Malware and Machine Learning

Charles LeDoux and Arun Lakhotia

Abstract Malware analysts use Machine Learning to aid in the fight against the unstemmed tide of new malware encountered on a daily, even hourly, basis. The marriage of these two fields (malware and machine learning) is a match made in heaven: malware contains inherent patterns and similarities due to code and code pattern reuse by malware authors; machine learning operates by discovering inherent patterns and similarities. In this chapter, we seek to provide an overhead, guiding view of machine learning and how it is being applied in malware analysis. We do not attempt to provide a tutorial or comprehensive introduction to either malware or machine learning, but rather the major issues and intuitions of both fields along with an elucidation of the malware analysis problems machine learning is best equipped to solve.

1 Introduction

Malware, short for malicious software, is the weapon of cyber warfare. It enables online sabotage, cyber espionage, identity theft, credit card theft, and many more criminal, online acts. A major challenge in dealing with the menace, however, is its sheer volume and rate of growth. Tens of thousands of new and unique malware are discovered *daily*. The total number of new malware has been growing exponentially, doubling every year over the last three decades.

Analyzing and understanding this vast sea of malware manually is simply impossible. Fortunately for the malware analyst, very few of these unique malware are truly novel. Writing software is a hard problem, and this remains the case whether said software is benign or malicious. Thus, malware authors often reuse code and code

C. LeDoux (✉) · A. Lakhotia
Center for Advanced Computer Studies, University of Louisiana at Lafayette,
PO Box 44330, Lafayette, LA 70504, USA
e-mail: charles.a.ledoux@gmail.com

A. Lakhotia
e-mail: arun@louisiana.edu

patterns in creating new malware. The result is the existence of inherent patterns and similarities between related malware, a weakness that can be exploited by malware analysts.

In order to capitalize on this inherent similarity and shared patterns between malware, the anti-malware industry has turned to the field of Machine Learning, a field of research concerned with "teaching" computers to recognize concepts. This "learning" occurs through the discovery of indicative patterns in a group of objects representing the concept being taught or by looking for similarities between objects. Though humans too use patterns in learning, such as using color, shape, sound, and smell to recognize objects, machines can find patterns in large swaths of data that may be gibberish to a humans, such as the patterns in sequences of bits of a collection of malware. Thus, Machine Learning has a natural fit with Malware Analysis since it can more rapidly learn and find patterns in the ever growing corpus of malware than humans.

Both Machine Learning and Malware Analysis are very diverse and varied fields with equally diverse and varied ways in which they overlap. In this chapter, we seek to provide a guiding, overhead cartography of these varied landscapes, focusing on the areas and ways in which they overlap. We do not seek to provide a comprehensive tutorial or introduction to either Malware or Machine Learning research. Instead, we strive to elucidate the major ideas, issues, and intuitions for each field; pointing to further resources when necessary. It is our intention that a researcher in either Malware Analysis or Machine Learning can read this chapter and gain a high-level understanding of the other field and the problems in Malware that Machine Learning has, is, and can be used to solve.

2 A Short History of Malware

The theory of malware is almost as old as the computer itself, tracing back to lectures by von Neumann in late 1940s on self-reproducing automata [1]. These early malware, if they can be called as such, did nothing significantly more than demonstrate self-reproduction and propagation. For example, one of the earliest malware to escape "into the wild" was called Elk Cloner and would simply display a small poem every 50th time an infected computer was booted:

```
Elk Cloner: The program with a personality

It will get on all your disks
It will infiltrate your chips
Yes it's Cloner!

It will stick to you like glue
It will modify ram too
Send in the Cloner!
```

The term computer *virus* was coined in early 1980s to describe such self-replicating programs [2]. The use of the term was influenced by the analogy of computer malware to biological viruses. A biological virus comes alive after it infects a living organism. Similarly, the early computer viruses required a host—typically another program—to be activated. This was necessitated by the limitations of the then computing infrastructure which consisted of isolated, stand-alone, machines. In order to propagate, that is infect currently uninfected machines, a computer virus necessarily had to copy itself in various drives, tapes, and folders that would be accessed by different machines. In order to ensure that the viral code was executed when it reached the new machine, the virus code would attach itself to, i.e. infect, another piece of code (a program or boot sector) that would be executed when the drive or tape reached another machine. When the now infected code would later execute, so would the viral code, furthering the propagation.

The early viruses remained mostly pranks. Any damage they caused, such as crashing a computer or exhausting disk space, was largely unintentional and a side effect of uncontrolled propagation. However, the number and spread of viruses quickly grew to enough of a nuisance that it led to the development of first anti-virus companies in the late 1980s. Those early viruses were simple enough that they could be detected by specific sequences of bytes, a la signatures.

The advent of networking, leading to the Internet, changed everything. Since data could now be transferred between computers without using an external storage device, so could the viruses. This freedom to propagate also meant that a virus no longer needed to infect a host program. A new class of malware called worm emerged. A worm was a stand alone program that could propagate from machine to machine without necessarily attaching to any other program.

Malware writing too quickly morphed from simple pranks into malicious vandalism, such as that done by the ILOVEYOU worm. This worm came as an attachment to an email with the (unsurprising) subject line "ILOVEYOU". When a user would open the attachment, the worm would first email itself to the user's contacts and then begin destroying data on the current computer. There were a number of similar malware created, designed only to wreak havoc and gain underground notoriety for their authors. These "graffiti" malware, however, soon gave way to the true threat: malware designed to make money and steal secrets.

Malware today has little if any resemblance to the malware of past. For one, gone are the simple days of pranks and vandalism conducted by bored teenagers and budding hackers. Modern malware is an well-organized activity forming a complete underground economy with its own supply chain. Malware is now a tool used by large underground organizations for making money and a weapon used by governments for espionage and attacks. Malware targeted towards normal, everyday computers can be designed to steal bank and credit card information (for direct theft of money), harvest email addresses (for selling to spammers), or gain remote control of the computer. The major threat from malware, however, comes from malware targeted not towards the average computer, but towards a particular corporation or government. These malware are designed to facilitate theft of trade or national secrets, steal crucial information (such as sensitive emails), or attack infrastructure. For example, Stuxnet was malware designed to attack and damage various nuclear facilities in Iran. These malware often have large organizations (such as rival corporations) or even governments behind them.

3 Types of Malware

Whenever there is a large amount of information or data, it helps to categorize and organize it so that it can be managed. Classification also aids in communication between people, giving them a common nomenclature. The same is true of malware. The industry uses a variety of methods to classify and organize malware. The classification is often based on the method of propagation, the method of infection, and the objective of the malware. There is, however, no known standard nomenclature that is used across the industry. Classifications sometimes also come with legal implications. For instance, can a program that inserts advertisements as you browse the web be termed as malicious. What if the program was downloaded and installed by the user, say after being enticed by some free offering? To thwart legal notices the industry invented the term *potentially unwanted program* or PUP to refer to such programs.

Though there is no accepted standard for classification of malware in the industry, there is a reasonable agreement on classifying malware on their method of propagation into three *types*: virus, worm, and trojan (short for Trojan horse).

Virus, despite being often used as a synonym for malware, technically refers to a malware that attaches a copy of itself to a host, as described earlier. Propagation by infecting removable media was the only method for transmission available prior to the Internet, and this method is still in use today. For instance, modern viruses travel by infecting USB drives. This method is still necessary to reach computer systems that are not connected to the Internet, and is hypothesized as the way Stuxnet was transmitted.

A **trojan** propagates the same way its name sake entered the city of Troy, by hiding inside something that seems perfectly innocent. The earliest trojan was a game called ANIMAL. This simple game would ask the user a serious of questions and attempt to guess what animal the user was thinking of. When the game was executed, a hidden program, named PERVADE, would install a copy of itself and ANIMAL to every location the user had access to. A common modern example of a trojan is a fake antivirus, a program that purports to be an anti-virus system but in fact is a malware itself.

A **worm**, as mentioned earlier, is essentially a self-propagating malware. Whereas a virus, after attaching itself to a program or document, relies on an action from a user to be activated and spread, a worm is capable of spreading between network connected computers all by itself. This is typically accomplished one of two ways: exploiting vulnerabilities on a networked service or through email. The worm CODE RED was an example of the first type of worm. CODE RED exploited a bug in a specific type of server that would allow a remote computer to execute code on the server. The worm would simply scan the network looking for a vulnerable server. Once found, it would attempt to connect to the server and exploit the known bug. If successful, it would create another instance of the worm that repeated the whole process. The ILOVEYOU worm, discussed earlier, is an example of an email worm and spread as an email attachment. When a user opened the attachment, the worm would email a copy of itself to everyone in the user's contact list and damage the current machine.

While the above methods of propagation are the mostly commonly known, they by no means represent all possible ways in which malware can propagate. In general, one of two methods are employed to get a malware onto a system: exploit a bug in software installed on the computer or exploit the trust (or ignorance) of the user of the computer through social engineering. There are many different types of software bugs that allow for arbitrary code to be executed and almost as many ways to trick a user into installing a malware. Complicating matters further, There is no technical reason for a malware to limit its use to only one method of propagation. It is entirely conceivable, as was demonstrated by Stuxnet, for a malware to enter a network through email or USB, and then spread laterally to other machines by exploiting bugs.

4 Malware Analysis Pipeline

The typical end goal of malware analysis is simple: automatically detect malware as soon as possible, remove it, and repair any damage it has done. To accomplish this goal, software running on the system being protected (desktop, laptop, server, mobile device, embedded device, etc.) uses some type of "signatures" to look for malware. When a match is made on a "signature", a removal and repair script is triggered. The various portions of the analysis "pipeline" all in one way or another support this end goal [3, 4].

The general phases of creating and using these signatures are illustrated by Fig. 1. Creating a signature and removal instructions for a new malware occurs in the "Lab." The input into this malware analysis pipeline is a feed of suspicious programs to be analyzed. This feed can come from many sources such as honeypots or other companies. This feed first goes through a triage stage to quickly filter out known programs and assign an analysis priority to the sample. The remaining programs are then analyzed to discover what it looks like and what it does. The results of the analysis phase are used to create a signature and removal/repair instructions which are then verified for correctness and performance concerns. Once verified, these signatures are propagated to the end system and used by a scanner to detect, remove, and repair malware.

Each of the various phases of the anti-malware analysis process is attempting to accomplish a related, but independent task and thus has its own unique goals and performance constraints. As a result, each phase can independently be automated and optimized in order to improve the performance of the entire analysis pipeline.

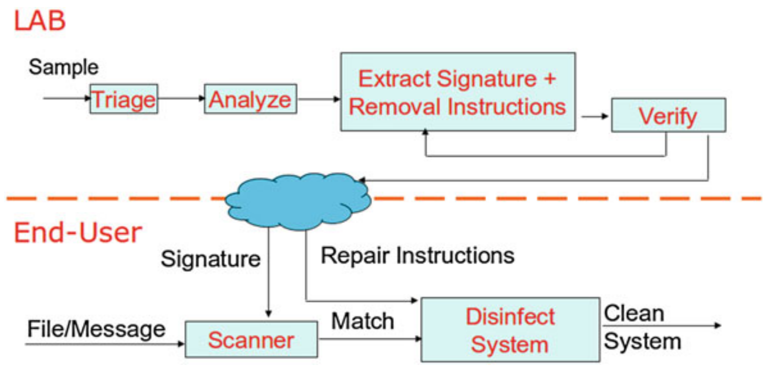


Fig. 1 Phases of the malware analysis pipeline

In fact, it is almost a requirement that automation techniques be tailored for the specific phase they are applied in, even if the technique could be applied to multiple phases. For example, a machine learning algorithm designed to filter out already analyzed malware in the triage stage will most likely perform poorly as a scanner. While both the triage stage and the scanner are accomplishing the same basic task, detect known malware, the standard by which they are evaluated is different.

4.1 Triage

The first phase of analysis, triage, is responsible for filtering out already analyzed malware and assigning analysis priority to the incoming programs. Malware analysts receive a *very* large number of new programs for analysis every day. Many of these programs, however, are essentially the same as programs that have already been analyzed and for which signatures exist. A time stamp or other trivial detail may have been changed causing a hash of the binary to be unique. Thus, while the program is technically unique, it does not need to reanalyzed as the differences are inconsequential. One of the purposes of triage is to filter these binaries out.

In addition to filtering out “exact” matches (programs that are essentially the same as already analyzed programs), triage is typically also tasked with assigning the incoming programs into malware families when possible. A malware family is a group of highly related malware, typically originating from common source code. If an incoming program can be assigned to a known malware family, any further analysis does not need to start with zero a priori knowledge, but can leverage general knowledge about the malware family, such as known intent or purpose.

A final purpose of the triage stage is to assign analysis priority to incoming programs. Humans still are and most likely will remain an integral part of the analysis pipeline. Like any other resource, what the available human labor is expended upon must be carefully chosen. Not all malware are created equal; it is more important that some malware have signatures created before others. For example, malware that only affects, say, Microsoft Windows 95 will not have the same priority as malware that affects the latest version of Windows.

The performance concerns for the triage phase are (1) ensuring that programs being filtered out truly should be removed and (2) efficient computation in order to achieve very high throughput. Programs filtered out by triage are not subjected to further analysis and thus it is very important that they do not actually need further analysis. Especially dangerous is the case of malware being filtered out as a benign program. In this case, that particular malware will remain undetectable. Marking a known malware or a benign program as malware for further processing, while undesirable, is not disastrous as it can still be filtered out in the later processing stages. Along the same lines, it is sufficient that malware be assigned to a particular family with only a reasonably high probability rather than near certainty. Finally, speed is of the utmost importance in this stage. This stage of the analysis pipeline examines the largest number of programs and thus requires the most efficient algorithms. Computationally expensive algorithms at this stage would cause a backlog so great that analysts would never be able to keep up with malware authors.

4.2 Analysis

In the analysis phase, information about what the program being analyzed does, i.e. its behavior, is gathered. This can be done in two ways: statically or dynamically.

Static analysis is performed without executing the program. Information about the behavior of the program is extracted by *disassembling* the binary and converting it back into human readable machine code. This is not high level source code, such as C++, but the low level assembly language. An assembly language is the human readable form of the instructions being given directly to the processor. ARM, PowerPC, and ×86 are the better known examples of assembly languages. After disassembly, the assembly code (often just called the malware "code" for short) can be analyzed to determine the behavior of the program. The methods for doing this analysis constitute an entire research field called program analysis and as such are outside the scope of this chapter. Nielson et al. [5] have a comprehensive tutorial to this field.

Static analysis can *theoretically* provide perfect information about the behavior of a program, but in practice provides an over approximation of the behaviors present. Only what is in the code is what can be executed, thus the code contains everything the program can do. However, extracting this information from a binary can be difficult, if not impossible. Perfectly solving many of the problems of static analysis is undecidable.

As an example of the problems faced by static analysis, binary disassembly is itself an undecidable problem. Binaries contain both data and code and separating the two from each other is undecidable. As a result some disassemblers treat the entire binary, including data, as if it were code. This results in a proper extraction of most of the original assembly code, along with much code that never originally existed.

There are many other methods of disassembly, such as the recent work by Schwarz et al. [6]. While these methods significantly improve on the resulting disassembly, none can guarantee correct disassembly. For instance, it is possible that there exists "dead code" in the original binary, i.e. code that can never be reached at runtime. In an ideal disassembly, such code ought to be excluded. Thus all of static analysis operates on approximations. Most disassemblers used in practice do not guarantee either over approximation or under approximation.

Dynamic analysis, in contrast with static analysis, is conducted by actually executing the program and observing what it does. The program can be observed from either within or without the executing environment. From within uses the same tools and techniques software developers use to debug their own programs. Tools that observe the operating system state can be utilized and the analyzed program run in a debugger. Observation from without the execution environment occurs by using a specially modified virtual machine or emulator. The analyzed program is executed within the virtual environment and the tools providing the virtualization observe and report the behavior of the program.

Dynamic analysis, as opposed to static analysis, generally provides an under approximation of the behaviors contained in the analyzed program, but guarantees that returned behaviors can be exhibited. Behaviors discovered by dynamic analysis are obviously guaranteed to be possible as the program was observed performing these behaviors. *Only* the observed behaviors can be returned, however. A single execution of a program is not likely to exhibit all the behaviors of the program as only a single path of execution through the binary is followed per run. A differing execution environment or differing input may reveal previously unseen behaviors.

4.3 Signatures and Verification

While the most common image conjured by the phrase "malware signatures" is specific patterns of bytes (often called strings) used by an Anti-Virus system to detect a malware, we do not use the term in that restricted sense. What we mean by signature is any method utilized for determining if a program is malware. This can include the machine learning system built to recognize malware, a set of behaviors marked as malicious, a white list (anything not on the white list is marked as malicious), and more. The important thing about a signature is that it can be used to determine if a program is malware or not.

Along with the signatures, instructions for how to remove malware that has infected the system and repair any damage it has done must also be created. This is usually done manually, utilizing the results of the analysis stage. Observe what the malware did, and then reverse it. One major concern here is ensuring that the repair instructions do not cause even more damage. If the malware changed a registry key, for example, and the original key is unknown, it may be safest to just leave the key alone. Changing it to a different value or removing it all together may result in corrupting the system being "protected." Thus repair instructions are often very conservative, many times only removing the malware itself.

Once created, the signatures need to be verified for correctness and, more importantly, for accuracy. Even more important than creating a signature that matches the malware is creating a signature that *only* matches the malware. Signatures that also match benign programs are worse than useless; they are acting like malware themselves! Saying that benign programs are actually malware, called a false positive, is an error that cannot be tolerated once the signatures have been deployed to the scanner.

4.4 Application

Once created, the signatures are deployed to the end user. At the end system, new files are scanned using the created signatures. When a file matches a signature, the associated repair instructions followed.

The functionality of the scanner will depend on the type of signature created. String based signatures will use a scanner that checks for existence of the string in the file. A scanner based on Machine Learning signatures will apply what has been learned through ML to detect malware. A rule based scanner will check if the file matches its rules, and so on and so forth.

5 Challenges in Malware Analysis

One of the fundamental problems associated with every step of the malware analysis pipeline is the reliance on incomplete approximations. In every stage of the pipeline, the exact solution is generally impossible. Triage cannot perfectly identify every part of every program that has already been identified. Analysis will generate either potentially inaccurate or incomplete information. All types of signatures are limited. Even verification is limited by what can be practically tested.

Naturally, malware authors have developed techniques that directly attack each stage of the analysis pipeline and shift the error in the inherent approximations to their favor. Packing and code morphing are used against triage to increase the number of "unique" malware that must be analyzed. Packing, tool detection, and obfuscation are used against the analysis stage to increase the difficultly of extracting any meaningful information.

While the ultimate goal of the malware authors is obviously to completely avoid detection, simply increasing the difficulty of achieving detection can be considered a "win" for the malware authors. The more resources consumed in analyzing a single malware, the less total malware that can be analyzed and detected. If this singular cost is driven high enough, then detection of any but the most critical malware simply becomes too expensive.

5.1 Code Morphing

The most common and possibly the most effective attack against the malware analysis pipeline targets the first stage: triage. The attack is to simply inundate the pipeline with as many unique malware as possible. Unique is not used here to mean novel, i.e. does something unique; here it simply means that the triage stage considers it something that has not been analyzed before. Analysis stages further down the pipe from Triage are allowed to be more expensive because it is assumed Triage has filtered out already analyzed malware, severely reducing the number of malware the expensive processes are run on. By slipping more malware past Triage and forcing the more expensive processes to run, the cost of analysis can be driven up, possibly prohibitively high.

One of the ways this attack is accomplished is through automated morphing of the malware's code into a different but semantical equivalent form. Such malware is often called metamorphic or polymorphic. Before infecting a new computer, a rewriting engine changes what the code looks like through such means as altering control flow, utilizing different instructions, and adding instructions that have no semantic effect. The changes performed by the rewriting engine only change the look or syntax of the code and leave its function or semantics intact. The result is that each "generation" of metamorphic malware is functionally equivalent, but the code can be radically different.

While several subtle variations in definitions exist, we view the difference between metamorphic and polymorphic malware as where the rewriting engine lies. Metamorphic malware contains its own, internal rewriting engine, that is, the malware binary rewrites itself. Polymorphic malware, on the other hand, have a separate mutating engine; a separate binary rewrites the malware binary. This mutating engine can either be distributed with the malware (client side) or kept on a distributing server and simply distribute a different version of malware every time (server side).

Metamorphic malware is more limited than polymorphic malware in the transformations it can safely perform. Any rewriting engine is going to contain limitations as to what it can safely take as input. If the engine is designed to modify the control flow of the program, for example, it will only be able to rewrite programs for which it can identify the existing control flow. Since metamorphic malware contains its own rewriting engine, the output of the rewriting engine must be constrained to acceptable input. Without this constraint, further mutations would not be possible. Polymorphic malware, however, does not contain this constraint. Since the rewriting engine is separate and can thus always operate over the exact some input, the output does not need to be constrained to only acceptable input.

5.2 Packing

Packing is a process whereby an arbitrary executable is taken and encrypted and compressed into a "packed" form that must be uncompressed and decrypted, i.e. "unpacked", before execution. This packed version of the executable is then packaged as data inside another executable that will decompress, decrypt, and run the original code. Thus, the end result is a new binary that looks very different from the original, but when executed performs the exact same task, albeit with some additional unpacking work. A program that does packing is referred to a packer and the newly created executable is called the packed executable.

Packing directly attacks Triage and static analysis. While packing a binary does not modify any of the malware's code, it drastically modifies the binary itself, potentially even changing a number of statistical properties. If there is some randomization within the packing routine, a binary that appears truly unique will result every time the exact same malware is packed. Unless the Triage stage can first unpack the binary, it will not be able to match it to any known malware.

Packing does more than simply complicate the triage stage, it also directly attacks any use of static analysis. As discussed in Sect. 4.2, the first step in static analysis is usually to disassemble the binary. Packing, however, often encrypts the original binary, preventing direct disassembly. A disassembler will not be able to meaningfully interpret the stored bits unless it is first unpacked and the original binary recovered.

The need to unpack a program (recover the original binary) is usually not a straight forward task—hence the existence of a challenge. As one might expect, there exists very complex packers intentionally designed to foil unpacking. Some packers, for example, only decrypt a single instruction at a time while others never fully unpack the binary and instead run the packed program in a virtual machine with a randomly created instruction set.

It might seem that simply detecting that an executable was packed would be sufficient to determine that it was malware. There are, however, legitimate uses for packing. First, packing is capable of reducing the overall size of the binary. The compression rate of the original binary is often large enough that even with the additional unpacking routine (which can be made fairly small), the packed binary is smaller in size than the original binary. Of course, when size is the only concern, the encryption part of packing is unnecessary. So, perhaps detecting encryption is sufficient? Unfortunately, no. Encryption has a legitimate application in protecting intellectual property. A software developer may compress and encrypt the executables they sell and ship to prevent a competitor from reversing the program and discovering trade secrets.

5.3 Obfuscation

While packing attempts to create code that cannot be interpreted at all, obfuscation attempts to make extracting meaning from the code, statically or dynamically, as difficult as possible. In general, obfuscation refers to writing or transforming a program into a form that hides its true functionality. The simplest example of a source code obfuscation is to give all variables meaningless names. Without descriptive names, the analyst must determine the purpose of each variable. At the binary level, examples of obfuscation include adding dead code (valid code that is never executed), interleaving several procedures within each other, and running all control flow through a single switch statement (called control flow flattening). An in depth treatment of code obfuscation, including methods for deobfuscating the code, is given by Collberg and Nagra [7].

5.4 Tool Detection

A major problem in dynamic analysis is malware detecting that it is being analyzed and modifying its behavior. Static analysis has a slight advantage in that the analyzed malware has no control over the analysis process. In dynamic analysis, however, the malware is actually being executed and so can be made capable of altering its behavior. Thus, malware authors will often check to see if any of the observation tools often used by malware analysis are present, and if so, perform only benign activities. For example, a malware may check to see if it is being run by a debugger and if so, exit. This effectively makes the malware invisible to dynamic analysis.

There are two types of checks that can be done by malware: check for a class of tool and check for a specific tool. There are specific types of tools normally used to observe malware in dynamic malware analysis such as debuggers and virtualization. When these tools are used, they usually leave some detectable artifact in the system. For example, in both cases of using a debugger or a virtualized environment, it will be necessity be the case that executing at least some instructions will take longer that if running unobserved. If a malware can detect this discrepancy, through a timer, perhaps, it can detect it is being observed.

Easier than checking for a class of tools, however, is to just check for the specific set of the most widely used tools. Finding a single check that can detect all tools of a particular type is difficult, and the test can be unreliable. A (usually) simpler test is to check for the existence of a specific tool. For example, an executing program could check if it is being run under one of the most common debuggers, Olly Debug, by looking for a process named *ollydbg.exe*. As a natural limitation of software, the number of mature, commonly used commercial or open source analysis tools available is relatively limited. Thus, a malware author can implement a number of simple checks and prevent a large portion of analysis. Naturally, the tool authors can remove the detected artifact, but completely eliminating every trace of a executing program is near impossible. As tool authors remove one artifact, malware authors can use another, resulting in a never ending game of cat and mouse.

5.5 Difficulty Obtaining Verification Data

An important part of the verification stage is obtaining "ground truth" information. This ground truth can simply be thought of as the correct answer. If evaluating a new technique's ability to detect malware, then the ground truth would be the labeling of each executable as malware or benign. If evaluating a classifier's ability to separate malware into families, the ground truth would consist of the labeling of each executable with the family it belongs to. The ground truth is needed to determine the correctness of the labels assigned by a machine learning system and to measure its performance.

Obtaining this ground truth is typically an expensive and error prone process. This is because the ground truth usually must be determined manually. Creating the labeling for executables often requires a human analyst to examine the program and give an expert opinion. This takes time and, as with any human judgment, is subject to potential errors. While this may not seem like an issue when the labeling is simply "malware" and "not malware," the challenge increases significantly when labeling a malware with its family. This task may involve manually, albeit with support of tools, viewing and comparing large amounts of information—such as disassembled code, strings, and API calls—for correct labeling. This is complicated by the fact that different malware families share similar characteristics, such as using the same method to trap keystrokes. In such cases, human judgment, inaccuracies, and fatigue may lead to errors in labeling.

6 Machine Learning Concepts

In general, the purpose of machine learning algorithms is to "teach" a program to recognize some type of concept [8]. The concept learned and the way it is taught are of course specific to both the exact machine learning algorithm and the application of the program. In the malware domain, these concepts can be as broad as "malware" or as focused as "implements x." They can be as abstract as "worm" or concrete as "written by Bob Doe." The concepts that can be recognized, and the applications of this recognition, are practically limitless.

The set of all concepts that can be learned by a particular machine learning algorithm is referred to as the *concept space*. This is not a set picked out by a researcher, but the theoretical set of *all* possible concepts for the particular machine learning algorithm selected. A single algorithm is not capable of learning just any concept from the universe of all concepts, but only a comparatively small subset of this infinite universe.

6.1 Features

Machine learning algorithms do not directly digest raw malware, but rather first extract *features* that provide an abstract view of the malware. These features can be thought of as the "language" of the classifier; a way for describing malware to a given machine learning algorithm. For example, features for representing fruit may include things such as shape, color, and whether or not it is firm. An apple would then have the features "shape = round, color = red, firm = yes" while a banana would have the features "shape = long, color = yellow, firm = no". An example feature type used in malware is the set of system calls made. The types of features used to represent malware are discussed in more detail in Sect. 7.

Defining the type of feature used by a machine learning algorithm is a crucial design decision as the feature space (the set of all possible features that can be taken as input) defines the concept space (the set of all possible concepts that can be learned). Only concepts that are capable of being represented or described by the type of features selected by the designer can be learned by the algorithm. For example, features that describe a binary will not, barring black magic, represent the concept "apple." Thus, it is important that the defined features create a concept space containing the concept to learn. Even more important, however, is that the concept space contain as little else as possible. If the concept space is too large, learning the desired concept becomes a "needle in the haystack" problem.

6.2 Classification and Clustering

In malware, the two basic tasks machine learning is used for are classification and clustering. Classification attaches one of a predetermined set of labels to an unknown program. Each of these labels represents a class or category of objects. Thus, assigning a label to a program is akin to marking that program as belonging to a specific class; hence the term classification. The simplest example of classification is labeling a program as malicious or benign. Clustering, on the other hand, partitions the given group of programs into clusters of related programs. The criteria for "related" is usually a similarity function that measures how much two programs resemble each other.

The general pipeline for performing classification in malware analysis is given in Fig. 2. This process has two stages: learning and classification. Both stages begin by extracting the feature representation of the malware. In the learning phase, the malware additionally contain attached labels providing the "correct answer" for classification. In other words, malware, and consequently features, are labeled with the concept they exemplify. A learning algorithm takes these features and labels and creates a model to represent the learned concepts. This model can be thought of as a function that takes a feature representation of a malware as input and outputs the concept that the malware best matches, i.e. a label. Classification, then, is simply

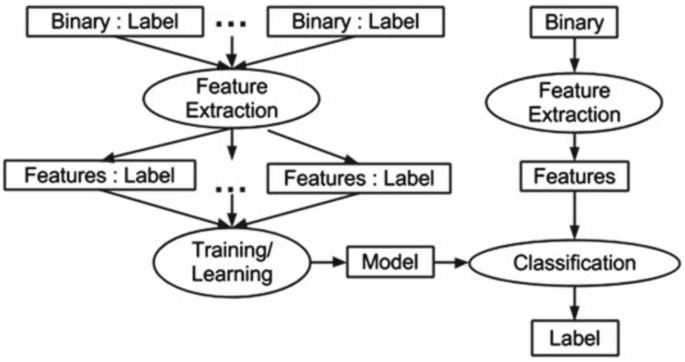


Fig. 2 General classification process

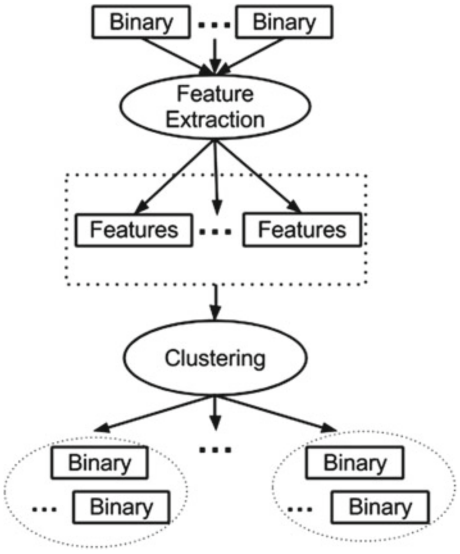


Fig. 3 General clustering process

applying this function to an unknown malware. The details of what a model is and how it operates are specific to the exact type of learning algorithm used and thus out of scope for this chapter.

The basic idea of clustering is to put malware into clusters such that malware within a given cluster are more closely related to each other than with malware outside the cluster. The basic process used to accomplish this is given in Fig. 3. Clustering, like classification, first begins by extracting feature representations of the malware. Clustering, however, does not contain a learning phase or utilize labels. The set of features is given directly to a clustering algorithm that partitions the input set into clusters of related malware. Like classification, the exact methods of defining "related" and partitioning are varied and out of scope for this discussion.

While both classification and clustering effectively partition a given group of malware, there are a number of key differences between the two tasks. Classification uses a predefined (usually human defined) set of labels; clustering uses the number of groupings that best fit the given notion of "related." Classification focuses on attaching labels to a *single* malware at a time; clustering operates on whole *groups* of malware. Labels in classification directly correspond to concepts (the concept to learn is "named" by the label); concepts represented by clusters are not named and are thus not always apparent (besides the overly-general "related to each other" concept).

As an illustrative example, let us consider the difference between classifying and clustering a group of binaries into malware families. The first difference is in the method of processing the binaries. After the learning stage, classification will examine each binary one at a time and attach a family name to it. Clustering, on the other hand, will immediately begin operating on the entire collection at once, returning a partitioning of the binaries into groups without labels. The second difference is the number of families that the binaries can be grouped into. In classification, this number is pre-specified and the learning algorithm can learn no more and *no less* than this number. Clustering, however, can theoretically create as many groups as there are binaries (one binary per group). It may, for example, split what an analyst considered one family into two sub-families. The final major difference is in interpreting the learned concepts. It is obvious how the labels of classification correspond to malware families; the labels were created by the analyst after all. The same is not necessarily true of clustering. Not only can the number of families differ from what is expected by the analyst, but none of the groups are labeled and so discovering which groups map to which family is not always straightforward.

A special case of clustering often used in malware analysis, called near neighbor search, is to find programs that are similar to a given "query" program. Knowing what an unknown malware is similar to has a number of uses, the most immediate being a leveraging of existing knowledge. If the unknown malware is 90 % similar to an already existing malware, only the 10 % dissimilar portion must be analyzed. Thus, for this and other reasons it is common to want to find the group of known malware that are very (very here being a relative term) similar to an unknown malware of interest. This is conceptually the same as creating a clustering of similar malware and discarding all clusters except the one containing the unknown malware. (Efficient near neighbor search algorithms, of course, do not actually create the clusters that will be discarded.)

6.3 Types of Learning

While the utilized feature type defines the concept space (Sect. 6.1), it is the type of learning algorithm that defines how this concept space is searched. There are a large number of machine learning algorithms, but they can generally be broken up into several categories based on how they learn concepts: Supervised Learning, Unsupervised Learning, Semi-Supervised Learning, and Ensemble Learning.

Supervised Learning (Sect. 8) can be described as "learning by example." Supervised Learning is often considered synonymous with classification (as supervised learning is often the type of learning algorithm used for classification) and follows the classification pipeline laid out in Fig. 2. The labeled features are fed to the learning algorithm as examples of what each concept looks like. The model is built based upon these examples and the concepts learned correspond the to labels provided.

Unsupervised Learning (Sect. 9) is learning without the "correct answer" given by labels. This form of learning is often considered synonymous with clustering and follows the clustering pipeline put forth in Fig. 3. Instead of forming a model of predefined concepts, unsupervised learning groups objects together based on a given concept of "relatedness" or "similarity." The idea is that objects within a cluster should be more closely related (more similar) to each other than objects outside the cluster. After clustering, every group of objects will represent some concept, though there are no labels or names attached to the represented concept.

Semi-supervised Learning (Sect. 11) combines both supervised and unsupervised learning. Semi-Supervised learning operates over a group of object where some, but not all, of the objects contain "correct answer" labels. The typical application of this sort of learning algorithm is to perform clustering and use the labels that do exist to improve the final clustering result.

Ensemble Learning (Sect. 12) is learning from a collection of classifiers or clusters. For a classifier ensemble, a number of classifiers are trained with their associated models. To classify a new malware, the results of applying all the created models are combined into a single classification. Cluster ensembles work similarly. A number of clusterings of the data are independently created and then merged to create a final clustering.

While Classification and Clustering are often used as synonyms for Supervised and Unsupervised Learning, a distinction can be made. Classification and clustering are tasks, while supervised and unsupervised learning are types of learning algorithms. It is possible to perform classification using an unsupervised learning algorithm. Given a set of malware, clusters can be formed in an unsupervised method (without using any labels), labels attached to the clusters after they have been formed, and classification done by assigning a new malware the label of the most similar cluster. This classification would be unsupervised learning because the concepts (the clusters) were learned without any labels being utilized. The labels were simply names attached to already learned concepts.

7 Malware Features

As initially discussed in Sect. 6.1, the type of feature defined and utilized in any machine learning application is of utmost importance. Features are the input to the machine learning algorithm and define the concept space, i.e. the space of all possible concepts that can be modeled by this algorithm. If the desired concept cannot be modeled, the features are useless. Similarly, if the concept space is too large, then learning how to model the desired concepts is analogous to finding a needle in a haystack. Thus the great importance of defining high quality features.

Just as the types of malware analysis can be divided into static and dynamic, so too can malware features. Static features [9] are features extracted from the binary of the malware without executing it, i.e. through static analysis. While this can refer to the actual bits of the binary or structural information contained in the header, it more commonly refers to features extracted from the disassembled binary. A disassembled binary is created by converting the bits of the binary back into human readable machine code (assembly language, not high level source such as C). Various transformations on this disassembly are then performed to create many different types of features.

Dynamic features [10], on the other hand, are features extracted by executing the malware and observing what it does, i.e. through dynamic analysis. There are several levels of abstraction that dynamic features can exist at ranging from a trace of the instructions executed in the processor to a predefined set of behaviors watched for.

Static and dynamic features have the issues discussed in Sect. 4.2. It is usually impossible to perfectly extract the precise of feature representation for a given malware and thus approximations are used. Static features often result in an over approximation and dynamic features often result in an under approximation. Take for example, the defined feature type “the set of system calls that can be made by the program.” Static analysis will usually extract almost all of the actual system calls that belong in the true set of features, but will also include potentially many system calls that the program will never actually execute. Features extracted by dynamic analysis are guaranteed to be in the true feature set, but not all system calls will be observed and recorded.

7.1 Binary Based Features

The simplest static features are structural features based directly on the raw binary. That is, features that treat the binary as nothing more than an executable file and do not attempt to extract any information more abstract than what the structure of the binary is. These types of features have been based on sequences of bytes in the binary, information contained in the binary’s header, and the strings visible in the binary.

7.1.1 N-Gram and N-Perm

One of the most common types of binary based static features is the *byte n-gram*. N-grams are a feature commonly used in text classification and they are created by sliding a window n characters long across a document and recording the unique strings

Table 1 Byte code 2-grams and 2-perms

Byte code	2-grams	2-perms
`0f0e` `0f0e` `b40f`	'0f0e' '0e0f' '0eb4' 'b40f'	'0e0f' '0fb4' '0eb4'

found (a technique often referred to as shingling or windowing in text classification literature). For example, the 2-grams for the string 'ababc' are 'ab', 'ba', and 'bc'. In the malware context, the n-grams are created over the byte code representation of the binary. N-grams were introduced to malware classification by members of the IBM TJ Watson Research Center [11–14].

A modification on n-grams proposed by Karim et al. [15] was *n-perms*. N-grams are extremely brittle, especially when n in large. Simply removing, adding, or swapping a few instructions can have a major impact on the set of n-grams retrieved. Karim et al. [15] proposed to treat all permutations of a single n-gram as equal. Thus, in the example given earlier, the string 'ababc' with 2-grams 'ab', 'ba', and 'bc' would only have 2-perms of 'ab' and 'bc'. The 2-grams 'ab' and 'ba' would be considered equal and only 'ab' stored.

Examples of byte code and the corresponding n-grams and n-perms are given in Table 1. The byte code is the hexadecimal representation of a binary. The 2-grams are created by sliding a window of size two across the individual bytes and recording unique sequences. The 2-grams are shown in the order encountered. The 2-perms are created by internally sorting each 2-gram. So, for example, the 2-gram '0f0e' becomes the n-perm '0e0f'. This has the desired effect of normalizing all permutations. In this example, there are four 2-grams, but only three 2-perms.

7.1.2 PE Header Based

A common source of structural information for creating features is the *PE header*. The PE Header is the data structure in a Windows executable that holds the information needed by the loader to place the program into memory and begin execution. Thus most of the information in the PE header relates to the locations and sizes of important pieces of the binary. For example, the loader needs to know the number, location, and size of the sections of the binary.[1] There are also a number of important data structures that the loader needs to be able to find, such as the import table. The import table is a list of the various external libraries that will need to be loaded into this program's address space.

Using the structural information provided by the PE Header in the triage analysis stage turns out to be quite effective. The information in the PE Header is very quick to extract (the OS does this every time the program is executed) and robust against

[1] Binaries are not one big blob, but separated into sections of logically related code and data. At a minimum, there will be two sections: one designated for data and the other for code.

minor changes in the binary. Compiling a program twice may result in binaries unique by hash because of trivial changes, but both binaries would contain almost identical PE Header information. This approach was proposed by Wang et al. [16].

Walenstein et al. [17] examined this effectiveness of using header information as features. They were especially interested in learning how much the information in the header contributed to the accuracy obtained when using byte based n-grams. When n-grams are computed over the entire binary, the header information is implicitly captured. What they found was that not only was header information quite useful in discovering malware variants, but that it also contributed a good deal to the usefulness of byte based features. Walenstein et al. [17], however, cautioned that it would be trivial for malware authors to remove or modify most, if not all, of the identifying information in the PE header and so this information should not be solely relied upon.

A non-structural feature that has been used from the PE header is the set of libraries and functions listed in the import table [18]. The import table is a data structure pointed to by the PE header and contains the list of functions from external libraries that are required by the program. Since external libraries are obviously external to the binary, hard coded addresses of the library functions cannot be used in the program. This is where the import table comes in. For each required function, the import table contains an entry with the name of the function and an address for the function. At load time, the loader pulls the libraries into the program's address space and writes the correct address to each entry in the import table.

The list of imported library functions contains a wealth of information. When dealing with well known libraries, such as libc or the set of Windows system calls, the intended task of each function is, well, known. For example, if functions from crypt32.dll (dll is short for dynamic link library) are imported, it is likely that this program performs cryptography tasks. Thus, if the import table hasn't been tampered with (usually a bad assumption in malware, but still true often enough), it can provide a nice high level idea of the types of behaviors the program is intended to perform.

7.1.3 Strings

A final type of feature based on the binary is *strings*. This was first explored by Schultz et al. [18] and is quite simply any sequence of bytes that can be validly interpreted as a printable sequence of characters. This idea was later refined by Ye et al. [19] to only include "interpretable" strings, that is only strings that make semantic sense. Ye et al. [19] posited that strings such as "8ibvxzciojrwqei" provided little useful information.

Table 2 illustrates the difference between strings and interpretable strings. The column to the left is the byte sequences found in the binary. The middle column is the result of treating the byte sequences as ASCII characters. Finally, the last column indicates whether the string would be considered interpretable.

Table 2 Strings and interpretable strings

Bytes	String	Interpretable?
44 65 74 65 63 74 65 64 20 6d 65 6d 6F 72 79 20 6C 65 61 6b 79 81 0A 00	"Detected Memory leaks! n"	✓
47 65 74 4C 61 73 74 41 63 74 69 76 65 50 6F 70 75 70 00	GetLastActivePopup	✓
31 39 32 2e 31 36 38 2e 38 33 2e 31 35 33	192.168.83.153	✓
3b 33 2b 23 3e 36 26 1e	;3+# 6.&	χ
77 73 72 65 77 6e 61 66 34 79 6F 77 33 69 37 35	wsrewnaf4yow3i75	χ

7.2 *Disassembly Based Features*

While structural information is useful for finding malware that *looks* the same, it isn't always useful for discovering malware that *behaves* the same. Using the list of imported library functions is a good, but limited first step. Ideally, we could convert the binary back into its original source code, complete with comments, and extract features from there. This, however, is a hopeless endeavor as much of the information is lost when the source code is compiled. Comments and variable names, for example, are usually completely tossed away and thus unrecoverable. As a next-best option, a disassembly of the binary (described below) is used.

Many static features begin with a disassembly stage, but perform further abstractions on the disassembly before extracting features. These types of features will be discussed in later sections. Here, we describe what disassembly is and the features that are extracted directly from it.

7.2.1 Disassembly

Disassembling a binary refers to extracting its assembly code (often simply referred to as "code" or "disassembly" for short). Assembly is a low level programming language that describes in human readable terms the actions of the processor. Recovering the assembly code is a matter of parsing the binary and mapping the bytes back into the human readable terms. The mapping between bytes and assembly instructions is one to one, but due to the complexity of the most prevalent architecture (the $\times 86$ family), statically accomplishing this task is not as straightforward as it may seem. It is further complicated by malware authors taking deliberate steps to break disassembly [20, 21]. Despite these complications, robust tools have been developed that can provide surprisingly accurate results.

It may seem that an easy way to extract the disassembly of a binary is to do it dynamically instead of statically. In other words, execute the binary and record the instructions as interpreted by the processor. This has been done [22–24], but what results isn't a full disassembly of the binary, but rather an *instruction trace*. Only the instructions that were actually executed by the processor will be disassembled and recorded. A single execution path through dynamic analysis is likely to only encounter a small percentage of the full binary. When it is desired to extract the full disassembly of the binary, it is still best done statically.

An example of disassembly is given below in Table 3. At the far left is the C program from which the binary was compiled. It is a simple program that does nothing useful; it just increments a variable ten times. Next are the instruction addresses and the bytes that comprise the individual instructions. The disassembled instructions are presented next to and on the same line as the byte code representation of the instructions.

The first three instructions set up the stack (the place in memory the variables are stored). The registers ebp and esp are the base pointer and stack pointer. The base pointer points to the base of the stack and the stack pointer points to the top. The next three instructions (addresses 80483ba–80483c8) initialize the variables i and j. The variable j in placed on the stack first (memory address [ebp-8]), i second (memory address [ebp-4]). Variable i is set to zero twice because this is the case in the source code (when it is initialized and when the loop starts). The top of the loop is at address 80483d1 where j is incremented by 1, followed by an increment of i and a check if i is equal to 9 (less than 10 condition in the source). If it is not, control flow is returned back to the top of the loop. If i is equal to 9, then the final instructions are executed, exiting the program.

Table 3 Example disassembly

Source code	Address	Byte code	Disassembly
int main() {	80483b4	55	push ebp
int j = 0;	80483b5	89 e5	mov ebp, esp
int i = 0;	80483b7	83 ec 10	sub esp, 0x08
for (i=0; i< 10; i++){	80483ba	c7 45 f8 00 00 00 00	mov [ebp-8], 0
j++;	80483c1	c7 45 fc 00 00 00 00	mov [ebp-4], 0
}	80483c8	c7 45 fc 00 00 00 00	mov [ebp-4], 0
}	80483cf	eb 08	jmp 80483d9
	80483d1	83 45 f8 01	add [ebp-8], 1
	80483d5	83 45 fc 01	add [ebp-4], 1
	80483d9	83 7d fc 09	cmp [ebp-4], 9
	80483dd	7e f2	jle 80483d1
	80483df	c9	leave
	80483e0	c3	retn

7.2.2 Opcodes and Mnemonics

One of the most common types of features directly based on the disassembly is opcodes [25–29] or mnemonics [30, 31] of the disassembled instructions, usually in ngram or nperm form. The opcode of an instruction is the first byte or two that tells the CPU what type the instruction is (ex. add, move, jump) and the types of operands to expect (register, memory address). A mnemonic is simply the human readable symbol that represents the opcode. Several opcodes will map to the same mnemonic because a mnemonic does not contain the operand information of the opcode. For example, the mnemonic for adding two values together is `add`, but there are around eight different opcodes for this one `add` mnemonic.

Examples of opcodes and mnemonics are given in Table 4. The assembly and corresponding byte codes are given first, followed by the instruction's opcode and mnemonic. All the instructions listed have unique opcodes, but there are only two unique mnemonics: push and mov. For push instructions, the opcode tells the processor which register to push onto the stack. The opcode will be 50+ a number representing which register was pushed. In a mov, the opcode indicates the kind of move that will occur. For example, opcode 8b indicates that the move will be from register to register, while opcode 89 tells the processor that the move will be from a register into a memory location. The next byte in the instruction tells the processor which register combinations will be used. The byte 'ec' explicitly indicates that the contents of esp will be moved into ebp. Byte 45 means the contents of eax will be moved into the address specified by the value currently in ebp plus a one byte offset; the next byte is this offset.

Using opcodes or mnemonics abstracts away the exact operands used for the instructions. This enables code that differs in only the relative jump addresses, for example, to produce the same set of features. The noise of structural based features is thus reduced without much loss of information. Obviously, mnemonics abstracts the features further than opcodes because opcodes still contain information of the types of operands. An instruction adding two registers will have a different opcode but the same mnemonic as an instruction adding a register address and an immediate (constant).

Table 4 Example opcodes and mnemonics

Assembly	Bytes	Opcode	Mnemonic
push eax	50	50	push
push ebp	55	55	push
mov ebp, esp	8b ec	8b	mov
mov [ebp, oxf8], eax	89 45 f8	89	mov

7.3 Control Flow Based

The control flow of a program is the way in which execution, or control, is capable of passing through the program. Control refers to the part of a program that is currently executing, i.e. is in control. Control flow, then, is the various execution paths that can be taken through the program.

Control flow information can either be captured globally or locally. In general, the complete information regarding all possible paths that can be taken through every single part of the program is too great and too complicated to be of use. Thus, global control flow is typically captured by recording how different pieces of the program interact with one another. Within each piece, local control flow will then be separately recorded. In malware, this is done using a call graph and control flow graphs. A call graph records which functions call each other and a control flow graph records for a single function the possible execution paths through that function.

7.3.1 Callgraph

A *callgraph* is a directed graph depicting the calling relationships between the procedures of the program, i.e. which procedures contain calls to which other procedures. A callgraph does not give information on how control flows through the procedure itself, just how it is transferred between procedures. This provides a coarse, high level overview of the flow of control and data through the entire binary. The use of call graphs for malware analysis was first proposed by Carrera and Erdelyi [32] and further refined by Briones and Gomez [33] and Kinable and Kostakis [34].

An example call graph with the C source it was derived from is given in Fig. 4. The call graph was derived from source code because this makes the concept easier to comprehend, but the exact same concepts apply at the assembly level as well. This source has five functions: main, doAwesome, doMoreAwesome, doWork, and doAwesomeWork. Each of these five functions is represented by a single node on the call graph. An edge between two node represent a “calls” relationship. For example, the edge between main and doWork represents the relationship “main calls doWork.” Loops indicate recursion, such as the loop from doMoreAwesome onto itself. The function doMoreAwesome recursively calls itself until sufficient awesome has been achieved.

While a callgraph can be constructed statically or dynamically, it is usually done statically. This is because the purpose of a call graph is generally to show which functions *can* call which other functions, not to only show that a function *did* call another function in a *single* execution path. Static analysis is more capable than dynamic analysis in extracting this full call graph. Dynamic analysis will only be able to determine calling relationships that it actually witnesses.

(a)

```
void doAwesome( ) {
    // Does Awesome
}
void doMoreAwesome( ) {
    // First doAwesome
    doAwesome( );
    if (Need More Awesome) {
        doMoreAwesome( );
    }
}
void doWork( ) {
     // Do work
}
void doAwesomeWork( ) {
    // First do work
    doWork( );
    // Then make it awesome
    doMoreAwesome( );
}
int main( ) {
    doAwesomeWork( );
}
```

(b)

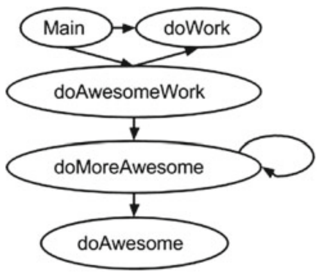


Fig. 4 Example callgraph

7.3.2 Control Flow Graph

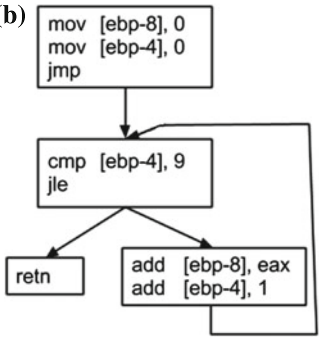
At a finer grain than the call graph is the *control flow graph* (CFG) [35]. A CFG is a graph of the control flow *within a single procedure*. Just as a call graph provides a view of control flow at the binary level, a CFG provides this view at a procedure level. The CFG is created by first breaking the sequential code of the procedure into discrete blocks called *basic blocks*. Basic blocks are constructed such that if control reaches the block, every instruction within the block is guaranteed to execute; basic blocks have a single entry and a single exit point. Edges in the CFG represent decisions made about the sequence of instructions to execute. For example, an if statement will create two edges, one representing the "TRUE" path, the other the "FALSE" path.

An example of creating a CFG is given in Fig. 5. The Assembly from which the CFG is created is given on the left and the corresponding CFG on the right. The assembly code initializes a variable at memory location ebp-8 and adds the value of eax to it 10 times. The is equivalent to multiplying eax by 10 and storing the value in the variable. The first basic block, consisting of two moves and a jump, initializes the variables and jumps to the loop condition at the end. The beginning of the new basic block will not be immediately after the jump instruction, but rather where the jump instruction goes to. In this case the basic block will start at the condition check. This condition check will exit if the counter stored at memory location ebp-4 is equal to 9. Otherwise it will go to code that adds eax and increments the counter before returning back to the check.

(a)

```
0: mov    [ebp -8], 0
1: mov    [ebp -4], 0
2: jmp    5
3: add    [ebp -8], eax
4: add    [ebp -4], 1
5: cmp    [ebp -4], 9
6: jle    3
7: retn
```

(b)



```
mov  [ebp-8], 0
mov  [ebp-4], 0
jmp

cmp  [ebp-4], 9
jle

retn

add  [ebp-8], eax
add  [ebp-4], 1
```

Fig. 5 Example control flow graph (CFG)

This example illustrates how the single entry, single exit property breaks apart the code into basic blocks. Just because instructions are sequential without a branching instruction between them does not mean that they will all be in a basic block together. The add instructions and condition check in addresses 3–6 are sequential, but they are broken into two basic blocks because the jump at address 2 branches into the middle of this sequence.

7.4 Semantics Based

In any programming language there are infinitely many ways to accomplish any given task, even a task as simple as assigning zero to a variable. As discussed in Sect. 5, this fact is often used by malware authors to create differing versions of malware that accomplish the same goal. In response, malware analysts have created features that do not rely on what the code *is* but rather what the code *does*, i.e. its *semantics*.

7.4.1 State Change

In static analysis, semantics is often defined as the effect executing the code will have on the hardware state [36–38], i.e. the change in values stored in registers and memory. Not every change need be recorded but rather the end result of a given portion of code. The size of these portions differ, but generally it is either a basic block [37, 38] or a procedure [36]. Computing the effect of executing a basic block is straightforward as, barring exception handling, execution flows through successive instructions in the block. The semantics of a block follows from the functional composition of semantics of individual instructions. Doing the same for a procedure, however, is not straightforward, especially, as is usually the case, when its CFG has branches and loops.

There are two methods of representing semantics: (a) enumerated concrete semantics and (b) symbolic expressions. In the first method the semantics is represented as a set of pairs of specific, concrete input and output states. For instance, the pair (`eax = 5, ebx = 20`) may represent that when executed with the value 5 in the register `eax`, the program upon termination will have the value 20 in `ebx`. A set of such pairs, of course with significantly more complex input and output states, would represent the semantics of the code. Using symbolic expressions instead, the semantics may be represented succinctly as a single expression. For instance, the expression `ebx = pre(eax)* 4` may represent that the value of `ebx` upon termination is the 4 times the value of `eax` at the start.

The benefit of using enumerated concrete semantics is that two different ways of effecting the same change of state will have exactly same input and output pairs. Alternatively, even a single difference in the input and output pair would imply a different semantics. The challenge, of course, is that a program may have exponentially large space of input, making it infeasible to compute and represent such semantics. On the other hand, symbolic expressions offer the advantage that a single expression may represent the entire semantics. However, since one cannot guarantee that two equivalent program always produce the same expression—this would solve the Halting Problem—differences in the semantic expression need not imply that the underlying code has different semantics.

The two different methods of representing, as well as computing, semantics have been used respectively by Jin et al. [37] and Lakhotia et al. [38]. Jin et al. [37] address the issue of exponentially large input space by randomly sampling the space. They pre-generate a large number of input states and use them to compute the semantics of all basic blocks. Each basic block is executed using the same set of input states and the corresponding output state recorded. A basic block can then be represented by a hash of all of its output states.

Lakhotia et al. [38] use symbolic interpretation to compute the semantics of blocks, without assigning any concrete values. Rather than execute a basic block with specific inputs, [38] execute it with symbolic inputs. Furthermore, they use algebraic properties, such as the distributive law, and simplifications to map expressions into a canonical form when possible. For example, they simplify the expression `(eax + 4) * 5` to `eax * 5 + 20`. This ensures that, for a large set of operations, all expressions that are functionally equivalent resolve to the same symbolic representation. After computing the symbolic output values, Lakhotia et al. [38] further abstract the semantics by converting the concrete register names and numbers into logical variables. This type of feature they call the "juice" of the basic block and term it "GenSemantics."

Tables 5 and 6 respectively illustrate how the features of [37, 38] work. In both tables, the left column contains two functionally identical basic blocks. Both tables contain the same two blocks. The two presented blocks both assign 3 to eax, set the value of edx to ecx, and multiply by 3 the value of ebx + 4. The first block uses a multiply instruction; the second block uses a series of adds to have the same effect.

Table 5 ([37]) shows the result of executing the two basic blocks on two sets of input values. The columns labeled *Input 1* and *Input 2* show the input states, as values

Table 5 Features by Jin et al. [37]

Basic block	Input 1	Output 1	Input 2	Output 2
add ebx,4	eax = 33	eax = 3	eax = 62	eax = 3
mov eax,3	ebx = 4	ebx = 21	ebx = 7	ebx = 33
mul ebx,eax	ecx = 25	ecx = 58	ecx = 72	ecx = 54
mov ecx,edx	edx = 58	edx = 58	edx = 54	edx = 54
add ebx,4				
mov eax,3	eax = 33	eax = 3	eax = 62	eax = 3
mov ecx,ebx	ebx = 4	ebx = 21	ebx = 7	ebx = 33
add ebx,ecx	ecx = 25	ecx = 58	ecx = 72	ecx = 54
add ebx,ecx	edx = 58	edx = 58	edx = 54	edx = 54
mov ecx,edx				

Table 6 Features by Lakhotia et al. [38]

Basic block	Semantics	GenSmantics
add ebx,4		
mov eax,3	eax = 3	A = N1
mul ebx,eax	ebx = pre(ebx) * 3 + 12	B = pre(B) × N1 + N2
mov ecx,edx	ecx = pre(edx)	C = pre(D)
add ebx,4		
mov eax,3		
mov ecx,ebx	eax = 3	A = N1
add ebx,ecx	ebx = pre(ebx) * 3 + 12	B = pre(B) × N1 + N2
add ebx,ecx	ecx = pre(edx)	C = pre(D)
mov ecx,edx		

in the registers eax, ebx, ecx, and edx. The columns labeled *Output 1* and *Output 2* show the corresponding states after executing the blocks.

As can be seen in the example, the outputs of the two blocks are exactly the same when using the same set of input values. It is important that the same list of input values be used across all blocks in all binaries that are going to be compared. As should be obvious, only features created with the same list of inputs can be meaningfully compared. Thus, as in this example, there will often be more generated values than needed because we have to generate enough values so that *all possible* basic blocks will have enough input values.

Table 6 gives [38] features extracted from the same two blocks. The middle column provides the initial semantics extracted through symbolic execution and simplification. The notation `pre(x)` represents the value of x when the basic block was reached; it is a symbolic notation for the input value. The right column contains the generalized semantics.

Chaki et al. [36] tackles the problem of computing the semantics at the level of the procedure instead of at the basic block level. Computing a single expression that represents the semantics of an entire expression becomes significantly more challenging, more so if the semantics is to be in a canonical form to ease comparison. For instance, computing the semantics of a loop would involve computing an expression that represents its fixed point. However, if a single path through the procedure is selected, the semantics can be computed as straightforwardly as in basic blocks. This is what [36] do. Given a single path through the procedure, they compute features very similar to the GenSemantics of [38]. This process is repeated for all bounded paths through the procedure. For unbounded paths, due to cycles, a bounded depth first traversal is used to limit the length of the path traversed. The features from all the traversed paths are unioned together to represent the whole procedure.

7.4.2 API Calls

Another way of defining the semantics of a program is through the set of API calls the program makes. An API call refers to when a program calls a function provided by some library, often the system library. The system library is a library of functions provided by the operating system to perform tasks that only the operating system has permissions to perform directly, such as writing a file to disk. Other well known libraries include the standard C library, encryption libraries, and more. If a program makes an API call to a well known and widely distributed library, then the purpose of that call can easily be determined.

When the order in which API calls are made is preserved in an *API trace*, the semantic information becomes even stronger. Sequences of API calls can be used to potentially determine the presence of high level behaviors [22, 39] such as "walks through a directory" or "copies itself to disk."

The set or trace of API calls provides a "behavioral profile" of the binary being examined. This idea was first proposed by bailey et al. [40] and later formalized by Trinius et al. [41]. Trinius et al. [41] designed what they called a *Malware Instruction Set* (MIST) for representing various layers of semantic information present in API calls. The layers are ordered from most to least abstract. Each "instruction" in MIST represents a single API call.

The first layer of a MIST instruction contains a category for the API call and the operation performed by the call. The category represents the type of object the API call operates on. For example, an API call that writes a file to disk would be in a "File System" category, one that opens a socket in a "Network" category, and an API that edits a registry value would be in a "Registry" category. The operation is, as the name implies, what was done to the object. The "File System" category, for example, has operations such as "open_file", "read_file", "write_file". Every category has a pre-specified set of possible operations. Both the category and operation are derived from the name of the API call.

Subsequent layers of a MIST instruction are derived from the parameters to the API call and ordered by the expected variability of their values. For example, an API call that manipulates a file will contain as one of its parameters the name of the file to manipulate. From this file path, we can separate out two parts, the path and the name.

The path will have a lower variability than the file name. A binary that writes to the windows directory is likely to write to the windows directory on every execution. However, the names of the files may well be different for every execution, since its common for malware to randomly generate filenames. Thus, the file path has a lower expected variability than the file name and is put at a higher level.

7.5 Hybrid Features

In an effort to increase the accuracy of classification, various authors have utilized feature fusion: combining multiple types of features to created hybrid features.

The first to use feature fusion was Masud et al. [42]. They combined features from three different levels of abstraction: binary n-grams, derived assembly features (DAF), and system calls. Binary n-grams are the same as described in Sect. 7.1. The feature on the next level of abstraction is the DAF. A DAF is basically just the disassembled binary n-grams. A DAF is created by extending the n-gram on the front and the back as needed to fit instruction boundaries and then disassembling these instructions. At the highest level of abstraction are system calls. The names of the system calls present in the header of the executable were extracted and used.

Lu et al. [43] combine static and dynamic features. The static features chosen were the names of system calls present in the binary of the executable and the dynamic features were expert defined behaviors commonly seen in malicious programs. Several examples of the behaviors searched for are packing the executable, DLL injection, and hiding files. There were twelve such features defined. These static and dynamic feature sets were fused by taking the union of the feature sets.

Islam et al. [44] simply combined two features they considered useful in an attempt to make an even more useful feature set. They take the union of function length frequency and printable string feature sets. Function length frequency is the number of functions a program has of a given length. It was found by Islam that malware within the same family tend to have functions of around the same length, thus motivating the use of function length frequency as a feature. Printable strings are simply sequences of bytes which can be interpreted as valid ASCII strings.

LeDoux et al. [45] takes a different approach. Whereas the above cited works all combined mutually exclusive feature sets, LeDoux et al. fuses features that are very similar, but collected in different ways. The hypothesis being that the differences present between two feature sets that should be identical is itself a feature and can help identify malware that is intentionally obfuscated against one of the methods of collecting features.

8 Supervised Learning

Supervised Learning describes a class of learning algorithms that build a model used in classification (see Sect. 6.2). These algorithms learn the concepts represented by the model by examining labeled examples of the concepts. The label provides the "correct answer," that is the concept that is being exemplified. This training data (the labeled examples) is often referred to as the ground truth. After the model has been learned, a never before seen example of a learned concept can be classified by matching it to the learned model.

The method of learning and classifying used by Supervised Learning can conceptually be thought of as teaching a child different types of animals by showing him flash cards and telling him the name of the animal. "This is a horse. This is a dog. This is another horse. This is a cat." By looking at many examples of horses, dogs, and cats, the child builds up an internal representation of each animal type. Then when shown a picture of a never before seen dog, and asked to "classify it", the child will be able to respond "It is a dog."

One of the limitation of classifiers built using supervised learning is that they will *only* be able to attach a label it has already learned and *must* attach one of these labels. Supervised learning can only explicitly learn concepts it is told about through the labels in the training data. Neither is it able to provide an "I don't know" answer. It must attach one of its learned labels. To extend our earlier example further, if our child is only shown pictures of horses, cats, and dogs, and then asked to "classify" a picture of a lion, he will respond "cat," not lion. If shown a picture of a tractor, however, the child will be able to respond "I don't know." A classifier built using supervised learning would respond "horse."

Another potential issue with supervised learning specific to the malware domain is the reliance on the ground truth labels. In malware analysis, ground truth can often only be reliably determined through manual reverse engineering and analysis of the binary by a human expert. The expert makes a judgment call based upon his experience and assigns an appropriate label to the binary. This process is expensive in terms of time and resources required, thus restricting the number of labels that can be reasonably generated. The required level of expertise for performing this task limits the reasonable size of ground truth even further (crowd sourcing can't solve this problem). As an added detriment, any process that relies on human decisions is prone to error, regardless of the level of expertise of the human. Training a Supervised Classifier on improperly labeled data can lead to very poor performance.

The main application of supervised learning to malware analysis has been in attempting to built automated malware detectors [46], in triage [47], and in evaluating the quality of malware features. The original intention of supervised learning in malware was to build a classifier that could act as the scanner on the end system. However, the false positive rate of these classifiers tend to be too high. A false positive is when the classifier labels a benign program as malware. On the end system, the false positive rate must be extraordinarily low as constantly labeling valid programs as malware will at best annoy the user into not using the scanner and at worst break the system by removing critical system files.

While not useful as a scanner, supervised learning has been of use in triage [47]. Supervised Learning can be used to build a classifier that identifies if a new, incoming program belongs to any of the already known and analyzed malware families. Knowing which family malware belongs to makes any manual analysis needed an easier prospect because the analyst does not have to start from nothing, but can leverage knowledge of what that family of malware does and how it usually operates. A higher false positive rate can be tolerated in the triage stage than in a scanner because the results are still verified later on.

A final use of supervised learning in malware is for controlled test and comparison of the quality of malware features. Due to the fact that Supervised Learning required that data with ground truth exists, determining performance metrics for classification is straightforward. A portion of the labeled data is set aside for testing purposes and the rest of the data used for creating the model. The testing data is classified using the created model and labels assigned by the model are compared against the ground truth labels to determine accuracy of classification. To compare the quality of features, the accuracy of classification can be compared. Features that are of a higher quality will result in a higher accuracy.

9 Unsupervised Learning

Unsupervised Learning [34, 48–51] algorithms learn concepts without the use of any labeled data. Labeled data may be used to perform a post hoc evaluation of the learned concepts, but they are in no way used for learning. Concepts are learned in unsupervised algorithms by clustering together related objects in such a way that objects within a cluster are more similar to each other than objects outside the cluster. Each cluster then represents a single concept. What this concept is, however, is not always easy to discover.

In order to determine which binaries are related and so belong in the same cluster, unsupervised learning algorithms rely on measuring the *similarity* of malware. A comprehensive overview of the various similarity functions that can be used to compare binaries is given by Cesare and Xiang [52]. It is important to note that similarity metrics do not directly measure the similarity of two binaries, but rather the similarity of the binary's *features*. The same pair of binaries may have very different similarities depending on the choice of feature type. Take, for example, two malware with dissimilar implementation but an almost identical set of performed system calls. Features that rely mostly on the implementation, such as opcode n-grams, will result in measuring a low similarity. Features that only look at system calls, however, will result in the exact opposite: high similarity.

One of the major issues with clustering is that many of the most common clustering algorithms require a priori knowledge of the number of clusters to create; the number of required clusters is a parameter to the clustering algorithm. When it is known how many malware families are present in the data set, it is common practice to use this number as the number of clusters. However, while the number of families present in a laboratory sample of malware may be determined, the same cannot be said of the sets of malware encountered "in the wild." If the incorrect number of clusters is selected, performance will quickly degrade.

Fortunately, not all unsupervised algorithms require prespecifying the number of clusters to create. Representative of these algorithms, and the most commonly used, is *hierarchical clustering*. Hierarchical clustering can be either agglomerative or divisive. Agglomerative Hierarchical clustering works by iteratively combining the two most similar items into a cluster. Items include both the individual objects and already created clusters. So a single iteration could consist of putting two objects into a cluster of size two, adding an object to an already existing cluster, or combining two clusters together. This process is repeated until everything has been put into a single cluster. Each step is tracked, resulting in a hierarchical tree of clusters called a dendrogram. Divisive hierarchical clustering works similarly, but starts with everything in a single cluster and iteratively splits clusters until everything is in a cluster of size one.

Hierarchical clustering changes the problem from needing to know a priori the optimal number of clusters to knowing when to stop the agglomerative (or divisive) process, i.e. where to cut the tree. This is determined using a type of metric known as a clustering validity index [53]. While a number of such indices exist, they are all a measure of some statistic regarding the similarity of objects within the clusters, the distance between objects in different clusters, or a combination of the two. A number of these metrics were evaluated by [50] for performance in the malware domain using a particular feature type.

10 Hashing: Improving Clustering Efficiency

In applications of clustering in malware, there are two inherent performance bottle necks: large feature space and the requirement to compute all pairwise similarities. The number of features generated for each individual malware is usually quite large. A single malware can have thousands, even hundreds of thousands of features representing itself. This has a two fold effect. First, it drives up the time cost of doing a single similarity computation. Second, storing this massive amount of features requires an equally massive amount of memory. Either a specialized computer with the required amounts of memory must be used, or the majority of the time spent computing similarities will be in swapping the features to and from disk.

Compounding the time and space requirements of clustering is the complexity of a typical clustering algorithm. Even efficient algorithms run at $O(n^2)$ due to the need to compute the similarity between every pair of objects.[2] This is tolerable for small to

[2] Objects don't need to be compared with themselves and similarity functions are (typically) symmetric, so the actual number of comparisons required is $(n^2 - n)/2$).

medium sized datasets, but severely limits the scalability of clustering. Parallelism can help up to a point, but handling millions of malware requires more efficient approaches.

The solutions being explored for both of the above problems are the same: hashing. Feature hashing based on Bloom filters [54] is being used to both reduce the memory overhead and the time a single similarity comparison takes. The number of required pairwise comparisons is being reduced or even eliminated by performing clustering based on a shared hash value. A few explorations have been made in using cryptographic hashes [51, 55, 56]. More commonly, however, Locality Sensitive Hashing is being used as a "fuzzy hash" to reduce the complexity of clustering [37, 48]. In some instances, constant time is even achieved [37]!

10.1 Feature Hashing

To reduce the memory requirement and time for a single similarity comparison, [54] present a method for hashing features into a very small representation using Bloom filters [57]. A Bloom filter is a probabilistic data structure used for fast set membership testing. It consists of a bit vector of size m with all bits initially set to zero and k hash functions, $h_1, h_2, \dots .h_k$, that hash objects into integers uniformly between 1 and m, inclusive. In practice, these k hash functions are approximated by simply splitting the MD5 into k even chunks, taking the modulo of each chunk with m, and then treating each resulting value as the result of one of k hash functions. To insert object $\times$ into the bloom filter, for each $h_i(x)$, set the bit at position $h_i(x)$ to one. Object y can then be tested for membership by checking that all bits at positions $h_i(x)$ for each i from 1 to k is set to one.

Jang et al. [54] perform feature hashing by inserting all features for a single binary into a bloom filter with only one hash function. The decision to use only one hash function was made in order to facilitate fast and intuitive similarity comparisons. Similarity between malware can be quickly approximated by measuring the number of shared and unique bits in each binary's corresponding bloom filter. In addition, only the bloom filters need to be stored, not every single feature, thus reducing the total storage requirement. Jang et al. [54] experienced compression rates of up to 82 times!

10.2 Concrete Hashes

Cryptographic hashes such as MD5 and SHA-1 are widely used to filter out exact binary duplicates. However, hashes at this level are not robust; changing a single bit can result in a dramatically different hash. Several approaches have been taken to improve this robustness by taking hashes of different types of features instead of the raw bits of the binary.

Wicherski [51] attempted to use the information stored in the PE header of a binary to define a hash for the purpose of filtering out duplicate binaries. The information in the PE header was utilized to make the hash robust against minor changes in the binary. For example, if the only difference between two binaries is a time stamp in the header, the binaries should be considered equal. To create this hash, [51] takes a subset of bits from several PE header fields, concatenates them together (in a fixed, reproducible ordering) and takes the SHA-1 of this bit sequence.

Wicherski [51] does not use all bits from all PE header fields, but rather judicially selects both the fields and bits used in the construction of the hash. The fields used are image flags such as whether the binary is a DLL, the Windows subsystem the binary is to be run in, the initial size of the stack and the heap, the initial address each section of the binary is loaded into memory at, the size of each section, and the flags for each section that indicate permissions and alignment. For each PE field included in the hash, only a subset of the full bits in the field are utilized. The exact bit range used for each field is chosen such that a change in one of bits indicates a major structural change in the binary. For example, the first 8 bits of the 32 bit initial stack size are almost always 0, and there is often minor changes in the lower bits in polymorphic malware.

In a different approach, [55, 56] use cryptographic hashes of functions in order to discover when malware is sharing code. The only modification made by Cohen and Havrilla [56] to the code before hashing is that all constants, such as jump and memory access addresses, are zeroed out. A cryptographic hash of the string representation of the code of the procedure is then taken. LeDoux et al. [55] use the abstractions defined by Lakhotia et al. [38] and described in Sect. 7.4. For each basic block in a procedure, the feature of Lakhotia et al. [38] is computed. A hash of the procedure is created by sorting and concatenating all the basic block features together and taking a cryptographic hash.

10.3 Locality Sensitive Hashing

To improve upon the robustness of concrete cryptographic hashes, MinHash, a type of Locality Sensitive Hash (LSH) [58] is being utilized in several different ways. A MinHash is a hash functions with the property that hashes of two arbitrary objects will be equivalent with a probability equal to the Jaccard Similarity between the objects. In other words, if two binaries are 90 % similar, there is a 90 % chance that they will produce the same LSH. Jaccard Similarity is a method for measuring the similarity of two sets and is defined as $|A \cap B|/|A \cup B|$, the size of the intersection divided by the size of the union. Two arbitrary MinHashes, then, will collide with a probability equal to the Jaccard Index of the two hashed objects, $P(MinHash(A) = MinHash(B)) = |A \cap B|/|A \cup B|$.

To create a MinHash of a set, an ordering over the universe of objects that can be placed in the set must be defined. This ordering can be arbitrary, but it must be fixed. A very simple example ordering is just “the order in which I first encountered the object” (this ordering does, however, need to be remembered across all sets that are to be compared). After this ordering is defined, several random permutations of the ordering are selected. A MinHash is then the minimum elements of the set as defined by the selected random order permutations.

For an illustrative example of MinHash, take the sets $A = 1, 2, 3$ and $B = 4, 5, 6$ with the ordering defined numerically. To create the MinHash, we first randomly select a number of permutations of the ordering, let’s say these are 5, 4, 2, 1, 3 and 3, 1, 4, 5, 2. Then the MinHash of set A is 2,1 and that of B is 5, 4.

There are two ways in which MinHashes are being used in malware analysis: for direct clustering and as a filter to reduce the complexity of clustering. The computed MinHashes can be used to define a clustering that has constant time complexity. Two binaries are considered to be in the same cluster if and only if they have the same MinHash. This was an approach taken by Jin et al. [37].

Rather than directly clustering, MinHashes can also be used as a filter to drastically reduce the size of n so that the clustering complexity of $O(n^2)$ becomes bearable. Bayer et al. [48] first utilized this approach in the malware space. They defined many MinHashes by selecting a different set of random permutations for each MinHash. Malware were first put into an initial cluster if *any* of the MinHashes matched. Bayer et al. [48] would then computed the Jaccard Similarity for only pairs of binaries *in the same cluster*. The resulting similarities were then used to create a final clustering. Since n now refers to the number of object in a single cluster, $O(n^2)$ becomes a tolerable performance.

11 Semi-supervised Learning

Halfway between Supervised and Unsupervised Learning is the wonderful world of Semi-supervised Learning. As the name implies, Semi-supervised Learning utilizes both labeled and unlabeled data. It is usually used for clustering with the available labels helping decide both the number and shape of clusters to create.

While there are a number of existing Semi-supervised learning algorithms [59], there currently exists only one instance of such an application in malware. Santos et al. [60] use a Semi-Supervised Learning algorithm known as “Learning with Local and Global Consistency” (LLGC) [61] to classify binaries as malicious or benign.

LLGC first starts with a directed, weighted graph representation of the data. The nodes of the graph are binaries being clustered and the weighted edges are the similarity values between the binaries. Every node keeps track of how much it “believes” it belongs to a particular label by attaching a weight to the different labels. When the graph is first constructed, unlabeled data has zero belief in any of the labels and labeled data has perfect belief in its label. So a binary labeled “malware” will start with a weight of 1 for the label “malware” and a weight of 0 for the label “benign.” An unlabeled binary will have a weight of 0 for both labels.

LLGC relies on two assumptions to create clusters. The first is that similar objects are likely to have the same label (“Local” consistency). The second is that objects in the same cluster are likely to have the same label ("Global" consistency). LLGC uses these two assumptions to "spread" beliefs through the initial graph. Nodes with higher similarity with each other will be incrementally updated so that their weights assigned to labels more closely mirror each other. The incremental updates continue until weights attached to labels converge. Each node is then assigned the label it has the highest weight attached to, i.e. the label it has the highest belief in.

12 Ensembles

Similar to the way in which features can be combined to create a more accurate classification, different learning algorithms can be combined to produce a single, more accurate learner. Such a learner is referred to as an *ensemble*. There are several ways in which classifiers can be combined [62]: voting, stacking, bagging, and boosting.

One of the simplest methods of creating an ensemble is *voting* [63–65]. This simply consists of creating a number of independent classifiers, running them separately, obtaining the different outputs, and then using some type of voting mechanism to determine which of the outputs to accept as the final answer. This voting mechanism most often takes the form of majority vote (simply select the answer that the most classifiers returned). Other forms of voting include weighted majority vote, a veto vote [64] where a "special" classifier can veto the decision of the majority, and a trust-based vote [65] where voting takes into account how much each individual classifier is "trusted" to provide a correct answer.

The above voting strategies really only apply to learning algorithms that perform classification and do not apply to clustering. A majority vote, for example, doesn't make sense for clustering as the number of possible ways to cluster is practically unbounded. It is unlikely that any two partitioning created by two different clustering algorithms will be the same. Instead, a *consensus partition* [66, 67] is used. Consensus partitioning can conceptually be thought of as a voting scheme for clustering. Like in voting ensembles, the first step is to cluster the data several times using independent methods each time. In the "combining phase" a new partitioning of the data is created that maximizes the "consensus" between each independent clustering. There are a number of ways that have been proposed to create this consensus clustering using graph based, combinatorial, or statistical methods. Various ways of creating this consensus clustering are covered by Strehl and Ghosh [68] and Topchy et al. [69].

Stacking [43, 70–72] is a type of ensemble in which a set of classifiers are connected in series, with each classifier taking as input the output of the classifier before it in the series. A simple example of such a classifier is [43] and their system called SVM-AR, depicted in Fig. 6. In this system, [43] first use a supervised classifier to decide if an executable is malicious or benign. After a decision is made, a trained rule-based classifier is used to "check" the results. If the supervised classifier said the executable was malware, the executable is checked against the rules for determining if an executable is benign. If any of these rules match, then final decision is "benign." A similar process is applied if the supervised classifier says the executable is benign.

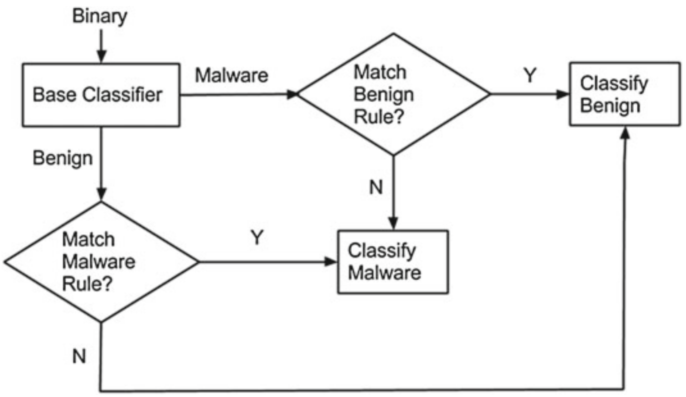


Fig. 6 SVM-AR by Lu et al. [43], an example stacking ensemble

Bagging [19] is an ensemble technique that instead of combining several different kinds of learners, uses the same learning algorithm, but trains the algorithm on random bootstrapped samples of the training data. There is a class of learning algorithms known as unstable algorithms. In unstable learning algorithms, slight changes in the training data can lead to very different models. Bagging seeks to solve this by combining many models created on slightly different data. The differing data is created through bootstrapped sampling of the original training data. A bootstrap sample is a random sample taken *with replacement*. Thus each bootstrapped sample will contain duplicates and be missing some of the original data. The learned models are usually combined through voting.

Boosting [46, 72] is meant to be an improvement over bagging. Like bagging, boosting combines many differently trained instances of the same learning algorithm. Where boosting differs is in the way the various training data sets are selected. Bagging selects training sets at random. Boosting selects training sets such that training focuses on data the already trained classifiers are getting incorrect. This results in building classifiers that specialize on specific portions of the data. Each individually trained classifier may have weak performance overall, but be extremely accurate on a specific subset of the data. Combining a number of such specialized classifiers (by voting, usually) results in more accurate ensemble overall.

References

1. Neumann, J.V. : Theory of Self-reproducing Automata. IEEE Trans. Neural Networks. **5**(1), 3–14 (1994)
2. Cohen, F.: Computer viruses. PhD thesis, University of Southern California (1985)
3. Measuring and optimizing malware analysis: An open model. L.L.C,Technical report, Securosis (2012)
4. Schon, B., Dmitry, G., Joel, S.: Automated sample processing, Technical Report, Mcafee AVERT, Auckland, New Zealand (2006)

5. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Berlin (1999). ISBN 9783540654100
6. Schwarz, B., Debray, S., Andrews, G.: Disassembly of executable code revisited. In: Proceedings of Ninth Working Conference on Reverse Engineering, IEEE, 2002, pp. 45–54
7. Collberg, C., Nagra, J.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Pearson Education (2010). ISBN 9780321549259
8. Mitchell, T.M.: Machine Learning. McGraw-Hill, New York (1997). ISBN 0070428077 9780070428072 0071154671 9780071154673
9. Shabtai, A., Moskovitch, R., Elovici, Y., Glezer, C.: Detection of malicious code by applying machine learning classifiers on static features: a state-of-the-art survey. Inf. Sec. Tech. Rep. **14**(1), 1629 (2009)
10. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware analysis techniques and tools. ACM Comput. Surv. **44**(2), 6:1–6:42 (2008). ISSN 0360–0300. doi:10.1145/2089125.2089126
11. Arnold, W. Tesauro, G.: Automatically generated WIN32 heuristic virus detection. In: 2000 Virus Bulletin International Conference, pp. 51–60. The Pentagon, Abingdon, Oxfordshire, OX14 3YP, England, Virus Bulletin Ltd (2000)
12. Kephart, J.O., Arnold, B.: Automatic extraction of computer virus signatures. In: Ford, R. (ed.) 4th Virus Bulletin International Conference, pp. 178–184, Abingdon, England, Virus Bulletin Ltd (1994)
13. Kephart, J.O., Arnold, B.: A biologically inspired immune system for computers. In: Fourth International Workshop on the Synthesis and Simulation of Living Systems, pp.130–139 (1994)
14. Kephart, J.O., Sorkin, G.B., Arnold, W.C., Chess, D.M., Tesauro, G.J., White, S.R.: Biologically inspired defenses against computer viruses. In: IJCAI 95, pp. 985–996 (1995)
15. Karim, M.E., Walenstein, A., Lakhotia, A., Parida, L.: Malware phylogeny generation using permutations of code. J. Comput. Virol. **1**(1), 13–23 (2005)
16. Wang, T.-Y., Wu, C.-H., Hsieh, C.-C.: Detecting unknown malicious executables using portable executable headers. In: Fifth International Joint Conference on INC, IMS and IDC, NCM 09, pp. 278–284 (2009). doi:10.1109/ncm.2009.385
17. Walenstein, A., Hefner, D.J., Wichers, J.: Header information in malware families and impact on automated classifiers. In: 2010 5th International Conference on Malicious and Unwanted Software (MALWARE), p. 1522 (2010). doi:10.1109/malware.2010.5665799
18. Schultz, M.G., Eskin, E., Zadok, F., Stolfo, S.J.: Data mining methods for detection of new malicious executables. In: Proceedings of 2001 IEEE Symposium on Security and Privacy, S P 2001, pp. 38–49 (2001). doi:10.1109/secpri.2001.924286
19. Ye, Y., Chen, L., Wang, D., Li, T., Jiang, Q., Zhao, M.: SBMDS: an interpretable string based malware detection system using SVM ensemble with bagging. J. Comput. Virol. **5**(4), 283–293 (2008). ISSN 1772–9890, 1772–9904. doi:10.1007/s11416-008-0108-y
20. Kruegel, C., Robertson, W., Valeur, F., Vigna, G.: Static disassembly of obfuscated binaries. In: Proceedings of the 13th USENIX Security Symposium, pp. 255–270. Usenix (2004)
21. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings of the 10th ACM Conference on Computer and Communications Security, pp. 290–299, ACM Press, New York, NY, USA (2003)
22. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: Proceedings of the 1st India Software Engineering Conference, ISEC '08, p. 514, New York, NY, USA (2008). ACM. ISBN 978-1-59593-917-3. doi:10.1145/1342211.1342215
23. Debray, S. Patel, J.: Reverse engineering self-modifying code: Unpacker extraction. In: 2010 17th Working Conference on Reverse Engineering (WCRE), pp. 131–140 (2010). doi:10.1109/WCRE.2010.22
24. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Automatic reverse engineering of malware emulators. In: 2009 30th IEEE Symposium on Security and Privacy, pp. 94–109 (2009). doi:10.1109/SP.2009.27
25. Alazab, M., Kadiri, M.A., Venkatraman, S., Al-Nemrat, A.: Malicious code detection using penalized splines on OPcode frequency. In: Cybercrime and Trustworthy Computing Workshop (CTC), 2012 Third, pp. 38–47 (2012). doi:10.1109/CTC.2012.15

26. Bilar, D.: Opcode as predictors for malware. Int. J. Electron. Sec. Digit. Forensics **1**(2), 156–168 (2007)
27. Hu, X., Bhatkar, S., Griffin, K., Shin, K.G.: MutantX-S: scalable malware clustering based on static features. In: USENIX Annual Technical Conference (USENIX ATC 13), pp. 187–198 (2013)
28. Moskovitch, R., Feher, C., Tzachar, N., Berger, E., Gitelman, M., Dolev, S., Elovici, Y.: Unknown malcode detection using opcode representation. Intell. Secur. Inform. **48**, 204–215 (2008)
29. Runwal, N., Low, R.M., Stamp, M.: Opcode graph similarity and metamorphic detection. J. Comput. Virol. **8**(1–2), 37–52 (2012). ISSN 1772–9890, 1772–9904, doi:10.1007/s11416-012-0160-5
30. Chouchane, M.R., Lakhotia, A.: Using engine signature to detect metamorphic malware. In: Proceedings of the 4th ACM Workshop on Recurring Malcode, WORM '06, pp. 73–78, New York, NY, USA (2006). ACM. ISBN 1-59593-551-7. doi:10.1145/1179542.1179558
31. Hu, X., Chiueh, T.-C., Shin, K.G.: Large-scale malware indexing using function-call graphs. In: Proceedings of the 16th ACM Conference on Computer and Communications security, pp. 611–620 (2009)
32. Carrera, E., Erdelyi, G.: Digital genome mapping: advanced binary malware analysis. In: Proceedings of the 2004 Virus Bulletin Conference, pp. 187–197 (2004)
33. Briones, I., Gomez, A.: Graphs, entropy and grid computing: automatic comparison of malware. Virus Bulletin, 1–12 (2008). http://pandalabs.pandasecurity.com/blogs/images/PandaLabs/2008/10/07/IsmaelBriones-VB2008.p
34. Kinable, J., Kostakis, O.: Malware classification based on call graph clustering. J. Comput. Virol. **7**(4), 233–245 (2011). ISSN 1772–9890, 1772–9904, doi:10.1007/s11416-011-0151-y
35. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic worm detection using structural information of executables. In: Valdes, A., Zamboni, D. (eds.) Recent Advances in Intrusion Detection, no. 3858. Lecture Notes in Computer Science, pp. 207–226. Springer, Berlin (2006). ISBN 978-3-540-31778-4, 978–3-540-31779-1
36. Chaki, S., Cohen, C., Gurfinkel, A.: Supervised learning for provenance-similarity of binaries. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '11, p. 1523, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0813-7. doi:10.1145/2020408.2020419
37. Jin, W., Chaki, S., Cohen, C., Gurfinkel, A., Havrilla, J., Hines, C., Narasimhan, P.: Binary function clustering using semantic hashes. In: Proceedings of the 11th International Conference on Machine Learning and Applications (ICMLA), vol. 1, pp. 386–391 (2012). doi:10.1109/ICMLA.2012.70
38. Lakhotia, A., Preda, M.D., Giacobazzi, R.: Fast location of similar code fragments using semantic 'juice'. In: Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, PPREW '13, p. 5:15:6, New York, NY, USA (2013). ACM. ISBN 978-1-4503-1857-0. doi:10.1145/2430553.2430558
39. Pfeffer, A., Call, C., Chamberlain, J., Kellogg, L., Ouellette, J., Patten, T., Zacharias, G., Lakhotia, A., Golconda, S., Bay, J., Hall, R., Scofield, D.: Malware analysis and attribution using genetic information. In: Proceedings of the 7th IEEE International Conference on Malicious and Unwanted Software (MALWARE 2012), pp. 39–45, IEEE Computer Society Press, Fajardo, Puerto Rico, Oct. (2012)
40. Bailey, M., Oberheide, J., Andersen, J., Mao, Z.M., Jahanian, F., Nazario, J.: Automated classification and analysis of internet malware. In: RAID07: Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection, pp. 178–197, Berlin, Heidelberg, Springer-Verlag (2007)
41. Trinius, P., Willems, C., Holz, T., Rieck, K.: A malware instruction set for behavior-based analysis, Technical Report, University of Mannheim (2009). http://citeseerx.ist.psu.edu/viewdoc/download
42. Masud, M.M., Khan, L., Thuraisingham, B.: A hybrid model to detect malicious executables. In: IEEE International Conference on Communications, ICC 07, pp. 1443–1448 (2007). doi:10.1109/icc.2007.242

43. Lu, Y.B., Din, S.C., Zheng, C.F., Gao, B.J.: Using multi-feature and classifier ensembles to improve malware detection. J. CCIT **39**(2), 57–72 (2010)
44. Islam, R., Tian, R., Batten, L., Versteeg, S.: Classification of malware based on string and function feature selection. In: Cybercrime and Trustworthy Computing, Workshop, p. 917 (2010)
45. LeDoux, C., Walenstein, A., Lakhotia, A.: Improved malware classification through sensor fusion using disjoint union. In: Information Systems, Technology and Management, pp. 360–371, Grenoble, France. Springer, Berlin Heidelberg (2012). ISBN 978-3-642-29166-1. doi:10.1007/978-3-642-29166-1_32
46. Kolter, J.Z., Maloof, M.A.: Learning to detect and classify malicious executables in the wild. J. Mach. Learn. Res. **7**, 2721–2744 (2006)
47. Walenstein, A., Venable, M., Hayes, M., Thompson, C., Lakhotia, A.: Exploiting similarity between variants to defeat malware. In: Proceedings of BlackHat Briefings DC 2007 (2007)
48. Bayer, U., Comparetti, P.M., Hlauschek, C., Kruegel, C., Kirda, E.: Scalable, behavior-based malware clustering (2009). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.148.7690&rep=rep1&type=pdf
49. Gurrutxaga, I., Arbelaitz, O., Ma Perez, J., Muguerza, J., Martin, J.I., Perona, I.: Evaluation of malware clustering based on its dynamic behaviour. In: Roddick, J.F., Li, J., Christen, P., Kennedy, P.J. (eds.) Seventh Australasian Data Mining Conference (AusDM 2008), Crpit, vol. 87, pp. 163–170, Glenelg, South Australia, Acs (2008)
50. Wang, Y., Ye, Y., Chen, H., Jiang, Q.: An improved clustering validity index for determining the number of malware clusters. In: 3rd International Conference on Anti-counterfeiting, Security, and Identification in Communication, 2009, ASID 2009, pp. 544–547. doi:10.1109/ICASID.2009.5277000
51. Wicherski, G.: peHash: a novel approach to fast malware clustering. In: Proceedings of LEET09: 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (2009)
52. Cesare, S., Xiang, Y.: Software Similarity and Classification. Springer, Heidelberg (2012)
53. Legany, C., Juhsz, S., Babos, A.: Cluster validity measurement techniques. In: Proceedings of the 5th WSEAS International Conference on Artificial Intelligence, Knowledge Engineering and Data Bases, AIKED'06, pp. 388–393, Stevens Point, Wisconsin, USA (2006). World Scientific and Engineering Academy and Society (WSEAS). ISBN 111-2222-33-9
54. Jang, J., Brumley, D., Venkataraman, S.: BitShred: feature hashing malware for scalable triage and semantic analysis. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11, pp. 309–320, ACM, New York, NY, USA (2011). ISBN 978-1-4503-0948-6. doi:10.1145/2046707.2046742
55. LeDoux, C., Lakhotia, A., Miles, C., Notani, V., Pfeffer, A.: FuncTracker: discovering shared code to aid malware forensics extended abstract (2013)
56. Cohen, C., Havrilla, J.S.: Function hashing for malicious code analysis. In: CERT Research Annual Report 2009, pp. 26–29. Software Engineering Institute, Carnegie Mellon University (2009)
57. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
58. Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets. Cambridge University Press, Cambridge (2012). ISBN 9781139505345
59. Zhu, X.: Semi-supervised learning literature survey, Technical Report, Computer Sciences, University of Wisconsin-Madison (2005). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.99.9681&rep=rep1&type=pdf. Accessed 14 Mar 2013
60. Santos, I., Nieves, J., Bringas, P.: Semi-supervised learning for unknown malware detection. In: International Symposium on Distributed Computing and Artificial Intelligence, pp. 415–422 (2011)
61. Zhou, D., Bousquet, O., Lal, T.N., Weston, J., Schlkopf, B.: Learning with local and global consistency. Adv. Neural Inf. Process. Syst. **16**, 321–328 (2004)
62. Kuncheva, L.I.: Combining Pattern Classifiers: Methods and Algorithms. Wiley-Interscience, Hoboken (2004). ISBN 0471210781

63. Dahl, G., Stokes, J.W., Deng, L., Yu, D.: Large-scale malware classification using random projections and neural networks. In: Proceedings IEEE Conference on Acoustics, Speech, and Signal Processing, pp. 3422–3426 (2013)
64. Shahzad, R., Lavesson, N.: Veto-based malware detection. In: 2012 Seventh International Conference on Availability, Reliability and Security (ARES), pp. 47–54 (2012). doi:10.1109/ARES.2012.85
65. Shahzad, R.K., Lavesson, N.: Comparative analysis of voting schemes for ensemble-based malware detection. Wireless Mob. Netw. Ubiquitous Comput. Dependable Appl. **4**, 76–97 (2013)
66. Ye, Y., Li, T., Chen, Y., Jiang, Q.: Automatic malware categorization using cluster ensemble. In: Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 95–104 (2010)
67. Zhuang, W., Ye, Y., Chen, Y., Li, T.: Ensemble clustering for internet security applications. IEEE Trans. Syst. Man Cybern. Part C Appl. Rev. **42**(6), 1784–1796 (2012). ISSN 1094-6977. doi:10.1109/TSMCC.2012.2222025
68. Strehl, A., Ghosh, J.: Cluster ensembles a knowledge reuse framework for combining multiple partitions. J. Mach. Learn. Res. **3**, 583–617 (2003). ISSN 1532–4435. doi:10.1162/153244303321897735
69. Topchy, A., Jain, A.K., Punch, W.: Clustering ensembles: models of consensus and weak partitions. IEEE Trans. Pattern Anal. Mach. Intell. **27**(12), 1866–1881 (2005). ISSN 0162–8828. doi:10.1109/TPAMI.2005.237
70. Barr, S.J., Cardman, S.J., Martin, D.M.Jr.: A boosting ensemble for the recognition of code sharing in malware. J. Comput. Virol. **4**(4), 335–345 (2008). ISSN 1772–9890, 1772–9904, doi:10.1007/s11416-008-0087-z
71. Menahem, E., Shabtai, A., Rokach, L., Elovici, Y.: Improving malware detection by applying multi-inducer ensemble. Comput. Stat. Data Anal. **53**(4), 1483–1494 (2009). ISSN 0167–9473. doi:10.1016/j.csda.2008.10.015
72. Zabidi, M., Maarof, M., Zainal, A.: Ensemble based categorization and adaptive model for malware detection. In: 2011 7th International Conference on Information Assurance and Security (IAS), pp. 80–85 (2011). doi:10.1109/ISIAS.2011.6122799