rules from the VRT Rule sets (Snort, 2015) of Snort, and anomalies occurring on the Protocol and Application level, respectively. Afterwards, a J48 Decision Tree classifies the threat.

Kheir (2013) presented a systematic approach to build detection signatures based on user agent anomalies within malware HTTP traffic. First, they extracted user agent header fields within HTTP traffic. Then, they performed an initial high-level clustering step to group user agents which are likely to have similar patterns. Afterwards, they applied a second clustering step to each group of user agent to group together those agents that can be described with a common set of signatures. Lastly, incremental K-means clustering was applied to regroup user agents that share similar pattern sequences in the same clusters. Then, the token-subsequence algorithm further extracted these shared patterns and built lists of token sequences that were translated into signatures that applied at either the network or the application layer using web proxies.

Boukhtouta et al. (2016) proposed a malware detection and classification system based on DPI and flow packed headers. Their approach executed malware in a sandbox for 3 min to generate representative malicious traffic. Then, bidirectional flow features were extracted from the traffic such as the number of forward and backward packets, the maximum and minimum inter-arrival times for forward packets,

the packet size, etc. The resulting features were provided as input to the following classification algorithms: Boosted J48, J48, Naïve Bayes, Boosted Naïve Bayes and SVMs, which detected whether or not the traffic was malicious. Once the traffic had been defined as malicious, Hidden Markov Models created non-deterministic models that profiled malware families using unidirectional flows represented as a set of 45 features including the total number of packets, the median, mean and first-quartile of inter-arrival times, etc.

Perdisci and Wenke Lee (2015) proposed a method to perform behavioral clustering of malware based on the HTTPs traffic obtained from monitoring the executables in a controlled environment. The method recorded the sequences of HTTP requests performed by malware and used this information to cluster malware using at least one of the following clustering algorithms: coarse-grain clustering, fine-grain clustering and cluster-merging. Finally, network signatures were extracted for each cluster and used to identify infected computers.

4.2.4. 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. Consequently, the Windows API call traces have been used in the literature to capture the behavior of malicious applications.

Galal et al. (2016) presented an approach to process raw information gathered by API call hooking to produce a set of actions representing the malicious behaviors of malware. An action was a representative semantic feature inferred from the sequences of API calls using a set of heuristic functions. Afterwards, the viability of actions was assessed by various classification algorithms such as Decision Trees, Random Forests and Support Vector Machines.

Ding et al. (2013) proposed an API (Application Programming Interface)-based association mining method for malware detection. To increase the detection speed of the objective-oriented association (OOA) mining, they improved the rule quality, changed the criteria for API selection to remove APIs that cannot become frequent items, find association rules with the strongest discriminant power, among others. These strategies improved the running speed of their approach by 32% and 15% of the time cost for data mining and classification, respectively.

Salehi et al. (2017) proposed a dynamic method to detect malicious activity in Android APKs based on the arguments and return values of API calls. They developed an “in-house” tool consisting of a virtual machine, a hooking tool and a logging system, which was used to analyze the binary files and monitor their behavior. Their approach is based on the hypothesis that API names alone may not represent intent of the operations that the function performs. For this reason, the feature set modeling malicious and benign behaviors was constructed using the API calls, their input arguments and return values. Afterwards, the feature set was reduced through a two-stage process. In the first stage, the Fisher score was applied to select the most discriminative features. In the second stage, Support Vector Machine based on Recursive Feature Elimination reduced the feature set even more. Then, the generated feature set was used as input to the classification algorithms.

Rieck et al. (2011) developed a framework for the automatic analysis of malware behavior using clustering techniques. The framework automatically identifies novel classes of malware with similar behavior and assigns unknown malware to these discovered classes. Malware is monitored in a sandbox and the API calls are inspected at runtime. Each execution of a binary is represented as a sequential report of MIST instructions. This information is embedded in a vector space using q-grams. Afterwards, the embedded reports are clustered using prototypes. Hierarchical clustering was employed to determine groups of malware behavior. For classification, the algorithm determines the nearest prototype of the training data.

Uppal et al. (2014) presented a malware identification approach based on features from the API sequences. The method monitors the execution of a binary to keep track of the API calls invoked. Then, API call grams are generated and the odds ratio of each gram is calculated. This odds ratio is used to rank the features and select the leading n features to form the feature vector. For classification, various algorithms were proposed including Naïve Bayes, Random Forest, Decision Tree and Support Vector Machine. The evaluation of their approach was performed on a dataset on 270 binaries obtained from VXHeavens.

5. Deep learning approaches

The above traditional machine learning approaches (see Section 4) rely mainly on manually designed features based on expert knowledge of the domain. These solutions provide an abstract view of malware that a machine learning classifier, e.g. Neural Network, Decision Tree, Support Vector Machine, etc, uses to make a decision. Feature engineering and feature extraction are key, time-consuming processes of the machine learning workflow. Following recent trends in computer vision and natural language processing fields, the development of M.L. solutions for malware detection has started heading towards deep learning architectures. These solutions have replaced the aforementioned feature engineering process of the M.L. workflow with a fully trainable system beginning from raw input to the final output of recognized objects.

Deep learning approaches for tackling the problem of malware detection and classification can be classified into various groups depending on how the input is preprocessed before feeding the learning algorithm. Tables 6 and 7 present a summary of recent developments. A detailed description of the distinct groups and methods is provided below.

5.1. Feature vector representation

The methods corresponding to this category perform feature engineering to extract a set of features which provide an abstract representation of an executable. Then, the resulting feature vector is fed as input to a feed-forward Neural Network. Notice that the feature vectors extracted by the methods presented in Sections 4.1 and 4.2 can also be used to train feed-forward networks.

Saxe et al. (2015) introduced a malware detection system, powered by a deep neural network, consisting of three main components: (1) the feature extraction component extracts 4 different types of features, byte/entropy histogram features, PE import features, String 2D histogram features, and PE metadata features; (2) the second component consists of the deep neural network classifier; and (3) the third component is the score calibrator, which calibrates the final score. Their system was evaluated on a dataset of 431926 executables retrieved from the Invencea database and achieved a detection rate of 95%.

Huang and Stokes (2016) proposed a multi-task deep learning architecture for malware detection and classification. They extracted a combined feature set consisting of null-terminated tokens, API event plus parameter value, and API trigrams from static and dynamic analysis. Due to the high dimensionality of the input space, mutual information was performed to generate features that best characterize each class. Afterwards, the resulting feature vector was reduced to 50000 features using random projections. Finally, a deep feed-forward Neural Network was trained using the projected feature vector.

Dahl et al. (2013) investigated a malware classification architecture which projects a high-dimensional feature vector to a much lower dimension using random projections. More specifically, random projections reduced the dimensionality of the feature vector from 179000 to 4000 features. Afterwards, a Neural Network classifier learned a non-linear model to classify malware. The system was evaluated on a dataset of 2.6 million labeled samples and achieved an error rate of 0.49%.

Table 6
A side-by-side comparison of the algorithms and input data of the reviewed deep learning methods. Algorithms: Convolutional Neural Network (CNN), Residual Network (ResNet), Autoencoder (AE), Recurrent Neural Network (RNN), Long Short-Term Memory Network (LSTM), Gated Redurrent Unit Network (GRU), Neural Network (NN).

Paper	Feature Type	Classification Algorithm
Saxe et al. (2015)	byte/entropy histogram features PE import features String 2D histogram feature, PE metadata features,	Feed-forward network
Huang and Stokes (2016)	Sequence of API calls events Sequence of NULL-terminated objects API 3-g	Feed-forward network
Dahl et al. (2013)	NULL-terminated patterns API 3-g API 1-g	Feed-forward network
Gibert et al. (2018c)	gray-scale image	CNN
Rezende et al. (2017)	gray-scale image	ResNet-50
Raff et al. (2018a)	bytes sequence	CNN
Krčál et al. (2018)	bytes sequence	CNN
Gibert et al. (2018a)	bytes sequence	Denoising AE + ResNet
Davis et al. (2017)	bytes sequence	CNN + RNN
Gibert et al. (2018b)	structural entropy	CNN
Athiwaratkun et al. (2017)	API call sequence	LSTM, GRU
Kolosnjaji et al. (2016)	API call sequence	CRNN
Gibert et al. (2017)	mnemonics sequence	Shallow CNN
Gibert et al. (2019)	mnemonics sequence	Hierarchical CNN
Prasse et al. (2017)	HTTP traffic	LSTM
AL-Hawawreh et al. (2018)	Network behavior	AE + NN

Table 7
A side-by-side comparison of the dataset characteristics of the reviewed deep learning methods.

Paper	Source	Total Size	Task
Saxe et al. (2015)	Invencea's private malware database	431.926	Detection
Huang and Stokes (2016)	In-house dataset	6.500.000	Detection, Classification
Dahl et al. (2013)	In-house dataset	2.600.000	Detection, Classification
Gibert et al. (2018c)	DB A: MalIMG DB B: Microsoft Malware Classification Challenge	DB A: 9339 DB B: 21741	Classification
Rezende et al. (2017)	MalIMG dataset	9339	Classification
Raff et al. (2018a)	In-house dataset	2.011.786	Detection
Krčál et al. (2018)	AVAST's repository	20.000.000	Detection
Gibert et al. (2018a)	Microsoft Malware Classification Challenge	21.741	Classification
Davis et al. (2017)	–	–	Detection, Classification
Gibert et al. (2018b)	Microsoft Malware Classification Challenge	21.741	Classification
Athiwaratkun et al. (2017)	In-house dataset	75.000	Detection
Kolosnjaji et al. (2016)	VirusShare Maltrieve private collection	–	Detection
Gibert et al. (2017)	Microsoft Malware Classification Challenge	21741	Classification
Gibert et al. (2019)	Microsoft Malware Classification Challenge	21741	Classification
Prasse et al. (2017)	In-house datasets	DB A: 44.348.879 DB B: 129.005.149	Detection
AL-Hawawreh et al. (2018)	DB A: KDD Cup 99 DB B: UNSW-NB15	DB A: 148.517 DB B: 257.673	Detection

5.2. IMG-based representation

Deep learning IMG-based approaches take as input the gray scale image representation of malware's binary content already described in Section 4.1.5. Instead of relying on hand-engineered feature extractors to gather relevant information about the gray scale image, they feed the images into a Convolutional Neural Network architecture that perform both feature learning and classification.

Gibert et al. (2018c) proposed a Convolutional Neural Network architecture composed of three convolutional blocks followed by one fully-connected and the output layer. Each convolutional block consisted of a convolutional operation, the ReLU activation, max-pooling and normalization. The convolutional layers acted as detection filters for the presence of specific features or patterns in the data and the subsequently fully-connected layers combine the learned features and determine a specific target output. Their approach was evaluated on the Microsoft Malware Classification Challenge (Ronen et al., 2018) against hand-crafted feature extractors (Nataraj et al., 2011; Kancherla et al., 2013; Ahmadi et al., 2016) and results demonstrate the superior performance of a deep learning architecture for classifying malware represented as gray scale images. Similarly, Rezende et al. (2017) proposed to use the ResNet-50 architecture with pretrained weights to classify malware images obtained from the MalImg dataset (Nataraj et al., 2011).

5.3. API call traces

Section 4.1.3 presented approaches that used as input a feature vector where each position of the vector indicated whether a particular API function was invoked by the program. However, this kind of feature representation does not take into account the order in which the API functions had been invoked. Alternatively, one can collect the ordered

sequence of API functions invoked and use this information to build classifiers that capture the dependencies in the API function traces.

Athiwaratkun et al. (2017) examined recurrent neural network architectures to better capture long-term dependencies in API call traces. They experimented with the Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) as language models. Their proposed method is composed of two stages. In the first stage, LSTM or GRU are used to construct the features associated with a particular API call trace. In the second stage, these features are classified with either a single fully-connected layer or Logistic Regression with softmax. In addition, they also proposed a character-level convolutional neural network (Zhang et al., 2015). This network takes as input a sequence of 1014 characters maximum length where each character is an event. Sequences with fewer than 1014 characters were padded in the end with end-of-sequence tokens. The character-level network presented consists of 9 layers, 6 convolutional and 3 fully-connected.

Kolosnjaji et al. (2016) investigated the utilization of neural networks to improve the classification of newly retrieved malware samples into a predefined set of malware families. They analyzed two types of neural network layers for modeling system call sequences: convolutional and recurrent layers. They constructed a Neural Network based on convolutional and recurrent layers that combines the convolution of n-grams with full sequential modeling. The input of the network is the API call sequences of malware without API calls repeated more than two times in a row. Each API call was encoded using one-hot encoding to find a unique vector for every API call. The convolutional part of the network consists of a convolutional layer followed by pooling with the convolution acting as feature extractor. The outputs of the convolutional part are connected to the recurrent part which models sequential dependencies in the kernel API traces. To extract the features of highest importance from the LSTM output, mean-pooling was used. Furthermore, they applied dropout to prevent overfitting and a softmax layer to output the class probabilities.

5.4. Instruction traces

Similarly, a program can be modeled as a sequence of instructions executed by the processor.

These sequences of instructions can be obtained from both static and dynamic analysis. On the one hand, it might be possible to obtain them by disassembling the binary executable and processing the resulting disassembled file. On the other hand, the executable can be monitored during runtime and extract the complete sequence of instructions executed on the system. These sequences of instructions can be used to train an end-to-end system to jointly learn the appropriate features and perform classification without having to explicitly enumerate millions of n-grams during training.

Gibert et al. (2017) proposed a Neural Network architecture with an embedding layer, one convolutional layer followed by a max-pooling and an output layer. The convolutional layer could intrinsically learn to detect n-gram-like signatures by learning to detect subsequences of opcodes that are indicative of malware. In addition, depending on the size of the kernel, the convolutional layer allows detecting very long n-gram-like signatures which would be impractical if explicit enumeration of all n-grams were required. This is achieved by defining filters of various sizes. For instance, in their work the convolutional layer contained 64 filters of size $h\,x\,k$ for every $h \in \{2, 3, 4, 5, 6, 7\}$, where k refers to the size of the embedding vector. Then, the maximum value was taken as the feature corresponding to the filter by applying the max-pooling operator over the feature map (also known as global max-pooling). This permits to extract n-gram-like signatures with n ranging from 2 to 7. Finally, the softmax layer outputs the probability distribution over the classes.

Alternatively, Gibert et al. (2019) proposed a Hierarchical Convolutional Neural Network (HCNN) to deal with the hierarchical structure of PE executables. In their work, instead of representing malware as a sequence of instructions, they grouped instructions in the same function to keep the hierarchical structure of a computer program. In consequence, the assembly language instructions were split into functions, where each function was represented by a sequence of mnemonics. In consequence, the hierarchical convolutional neural network captured features at the mnemonic-level and at the function-level.

5.5. Bytes-based representation

The simplest way to represent a computer program is as a sequence of bytes. In other words, each byte is treated as a unit in an input sequence. The main advantage of this representation is that it could be used to represent malware indistinctly of the O.S. and hardware because it is not affected by the file format of the executable, whether it is a Portable Executable (PE) file, or an Executable and Linkable Format (ELF) file, etc. However, representing an executable as a sequence of bytes presents considerable challenges not found in other domains. First, by treating each byte as a unit in a sequence, the size of the resulting byte sequences could consist of several million time steps, making it among the most challenging sequence classification problems. Second, the meaning of any particular byte depends on its context and could encode any type of information such as binary code, human-readable text, images, sound, etc. Third, binary files exhibit various levels of spatial correlation. Adjacent machine instructions tend to be correlated spatially, but, due to jumps and function calls, this correlation might not always hold, as they transfer the control of the program into other addresses in memory and the execution continues from there. Consequently, these discontinuities are maintained on the binary file and in its hexadecimal representation. Therefore, when designing a model to detect malware from a sequence of bytes, (1) its ability to scale well with sequence length and (2) its ability to consider both local and global context while examining an entire file must be taken into account.

Raff et al. (2018a) proposed a Convolutional Neural Network architecture to capture such high level location invariance. They combined the convolutional activations with a global max-pooling before the fully connected layer to allow the model to produce its activations regardless of the location of the detected features in the bytes sequence. Rather than performing convolutions on the raw byte values, they used an embedding layer to map each byte to a fixed length feature vector.

Krčál et al. (2018) explored a deeper architecture composed of an embedding layer followed by four convolutions with strides separated by a max-pooling layer between the second and third convolutional layers, followed by global average pooling and four fully connected layers. They evaluated their model against the MalConv architecture and observed that they slightly increased the performance of the MalConv in their dataset from 94.6% to 96.0% of accuracy. In addition, they enriched the feature vector obtained after the global pooling with hand-crafted features to build a stronger classifier.

As part of an analysis of the likelihood that a given input includes malicious code, an executable can be divided into chunks of code. Afterwards, the information at each chunk can be encoded or codified as a single value. Thus, the resulting output would be a time series $m = \{m_1, m_2, \ldots, m_n\}$, where m_i is the corresponding codification of the i-th chunk and n is the number of chunks into which a binary has been divided. Gibert et al. (2018b) proposed a method for classifying malware represented as a stream of entropy values using Convolutional Neural Networks. Thus, they calculated the entropy of each chunk of code. Afterwards, they applied the single-level discrete wavelet transform to the entropy time series to compress the signal and reduce the noise. The wavelet transformation generated two time series, the approximation coefficients and the details coefficients. Then, both time series were fed into a Convolutional Neural Network that performs feature learning on both time series and classifies a given malware sample into its corresponding family.

Gibert et al. (2018a) encoded the information stored by each chunk using Denoising Autoencoders (DAE). In their work, they first divided a

binary file into contiguous, non-overlapping chunks of fixed size. Afterwards, a denoising autoencoder takes as input every chunk of bytes and projects it into a single value that captures the main factors of variation in the data. The resulting time series is then fed into a Dilated Residual Network which learns descriptive patterns from the encoding of the bytes sequence and assigns a label indicating the family to which the malware belongs.

In Cylance's patent (Davis et al., 2017), instead of codifying the information at a particular chunk into a single value, they implemented a computer method to detect malicious code that comprises three phases:(1) the examination of a sequence of chunks with a Convolutional Neural Network, (2) an analysis of at least some of the chunks using a Recurrent Neural Network (RNN), and (3) determining the likelihood that the input includes malicious code, based on at least some of the chunks analyzed using the RNN.

5.5.1. Network traffic

The methods that fall under this category are those that aim to classify network traffic. More specifically, they try to detect malicious traffic by identifying the type and quantity of traffic flowing through a network.

Prasse et al. (2017) proposed a framework to detect malware on client computers based on the analysis of HTTP traffic. They extracted various features from the sequences of flows sent or received by client computers and domain-name features. Then, an LSTM classifier takes sequences of flows as input and learns to determine whether or not the flows originate from malicious applications.

AL-Hawawreh et al. (2018) proposed an anomaly detection technique for detecting intrusions in Internet Industrial Control Systems (IICSs) based on deep learning models. The system includes an unsupervised learning phase, where a Deep Autoencoder learns normal network behaviors, and a supervised learning phase, where a Deep Neural Network uses the estimated parameters of the Autoencoder to fine-tune its parameters and classify incoming network observations.

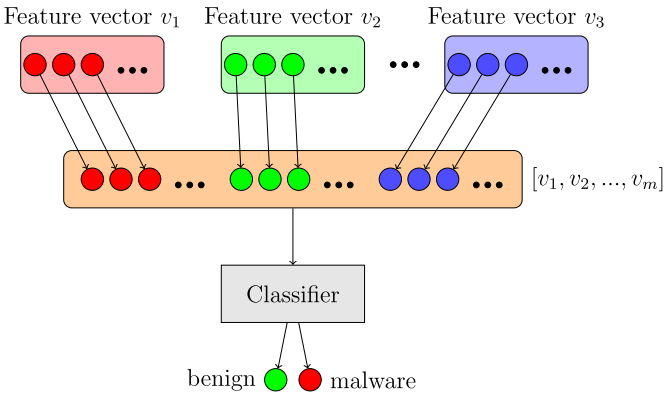

Fig. 6. Early fusion strategy.

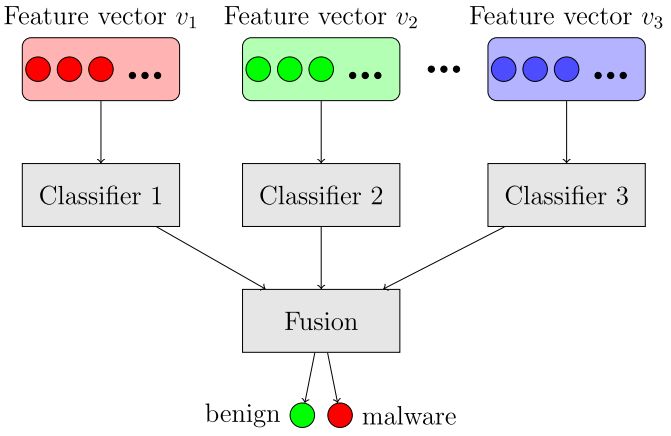

Fig. 7. Late fusion strategy.

6. Multimodal approaches

So far, we have presented approaches that largely rely on one type of feature or modality of data to detect and classify malware. However, malware detection is a research problem characterized as multimodal as it includes multiple modalities of data. Multimodal learning is the field that studies how to be able to interpret such multimodal signals together. Though combining different modalities or types of information for improving performance seems an intuitively appealing task, it is very challenging to combine the varying levels of noise and conflict between modalities. Multimodal approaches can be categorized into three groups considering how the multiple modalities are combined.

- Input-level or early fusion. Early fusion methods create a joint representation of the unimodal features extracted separately from multiple modalities. The simplest way to combine these unimodal feature vectors is to concatenate them to obtain a fused representation. Cf. Fig. 6. Next, a single model is trained to learn the correlation and interactions between the features of each modality. The final outcome of the model can be written as

  $$p = h\left(\left[v_1, v_2, \ldots, v_m\right]\right)$$

  where h denotes the single model, $\left[v_1, v_2, \ldots, v_m\right]$ represents the concatenation of the feature vectors, and m is the number of distinct unimodal feature vectors.
- Decision-level or late fusion. In contrast to early fusion, late fusion methods train one model per modality and fuses the learned decision values with a fusion mechanism such as averaging, voting, a learned model, etc. Cf. Fig. 7. The main advantage of late fusion is that it allows using different models on different modalities, thus being more flexible. In addition, as the predictions for each modality are made separately, it is easier to handle missing modalities. Supposing that model h_i is the decision value on modality i, the final prediction is

  $$p = F\left(h_1(v_1), h_2(v_2), \ldots h_m(v_m)\right)$$

  where F denotes the type of fusion strategy.
- Intermediate fusion. Intermediate fusion methods construct a shared representation by merging the intermediate features obtained by separate machine learning models. Afterwards, these intermediate features are concatenated and then a machine learning model is trained to capture the interactions between modalities. Cf. Fig. 8.

In addition, features extracted from both types of analysis, static and dynamic, can be combined to build more robust classifiers. Approaches that combine static analysis and dynamic analysis are known as hybrid approaches.

On the one hand, static analysis aims at finding malicious characteristics of an executable, app or program without actually running it. Static analysis is faster but suffers from code obfuscation. That is, malicious characteristics can be concealed using different obfuscation techniques (You et al., 2010) or by polymorphic and metamorphic malware (Moser et al., 2007). On the other hand, this obfuscation technique fails at dynamic analysis as it monitors and analyses the runtime behavior of a program during its execution in a controlled environment. But there are some limitations to dynamic analysis. The monitoring process is time consuming and the environment where the program is run must be secured as not to infect the platform. In addition, the controlled environment might be different from the real runtime environment and the malware may behave differently, causing an inexact behavior logging. Moreover, some actions of the program are only triggered if certain conditions are satisfied and may not be detected/activated in a controlled

Table 8
A side-by-side comparison of the features and fusion strategies of the reviewed multimodal and hybrid methods. Algorithms: Support Vector Machine (SVM), Naive Bayes (NB), Decision Tree (DT), Random Forest (RF), Convolutional Neural Network (CNN), Neural Network (NN), K-Nearest Neighbor (K-NN), Passive-Aggressive I (PA-I), Passive-Aggressive II (PA-II), Confidence Weighted Learning (CW), Adaptive Regularization of Weight (AROW), Normal Herd (NHERD), Logistic Regression (LR).

Paper	Fusion Strategy	Static Analysis	Dynamic Analysis	Classification Algorithm
Ahmadi et al. (2016)	Early and late fusion	Bytes 1-g, metadata features, entropy statistics, Haralick and Local Binary Pattern features, ASCII strings, symbol frequencies, opcode 1-g, register's usage, API function calls, section sizes, frequency of keywords	–	Ensemble of Gradient Boosting Trees
Microsoft Challenge winner's solution	Early and late fusion	Opcode 2, 3, 4-g, segment counts, asm pixel intensity, byte 4-g, single byte frequency, function names, derived assembly features	–	Ensemble of Gradient Boosting Trees
Kolosnjaji et al. (2017)	Intermediate fusion	Instruction traces, PE header features, imported functions and DLL files	–	CNN + Feedforward NN
Bayer et al. (2009)	Early fusion	–	API call traces and network traffic	Approximate Nearest Neighbor
Mohaisen and Alrawi (2013)	Early fusion	–	files created, modified or deleted, registry keys created, modified or deleted, destination IP addresses, hosts, TCP and UDP connections, requests and DNS records	SVM, LR, DT and K-NN
Dhammi and Singh (2015)	Early fusion	–	File details, signatures, hosts involved, affected files, registry keys, mutexes, section details, imports and strings	LMT, NB, SVM, Rider and K-NN
Pektaş and Acarman (2017)	Early fusion	–	File system, network and registry features, API call N-grams	PA-I, PA-II, CW, AROW, NHERD
Mohaisen et al. (2015)	Early fusion	–	File system, memory, network and registry based features	SVM, DT, LR, K-NN
Islam et al. (2013)	Early fusion	Function length frequency, string information	API function calls	SVM, RF, DT, Instance-based
Han et al. (2019a)	Early fusion based on semantic blocks	Static API sequences	Dynamic API sequences	K-NN, DT, RF, Extreme Gradient Boosting
Han et al. (2019b)	Early fusion	PE sections size, API sequence, DLL information	IP, port, DNS and domain request, file manipulation operations, registry modification operations	K-NN, DT, RF, Extreme Gradient Boosting
Kumar et al. (2019)	Early fusion	PE file metadata	Network data, system calls, process and registry features	RF, DT, XGBoost, NN, K-NN
Rhode et al. (2019)	Early fusion	Machine metrics	API calls	NN, RF, SVM

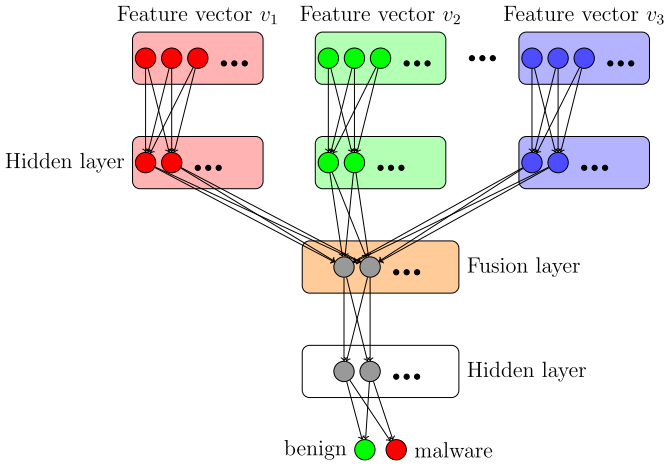

Fig. 8. Intermediate fusion strategy.

environment. Considering the advantages and disadvantages of static and dynamic malware detection, a natural improvement and line of research is to focus on hybrid schemes that combine elements of both. Notice that hybrid approaches also include various modalities of data and could be included under the same category. The main difference between hybrid and multimodal approaches is that hybrid approaches combine features from both static and dynamic analysis while multimodal approaches do not have to.

A summary of the main characteristics of the multimodal and hybrid approaches for malware detection and classification is presented in Tables 8 and 9. A description of each of them is provided below.

Ahmadi et al. (2016) proposed a system that uses different malware features to effectively classify malware samples according to their corresponding family. For each malware sample, they extract a set of content-based and statistical features that reflect the structure of PE files. Then, these features are combined by stacking the feature categories into a single feature vector using a variation of the forward stepwise selection technique. Instead of gradually increasing the feature set by adding features to the model, one by one, they considered all the subset of features belonging to a category. The classification algorithm of their choice was a parallel implementation of the Gradient Boosting Tree classifier, XGBoost. Additionally, they used bagging to boost the classifier stability and accuracy. Their approach was evaluated on the Microsoft Malware Classification Challenge dataset (Ronen et al., 2018) and it achieved accuracy comparable to the winner of the competition[21]

Table 9
A side-by-side comparison of the dataset characteristics of the reviewed multimodal approaches.

Paper	Source	Total Size	Task
Ahmadi et al. (2016)	Microsoft Malware Classification Challenge	21741	Classification
Microsoft Challenge winner's solution	Microsoft Malware Classification Challenge	21741	Classification
Kolosnjaji et al. (2017)	–	22757	Classification
Bayer et al. (2009)	ANUBIS	2658	Classification
Mohaisen and Alrawi (2013)	–	3980	Detection
Dhammi and Singh (2015)	–	1270	Detection
Pektaş and Acarman (2017)	VirusShare	17900	Classification
Mohaisen et al. (2015)	–	115157	Classification
Islam et al. (2013)	–	2939	Detection
Han et al. (2019a)	VirusShare, Windows 7	6471	Classification
Han et al. (2019b)	VirusShare, Windows 7	4250	Classification
Kumar et al. (2019)	MalShare, VirusShare	120000	Classification
Rhode et al. (2019)	VirusShare, Commercial data	6809	Detection

but without requiring the same computational resources. On the other hand, the winning team relied on a large set of well-known features including, but not limited to, byte N-grams and opcode N-grams, which require large computational resources both during the training and the testing phases.

Kolosnjaji et al. (2017) proposed a neural network architecture that consists of convolutional and feed-forward subnetworks. The convolutional subnetwork learns features from sequences of disassembled malicious binaries. Conversely, the feed-forward network takes as input a set of features extracted from the metadata contained in the PE Header and the list of imported functions and their DLL files. Then, the final neural network-based classifier combines the feedforward and convolutional neural network architectures along with their corresponding features into a single network. This network generates the final classification output after aggregating the features learned by both subnetworks.

Bayer et al. (2009) built behavioral profiles of malware based on the system calls, their dependencies and network activities. This generalized representation serves as input to a clustering algorithm that groups malware samples that exhibit similar behavior. Clustering malware is a multi-step process. The first step is the automated analysis of the executables performed by an extended version of ANUBIS.[22] The second step is the extraction of the behavioral profile. Lastly, in the third step samples that exhibit similar behavior are grouped in the same cluster using an approximate, probabilistic approach based on locality sensitive hashing (Indyk and Motwani, 1998).

Mohaisen and Alrawi (2013) proposed a behavior based approach for identifying malware belonging to the Zeus family. The Zeus banking trojan is a form of malware that targets the Windows OS and is often used to steal money and credentials from the infected victim. For classification purposes, a set of 65 unique and robust features are extracted including files created, modified or deleted, registry keys created, modified or deleted, destination IP addresses, ports, TCP and UDP connections, requests, DNS records, etc. Then, the resulting feature vector is used to evaluate the performance of various M.L. algorithms such as SVM, LR, DT and K-NN.

Dhammi and Singh (2015) proposed a malware detection system based on the dynamic analysis of malware using the Cuckoo sandbox. Their approach extracted various features from the malware execution such as file details, signatures, hosts involved, affected files, registry keys, mutexes, section details, imports and strings. All the features obtained from Cuckoo are mapped into an Attribute Relation File Format (ARFF) file, and later, the resulting ARRF file is fed into WEKA (Hall et al., 2009) for classification.

Pektaş and Acarman (2017) presented a malware classification system based on runtime behavior by applying online machine learning. The system entails three stages. The first stage consists of monitoring the behavior of the file in sandbox environments; VirMon and Cuckoo. During the second stage, feature extraction is applied to build a feature vector consisting of features based on the file system, network and registry activities and API call N-grams. Finally, the third stage performs classification using online learning algorithms.

Mohaisen et al. (2015) presented AMAL, an automated behavior-based malware analysis system that provides tools to collect behavioral features that characterize malware based on the usage of the file system, memory, network and registry. Then, the resulting feature vector is used to perform classification with Support Vector Machine, Decision Tree, Logistic Regression and K-Nearest Neighbor algorithms.

Islam et al. (2013) presented a method integrating static and dynamic features into a single classification system. For each executable file, they extracted and converted to vector representations both function length frequency and printable string information. After running the executables and logging the Windows API calls, they extracted API features comprising API function names and parameters. Then, all feature vectors are combined into a single vector for each executable. Next, the resulting vector is used as input to four base classifiers: Support Vector Machine, Random Forest, Decision Tree and Instance-based.

Han et al. (2019a) built a malware detection framework based on the correlation and fusion of static and dynamic API call sequences. In their work, they explored the difference and relation between static and dynamic API call sequences by defining a number of types of malicious behaviors. After correlation and fusion, a hybrid feature vector space is established for detection and classification. To evaluate the effectiveness of their approach, they trained four classifiers to detect/classify malware including K-Nearest Neighbor, Decision Tree, Random Forest and Extreme Gradient Boosting.

Han et al. (2019b) presented MalInsight, a malware detection framework based on programs profiling of: (1) their basic structure, (2) their low-level behavior, and (3) their high-level behavior. These three aspects reflect structural features; the primary operations interacting with the OS, files, the registry and the network. The resulting feature set is used to train various machine learning classifiers: K-Nearest Neighbor, Decision Tree, Random Forests and Extreme Gradient Boosting. These classifiers were evaluated on a dataset consisting of 4250 samples obtained from VirusShare and from the Windows 7 Pro operative system. Results show accuracy of 97.21% in detecting unknown malware.

Kumar et al. (2019) used a combination of static and dynamic approaches to classify malware into types in the initial 4 s of its execution using a Random Forest classifier. Stopping the process early before the analysis is fully executed is known as early-stage detection. From static analysis they extracted information from the PE header such as file header, optional header and section header. They also extracted information from the section table and sections such as the number of sections, their size, the section virtual address, etc. From dynamic anal-

[21] http://blog.kaggle.com/2015/05/26/microsoft-_malware-_winners-_interview-_1st-_place-_no-_to-_overfitting/.
[22] http://anubis.iseclab.org.

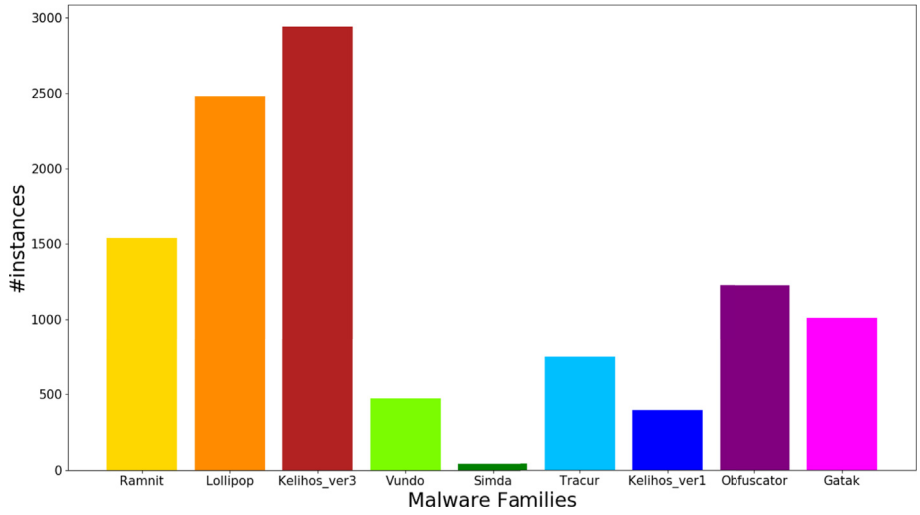

Fig. 9. Class distribution in the Microsoft Malware Classification Challenge dataset.

ysis they extracted features based on critical resources such as network data, system calls, process and registry. Afterwards, the feature set is reduced using the Information Gain algorithm. Finally, the resulting feature vector is used for classification purposes by training a Random Forest, Decision Tree, XGBoost, Neural Network and K-NN classifiers.

Rhode et al. (2019) collected two types of features: (1) API calls and (2) machine metrics. The Cuckoo sandbox was used to collect the API call features while Psutil library was used to collect the machine metrics. Machine metrics include user CPU usage, system CPU usage, memory use, swap use, bytes received and transmitted, number of packets received and transmitted, total number of processes, maximum process ID and time in seconds since execution began. Then, the two feature vectors are fused into a single vector that is used to detect malware using a Neural Network, Random Forest or Support Vector Machine as classifiers.

7. Research issues and challenges

This section presents some of the issues and challenges faced by security researchers. It is structured as follows: Section 7.1 presents the class imbalance problems. Section 7.2 reviews the availability of public benchmarks of malware for research. Finally, Section 7.3 discusses the problem of concept drift and presents various adversarial learning techniques to fool machine learning detectors.

7.1. 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.

Additionally, there are various disciplines including fraud detection, malware detection, malware classification, medical diagnosis, etc, where it is common to have a disproportional number of samples per class. For instance, the number of benign samples might not be proportionally equal to the number of malicious samples, or the number of samples belonging to one family might far exceed the number of samples from other families. This is known as the class imbalance problem (Japkowicz and Stephen, 2002; Guo et al., 2008).

By way of an example, let's look at the distribution of classes of the Microsoft dataset in Fig. 9. Families Kelihos_ver3, Lollipop and Ramnit have 2942, 2478, 1541 samples, respectively. On the other hand, families Simda and Kelihos_ver1 have 42 and 398 samples, respectively. 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.

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. In these cases, accuracy is a misleading measure. It may be desirable to select a less accurate model but with greater predictive power. For problems like this, additional measures are required to evaluate a classifier such as precision 1, recall 2 and the F1 score 3. Alternatively, the Receiver Operating Characteristic (ROC) curve graphically illustrates the discriminative ability of a binary classifier.

Precision (P) is the number of true positives (T_p) over the number of true positives plus the number of false positives (F_p).

$$P = \frac{T_p}{T_p + F_p}. \tag{1}$$

Recall (R) is the number of true positives (T_p) over the number of true positives plus the number of false negatives (F_n).

$$R = \frac{T_p}{T_p + F_n}. \tag{2}$$

The F1 score is the weighted average of precision, defined as following:

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R}. \tag{3}$$

Finally, the ROC curve is created by plotting the True Positive Rate (TPR) or recall against the False Positive Rate. The FPR is also known as the probability of false alarm and can be calculated as (1-Specificity) where Specificity is equal to $\frac{TN}{TN+FP}$. The higher the AUC, the better the model is at predicting the correct label of classes.

7.2. Open and public benchmarks

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. These include digit classification, image labelling, speech recognition, etc. This situation has been exacerbated by legal restrictions. Even though malware binaries are shared generously through web sites such as VirusShare and VX Heaven, benign binaries are often protected by copyright laws that prevent sharing. 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. In addition, unlike other domains where data may be labeled very quickly and in many cases by a non-expert, determining whether a file is malicious or benign can be a time-consuming process, even for security experts. Furthermore, services like VirusTotal specifically restrict the public sharing of vendor antimalware labels.

The aforementioned issues render it impossible to meaningfully compare accuracy numbers across works, as different datasets are used with different labeling procedures. At the present time, the only standard benchmark available to the research community regarding Windows Portable Executables is the one provided by Microsoft (Ronen et al., 2018) for the Big Data Innovators Gathering Anti-Malware Prediction Challenge. The dataset is hosted on Kaggle and includes almost half a terabyte of malware consisting of around 20 K malicious samples from nine families. Each sample is comprised of two files: (1) the hexadecimal representation of the malware's binary content and (2) their corresponding disassembled file. Unfortunately, the byte code does not include the headers and thus, it is not possible to analyze dynamically the executables or to reproduce the disassembly process. In consequence, researchers are constrained to using only the provided byte code and disassembly files (generated with the IDA Pro disassembler).

7.3. 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. Supervised learning is the machine learning task of learning a function that maps an input to an output based on a set of input-output samples. Technically speaking, it is the problem of approximating a mapping function (f) given input data (x) to predict an output value (y), $y = f(x)$. Traditional machine learning applications such as digit classification, text categorization or speech recognition, assume that training data is sampled from a stationary population. 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.

Software applications, including malware, naturally evolve over time due to changes resulting from adding features, fixing bugs, porting to new environments and platforms (Lehman, 1996). These changes are expected to be introduced relatively infrequently. Additionally, successive versions of the software are expected to be highly similar to previous versions, with few exceptions such as when the code base undergoes significant refactoring and there are changes in the compilers or libraries linked to the software. Moreover, the similarity between previous and future versions is expected to degrade slowly over time. In consequence, the prediction quality decays over time as malware evolves and new variants and families appear (Jordaney et al., 2017). 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. Existing solutions (Kantchelian et al., 2013; Gama et al., 2014) aim at periodically retrain the model with the hope that it will automatically adapt to changes in malware over time. The process of retraining the model can be done from scratch, partially and incrementally, were incremental retraining refers to the process of retraining a given model with new labeled malware samples and all previous training samples without forgetting the knowledge obtained from prior datasets.

7.4. Adversarial learning

Malware is pushed to evolve in order to survive and operate. That is, malicious software has to constantly evolve to avoid detection by anti-malware engines. In consequence, malware writers are well-motivated to intentionally seek evasion by employing a wide range of obfuscation techniques (You et al., 2010; OKane et al., 2011).

To put it in the machine learning context, an attacker's aim is to fool the machine learning detector by camouflaging a piece of malware in feature space by inducing a feature representation highly correlated to benign behavior. The ability of the attacker to bypass machine learning solutions is related to their knowledge about features and machine learning models to target. 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.

Adversarial machine learning (Huang et al., 2011) is a technique employed to attempt to fool machine learning by automatically crafting adversarial examples. That is, samples with small, intentional feature perturbations that cause a machine learning model to make an incorrect prediction. Machine learning-based detectors are vulnerable to adversarial examples, and the application of machine learning to the cybersecurity domain does not constitute an exception. For a detailed overview of the evolution of adversarial machnine learning over the past decade we refer to Biggio and Roli (2018). They reviewed the work done in the context of various applications, including computer security and its notion of arms race and proposed a comprehensive threat model that accounts for the presence of the attacker during the system design. Recent classifiers proposed for malware detection, have indeed shown to be easily fooled by well-crafted adversarial manipulations (Demetrio et al., 2019; Chen et al., 2017; Huang et al., 2018; Suciu et al., 2018; Maiorca et al., 2019). Chen et al. (2017) explored adversarial machine learning to attack a malware detector based on the input of Windows Application Programming Interface (API) calls extracted from the PE files.

Suciu et al. (2018) analyzed various append-based strategies to generate adversarial examples to conceal malware and bypass the MalConv (Raff et al., 2018a) model.

Furthermore, Demetrio et al. (2019) proposed a novel attack algorithm to generate adversarial malware binaries which only change a few tens of bytes of the file header. Their algorithm was evaluated against MalConv. They found that MalConv learns discriminative features mostly from the characteristics of the file header and used their findings to exploit and bypass the model. Contrarily, Maiorca et al. (2019) explored the types of adversarial attacks that have exploited the vulnerabilities of the components of PDFs to bypass malware detectors, including JavaScript-based attacks, ActionScript-based attacks and file embedding-based attacks.

7.5. Interpretability of the models

The interpretation of machine learning models is a new and open challenge (Shirataki and Yamaguchi, 2017; Gilpin et al., 2018). Most of the models used at the present time are treated as a black box. This black box is given an input X and it produces an output Y through a sequence of operations hardly understandable to a human. 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. For this reason, cybersecurity analysts have preferred solutions that are more interpretable and understandable such as rule-based and signature-based systems rather than neural-based methods because they are easier to tune and optimize to mitigate and control the effect of false positives and false negatives. However, there is no work in the literature that investigates the interpretability of machine learning models for malware detection and classification.

8. Conclusions

This paper presents a systematic review of malware detection and classification approaches using machine learning. To sum up, a total of 67 research papers for tackling the problem of malware detection and classification on the Windows platform are reviewed. The reviewed papers are compared and analyzed according to various essential factors including the input features, the classification algorithm, the characteristics of the dataset and the objective task. There are four main contributions of our work.

First, we provide a detailed description of the methods and features in a traditional machine learning workflow, from the feature extraction, selection and reduction steps to classification. The traditional approaches are classified into three main categories: (1) static-based and (2) dynamic-based approaches and (3) hybrid approaches. On the one hand, static-based approaches extract features derived from a piece of program without involving its execution. On the other hand, dynamic-based approaches include those approaches that extract features from the execution of malware during runtime. Lastly, hybrid approaches are those that combine static and dynamic analysis to extract features.

Second, it arranges the existing literature on malware detection through deep learning and provides a comparative analysis of the approaches based on the network architecture and its input. Deep learning approaches are grouped considering the type of input of the networks: (1) methods that perform feature engineering to extract a feature vector representing the executable; (2) methods that take the gray scale representation of an executable as input; (3) methods that are fed with the sequence of API function invocations; (4) methods that model a program as a sequence of instructions; (5) methods that represent a computer program as a sequence of bytes; and (6) methods that aim to classify a program from its network traffic.

Third, it introduces new directions of research and present classifiers that rely on more than one type of feature or modality of data to detect malware. It organizes multimodal approaches into three groups, depending on how the different modalities of data are fused: (i) early-fusion methods create a joint representation of the unimodal feature vectors; (ii) late-fusion methods train one model per modality and fuses the output decision values; and (iii) intermediate-fusion methods construct a shared representation by merging the intermediate features obtained by separate models.

Fourth, it discusses the most important research issues and challenges faced by researchers. Special emphasis is placed on the problem of concept drift and the challenges of adversarial learning. Furthermore, it examines the status of the benchmarks used by the scientific community to evaluate the performance of their methods and reviews the problem of class imbalance.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that may appear to influence the work reported in this paper.

Acknowledgements

This research has been partially funded by the Spanish MICINN Projects TIN2015-71799-C2-2-P, ENE2015-64117-C5-1-R, and is supported by the University of Lleida. This research article has received a grant (2019 call) from the University of Lleida Language Institute to review the English.

References

Ahmadi, M., Ulyanov, D., Semenov, S., Trofimov, M., Giacinto, G., 2016. Novel feature extraction, selection and fusion for effective malware family classification. CODASPY 16. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. ACM, New York, NY, USA, pp. 183–194, https://doi.org/10.1145/2857705.2857713.

AL-Hawawreh, M., Moustafa, N., Sitnikova, E., 2018. Identification of malicious activities in industrial internet of things based on deep learning models. Journal of Information Security and Applications. 41, 1–11. http://www.sciencedirect.com/science/article/pii/S2214212617306002.

Anderson, B., Quist, D., Neil, J., Storlie, C., Lane, T., Nov 2011. Graph-based malware detection using dynamic analysis. J. Comput. Virol. 7 (4), 247–258, https://doi.org/10.1007/s11416-_011-_0152-_x.

Athiwaratkun, B., Stokes, J.W., March 2017. Malware classification with lstm and gru language models and a character-level cnn. In: 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 2482–2486.

Bayer, U., Comparetti, P.M., Hlauschek, C., Krgel, C., Kirda, E., 2009. Scalable, behavior-based malware clustering. In: NDSS. The Internet Society. p. Section4 http://dblp.uni-_trier.de/db/conf/ndss/ndss2009.html#BayerCHKK09.

Baysa, D., Low, R.M., Stamp, M., Nov 2013. Structural entropy and metamorphic malware. Journal of Computer Virology and Hacking Techniques 9 (4), 179–192, https://doi.org/10.1007/s11416-_013-_0185-_4.

Bazrafshan, Z., Hashemi, H., Fard, S.M.H., Hamzeh, A., May 2013. A survey on heuristic malware detection techniques. In: The 5th Conference on Information and Knowledge Technology, pp. 113–120.

Bekerman, D., Shapira, B., Rokach, L., Bar, A., 2015. Unknown malware detection using network traffic classification. 09. In: 2015 IEEE Conference on Communications and Network Security (CNS), pp. 134–142.

Biggio, B., Roli, F., 2018. Wild patterns: ten years after the rise of adversarial machine learning. Pattern Recognit. 84, 317–331. http://www.sciencedirect.com/science/article/pii/S0031320318302565.

Boukhtouta, A., Mokhov, S.A., Lakhdari, N.-E., Debbabi, M., Paquet, J., May 2016. Network malware classification comparison using dpi and flow packet headers. Journal of Computer Virology and Hacking Techniques 12 (2), 69–100, https://doi.org/10.1007/s11416-_015-_0247-_x.

Carlin, D., Cowan, A., O'Kane, P., Sezer, S., 2017a. The effects of traditional anti-virus labels on malware detection using dynamic runtime opcodes. IEEE Access 5, 17742–17752.

Carlin, D., O'Kane, P., Sezer, S., 2017b. Dynamic Analysis of Malware Using Run-Time Opcodes. Springer International Publishing, Cham, https://doi.org/10.1007/978-_3-_319-_59439-_2\T1\textbackslash_4.

Chen, L., Ye, Y., Bo, ai, T., Sep. 2017. Adversarial machine learning in malware detection: arms race between evasion attack and defense. In: 2017 European Intelligence and Security Informatics Conference (EISIC), pp. 99–106.

Corporation, S., 2018. Symantec 2018 Internet Security Threat Report. Tech. rep. Symantec Corporation, https://www.symantec.com/content/dam/symantec/docs/reports/istr-_23-_executive-_summary-_en.pdf.

Dahl, G.E., Stokes, J.W., Deng, L., Yu, D., May 2013. Large-scale malware classification using random projections and neural networks. In: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 3422–3426.

Davis, A., Wolff, M., Wojnowicz, M., Soeder, D.A., Zhao, X., 2017. Neural Attention Mechanisms for Malware Analysis. 07. https://patentimages.storage.googleapis.com/4f/e5/74/b62fd3b08788bd/US9705904.pdf.

Demetrio, L., Biggio, B., Lagorio, G., Roli, F., Armando, A., 2019. Explaining Vulnerabilities of Deep Learning to Adversarial Malware Binaries. CoRR abs/1901.03583. http://arxiv.org/abs/1901.03583.

Dhammi, A., Singh, M., Aug 2015. Behavior analysis of malware using machine learning. In: 2015 Eighth International Conference on Contemporary Computing (IC3), pp. 481–486.

Dinaburg, A., Royal, P., Sharif, M., Lee, W., 2008. Ether: malware analysis via hardware virtualization extensions. CCS 08. In: Proceedings of the 15th ACM Conference on Computer and Communications Security. ACM, New York, NY, USA, pp. 51–62, https://doi.org/10.1145/1455770.1455779.

Ding, Y., Yuan, X., Tang, K., Xiao, X., Zhang, Y., 2013. A fast malware detection algorithm based on objective-oriented association mining. Comput. Secur. 39, 315–324. http://www.sciencedirect.com/science/article/pii/S0167404813001259.

Eskandari, M., Hashemi, S., 2011. Metamorphic malware detection using control flow graph mining. 06 International Journal of Computer Science and Network Security 11 (12).

Faruki, P., Laxmi, V., Gaur, M.S., Vinod, P., 2012. Mining control flow graph as api call-grams to detect portable executable malware. SIN 12. In: Proceedings of the Fifth International Conference on Security of Information and Networks. ACM, New York, NY, USA, pp. 130–137, https://doi.org/10.1145/2388576.2388594.

Fraley, J.B., Cannady, J., March 2017. The promise of machine learning in cybersecurity. In: SoutheastCon 2017, pp. 1–6.

Fuyong, Z., Tiezhu, Z., July 2017. Malware detection and classification based on n-grams attribute similarity. In: 2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC), vol. 1, pp. 793–796.

Galal, H.S., Mahdy, Y.B., Atiea, M.A., May 2016. Behavior-based features model for malware detection. Journal of Computer Virology and Hacking Techniques 12 (2), 59–67, https://doi.org/10.1007/s11416-_015-_0244-_0.

Gama, J.a., liobait, I., Bifet, A., Pechenizkiy, M., Bouchachia, A., Mar. 2014. A survey on concept drift adaptation. ACM Comput. Surv. 46 (4), https://doi.org/10.1145/2523813 44:144:37.

Ghiasi, M., Sami, A., Salehi, Z., Sep. 2012. Dynamic malware detection using registers values set analysis. In: 2012 9th International ISC Conference on Information Security and Cryptology, pp. 54–59.

Ghiasi, M., Sami, A., Salehi, Z., 2015. Dynamic vsa: a framework for malware detection based on register contents. Eng. Appl. Artif. Intell. 44, 111–122. http://www.

sciencedirect.com/science/article/pii/S0952197615001190.
Gibert, D., Bjar, J., Mateu, C., Planes, J., Solis, D., Vicens, R., 2017. Convolutional neural networks for classification of malware assembly code. In: Recent Advances in Artificial Intelligence Research and Development - Proceedings of the 20th International Conference of the Catalan Association for Artificial Intelligence, Deltebre, Terres de l'Ebre, Spain, October 25-27, 2017, pp. 221–226, https://doi.org/10.3233/978-_1-_61499-_806-_8-_221.
Gibert, D., Mateu, C., Planes, J., 2018a. An end-to-end deep learning architecture for classification of malware's binary content. In: Krkov, V., Manolopoulos, Y., Hammer, B., Iliadis, L., Maglogiannis, I. (Eds.), Artificial Neural Networks and Machine Learning ICANN 2018. Springer International Publishing, Cham, pp. 383–391.
Gibert, D., Mateu, C., Planes, J., Vicens, R., 2018b. Classification of malware by using structural entropy on convolutional neural networks. In: IAAI Conference on Artificial Intelligence, pp. 7759–7764. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16133.
Gibert, D., Mateu, C., Planes, J., Vicens, R., Aug 2018c. Using convolutional neural networks for classification of malware represented as images. Journal of Computer Virology and Hacking Techniques, https://doi.org/10.1007/s11416-_018-_0323-_0.
Gibert, D., Mateu, C., Planes, J., 2019. A hierarchical convolutional neural network for malware classification. In: The International Joint Conference on Neural Networks 2019. IEEE, pp. 1–8.
Gilpin, L.H., Bau, D., Yuan, B.Z., Bajwa, A., Specter, M., Kagal, L., 2018. Explaining Explanations: an Approach to Evaluating Interpretability of Machine Learning. CoRR abs/1806.00069 http://arxiv.org/abs/1806.00069.
Guo, X., Yin, Y., Dong, C., Yang, G., Zhou, G., Oct 2008. On the class imbalance problem. In: 2008 Fourth International Conference on Natural Computation, vol. 4, pp. 192–201.
Hall, M.A., 1999. Correlation-based Feature Selection for Machine Learning. Ph.D. thesis. The University of Waikato.
Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H., Nov. 2009. The weka data mining software: an update. SIGKDD Explor. Newsl. 11 (1), 10–18, https://doi.org/10.1145/1656274.1656278.
Han, W., Xue, J., Wang, Y., Huang, L., Kong, Z., Mao, L., 2019a. Maldae: detecting and explaining malware based on correlation and fusion of static and dynamic characteristics. Comput. Secur. 83, 208–233. http://www.sciencedirect.com/science/article/pii/S016740481831246X.
Han, W., Xue, J., Wang, Y., Liu, Z., Kong, Z., 2019b. Malinsight: a systematic profiling based malware detection framework. J. Netw. Comput. Appl. 125, 236–250. http://www.sciencedirect.com/science/article/pii/S1084804518303503.
Hassen, M., Chan, P.K., 2017. Scalable function call graph-based malware classification. CODASPY 17. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy. ACM, New York, NY, USA, pp. 239–248, https://doi.org/10.1145/3029806.3029824.
Hu, X., Shin, K.G., Bhatkar, S., Griffin, K., 2013. Mutantx-s: scalable malware clustering based on static features. In: Presented as Part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13). USENIX, San Jose, CA, pp. 187–198. https://www.usenix.org/conference/atc13/technical-_sessions/presentation/hu.
Huang, W., Stokes, J.W., 2016. Mtnet: a multi-task neural network for dynamic malware classification. In: Caballero, J., Zurutuza, U., Rodrguez, R.J. (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment. Springer International Publishing, Cham, pp. 399–418.
Huang, L., Joseph, A.D., Nelson, B., Rubinstein, B.I., Tygar, J.D., 2011. Adversarial machine learning. AISec 11. In: Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence. ACM, New York, NY, USA, pp. 43–58, https://doi.org/10.1145/2046684.2046692.
Huang, A., Al-Dujaili, A., Hemberg, E., O'Reilly, U., 2018. Adversarial Deep Learning for Robust Detection of Binary Encoded Malware. CoRR abs/1801.02950. http://arxiv.org/abs/1801.02950.
Indyk, P., Motwani, R., 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. STOC 98. In: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing. ACM, New York, NY, USA, pp. 604–613, https://doi.org/10.1145/276698.276876.
Islam, R., Tian, R., Batten, L.M., Versteeg, S., 2013. Classification of malware based on integrated static and dynamic features. J. Netw. Comput. Appl. 36 (2), 646–656. http://www.sciencedirect.com/science/article/pii/S1084804512002214.
Jain, S., Meena, Y.K., 2011. Byte level ngram analysis for malware detection. In: Venugopal, K.R., Patnaik, L.M. (Eds.), Computer Networks and Intelligent Computing. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 51–59.
Japkowicz, N., Stephen, S., 2002. The class imbalance problem: a systematic study. Intell. Data Anal. 429–449.
Jordaney, R., Sharad, K., Dash, S.K., Wang, Z., Papini, D., Nouretdinov, I., Cavallaro, L., Aug. 2017. Transcend: detecting concept drift in malware classification models. In: 26th USENIX Security Symposium (USENIX Security 17). USENIX Association, Vancouver, BC, pp. 625–642. https://www.usenix.org/conference/usenixsecurity17/technical-_sessions/presentation/jordaney.
Kancherla, K., Mukkamala, S., April 2013. Image visualization based malware detection. In: 2013 IEEE Symposium on Computational Intelligence in Cyber Security (CICS), pp. 40–44.
Kantchelian, A., Afroz, S., Huang, L., Islam, A.C., Miller, B., Tschantz, M.C., Greenstadt, R., Joseph, A.D., Tygar, J.D., 2013. Approaches to adversarial drift. AISec 13. In: Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security. ACM, New York, NY, USA, pp. 99–110, https://doi.org/10.1145/2517312.2517320.
Kheir, N., 2013. Behavioral classification and detection of malware through http user agent anomalies. Journal of Information Security and Applications 18 (1), 2–13 sETOP2012 and FPS2012 Special Issue, http://www.sciencedirect.com/science/article/pii/S2214212613000331.
Kinable, J., Kostakis, O., Nov 2011. Malware classification based on call graph clustering. J. Comput. Virol. 7 (4), 233–245, https://doi.org/10.1007/s11416-_011-_0151-_y.
Kolias, C., Kambourakis, G., Stavrou, A., Voas, J., 2017. Ddos in the iot: Mirai and other botnets. Computer 50 (7), 80–84.
Kolosnjaji, B., Zarras, A., Webster, G., Eckert, C., 2016. Deep learning for classification of malware system call sequences. In: Kang, B.H., Bai, Q. (Eds.), AI 2016: Advances in Artificial Intelligence. Springer International Publishing, Cham, pp. 137–149.
Kolosnjaji, B., Eraisha, G., Webster, G., Zarras, A., Eckert, C., May 2017. Empowering convolutional networks for malware classification and analysis. In: 2017 International Joint Conference on Neural Networks (IJCNN), pp. 3838–3845.
Konopisky, D., 10 2018. Malware Detection in Application Based on Presence of Computer Generated Strings. https://patentscope.wipo.int/search/en/detail.jsf?docIdWO2018177602&tabPCTBIBLIO&queryString&recNum29&maxRec71152078.
Krl, M., vec, O., Blek, M., Jaek, O., 2018. Deep Convolutional Malware Classifiers Can Learn from Raw Executables and Labels Only. https://openreview.net/forum?idHkHrmM1PM.
Kumar, N., Mukhopadhyay, S., Gupta, M., Handa, A., Shukla, K.S, Aug 2019. Malware classification using early stage behavioral analysis. In: 2019 14th Asia Joint Conference on Information Security (AsiaJCIS), pp. 16–23.
Lee, J., Im, C., Jeong, H., 2011. A study of malware detection and classification by comparing extracted strings. ICUIMC 11. In: Proceedings of the 5th International Conference on Ubiquitous Information Management and Communication. ACM, New York, NY, USA, p. 75, https://doi.org/10.1145/1968613.1968704. 175:4.
Lehman, M.M., 1996. Laws of software evolution revisited. In: Montangero, C. (Ed.), Software Process Technology. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 108–124.
Ligh, M., Adair, S., Hartstein, B., Richard, M., 2010. Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code. Wiley Publishing.
Lyda, R., Hamrock, J., March 2007. Using entropy analysis to find encrypted and packed malware. IEEE Security Privacy 5 (2), 40–45.
Maiorca, D., Biggio, B., Giacinto, G., Aug. 2019. Towards adversarial malware detection: lessons learned from pdf-based attacks. ACM Comput. Surv. 52 (4), https://doi.org/10.1145/3332184 78:178:36.
Mohaisen, A., Alrawi, O., 2013. Unveiling zeus: automated classification of malware samples. In: Proceedings of the 22Nd International Conference on World Wide Web. WWW 13 Companion. ACM, New York, NY, USA, pp. 829–832, https://doi.org/10.1145/2487788.2488056.
Mohaisen, A., Alrawi, O., Mohaisen, M., 2015. Amal: high-fidelity, behavior-based automated malware analysis and classification. Comput. Secur. 52, 251–266. http://www.sciencedirect.com/science/article/pii/S0167404815000425.
Monnappa, 2018. Learning Malware Analysis: Explore the Concepts, Tools, and Techniques to Analyze and Investigate Windows Malware. Packt Publishing.
Moser, A., Kruegel, C., Kirda, E., Dec 2007. Limits of static analysis for malware detection. In: Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), pp. 421–430.
Moskovitch, R., Stopel, D., Feher, C., Nissim, N., Elovici, Y., June 2008. Unknown malcode detection via text categorization and the imbalance problem. In: 2008 IEEE International Conference on Intelligence and Security Informatics, pp. 156–161.
Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B.S., 2011. Malware images: visualization and automatic classification. VizSec 11. In: Proceedings of the 8th International Symposium on Visualization for Cyber Security. ACM, New York, NY, USA, https://doi.org/10.1145/2016904.2016908. pp. 4:14:7.
OKane, P., Sezer, S., McLaughlin, K., Sept 2011. Obfuscation: the hidden malware. IEEE Security Privacy 9 (5), 41–47.
on Cybersecurity for the 44th Presidency, C. C., Langevin, J., Lewis, J., for Strategic, C., International Studies (Washington, D, 2010. A Human Capital Crisis in Cybersecurity: Technical Proficiency Matters. a White Paper of the CSIS Commission on Cybersecurity for the 44th Presidency. Center for Strategic and International Studies. https://books.google.es/books?idZa-_MnQAACAAJ.
Okane, P., Sezer, S., McLaughlin, K., May, 2016. Detecting obfuscated malware using reduced opcode set and optimised runtime trace. Security Informatics 5 (1), 2, https://doi.org/10.1186/s13388-_016-_0027-_2.
Pekta, A., Acarman, T., 2017. Classification of malware families based on runtime behaviors. Journal of Information Security and Applications. 37, 91–100. http://www.sciencedirect.com/science/article/pii/S2214212617301643.
Perdisci, Roberto, Wenke Lee, G.O., 01 2015. Method and System for Network-Based Detecting of Malware from Behavioral Clustering. https://patentimages.storage.googleapis.com/42/60/cd/37786f1ef6be24/US20150026808A1.pdf.
Prasse, P., Machlica, L., Pevn, T., Havelka, J., Scheffer, T., 2017. Malware detection by analysing encrypted network traffic with neural networks. In: Ceci, M., Hollmn, J., Todorovski, L., Vens, C., Deroski, S. (Eds.), Machine Learning and Knowledge Discovery in Databases. Springer International Publishing, Cham, pp. 73–88.
Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., Nicholas, C.K., 2018a. Malware detection by eating a whole EXE. In: The Workshops of the the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018, pp. 268–276. https://aaai.org/ocs/index.php/WS/AAAIW18/paper/view/16422.
Raff, E., Zak, R., Cox, R., Sylvester, J., Yacci, P., Ward, R., Tracy, A., McLean, M., Nicholas, C., Feb, 2018b. An investigation of byte n-gram features for malware classification. Journal of Computer Virology and Hacking Techniques 14 (1), 1–20, https://doi.org/10.1007/s11416-_016-_0283-_1.
Razak, M.F.A., Anuar, N.B., Salleh, R., Firdaus, A., 2016. The rise of malware: bibliometric analysis of malware study. J. Netw. Comput. Appl. 75, 58–76. http://

www.sciencedirect.com/science/article/pii/S1084804516301904.
Rezende, E., Ruppert, G., Carvalho, T., Ramos, F., de Geus, P., Dec 2017. Malicious software classification using transfer learning of resnet-50 deep neural network. In: 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA), pp. 1011–1014.
Rhode, M., Tuson, L., Burnap, P., Jones, K., June 2019. Lab to soc: robust features for dynamic malware detection. In: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Industry Track, pp. 13–16.
Rieck, K., Trinius, P., Willems, C., Holz, T., Dec. 2011. Automatic analysis of malware behavior using machine learning. J. Comput. Secur. 19 (4), 639–668. http://dl.acm.org/citation.cfm?id2011216.2011217.
Ronen, R., Radu, M., Feuerstein, C., Yom-Tov, E., Ahmadi, M., Feb. 2018. Microsoft malware classification challenge. ArXiv e-prints.
Salehi, Z., Sami, A., Ghiasi, M., 2017. Maar: robust features to detect malicious activity based on api calls, their arguments and return values. Eng. Appl. Artif. Intell. 59, 93–102. http://www.sciencedirect.com/science/article/pii/S0952197616302512.
Sami, A., Yadegari, B., Rahimi, H., Peiravian, N., Hashemi, S., Hamze, A., 2010. Malware detection based on mining api calls. In: Proceedings of the 2010 ACM Symposium on Applied Computing. SAC 10. ACM, New York, NY, USA, pp. 1020–1025, https://doi.org/10.1145/1774088.1774303.
Santos, I., Brezo, F., Ugarte-Pedrero, X., Bringas, P.G., 2013. Opcode sequences as representation of executables for data-mining-based unknown malware detection. Inf. Sci. 231, 64–82 data Mining for Information Security, http://www.sciencedirect.com/science/article/pii/S0020025511004336.
Saxe, J., Berlin, K., Oct 2015. Deep neural network based malware detection using two dimensional binary program features. In: 2015 10th International Conference on Malicious and Unwanted Software (MALWARE), pp. 11–20.
Shabtai, A., Moskovitch, R., Elovici, Y., Glezer, C., 2009. Detection of Malicious Code by Applying Machine Learning Classifiers on Static Features: A State-Of-The-Art Survey. Information Security Technical Report 14 (1). , pp. 16–29 malware http://www.sciencedirect.com/science/article/pii/S1363412709000041.
Shabtai, A., Moskovitch, R., Feher, C., Dolev, S., Elovici, Y., Feb, 2012. Detecting unknown malicious code by applying classification techniques on opcode patterns. Security Informatics 1 (1), 1, https://doi.org/10.1186/2190-_8532-_1-_1.
Shirataki, S., Yamaguchi, S., Dec 2017. A study on interpretability of decision of machine learning. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 4830–4831.
Sikorski, M., Honig, A., 2012. Practical Malware Analysis: the Hands-On Guide to Dissecting Malicious Software, first ed. No Starch Press, San Francisco, CA, USA.
Snort, 2015. Snort Network Intrusion Detection System. Tech. rep., Snort.. https://www.snort.org/.
Sorokin, I., Jun 2011. Comparing files using structural entropy. J. Comput. Virol. 7 (4), 259, https://doi.org/10.1007/s11416-_011-_0153-_9.
Souri, A., Hosseini, R., Jan 2018. A state-of-the-art survey of malware detection approaches using data mining techniques. Human-centric Computing and Information Sciences 8 (1), 3, https://doi.org/10.1186/s13673-_018-_0125-_x.
Storlie, C., Anderson, B., Wiel, S., Quist, D., Hash, C., Brown, N., 2014. Stochastic identification of malware with dynamic traces. Ann. Appl. Stat. 8 (1), 1–18.
Suciu, O., Coull, S.E., Johns, J., 2018. Exploring Adversarial Examples in Malware Detection. CoRR abs/1810.08280. http://arxiv.org/abs/1810.08280.
Ucci, D., Aniello, L., Baldoni, R., 2019. Survey of machine learning techniques for malware analysis. Comput. Secur. 81, 123–147. http://www.sciencedirect.com/science/article/pii/S0167404818303808.
Uppal, D., Sinha, R., Mehra, V., Jain, V., Sep. 2014. Malware detection and classification based on extraction of api sequences. In: 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI), pp. 2337–2342.
VirusShare, 2011. Vxshare. https://virusshare.com/.
Wojnowicz, M., Chisholm, G., Wolff, M., Zhao, X., 2016. Wavelet decomposition of software entropy reveals symptoms of malicious code. Journal of Innovation in Digital Ecosystems. 3 (2), 130–140. http://www.sciencedirect.com/science/article/pii/S2352664516300220.
Yan, X., Han, J., Afshar, R., 2003. Clospan: mining closed sequential patterns in large datasets. In: Proceedings of the 2003 SIAM International Conference on Data Mining, pp. 166–177.
Ye, Y., Wang, D., Li, T., Ye, D., Jiang, Q., Nov 2008b. An intelligent pe-malware detection system based on association mining. J. Comput. Virol. 4 (4), 323–334, https://doi.org/10.1007/s11416-_008-_0082-_4.
Ye, Y., Li, T., Adjeroh, D., Iyengar, S.S., Jun 2017. A survey on malware detection using data mining techniques. ACM Comput. Surv. 50 (3), https://doi.org/10.1145/3073559 41:141:40.
Ye, Y., Chen, L., Wang, D., Li, T., Jiang, Q., Zhao, M., Nov 2008a. Sbmds: an interpretable string based malware detection system using svm ensemble with bagging. J. Comput. Virol. 5 (4), 283, https://doi.org/10.1007/s11416-_008-_0108-_y.
You, I., Yim, K., Nov 2010. Malware obfuscation techniques: a brief survey. In: 2010 International Conference on Broadband, Wireless Computing, Communication and Applications, pp. 297–300.
Yuval Nativ, L.L., 2015. 5fingers. The zoo https://github.com/ytisf/theZoo.
Yuxin, D., Siyi, Z., Feb 2019. Malware detection based on deep learning algorithm. Neural Comput. Appl. 31 (2), 461–472, https://doi.org/10.1007/s00521-_017-_3077-_6.
Zhang, X., Zhao, J., LeCun, Y., 2015. Character-level convolutional networks for text classification. In: Proceedings of the 28th International Conference on Neural Information Processing Systems, ume 1. MIT Press, Cambridge, MA, USA, pp. 649–657 NIPS15, http://dl.acm.org/citation.cfm?id2969239.2969312.
Zhao, G., Xu, K., Xu, L., Wu, B., 2015. Detecting apt malware infections based on malicious dns and traffic analysis. IEEE Access 3, 1132–1142.

Daniel Gibert is a PhD student at the Department of Computer Science and Industrial Engineering of the University of Lleida, in Spain. He graduated in 2014 and received a Masters degree in Artificial Intelligence in 2016 from the Polytechnic University of Catalonia. His research interests include intrusion detection and machine learning.

Carles Mateu received his B. Sc. from Universitat de Lleida, M. Sc. from Open University of Catalonia, and Ph.D. from University of Lleida in 2009. He is currently an Associate Professor at the University of Lleida. His research interests include Security, Energy Storage, Energy Efficiency and Artificial Intelligence.

Jordi Planes received his B. Sc. from Universitat de Lleida, M. Sc. from University Rovira i Virgili, and Ph.D. from University of Lleida in 2007. He is currently an Associate Professor at the University of Lleida. His research interests include Applications of Neural Networks.