UNIVERSITY OF SANNIO

Department Of Engineering
Master's Degree Course in
Computer Engineering

Master's Degree Thesis

Realization of an engine for GAN-driven malware manipulation

Thesis Advisor:
Prof. Corrado Aaron Visaggio

Research Supervisor:
Engr. Antonio Pirozzi

Author:
Laura Nardi
Num. 399000289

Academic Year 2019-2020

[This Page Intentionally Left Blank]

Contents

List of Figures **III**

List of Tables **IV**

Listing **V**

Introduction **1**

1 Overview of Files Structure and GAN **4**
1.1 PE File Format Specification 4
1.1.1 General Concept 4
1.1.2 File Headers 5
1.1.3 File Sections 10
1.2 Generative Adversarial Networks 15
1.2.1 What Are Generative Models 16
1.2.2 What Are Generative Adversarial Networks 17
1.2.3 Why Generative Adversarial Networks? 20

2 Engine Methodology and Architecture **22**
2.1 Generative Adversarial Attacks against Malware Detection Systems 22
2.2 Engine Architecture 23
2.3 Engine Design 24
2.3.1 Requirements 24
2.3.2 Manipulated Features 26
2.3.3 Constraints 35

3 Experimentation and Results **37**
3.1 Case Study Design 38
3.2 Data Set 38
3.3 Case Study Toolchain 41
3.4 Results 43

Conclusions **51**

Bibliography **53**

List of Figures

1.1 Typical Portable .EXE File Layout 6
1.2 File Layout for the Import Information 14
1.3 A comparison between supervised and unsupervised learning model 17
1.4 Generative Adversarial Network (GAN) Architecture 20

2.1 The architecture of the proposed engine 23

3.1 An instance example from EMBER data set 40
3.2 Toolchain for the experimentation 42
3.3 Interval Plot of number of strings, number of imports and number of sections 45
3.4 Interval Plot of TimeDateStamp field 46
3.5 Interval Plot of Size field 46
3.6 Interval plot of Byte Histogram [0-127] 48
3.7 Interval plot of Byte Histogram [128-255] 49
3.8 Interval plot of File Header Characteristics and Optional Header DLL Characteristics 50

List of Tables

1.1 General Concept 5
1.2 Export Section Structure 13

2.1 List of features manipulated by the engine 26

3.1 PEs Feature Groups in the EMBER data set 42

Listing

1.1 Data Directory Field 10
2.1 Access and modification of the COFF File Header 28
2.2 Access and modification of the Optional File Header 29
2.3 PE byte count 29
2.4 Section header structure 31
2.5 Alignment of Section values 31
2.6 Resize the executable 32
2.7 Add the new Section Header and validate values for the new Section Header 33
2.8 Set new Section Header characteristics and name 33
2.9 Create the new section and edit the Main Header 34
2.10 Inject code into the executable 34
2.11 Using PeNet to add imports to executable 35

Introduction

Machine learning has been used to detect new malware in recent years, while malware authors have strong motivation to attack such algorithms. Malware authors usually have no access to the detailed structures and parameters of the machine learning models used by malware detection systems, and therefore they can only perform black-box attacks [39].

Adversarial attacks are essentially a means of assessing network robustness. Broadly, there are two different types of attack: white-box and black-box [42, 43]. In white-box attacks an attacker has full access to the network model's parameters, whereas in black-box attacks the attacker has access only to network inputs and outputs, but not to any internal parameters. In [44] two white-box methods, iterative-Fast Gradient Sign Method (iFGSM) and DeepFool, are utilized, and it is proposed a low-cost method to explore marginal sample data near trained classifier decision boundaries, thus identifying potential adversarial samples. In [45] is proposed a gray-box adversarial attacks, that uses intermediate versions of the models (white-box and black-box)to seed the adversaries. Experimental evaluation demonstrates that the models trained using their method exhibit better robustness compared to both undefended and adversarially trained models.

This work proposes an engine for Generative Adversarial Network (GAN) driven malware manipulation and, thus, to generate adversarial malware examples.

Most researchers focused their efforts on improving the detection performance (e.g. true positive rate, accuracy and AUC) of such algorithms, but ignored the robustness of these algorithms. Generally speaking, the propagation of malware will benefit malware authors. Therefore, malware authors have sufficient motivation to attack malware detection algorithms.

Many machine learning algorithms are very vulnerable to intentional attacks. Machine learning based malware detection algorithms cannot be used in real-world applications if they are easily to be bypassed by some adversarial techniques. Recently, adversarial examples of deep learning models have attracted the attention of many researchers. Szegedy et al. added imperceptible perturbations to images to maximize a trained neural network's classification errors, making the network unable to classify the images correctly [35]. The examples after adding perturbations are called adversarial examples. Goodfellow et al. proposed a gradient based algorithm to generate adversarial examples [36]. Papernot et al. used the Jacobian matrix to determine which features to modify when generating adversarial examples [38]. The Jacobian matrix based approach is also a kind of gradient based algorithm. Grosse et al. proposed to use the gradient based approach to generate adversarial Android malware examples [37]. The adversarial examples are used to fool a neural network based malware detection model. They assumed that attackers have full access to the parameters of the malware detection model. For different sizes of neural networks, the misclassification rates after adversarial crafting range from 40% to 84%.

Machine learning based malware detection algorithms are usually integrated into antivirus software or hosted on the cloud side, and therefore they are black-box systems to malware authors. It is hard for malware authors to know which classifier a malware detection system uses and the parameters of the classifier. However, it is possible to figure out what features a malware detection algorithm uses by feeding some carefully designed test cases to the black-box algorithm. For example, if a malware detection algorithm uses static DLL or API features from the import directory table or the import lookup tables of PE programs, malware authors can manually modify some DLL or API names in the import directory table or the import lookup tables. They can modify a benign program's DLL or API names to malware's DLL or API names, and vice versa. If the detection results change after most of the modifications, they can judge that the malware detection algorithm uses DLL or API features.

Existing algorithms mainly use gradient information and hand-crafted rules to transform original samples into adversarial examples. This work proposes a generative based approach which takes original samples as inputs and outputs adversarial examples.

The thesis is divided into the following chapters:

- Overview of Files Structure and GAN. Explanation of the basic concepts of the PE file structure and specifications. In addition, an overview of what GANs are, therefore, the context of a GAN, what a GAN is and what it is used for is shown.

- Engine Methodology and Architecture. An overview of GAN-based attacks against Malware Detection Systems is first made in order to make clear the difference between the different contexts in which GANs have been presented. Subsequently, the design of the engine and where it is positioned within the architecture of the entire system is explained.

- Experimentation and Results. Explanation of the testing and validation process of the engine. Furthermore, the results obtained following the testing phase are shown.

Experimental results show that almost all of the adversarial examples generated by the Generator developed successfully bypass the detection algorithms and the Generator is very flexible to fool further defensive methods of detection algorithms.

Chapter 1

Overview of Files Structure and GAN

1.1 PE File Format Specification

The main focus of this first part of the chapter is to describe the structure of executable (image) files under the Windows family of operating systems. These files are referred to as Portable Executable (PE).

The PE file format describes the predominant executable format for Microsoft Windows operating systems, and includes executables, dynamically-linked libraries (DLLs), and FON font files. The format is currently supported on Intel, AMD and variants of ARM instruction set architectures [12].

1.1.1 General Concept

Certain concepts that appear throughout this section are described in the Table 1.1.

Name	Description
Attribute Certificate	A certificate that is used to associate verifiable statements with an image. A number of different verifiable statements can be associated with a file.
Date/Time stamp	A stamp that is used for different purposes in several places in a PE or COFF[1] file. In most cases, the format of each stamp is the same as that used by the time functions in the C run-time library.

[1]Common Object File Format [1]

File pointer	The location of an item within the file itself, before being processed by the loader. In other words, this is a position within the file as stored on disk.
Linker	A reference to the linker that is provided with Microsoft Visual Studio.
Object file	A file that is given as input to the linker. The linker produces an image file, which in turn is used as input by the loader. The term “object file” does not necessarily imply any connection to object-oriented programming.
Reserved, must be 0	A description of a field that indicates that the value of the field must be zero for generators and consumers must ignore the field.
Relative virtual address (RVA)	This is the address of an item after it is loaded into memory, with the base address of the image file subtracted from it. The RVA of an item almost always differs from its position within the file on disk (file pointer).
Section	The basic unit of code or data within a PE or COFF file.
Virtual Address (VA)	Same as RVA, except that the base address of the image file is not subtracted.

Table 1.1: General Concept

Figure 1.1 illustrate the Microsoft PE executable format.

1.1.2 File Headers

The PE file header consists of a Microsoft MS-DOS stub, the PE signature, the COFF file header, and an optional header. The file headers are followed immediately by section headers.

MS-DOS Stub

The MS-DOS stub is a valid application that runs under MS-DOS. It is placed at the front of the EXE image. The linker places a default stub here, which prints out the message “This program cannot be run in DOS mode” when the image is run in MS-DOS. The user can specify a different stub by using the `/STUB` linker option.

At location `0x3c`, the stub has the file offset to the PE signature. This information enables Windows to properly execute the image file, even though it has an MS-DOS stub. This file offset is placed at location `0x3c` during linking.

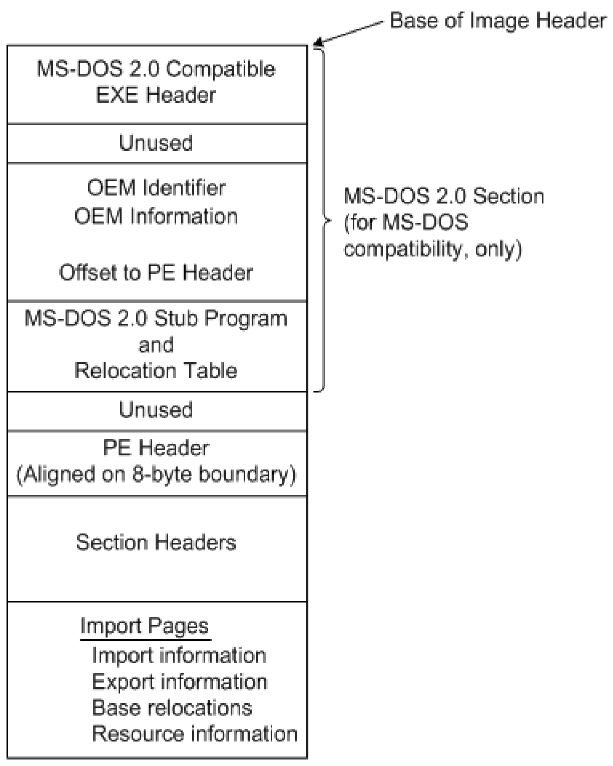


Figure 1.1: Typical Portable .EXE File Layout

The first field, `e_magic`, is the so-called magic number. This field is used to identify an MS-DOS-compatible file type. All MS-DOS-compatible executable files set this value to `0x5A4D`, which represents the ASCII characters MZ. MS-DOS headers are sometimes referred to as MZ headers for this reason.

Signature

After the MS-DOS stub, at the file offset specified at offset `0x3c`, is a 4-byte signature that identifies the file as a PE format image file. This signature is "`PE\0\0`" (the letters "P" and "E" followed by two null bytes).

COFF File Header

Immediately after the signature of an image file, is a standard COFF file header. Its format is described in the following. Note that the Windows loader limits the number of sections to 96.

- `Machine`. The number that identifies the type of target machine.
- `NumberOfSections`. The number of sections. This indicates the size of the section table, which immediately follows the headers.

- `TimeDateStamp`. The low 32 bits of the number of seconds since 00:00 January 1, 1970 (a C run-time time_value), which indicates when the file was created.

- `PointerToSymbolTable`. The file offset of the COFF symbol table, or zero if no COFF symbol table is present. This value should be zero for an image because COFF debugging information is deprecated.

- `NumberOfSymbols`. The number of entries in the symbol table. This data can be used to locate the string table, which immediately follows the symbol table. This value should be zero for an image because COFF debugging information is deprecated.

- `SizeOfOptionalHeader`. The size of the optional header, which is required for executable files.

- `Characteristics`. The flags that indicate the attributes of the file.

The information in the PE file is basically high-level information that is used by the system or applications to determine how to treat the file. The first field is used to indicate what type of machine the executable was built for, such as the DEC® Alpha, MIPS R4000, Intel® x86, or some other processor. The system uses this information to quickly determine how to treat the file before going any further into the rest of the file data.

The Characteristics field identifies specific characteristics about the file. For example, consider how separate debug files are managed for an executable. It is possible to strip debug information from a PE file and store it in a debug file (.DBG) for use by debuggers. To do this, a debugger needs to know whether to find the debug information in a separate file or not and whether the information has been stripped from the file or not. A debugger could find out by drilling down into the executable file looking for debug information. To save the debugger from having to search the file, a file characteristic that indicates that the file has been stripped (`IMAGE_FILE_DEBUG_STRIPPED`) was invented. Debuggers can look in the PE file header to quickly determine whether the debug information is present in the file or not.

Optional Header

Every image file has an optional header that provides information to the loader. This header is optional in the sense that some files (specifically, object files) do not have it. For image files, this header is required. An object file can have an optional header, but generally this header has no function in an object file except to increase its size.

Note that the size of the optional header is not fixed. The `SizeOfOptional Header` field in the COFF header must be used to validate that a probe into the file for a particular data directory does not go beyond `SizeOfOptionalHeader`.

The `NumberOfRvaAndSizes` field of the optional header should also be used to ensure that no probe for a particular data directory entry goes beyond the optional header. In addition, it is important to validate the optional header magic number for format compatibility.

The optional header magic number determines whether an image is a PE32 or PE32+ executable. PE32+ images allow for a 64-bit address space while limiting the image size to 2 gigabytes. Other PE32+ modifications are addressed in their respective sections.

The optional header itself has three major parts: Standard fields, Windows-specific fields and Data directories.

The first eight fields of the optional header are standard fields that are defined for every implementation of COFF. These fields contain general information that is useful for loading and running an executable file. They are unchanged for the PE32+ format. These fields are:

- `Magic`. The unsigned integer that identifies the state of the image file. The most common number is `0x10B`, which identifies it as a normal executable file. `0x107` identifies it as a ROM image, and `0x20B` identifies it as a PE32+ executable.

- `MajorLinkerVersion`, `MinorLinkerVersion`. Indicate major and minor version number of the linker that linked this image.

- `SizeOfCode`. The size of the code (text) section, or the sum of all code sections, if there are multiple sections.

- `SizeOfInitializedData`. The size of the initialized data section, or the sum of all such sections, if there are multiple data sections.

- `SizeOfUninitializedData`. The size of the uninitialized data section (BSS), or the sum of all such sections, if there are multiple BSS sections.

- `AddressOfEntryPoint`. The address of the entry point relative to the image base when the executable file is loaded into memory. For program images, this is the starting address. For device drivers, this is the address of the initialization function. An entry point is optional for DLLs. When no entry point is present, this field must be zero.

- `BaseOfCode`. The address that is relative to the image base of the beginning-of-code section when it is loaded into memory.

PE32 contains an additional field, which is absent in PE32+. `BaseOfData`, is the address that is relative to the image base of the beginning-of-data section when it is loaded into memory.

The additional fields added to the Windows NT PE file format provide loader support for much of the Windows NT-specific process behavior. Following is a summary of these fields:

- `ImageBase`. The preferred address of the first byte of image when loaded into memory; must be a multiple of 64 K. The default for DLLs is `0x10000000`. The default for Windows CE EXEs is `0x00010000`. The default for Windows NT, Windows 2000, Windows XP, Windows 95, Windows 98, and Windows Me is `0x00400000`.

- `SectionAlignment`. The alignment (in bytes) of sections when they are loaded into memory. It must be greater than or equal to `FileAlignment`. The default is the page size for the architecture.

- `FileAlignment`. The alignment factor (in bytes) that is used to align the raw data of sections in the image file. The value should be a power of 2 between 512 and 64 K, inclusive. The default is 512. If the `SectionAlignment` is less than the architecture's page size, then `FileAlignment` must match `SectionAlignment`.

- `MajorOperatingSystemVersion`, `MinorOperatingSystemVersion`. The major and minor version number of the required operating system.

- `MajorImageVersion`, `MinorImageVersion`. The major and minor version number of the image.

- `MajorSubsystemVersion`, `MinorSubsystemVersion`. The major and minor version number of the subsystem.

- `Win32VersionValue`. Reserved, must be zero.

- `SizeOfImage`. The size (in bytes) of the image, including all headers, as the image is loaded in memory. It must be a multiple of `SectionAlignment`.

- `SizeOfHeaders`. The combined size of an MS-DOS stub, PE header, and section headers rounded up to a multiple of `FileAlignment`.

- `CheckSum`. The image file checksum. The algorithm for computing the checksum is incorporated into IMAGHELP.DLL. The following are checked for validation at load time: all drivers, any DLL loaded at boot time, and any DLL that is loaded into a critical Windows process.

- `Subsystem`. The subsystem that is required to run this image.

- `DllCharacteristics`. Flags used to indicate if a DLL image includes entry points for process and thread initialization and termination.
- `SizeOfStackReserve`. The size of the stack to reserve. Only `SizeOfStackCommit` is committed; the rest is made available one page at a time until the reserve size is reached.
- `SizeOfStackCommit`. The size of the stack to commit.
- `SizeOfHeapReserve`. The size of the local heap space to reserve. Only `SizeOfHeapCommit` is committed; the rest is made available one page at a time until the reserve size is reached.
- `SizeOfHeapCommit`. The size of the local heap space to commit.
- `LoaderFlags`. Reserved, must be zero.
- `NumberOfRvaAndSizes`. The number of data-directory entries in the remainder of the optional header. Each describes a location and size.

Each data directory gives the address and size of a table or string that Windows uses. These data directory entries are all loaded into memory so that the system can use them at run time. A data directory is an 8-byte field that has the following declaration:

Listing 1.1: Data Directory Field

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

The first field, `VirtualAddress`, is actually the RVA of the table. The RVA is the address of the table relative to the base address of the image when the table is loaded. The second field gives the size in bytes. The data directories form the last part of the optional header.

Note that the number of directories is not fixed. Before looking for a specific directory, check the `NumberOfRvaAndSizes` field in the optional header.

Also, do not assume that the RVAs in this table point to the beginning of a section or that the sections that contain specific tables have specific names.

1.1.3 File Sections

The PE file specification consists of the headers defined so far and a generic object called a section. Sections contain the content of the file, including code, data, resources, and other executable information. Each section has a header and a body (the raw data). Section headers are described below, but section bodies lack a rigid file structure. They can be organized in almost any way a linker wishes to organize them, as long as the header is filled with enough information to be able to decipher the data.

Section Headers

Section headers are located sequentially right after the optional header in the PE file format. Each section header is 40 bytes with no padding between them.

Since section headers are organized sequentially in no specific order, section headers must be located by name. Below the section header fields:

- `Name`. An 8-byte, null-padded UTF-8 encoded string. If the string is exactly 8 characters long, there is no terminating null. For longer names, this field contains a slash (/) that is followed by an ASCII representation of a decimal number that is an offset into the string table. Executable images do not use a string table and do not support section names longer than 8 characters.

- `VirtualSize`. The total size of the section when loaded into memory. If this value is greater than `SizeOfRawData`, the section is zero-padded.

- `VirtualAddress`. The address of the first byte of the section relative to the image base when the section is loaded into memory.

- `SizeOfRawData`. The size of the initialized data on disk (for image files). For executable images, this must be a multiple of `FileAlignment` from the optional header. If this is less than `VirtualSize`, the remainder of the section is zero-filled. Because the `SizeOfRawData` field is rounded but the `VirtualSize` field is not, it is possible for `SizeOfRawData` to be greater than `VirtualSize` as well. When a section contains only uninitialized data, this field should be zero.

- `PointerToRawData`. The file pointer to the first page of the section within the COFF file. For executable images, this must be a multiple of `FileAlignment` from the optional header. For object files, the value should be aligned on a 4-byte boundary for best performance. When a section contains only uninitialized data, this field should be zero.

- `PointerToRelocations`. The file pointer to the beginning of relocation entries for the section. This is set to zero for executable images or if there are no relocations.

- `PointerToLinenumbers`. The file pointer to the beginning of line-number entries for the section. This is set to zero if there are no COFF line numbers. This value should be zero for an image because COFF debugging information is deprecated.
- `NumberOfRelocations`. The number of relocation entries for the section. This is set to zero for executable images.
- `NumberOfLinenumbers`. The number of line-number entries for the section. This value should be zero for an image because COFF debugging information is deprecated.
- `Characteristics`. The flags that describe the characteristics of the section.

Section Data

Initialized data for a section consists of simple blocks of bytes. However, for sections that contain all zeros, the section data need not be included.

The data for each section is located at the file offset that was given by the `PointerToRawData` field in the section header. The size of this data in the file is indicated by the `SizeOfRawData` field. If `SizeOfRawData` is less than `VirtualSize`, the remainder is padded with zeros.

In an image file, the section data must be aligned on a boundary as specified by the `FileAlignment` field in the optional header. Section data must appear in order of the RVA values for the corresponding sections (as do the individual section headers in the section table).

There are additional restrictions on image files if the `SectionAlignment` value in the optional header is less than the page size of the architecture. For such files, the location of section data in the file must match its location in memory when the image is loaded, so that the physical offset for section data is the same as the RVA.

Special Sections

The export data section, named `.edata`, contains information about symbols that other images can access through dynamic linking. Exported symbols are generally found in DLLs, but DLLs can also import symbols.

An overview of the general structure of the export section is described in Table 1.2. The tables described are usually contiguous in the file in the order shown (though this is not required). Only the export directory table and export address table are required to export symbols as ordinals. (An ordinal is an export that is accessed directly by its export address table index.) The

Table Name	**Description**
Export directory table	A table with just one row (unlike the debug directory). This table indicates the locations and sizes of the other export tables.
Export address table	An array of RVAs of exported symbols. These are the actual addresses of the exported functions and data within the executable code and data sections. Other image files can import a symbol by using an index to this table (an ordinal) or, optionally, by using the public name that corresponds to the ordinal if a public name is defined.
Name pointer table	An array of pointers to the public export names, sorted in ascending order.
Ordinal table	An array of the ordinals that correspond to members of the name pointer table. The correspondence is by position; therefore, the name pointer table and the ordinal table must have the same number of members. Each ordinal is an index into the export address table.
Export name table	A series of null-terminated ASCII strings. Members of the name pointer table point into this area. These names are the public names through which the symbols are imported and exported; they are not necessarily the same as the private names that are used within the image file.

Table 1.2: Export Section Structure

name pointer table, ordinal table, and export name table all exist to support use of export names.

When another image file imports a symbol by name, the Win32 loader searches the name pointer table for a matching string. If a matching string is found, the associated ordinal is identified by looking up the corresponding member in the ordinal table (that is, the member of the ordinal table with the same index as the string pointer found in the name pointer table). The resulting ordinal is an index into the export address table, which gives the actual location of the desired symbol. Every export symbol can be accessed by an ordinal.

When another image file imports a symbol by ordinal, it is unnecessary to search the name pointer table for a matching string. Direct use of an ordinal is therefore more efficient. However, an export name is easier to remember and

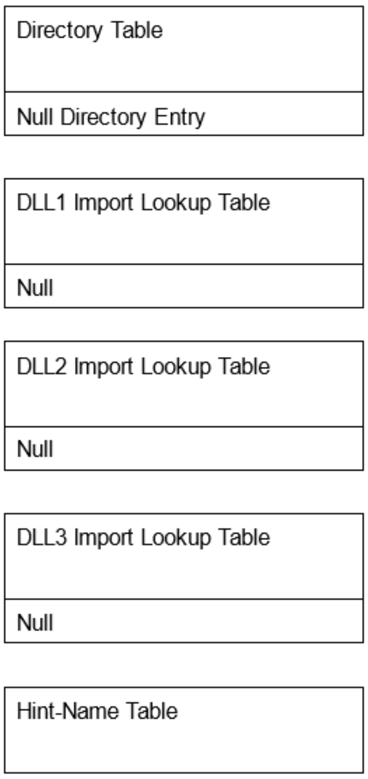


Figure 1.2: File Layout for the Import Information

does not require the user to know the table index for the symbol.

All image files that import symbols, including virtually all executable (EXE) files, have an `.idata` section. A typical file layout for the import information is shown in Figure 1.2.

The import information begins with the import directory table, which describes the remainder of the import information. The import directory table contains address information that is used to resolve fixup references to the entry points within a DLL image. The import directory table consists of an array of import directory entries, one entry for each DLL to which the image refers. The last directory entry is empty (filled with null values), which indicates the end of the directory table. An import lookup table is an array of 32-bit numbers for PE32 or an array of 64-bit numbers for PE32+. Each entry uses the bit-field format that is described in the following table. The collection of these entries describes all imports from a given DLL. The last entry is set to zero (NULL) to indicate the end of the table. Moreover, there is one hint/name table suffices for the entire import section. It represents an index into the export name pointer table and an ASCII string that contains the name to import. This is the string that must be matched to the public name in the DLL. The structure and content of the import address table are identical to those of the import lookup table, until the file is bound. During binding, the entries in the import address table are overwritten with the 32-bit (for PE32) or 64-bit (for PE32+) addresses of the symbols that are being imported. These addresses are the actual memory addresses of the symbols, although technically they are still called "virtual addresses". The loader typically processes the binding.

A resources section may contain resources such as required for user interfaces: cursors, fonts, bitmaps, icons, menus, etc. A basic PE file would normally contain a `.text` code section and one or more data sections (`.data`, `.rdata` or `.bss`). Relocation tables are typically stored in a `.reloc` section, used by the Windows loader to reassign a base address from the executable's preferred base. A `.tls` section contains special thread local storage (TLS) structure for storing thread-specific local variables, which has been exploited to redirect the entry point of an executable to first check if a debugger or other analysis tool are being run [3]. Section names are arbitrary from the perspective of the Windows loader, but specific names have been adopted by precedent and are overwhelmingly common. Packers may create new sections, for example, the UPX packer creates UPX1 to house packed data and an empty section UPX0 that reserves an address range for runtime unpacking [4].

This first part of the chapter aims to give a smattering of the PE file format and some of the concepts that will be taken up in subsequent chapters. Not everything concerning the format has been described in detail, but only those useful for understanding the work have been mentioned. To learn more, you can use [1], which also describes the COFF file format.

1.2 Generative Adversarial Networks

Generative Adversarial Networks, or GANs for short, are an approach to generative modeling using deep learning methods, such as convolutional neural networks [10].

Generative modeling is an unsupervised learning task in machine learning that involves automatically discovering and learning the regularities or patterns in input data in such a way that the model can be used to generate or output new examples that plausibly could have been drawn from the original dataset.

GANs are a clever way of training a generative model by framing the problem as a supervised learning problem with two sub-models: the *generator model* that we train to generate new examples, and the *discriminator model* that tries to classify examples as either real (from the domain) or fake (generated) [23]. The two models are trained together in a zero-sum game, adversarial, until the discriminator model is fooled about half the time, meaning the generator model is generating plausible examples [10].

The promise of deep learning is to discover rich, hierarchical models [6] that represent probability distributions over the kinds of data encountered in artificial intelligence applications, such as natural images, audio waveforms containing speech, and symbols in natural language corpora. So far, the most striking successes in deep learning have involved discriminative models, usually those that map a high-dimensional, rich sensory input to a class label [7, 8]. These striking successes have primarily been based on the backpropagation and dropout algorithms, using piecewise linear units [11, 9, 10] which have a particularly well-behaved gradient. Deep generative models have had less of an impact, due to the difficulty of approximating many intractable probabilistic computations that arise in maximum likelihood estimation and related strategies, and due to difficulty of leveraging the benefits of piecewise linear units in the generative context [5].

The main objective of this section is to clarify several aspects, including:

- Context for GANs, including supervised vs. unsupervised learning and discriminative vs. generative modeling;
- What is a GAN;
- What is GAN for, so how it provide a path to sophisticated domain-specific data augmentation and a solution to problems that require a generative solution, such as image-to-image translation.

1.2.1 What Are Generative Models

In this section, we will review the idea of generative models, stepping over the *supervised* vs. *unsupervised* learning paradigms and *discriminative* vs. *generative* modeling.

Supervised vs. Unsupervised Learning

A typical machine learning problem involves using a model to make a prediction, e.g. predictive modeling. This requires a training dataset that is used to train a model, comprised of multiple examples, called samples, each with input variables and output class labels.

A model is trained by showing examples of inputs, having it predict outputs, and correcting the model to make the outputs more like the expected outputs. This correction of the model is generally referred to as a supervised form of learning, or supervised learning. Examples of supervised learning problems include classification and regression, and examples of supervised learning algorithms include logistic regression and random forest.

There is another paradigm of learning where the model is only given the input variables and the problem does not have any output variables. A model is constructed by extracting or summarizing the patterns in the input data. There is no correction of the model, as the model is not predicting anything.

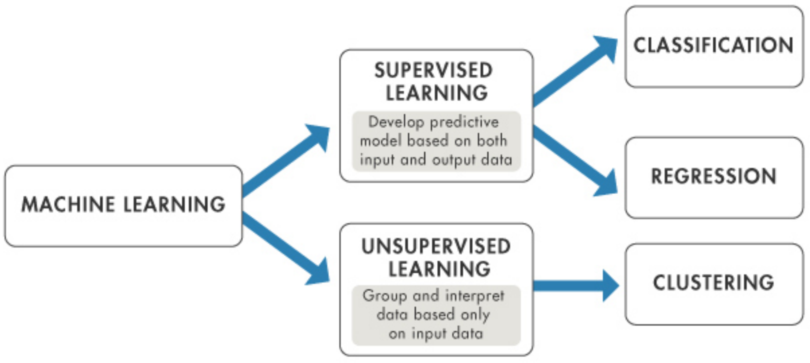

MACHINE LEARNING
SUPERVISED LEARNING
Develop predictive model based on both input and output data
UNSUPERVISED LEARNING
Group and interpret data based only on input data
CLASSIFICATION
REGRESSION
CLUSTERING

Figure 1.3: A comparison between supervised and unsupervised learning model

This lack of correction is generally referred to as an unsupervised form of learning, or unsupervised learning. Examples of unsupervised learning problems include clustering and generative modeling, and examples of unsupervised learning algorithms are K-means and Generative Adversarial Networks [24].

In figure 1.3 is shown a comparison between supervised and unsupervised learning model.

Discriminative vs. Generative Modeling

In supervised learning, we may be interested in developing a model to predict a class label given an example of input variables. This predictive modeling task is called *classification*. Classification is also traditionally referred to as discriminative modeling. This is because a model must discriminate examples of input variables across classes; it must choose or make a decision as to what class a given example belongs.

Alternately, unsupervised models that summarize the distribution of input variables may be able to be used to create or generate new examples in the input distribution. As such, these types of models are referred to as generative models. For example, a single variable may have a known data distribution, such as a Gaussian distribution, or bell shape. A generative model may be able to sufficiently summarize this data distribution, and then be used to generate new variables that plausibly fit into the distribution of the input variable. In fact, a really good generative model may be able to generate new examples that are not just plausible, but indistinguishable from real examples from the problem domain.

1.2.2 What Are Generative Adversarial Networks

Generative Adversarial Networks, or GANs, are a deep-learning-based generative model. GANs are an architecture for automatically training a generative model by treating the unsupervised problem as supervised and using both a generative and a discriminative model.

The GAN architecture was first described in [10]. A standardized approach that led to more stable models was later formalized by Alec Radford, et al. in [26].

Several studies have investigated the effectiveness [13, 14, 15, 16, 17, 18] and drawbacks [19, 20] of machine (and recently also deep) learning in detecting and classifying malware. Independently from the inherent limitations of malware detectors based on machine learning, the generative adversarial networks, GANs in short, become a menace to the effectiveness of these tools [21].

A GAN is a tool that produces adversarial samples by using the adversarial machine learning [22]: this is a technique that leverages machine learning for fooling classifiers trained with a machine learning algorithm, leading them to wrongly classify some samples.

Alike the fields where GANs have been experimented, they could be successfully used for generating samples of malware that are recognized by malware detectors based on machine learning as goodware. The research community is now investigating the application of GANs to malware analysis, and so far the main result consists of some models of GANs for producing adversarial malware samples.

The GAN model architecture involves two sub-models: a **generator model** for generating new examples and a **discriminator model** for classifying whether generated examples are real, from the domain, or fake, generated by the generator model. Thus, generator is a model that is used to generate new plausible examples from the problem domain and, discriminator is a model that is used to classify examples as real (from the domain) or fake (generated).

Generative adversarial networks are based on a game theoretic scenario in which the generator network must compete against an adversary. The generator network directly produces samples. Its adversary, the discriminator network, attempts to distinguish between samples drawn from the training data and samples drawn from the generator [27].

The Generator Model

The generator model takes a fixed-length random vector as input and generates a sample in the domain. The vector is drawn from randomly from a Gaussian distribution, and the vector is used to seed the generative process. After training, points in this multidimensional vector space will correspond to points in the problem domain, forming a compressed representation of the data distribution.

This vector space is referred to as a latent space, or a vector space comprised of latent variables. Latent variables, or hidden variables, are those variables that are important for a domain but are not directly observable.

We often refer to latent variables, or a latent space, as a projection or compression of a data distribution. That is, a latent space provides a compression or high-level concepts of the observed raw data such as the input data distribution. In the case of GANs, the generator model applies meaning to points in a chosen latent space, such that new points drawn from the latent space can be provided to the generator model as input and used to generate new and different output examples.

After training, the generator model is kept and used to generate new samples [27].

The Discriminator Model

The discriminator model takes an example from the domain as input (real or generated) and predicts a binary class label of real or fake (generated). The real example comes from the training dataset. The generated examples are output by the generator model.

The discriminator is a normal (and well understood) classification model. After the training process, the discriminator model is discarded as we are interested in the generator.

Sometimes, the generator can be repurposed as it has learned to effectively extract features from examples in the problem domain. Some or all of the feature extraction layers can be used in transfer learning applications using the same or similar input data [26].

GANs as a Two Player Competition

Generative modeling is an unsupervised learning problem, as we discussed previously, although a clever property of the GAN architecture is that the training of the generative model is framed as a supervised learning problem.

The two models, the generator and discriminator, are trained together. The generator generates a batch of samples, and these, along with real examples from the domain, are provided to the discriminator and classified as real or fake. The discriminator is then updated to get better at discriminating real and fake samples in the next round, and importantly, the generator is updated based on how well, or not, the generated samples fooled the discriminator.

In this way, the two models are competing against each other, they are adversarial in the game theory sense, and are playing a *zero-sum game* [28].

In this case, zero-sum means that when the discriminator successfully identifies real and fake samples, it is rewarded or no change is needed to the model parameters, whereas the generator is penalized with large updates to model parameters.

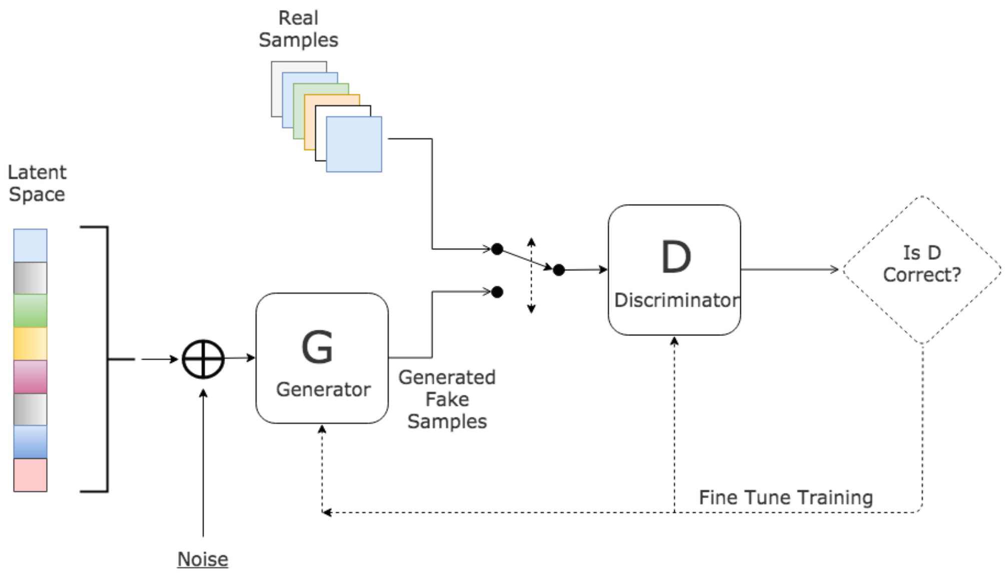


Figure 1.4: Generative Adversarial Network (GAN) Architecture

Alternately, when the generator fools the discriminator, it is rewarded, or no change is needed to the model parameters, but the discriminator is penalized and its model parameters are updated. In figure 1.4 is shown a GAN architecture to understand more easily how it works.

At a limit, the generator generates perfect replicas from the input domain every time, and the discriminator cannot tell the difference and predicts "unsure" (e.g. 50% for real and fake) in every case. This is just an example of an idealized case; we do not need to get to this point to arrive at a useful generator model.

1.2.3 Why Generative Adversarial Networks?

In complex domains or domains with a limited amount of data, generative modeling provides a path towards more training for modeling. GANs have seen much success in this use case in domains such as deep reinforcement learning.

There are many research reasons why GANs are interesting, important, and require further study. Ian Goodfellow outlines a number of these in [28].

Among these reasons, he highlights GANs' successful ability to model high-dimensional data, handle missing data, and the capacity of GANs to provide multi-modal outputs or multiple plausible answers.

Perhaps the most compelling application of GANs is in conditional GANs for tasks that require the generation of new examples. Here, Goodfellow indicates three main examples:

- Image Super-Resolution. The ability to generate high-resolution versions of input images.
- Creating Art. The ability to great new and artistic images, sketches, painting, and more.
- Image-to-Image Translation. The ability to translate photographs across domains, such as day to night, summer to winter, and more.

Perhaps the most compelling reason that GANs are widely studied, developed, and used is because of their success. GANs have been able to generate photos so realistic that humans are unable to tell that they are of objects, scenes, and people that do not exist in real life.

An exemplar case is the automatic recognition of street signs: a street sign is decoded as another street sign. Alike the fields where GANs have been experimented, they could be successfully used for generating samples of malware that are recognized by malware detectors based on machine learning as goodware [21].

As mentioned by [21] machine and deep learning promise to provide valid countermeasures against modern malware because of their capability to potentially detect malware applications without specific signatures of their behavior or data. Generally, ML-based malware detectors work on the extraction of the malware (and benign programs) features and static and/or dynamic analysis can be performed. Such systems learn from examples for creating models by which they will be able to discriminate whether a given program is a malware or not. These models are then used to estimate the likelihood that a given program is malware. One of the bottlenecks exhibited by ML-based malware detection is represented by the high time required to learn when the number or size of features is wide or the number of sample programs is large. Although reducing the number of features could shorten the learning time, the accuracy in the detection task likely decreases. So, finding an acceptable trade-off among the detection accuracy, short learning times and limiting the size of data, obtainable by selecting a convenient combination of sensitive features, is far from being a trivial problem.

A very accurate review of recent findings of adversarial examples in deep neural networks and a deep investigation of existing methods for generating adversarial examples is provided in [29].

Chapter 2

Engine Methodology and Architecture

2.1 Generative Adversarial Attacks against Malware Detection Systems

Generative Adversarial Algorithms have been mainly applied to image recognition, where Generative Adversarial Networks (GANs) were used to generate images that were indistinguishable from real ones. In the process of image generation, for example, the GAN network modifies some features like pixels, while a human eye does not perceive the difference from an original one. Using GAN to create a binary file poses more difficulties than an image, because changing a bit in a binary may corrupt the file. For this reason, generating executable files with a GAN could be challenging [21].

The main difference between image and malware is that images are continuous while malware features are binary. Changing byte arbitrarily could break semantics and syntax of portable executable (PE) so we are limited in the types of modification that can be done without breaking the malware functionality. For this reason, different approaches have been proposed in the literature such as adding padding bytes (adversarial noise) at the end of a file beyond PE boundaries [30]. Another approach consists of injecting the adversarial noise in an unused PE region, that is not mapped in memory [31]. Most works in literature simply ignore this problem. In order to overcome this limitation, attackers must have a white box model in which the type of the ML algorithm used and the features to be used are known.

One of the first demonstrations of an adversarial creation of a PE is the work [39]; in this paper, authors adopt a grey-box model in which they only know the set of used features based on API calls but do not know the ML model used by the classifier. In MalGAN, authors generate adversarial examples by

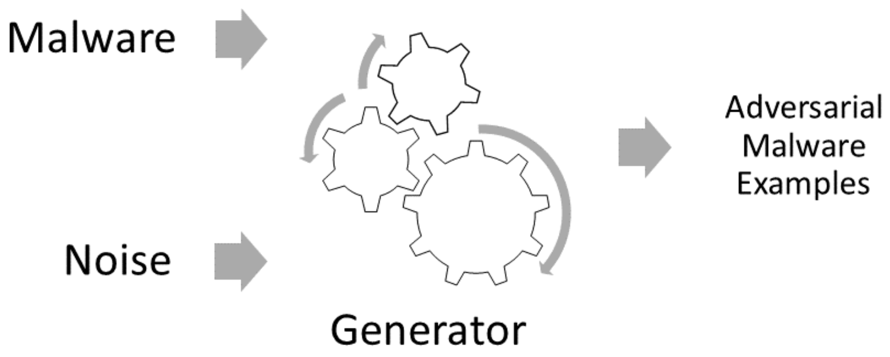


Figure 2.1: The architecture of the proposed engine

adding some irrelevant features to the binary files because removing features may crack the executable or its intended behaviour. While MalGAN and the detector use the same API as features quantity and this could affect the performance of avoidance, in [46] authors add some noise to malware, extracting features (API list) from clean malware and input them to the generator.

GANs are also used to generate a malicious document. In [32], the authors propose a method based on Wasserstein Generative Adversarial Network (WGAN) to generate a malicious PDF with an evasion rate of 100% as stated. In this work, the authors generate adversarial examples by modifying 68 features extracted from various attributes: size, metadata, and structural attributes.

With these premises, among the different approaches already addressed in the literature, the approach taken into consideration is similar to the one discussed in [39] and in [32], where a vector of features is used to manipulate a PE.

2.2 Engine Architecture

The architecture of proposed Generator is shown in Figure 2.1.

For the generator model, it has been written a command line tool in order to manipulate executables. For the generation of adversarial examples, we set two main constraints:

- **functionality preserving**: adding noise for generating adversarial examples should not break the sample's behavior;
- **features probability distributions invariance**: when we worked only on surface features, we don't manipulate the content of binaries but we try to change the surface information as their probability distribution looks like more close to the distribution of good samples. When we

worked not only on surface features, but even on the content of binaries, this property cannot be achieved. So, in order not to stray far from the original instance of the executable, the engine should bring a minimum of perturbation.

In order to modify the executable files to fool the discriminator within the GAN, it is necessary to have a *feature noise vector* that tells the engine which features to manipulate. Noise vectors adopted for manipulating genuine inputs have the same dimensions of the input, according to the number of features that are considered in each couple of Generator-Discriminator.

The generator is used to transform a malware feature vector into its adversarial version. It takes a malware feature vector `m` and a noise vector `z` as input. Each element of `m` corresponds to the actual feature in the executable to manipulate. In the case of binary/discrete features each element of `z` corresponds to the presence or absence of a feature. In the case of continuous features each element of `z` is a a random number. This random number will be generated by the GAN itself into which the engine will integrate, based on a classification model on which the GAN has already been trained. The effect of z is to allow the generator to generate diverse adversarial examples from a single malware feature vector.

When generating adversarial examples for binary malware features we only consider to add some irrelevant features to malware. Removing a feature from the original malware may crack it. For example, if the "WriteFile" API is removed from a program, the program is unable to perform normal writing function and the malware may crack. The non-zero elements of the noise vector act as the irrelevant features to be added to the original malware.

2.3 Engine Design

As previously mentioned, the goal of this work is to implement an engine which, within the GAN, represents what is commonly called *Generator*. This engine does nothing but modify a series of features of the executables that are present in an input noise vector.

2.3.1 Requirements

The engine is a command line tool that is written in Python and C#. The choice of these programming languages is due to their ease of use and the wide spectrum of constructs present. Furthermore, in these languages are available the libraries that were used for writing the engine and, thus, to manipulate the executables.

The library mainly used by the tool, in order to manipulate executable files, is *pefile*[1]. *pefile* is a multi-platform Python module to parse and work with Portable Executable (PE) files. Most of the information contained in the PE file headers is accessible, as well as all the sections' details and data. The structures defined in the Windows header files will be accessible as attributes in the PE instance. The naming of fields/attributes will try to adhere to the naming scheme in those headers. Only shortcuts added for convenience will depart from that convention. *pefile* requires some basic understanding of the layout of a PE file — with it, it's possible to explore nearly every single feature of the PE file format. Some of the tasks that pefile makes possible are:

- Inspecting headers
- Analyzing of sections' data
- Retrieving embedded data
- Reading strings from the resources
- Warnings for suspicious and malformed values
- Basic butchering of PEs, like writing to some fields and other parts of the PE
  - This functionality won't rearrange PE file structures to make room for new fields, so use it with care.
  - Overwriting fields should mostly be safe.
- Packer detection with PEiD's signatures
- PEiD signature generation

pefile runs in several pipelines scanning hundreds of thousands of new PE files every day, and, while not perfect, it has grown to be pretty robust over time. That being said, small glitches are found now and then.

Another library used to support *pefile*, again in order to manipulate executable files, is *PeNet*[2]. *PeNet* is a parser for Windows Portable Executable headers. It completely written in C# and does not rely on any native Windows APIs. Furthermore it supports the creation of Import Hashes (ImpHash), which is a feature often used in malware analysis. You can extract Certificate Revocation List, compute different hash sums and other useful stuff for working with PE files.

The main reason why this library was used concerns the fact that *pefile* does not allow adding an import inside the executable file. The choice fell on *PeNet* for its ease of use.

[1]`https://github.com/erocarrera/pefile` (visited on 12/14/2020)
[2]`https://github.com/secana/PeNet` (visited on 12/14/2020)

Feature	**Description**
Number of Strings	Number of strings present within the executable file. Such strings are counted through the following Regular Expression `[\x20-\x7f]{5,}`.
Size	Size of the executable file.
Number of Imports	Number of functions imported into the executable file.
TimeDateStamp	It refers to `TimeDateStamp` field in COFF File Header.
Machine	It refers to `Machine` field in COFF File Header.
Subsystem	It refers to `Subsystem` field in Optional File Header.
Magic	It refers to `Magic` field in Optional File Header.
Number of Sections	Number of Sections present within the executable file.
Byte Histogram	Byte histogram contains 256 integer values, representing the count of each byte value within the file. The byte histogram is normalized to a distribution, since the file size is represented as a feature in the general file information.
Characteristics	It refers to `Characteristics` field in COFF File Header.
DLL Characteristics	It refers to `DllCharacteristics` field in Optional File Header.

Table 2.1: List of features manipulated by the engine

2.3.2 Manipulated Features

The features manipulated by the engine are shown in Table 2.1. These features have been chosen on the basis of those that we have currently managed to manipulate with the present libraries.

Number of Strings

To get the number of strings, as previously mentioned in Table 2.1, it has been used a regular expression. This regular expression, `[\x20-\x7f]{5,}`, considers all strings with a number of characters greater than five and, at the same time, these characters must belong to the range `0x20 - 0x7f` of the ASCII table.

So, to understand how many strings are contained in the initial executable file, this is processed with the regex, after which, to respect the condition within the input noise vector, being a continuous variable, the two numbers are compared in order to understand if it is necessary to add or not the strings in the adversarial example in output.

In order to add strings to the executable you first need to add a section to the executable. After that you can inject the strings as if it were code. Obviously these strings must first be encoded in hexadecimal so that the *pefile* library does not generate errors.

Size

To resize an executable, as well as when you need to add a section, you need to go and act on its size. So in a similar way to what we saw in Listing 2.6, we open the file with the "a+" option, or *append*, in order to add more data to the existing file. Additionally, the "b" option is used to open the file in binary mode.

As you can see from the code, you can only do a resize to increase the size of the executable and not to decrease it. This is because otherwise the executable gets corrupted. The resize size in Listing 2.6 is just an example to render the concept, but this can be as large as you like. Obviously, to bring minimal disturbances to the executable and not to arouse suspicion in using, the size must be as close to the original one.

Also in this case, as well as in adding strings, the desired size in the input noise vector is taken as input and compared with that of the original executable to try to understand if this resizing is possible.

TimeDateStamp, Machine and Characteristics

The `TimeDateStamp` field indicates the low 32 bits of the number of seconds since 00:00 January 1, 1970 (a C run-time time_t value), which indicates when the file was created.

The `Machine` field has different values, which specify the CPU type. An image file can only be run on the specified machine or on a system that emulates the specified machine.

The `Characteristics` field contains flags that indicate attributes of the object or image file.

All these fields are contained in the COFF File Header and can be modified accessing to it. Listing 2.1 shows how to access these fields and how to modify them. As you can see, the `TimeDateStamp` field accepts an integer representing a timestamp in milliseconds. The `Machine` field accepts an integer that represents the type of machine on which the executable will be run. This field can also be expressed in hexadecimal. In this case the target machine indicated by the integer 467 represents the constant `IMAGE_FILE_MACHINE_AM33`, which is a Matsushita AM33 machine. Finally, the `Characteristics` field accepts a hexadecimal or an integer. Since this field is expressed in the form

Listing 2.1: Access and modification of the COFF File Header

```
pe = pefile.PE(exe_path)
pe.FILE_HEADER.TimeDateStamp = 1365446976
pe.FILE_HEADER.Characteristics = 0x0020 | 0
    x0001 | 0x0080
pe.FILE_HEADER.Machine = 467
pe.write(exe_path)
```

of a list, the hexadecimal is obtained in OR with the different characteristics. So, in this case, the executable has the characteristics `0x0020 | 0x0001 | 0x0080`, which respectively represent `IMAGE_FILE_LARGE_ADDRESS_AWARE`, `IMAGE_FILE_RELOCS_STRIPPED`, `IMAGE_FILE_BYTES_REVERSED_LO`.

Subsystem, Magic and DllCharacteristics

The `Subsystem` field represents the Windows subsystem that is required to run this image.

The `Magic` field represents the unsigned integer that identifies the state of the image file. The most common number is `0x10B`, which identifies it as a normal executable file. `0x107` identifies it as a ROM image, and `0x20B` identifies it as a PE32+ executable.

The `DllCharacteristics` field contains, in a similar way to `Characteristics` field, flags that indicate attributes of the object or image file.

All these fields are contained in the Optional File Header and can be modified accessing to it. Listing 2.2 shows how to access these fields and how to modify them. As you can see, the `Subsystem` field accepts a hexadecimal or an integer representing it. In this case, 16 stands for `IMAGE_SUBSYSTEM_WINDOWS_BOOT_APPLICATION`, that is a Windows Boot Application. The `Magic` field accepts an integer, but this field can also be expressed in hexadecimal. In this case the magic indicated by the integer 267 represents the constant `PE32`, which is a PE32 executable. Finally, the `DllCharacteristics` field accepts a hexadecimal or an integer. Since this field is expressed in the form of a list, the hexadecimal is obtained in OR with the different characteristics. So, in this case, the executable has the characteristics `512 | 1024 | 2048`, which respectively represent `IMAGE_DLLCHARACTERISTICS_NO_ISOLATION`, `IMAGE_DLLCHARACTERISTICS_NO_SEH`, `IMAGE_DLLCHARACTERISTICS_NO_BIND`.

Listing 2.2: Access and modification of the Optional File Header

```
pe = pefile.PE(exe_path)
pe.OPTIONAL_HEADER.Subsystem = 16
pe.OPTIONAL_HEADER.DllCharacteristics = 512 |
    1024 | 2048
pe.OPTIONAL_HEADER.Magic = 267
pe.write(exe_path)
```

Listing 2.3: PE byte count

```
counts = np.bincount(np.frombuffer(open(
    exe_path, "rb").read(), dtype=np.uint8),
    minlength=256)
```

Byte Histogram

As previously mentioned, byte histogram contains 256 integer values, representing the count of each byte value within the file.

This feature, although it may seem trivial, is very important as it considers the frequency of the opcodes within the executable. In fact, several studies [33, 34, 12], consider this feature. In [33], the authors implemented and evaluated a classification method based on the opcodes distribution, using histogram. In the presented method, a histogram of opcodes extracted from portable executables is generated to show the average distribution of opcodes in files belonged to the same family. Later, they used this histogram to be compared with any executable file histogram to classify it whether is belonged to the same family or not. Their experiment results prove that this method can be used as a reliable technique for classifying the metamorphic PE-malware. Furthermore, it has shown that the simple proposed methodology not only is able to recognize the malware variants, but also it can differentiate benign executable programs. The most significant advantages of this method are the simplicity of implementation and the high rate of accuracy.

To manipulate this feature, as for the number of strings, to respect the condition within the input noise vector, the two numbers (occurrence of the i-th byte in the malware feature vector and occurrence of the i-th in the noise vector) are compared in order to understand if it is necessary to add or not one or more bytes in the adversarial example.

The histogram is counted with the code in Listing 2.3, where `np` represents the *numpy* library[3].

[3]`https://numpy.org/` (visited on 14/12/2020)

Number of Sections

Adding a section to the executable will slightly increase the size of the executable. Before adding a new section, we need to know the structure details to not break our executable. In a PE executable, the section is composed of 2 parts:

- The **section**, containing the executable code.
- The **section header**, containing the description of the section (address, code, size, and so on).

The section header length is 40 bytes and follow the structure in Listing 2.4. Each field help Windows to load the sections properly into the memory. Here, we are only interested by the following fields, the others will be initialized at zero.

- `Name`, contains the section name with a padding of null bytes if the size of the name is not equal to 8 bytes.
- `VirtualSize`, contains the size of the section in memory.
- `VirtualAddress`, contains the relative virtual address of the section.
- `SizeOfRawData`, contains the size of the section on the disk.
- `PointerToRawData`, contains the offset of the section on the disk.
- `Characteristics`, contains the flags describing the section characteristics (RWX).

Note that it's really important to differentiate VA (Virtual Address) and RVA (Relative Virtual Address). A relative virtual address is the virtual address of an object from the file once it is loaded into memory, minus the base address (often equal to 0x00400000) of the file image.

Finally, we have to take care of the alignment. The value we will set into the section header should be aligned to the value set into the OPTIONAL_HEADER of the PE file.

- `SectionAlignment`, section alignment in memory.
- `FileAlignment`, section alignment on the disk.

For example, let's say `FileAlignment` equals 512 bytes and `SectionAlignment` equals 4096 bytes. If you new section contains 515 bytes on the disk, the section size value on the disk (`SizeOfRawData`) will be 1024 because $515 > 512$, so we round it up. Same thing for the `VirtualSize`, it will be equal to 4096 bytes, because $515 < 4096$. So a very simple function that describe this alignment is in Listing 2.5.

Listing 2.4: Section header structure

```
class SECTION_HEADER(Structure):
    _fields_ = [
        ("Name",                    BYTE * 8),
        ("VirtualSize",             DWORD),
        ("VirtualAddress",          DWORD),
        ("SizeOfRawData",           DWORD),
        ("PointerToRawData",        DWORD),
        ("PointerToRelocations",    DWORD),
        ("PointerToLinenumbers",    DWORD),
        ("NumberOfRelocations",     WORD),
        ("NumberOfLinenumbers",     WORD),
        ("Characteristics",         DWORD)
    ]
```

Listing 2.5: Alignment of Section values

```
def align(val_to_align, alignment):
    return ((val_to_align + alignment - 1) /
        alignment) * alignment
```

For `VirtualAddress` and `PointerToRawData`, we have to be sure that we don't overwrite the other sections. If one of the pointer point to an existing header, we would corrupt the executable. To avoid this issue, we will set our pointers to go after the last section of the executable. Once done this, the new section will be located right after the last section on the disk and in memory. To comply with the alignment, we will modify the code to dynamically compute the right values for the section size.

Now we can write the new header properly, but we have to take care of two things:

- We have to be careful and not break the current headers, it means that our value have to comply with the header format.
- We have to write them in little-endian (*pefile* will take care of that).

Our new section header have been added to the executable, but the loader can't see it yet. We need to modify some value into the main structure header of the file first:

- `NumberOfSections`, in the FILE_HEADER must be increased by 1.
- `SizeOfImage`, in the OPTIONAL_HEADER, must be equal to the (VirtualAddress + VirtualSize (size of our new header)).

- Enlarge the size of the executable.

In this way, we told to the executable that there is a new section of some bytes somewhere, so we have to add some empty space to comply with the header information but also to add some code. We only created the section header not the section itself.

Shown below are several snippets (Listings 2.6, 2.7, 2.8, 2.9, 2.10) to: resize the executable, add the new section header, align the values of the new section header, set characteristics and name, create the section, edit the main header and inject code into the executable.

Listing 2.6: Resize the executable

```
pe = pefile.PE(exe_path)
fd = open(exe_path, "a+b")
map = mmap.mmap(fd.fileno(), 0, access=mmap.
    ACCESS_WRITE)
map.resize(original_size + 0x2000)
map.close()
fd.close()
pe.write(exe_path)
```

Number of Imports

One of the cool new features in .NET Core 3.0 is that you can easily build a single EXE file that contains the entire application. The merged EXE will contain all .NET Core files, and this means that you do not need to install anything on the client machine. You do not have to know if a specific framework is installed as well.

Precisely for this reason, having to use the *PeNet* library, it was easy to integrate the functions offered by this library into the engine. In fact, a simple console application has been written, which once compiled and produced the executable, is invoked from the terminal with some parameters that allow you to understand what is the executable to manipulate, what is the DLL and what is the function to add to the IAT.

In Listing 2.11 is shown the C# code that exploits the *PeNet* library. As you can see, the application accepts command line parameters. In particular, the first argument, `args[0]`, must be `-a` to indicate that imports should be added to the IAT, the second argument, `args[1]` refers to the name of the executable to be manipulated, third and fourth arguments, `args[2]` and `args[3]`, refer to the DLL name and the number of imports to be added. Once the name of the file is known, an object of type `PeFile` is created. Then is created an object of type `List<AdditionalImport>`, which includes all the import to be added. From the `PeFile` object is invoked the `AddImports()` method, which

Listing 2.7: Add the new Section Header and validate values for the new Section Header

```
pe = pefile.PE(exe_path)
number_of_section = pe.FILE_HEADER.
    NumberOfSections
last_section = number_of_section - 1
file_alignment = pe.OPTIONAL_HEADER.
    FileAlignment
section_alignment = pe.OPTIONAL_HEADER.
    SectionAlignment
new_section_offset = (pe.sections[
    number_of_section - 1].get_file_offset() +
    40)
raw_size = align(0x1000, file_alignment)
virtual_size = align(0x1000, section_alignment)
raw_offset = align((pe.sections[last_section].
    PointerToRawData + pe.sections[last_section
    ].SizeOfRawData), file_alignment)
virtual_offset = align((pe.sections[
    last_section].VirtualAddress + pe.sections[
    last_section].Misc_VirtualSize),
    section_alignment)
```

Listing 2.8: Set new Section Header characteristics and name

```
# CODE | EXECUTE | READ | WRITE
characteristics = 0xE0000020
# Section name must be equal to 8 bytes
name = ".new" + (4 * "\x00")
```

Listing 2.9: Create the new section and edit the Main Header

```
pe.set_bytes_at_offset(new_section_offset, name
    )
pe.set_dword_at_offset(new_section_offset + 8,
    virtual_size)
pe.set_dword_at_offset(new_section_offset + 12,
     virtual_offset)
pe.set_dword_at_offset(new_section_offset + 16,
     raw_size)
pe.set_dword_at_offset(new_section_offset + 20,
     raw_offset)
pe.set_bytes_at_offset(new_section_offset + 24,
     (12 * "\x00"))
pe.set_dword_at_offset(new_section_offset + 36,
     characteristics)
pe.FILE_HEADER.NumberOfSections += 1
pe.OPTIONAL_HEADER.SizeOfImage = virtual_size +
     virtual_offset
pe.write(exe_path)
```

Listing 2.10: Inject code into the executable

```
pe = pefile.PE(exe_path)
raw_offset = pe.sections[last_section].
    PointerToRawData
pe.set_bytes_at_offset(raw_offset,
    code_to_inject)
pe.write(exe_path)
```

Listing 2.11: Using PeNet to add imports to executable

```
using PeNet;
...
public static void AddMultipleImport(string fileName, string dll, List<string> functions) {
    var peFile = new PeFile(fileName);
    var ai1 = new AdditionalImport(dll, functions);
    var importList = new List<AdditionalImport> { ai1 };
    peFile.AddImports(importList);
    File.WriteAllBytes(fileName, peFile.RawFile.ToArray());
}
```

accepts the list of imports. Once this is done, the file is overwritten with the new noise feature.

2.3.3 Constraints

Obviously, to manipulate the executables, some constraints must be respected between all the features to be manipulated.

Once you understand how to modify all the features, it is necessary to understand how, given an input noise vector, the required features can "get stuck" with each other, i.e. whether the input executable can somehow be modified as required or not. Then, considering all the features, we will define the constraints to be respected and which have been described in the code.

All features that refer to the COFF Header or the Optional Header, such as TimeDateStamp, Machine, Subsystem, Magic, Characteristics and DLL Characteristics, can be modified independently of the others. So in this case no constraint has been defined.

As for the number of strings to add, since these can be interpreted as code to be injected into the executable, it is necessary to add a section to the executable. By adding a section, it is necessary to perform a resizing of the executable in order to avoid that the whole payload is not written inside the new section.

As for the number of imports, the *PeNet* library allows you to add up to a maximum of 19 imports at one time. Obviously, even in this case, to add import to the executable it is necessary to add a new section. However, in this case, unlike the previous case, everything is handled implicitly when the code written in C# is invoked. So, we don't have to worry about adding the section and therefore also resizing the executable, but we must still take into account that these other two features are also affected.

As for adding opcodes inside the executable, these are treated as full-fledged code. As you can imagine, also in this case it is necessary to add a new section and then resize the executable.

Regarding the size and number of sections, these two features can be modified independently from the previous ones. However, considering them in a broader context, such as that of the input noise vector, most likely, you will be required to modify more features at a time. For this reason, since these two features are the basis of those mentioned above, before going to modify the executable, a feasibility study is carried out on the input noise vector. If the required features are consistent with each other, proceed with the modification, otherwise, if more input noise vectors are available, move to the next vector.

Chapter 3

Experimentation and Results

It has been carried out a case study to examine the effectiveness of adversarial models against malware detectors based on deep learning.

Under the realistic hypothesis that an attacker knows very little about the system he wants to attack (the case of a black-box attack), the attacker could set up some sort of brute-force attack by deploying a pool of specific generators built for interacting with a corresponding pool of specific targets (discriminators).

The attack strategy envisages multiple stages (steps). As first, the only knowledge owned by the attacker consists of knowing that the target victim could behave according to a ML or DL model and the kinds of input it could accept; so, the attacker trains several generators working over different groups of features (possibly, he could try over all the sensible combinations) for refining the generation of artificial adversarial samples.

This training stage is performed without effectively interacting yet with the real target. Discriminators play the role of the potential victims, as substitutes of real victim systems. In the middle of attacking time, the attacker will start a smooth interaction with its victim, this time represented by a black box. By carefully analyzing responses from the black-box, it is able to figure out what features are used by the malware detector black-box. Adversarial test cases are produced by exploiting all the trained Generators in the attacker's wallet.

Most of these samples will be harmless since they will not act on the right set of features but we can suppose that almost one of these generators will be able to generate samples that will produce some effects. By this way, the attacker will gain knowledge about its adversary and can implement a gray box attack, basing on the features set its victim works over. At the real attack time, the attacker will exploit only the right generator and proceed to attack and refine its generation model until its target is reached out. The case study we propose should not be regarded as exhaustive but it can be regarded as proof of the concept that adversary attacks pointing ML and DL systems can be implemented in many alternative and successful ways, for tampering with real existing defense systems.

3.1 Case Study Design

Static analysis has the advantage that does not require the execution of samples in a sand-box or safe environment for studying their behaviors, and the features for training the detector and/or classifiers can be extracted over specific subsets of features. Conversely, the dynamic analysis could reveal more information about malicious behaviors by the applications (e.g, actions relationships, and patterns) but the operative conditions are more difficult to achieve. Challenging results obtained by adopting static analysis in training machine and deep learning algorithms for malware detection are described in [12], where the authors provided, as first, an open source data set, namely "EMBER", consisting in a collection surface features extracted from a little under a million of malicious applications targeting Windows O.S. environment; furthermore, they provide experiments that compare a baseline gradient boosted decision tree model trained using LightGBM [40] with default settings to MalConv [41], an end-to-end but featureless deep learning model for malware detection, recently become a very popular benchmark in this kind of experiments.

In the case of malware detection, unlike other application domains, like image and speech recognition, manipulating bytes can severely compromise application functionalities and validity; therefore generating adversarial examples is not straightforward. An unavoidable requirement that should lay down every manipulation strategy consists in adopting generation techniques that are able to guarantee the preservation of malware functionality in the adversarially manipulated samples. In the evaluation it has been trained, validated and tested generator models for adversarial examples, by adopting for all the same noise vectors extracted from the EMBER data set classification.

3.2 Data Set

It has been chosen EMBER released by Endgame as the data set for the case study. EMBER is a collection of features extracted from a large corpus of Windows portable executables.

The first version of the data set is a collection of 1.1 million PEs that were all scanned by VirusTotal in 2017. The second EMBER dataset release consisted of features extracted from samples collected in or before 2018. The set of binary files is divided as follow:

- 900,000 training samples grouped in:

  - 300,000 malicious;
  - 300,000 benign;
  - 300,000 unlabeled.

- 200,000 test samples grouped in:
  - 100,000 malicious;
  - 100,000 benign.

The data set is made up of JSONL (JSON Lines) files, where on each line there is a JSON instance. Each sample includes:

- the sha256 hash of the original file as a unique identifier;
- the month and the year the file was first seen;
- a label, which may be 0 for benign, 1 for malicious or -1 for unlabeled;
- eight groups of raw features that include both parsed values as well as format-agnostic histograms.

An instance example from EMBER data set JSONL files describing PEs features is shown in Figure 3.1.

The raw features include both parsed features and format-agnostic histograms and counts of strings. Parsed features, extracted from the PE file, are:

- *general file*: information including the file size and basic information obtained from the PE header;
- *header information*: reporting the timestamp, the target machine (string) and a list of image characteristics (list of strings). From the optional header, the target subsystem (string), DLL characteristics (a list of strings), the file magic as a string (e.g., "PE32"), major and minor image versions, linker versions, system versions and subsystem versions, and the code, headers and commit sizes are provided;
- *imported functions*: after having parsed the import address table, the imported functions by the library are reported;
- *exported functions*: the raw features include a list of the exported functions;
- *section information*: properties of each section are provided, including the name, the size, the entropy, the virtual size, and a list of strings representing section characteristics.

```
  "sha256": "000185977be72c8b007ac347b73ceb1ba3e5e4dae4fe98d4f2ea92250f7f580
  "appeared": "2017-01",
  "label": -1,
  "general": {
    "file_size": 33334,
    "vsize": 45056,
    "has_debug": 0,
    "exports": 0,
    "imports": 41,
    "has_relocations": 1,
    "has_resources": 0,
    "has_signature": 0,
    "has_tls": 0,
    "symbols": 0
  },
  "header": {
    "coff": {
      "timestamp": 1365446976,
      "machine": "I386",
      "characteristics": [ "LARGE_ADDRESS_AWARE", ..., "EXECUTABLE_IMAGE" ]
    },
    "optional": {
      "subsystem": "WINDOWS_CUI",
      "dll_characteristics": [ "DYNAMIC_BASE", ..., "TERMINAL_SERVER_AWARE"
      "magic": "PE32",
      "major_image_version": 1,
      "minor_image_version": 2,
      "major_linker_version": 11,
      "minor_linker_version": 0,
      "major_operating_system_version": 6,
      "minor_operating_system_version": 0,
      "major_subsystem_version": 6,
      "minor_subsystem_version": 0,
      "sizeof_code": 3584,
      "sizeof_headers": 1024,
      "sizeof_heap_commit": 4096
    }
  },
  "imports": {
    "KERNEL32.dll": [ "GetTickCount" ],
    ...
  },
  "exports": []
  "section": {
    "entry": ".text",
    "sections": [
      {
        "name": ".text",
        "size": 3584,
        "entropy": 6.368472139761825,
        "vsize": 3270,
        "props": [ "CNT_CODE", "MEM_EXECUTE", "MEM_READ"]
      },
      ...
    ]
  },
  "histogram": [ 3818, 155, ..., 377 ],
  "byteentropy": [0, 0, ... 2943 ],
  "strings": {
    "numstrings": 170,
    "avlength": 8.170588235294117,
    "printabledist": [ 15, ... 6 ],
    "printables": 1389,
    "entropy": 6.259255409240723,
    "paths": 0,
    "urls": 0,
    "registry": 0,
    "MZ": 1
  },
}
```

Figure 3.1: An instance example from EMBER data set

The EMBER data set also includes three groups of features that are format-agnostic, as they do not require parsing the PE file:

- *byte histogram* contains 256 integer values, representing the count of each byte value within the file. The byte histogram is normalized to a distribution, since the file size is represented as a feature in the general file information;
- *byte-entropy histogram* approximates the joint distribution p(H,X) of entropy H and byte value X;
- *string information* reported is the number of strings, their average length, a histogram of the printable characters within those strings, and the entropy of characters across all the printable strings.

3.3 Case Study Toolchain

The toolchain defined, in order to conduct a testing and validation phase of the engine, is shown in Figure 3.2.

We can divide the toolchain into four macro-steps:

- construction of different classification models and choice of the most accurate and precise one;
- extraction of feature thresholds and generation of noise vectors;
- manipulation of executables and generation of adversarial examples;
- extraction of the features from the adversarial examples and construction of a test set to be given as input to the built model in order to validate the engine.

A subset of EMBER data set was considered for the construction of the training set. In particular, this subset is composed of 80,000 genuine samples, divided into 40,000 goodware and 40,000 malware.

The total number of features comprised in each PE is equal to 2,351, grouped in 8 families, according to the PE specifications [2]. Families' names and their quantity are provided in Table 3.1.

Not all the features have been considered for the classification phase, but only a subset, as the study is of an exploratory type and wants to try to understand the feasibility of this approach, that is to modify not only the surface features of the binaries files, but also the content itself. Manipulating the content of a binary file, even also changing a small number of byte, can severely compromise the behavior and the functionalities of the application.

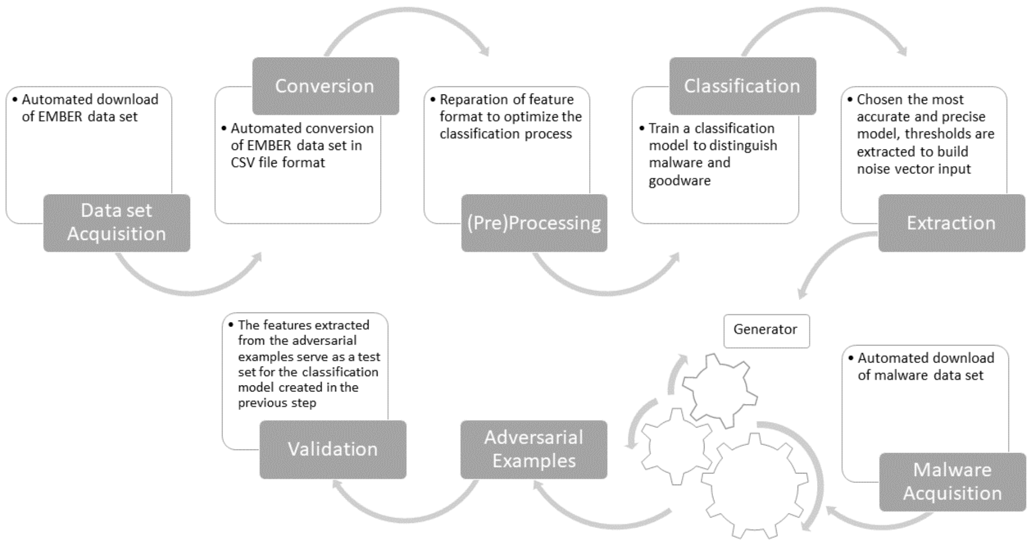

Data set Acquisition
• Automated download of EMBER data set
Conversion
• Automated conversion of EMBER data set in CSV file format
(Pre)Processing
• Reparation of feature format to optimize the classification process
Classification
• Train a classification model to distinguish malware and goodware
Extraction
• Chosen the most accurate and precise model, thresholds are extracted to build noise vector input
Generator
Malware Acquisition
• Automated download of malware data set
Adversarial Examples
Validation
• The features extracted from the adversarial examples serve as a test set for the classification model created in the previous step

Figure 3.2: Toolchain for the experimentation

Description and Original Name	**Number of Features**
General File Info (General)	10
Header Info (Header)	62
Imported Functions (Imports)	1280
Exported Functions (Exports)	128
Section Info (Section)	255
Byte Histogram (Histogram)	256
Byte-Entropy Histogram (Byte entropy)	256
String Info (Strings)	104

Table 3.1: PEs Feature Groups in the EMBER data set

This aspect, also investigated in the works of [30] and [31], will be a matter of further investigations, possibly joining both surface features and payload of binary files.

As classification algorithms, several were considered and, subsequently, the most accurate and precise one was chosen. This algorithm, with an accuracy of 99.9975%, is the *Random forest* or *random decision forest* is an ensemble learning method for classification, regression and other tasks that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean/average prediction (regression) of the individual trees. It uses bagging and feature randomness when building each individual tree to try to create an uncorrelated forest of trees whose prediction by committee is more accurate than that of any individual tree. The number of iterations with which the algorithm has been executed is equal to 100, a good compromise between computational resources and execution times.

The classification algorithm, of course, outputs as many trees as there are iterations. Having chosen a number of iterations equal to 100, it is clear that 100 decision trees have been obtained as output. Thinking of treating all 100 trees, given the number of attributes (290) for their construction, is paradoxical, as there are about 50,000 tree leaves that categorize software as goodware. This means that in traversing the tree you will get about 50,000 noise vectors. For this reason, we have chosen to parse only one tree of the 100 products, in order to consider at least 10% of the possible noise vectors that can be obtained.

Once the noise vectors have been obtained, having available a data set of malware that have been downloaded from MalwareBazaar[1], a project from *abuse.ch* with the goal of sharing malware samples with the infosec community, AV vendors and threat intelligence providers, it is possible to activate the key component of the whole toolchain, the generator.

As mentioned in the previous chapter, the generator takes as input a series of noise vectors, extracted from the classification model and an executable. This is because, if we put ourselves in the shoes of a threat actor, having more ways to manipulate the executable and deceive the AV, we try to choose the one that best suits our needs. The engine, for each input noise vector, checks its consistency with the features present within the executable to be manipulated. Checking the consistency means respecting all those constraints that have been previously defined starting from the features required within the input vector. If these constraints are respected, it is possible to go and manipulate the executable with these features. Obviously, it can happen that for a single executable there can be more noise vectors that respect the constraints, just as it can happen that none of the noise vectors respect them. In the latter case, the executable cannot be manipulated. Its modification would lead to the violation of one or more features within the input vector.

Once the adversarial example has been created, it is possible to extract the features that have been used for the classification and manipulation of the executable. In particular, every time an adversarial example is generated, its features are appended in a CSV file, which will then be the test set of the prediction algorithm to validate the engine itself.

3.4 Results

The described toolchain was run on a malware data set containing 120 instances. This reduction is due to the fact that the time it takes to increase the

[1]`https://bazaar.abuse.ch/` (visited on 12/14/2020)

number of malware is quite high. It should be noted that only 5 malware are manipulated in two hours. Furthermore, quite high computational capabilities are required, since the algorithm inside the engine has a complexity equal to `O(N*M*F)`, where `N` is the number of noise vectors, `M` is the number of malware and `F` is the number of manipulated features.

With such instances, a test set of 167 instances was produced. This means that more than one manipulation has been possible for some malware, and, therefore, the threat actor could choose which of the noise vectors is best suited to their needs.

Of the 167 instances produced, 165 were classified as goodware, while the remaining 2 were classified as malware. The **evasion rate** has been chosen as the performance metric, that is the ability of the generator to produce adversarial samples that are missclassified. In this perspective, the evasion rate can be adopted as an indicator for measuring the generation ability and is defined, according to the definition provided in [47] as follows:

$$Evasion\,Rate\,(EV) = \frac{FN_{AEs}}{N_{AEs}} \tag{3.1}$$

where $\mathrm{N_{AEs}}$ represents the number of artificially generated adversarial samples of malware submitted to the detector; $\mathrm{FN_{AEs}}$ is the fraction of the overall counted False Negatives (malware incorrectly classified as goodware) represented by adversarial samples set (that is to say artificially generated malware incorrectly classified as goodware). Thus, we have an evasion rate of 98.80%. The result is obviously positive, as it means that the engine is able to manipulate the features correctly fooling the discriminator and, therefore, respecting the noise parameters in the input vector.

Following the classification, an analysis of test set was conducted on the different features manipulated, to understand if they were specific of goodware and if there were any patterns. Obviously these results will be related to the noise vector that is used. As other values are generated, the results may change.

Figure 3.3 represents an interval plot with respect to the variables `numstrings`, `imports` and `numsections`, which respectively represent the number of strings, imports and sections within the executable. These values are then grouped by the *decision variable* `label`, which is -1 for malware and 0 for goodware. As you can see, within the malware we find a fairly low number of strings, with an average of 25 with a range that goes from 12 to 38. This happens because very often the code is obfuscated and, therefore, even if present, the strings are not recognized. For obvious reasons, in goodware the number of strings is quite high with an average value of 303.54 with a range that goes from 228 to 379. As far as imports are concerned, the speech is similar to that of strings. For malware the average value is equal to 1, while for

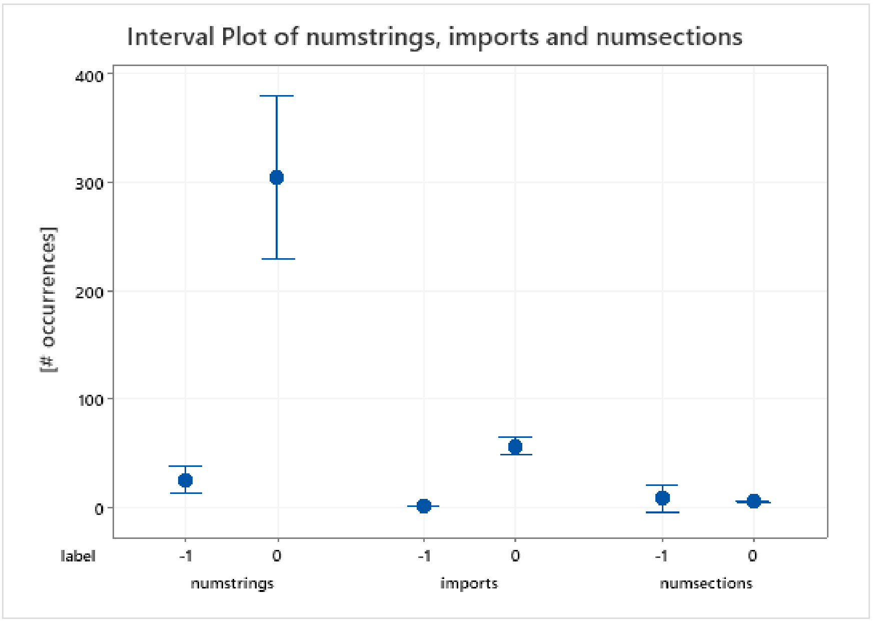


Figure 3.3: Interval Plot of number of strings, number of imports and number of sections

goodware it is equal to 56.26 with a range from 48 to 64. Finally, the number of sections is a similar value for both types of executables. For malware there is an average value of 8 with a range from 0 to 20. While for goodware there is an average value of 5 with a range from 4 to 5.

Figure 3.4 represents an interval plot with respect to the variables `timestamp`. As you can see, for malware this value has a very high range (319741981 - 24737222318). This happens because very often there is a tendency to disguise malicious files through this value. For goodware, on the other hand, the range is narrower. We have a range from 1260105867 to 1389021232 with an average value of 1324563549, which is Thursday 22 December 2011 14:19:09.

The same goes for the size of the executable that is represented in Figure 3.5. For a malware we have a range that goes from -1601439 to 10623646. The lower limit is negative as this means that the executable has a size greater than 4 GB. While the goodware has a range that goes from 1591377 to 2312888 that is, we range from 1 to 2 MB.

Figure 3.6 and 3.7 show the interval plots for the opcodes that we find inside the executable. For greater visibility, they have been reported in more graphs. As it can be noted, almost all bytes have a similar average value, but some are particular, as for malware they do not occur with a wider range, but also an average value that differs a lot from the average value present in the case some goodware. An example are bytes 0, 51, 56, 68, 130, 244 and 255.

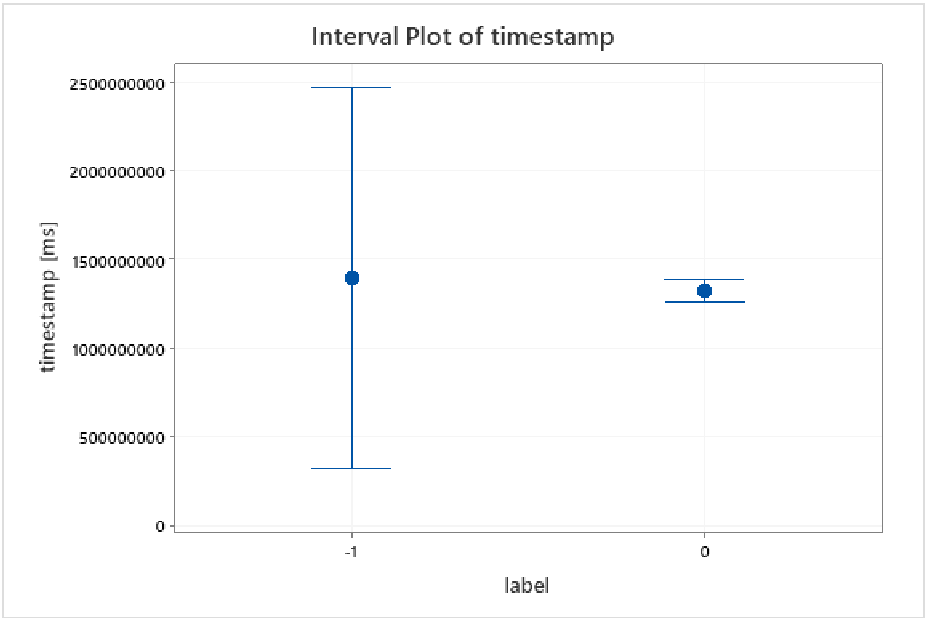


Figure 3.4: Interval Plot of TimeDateStamp field

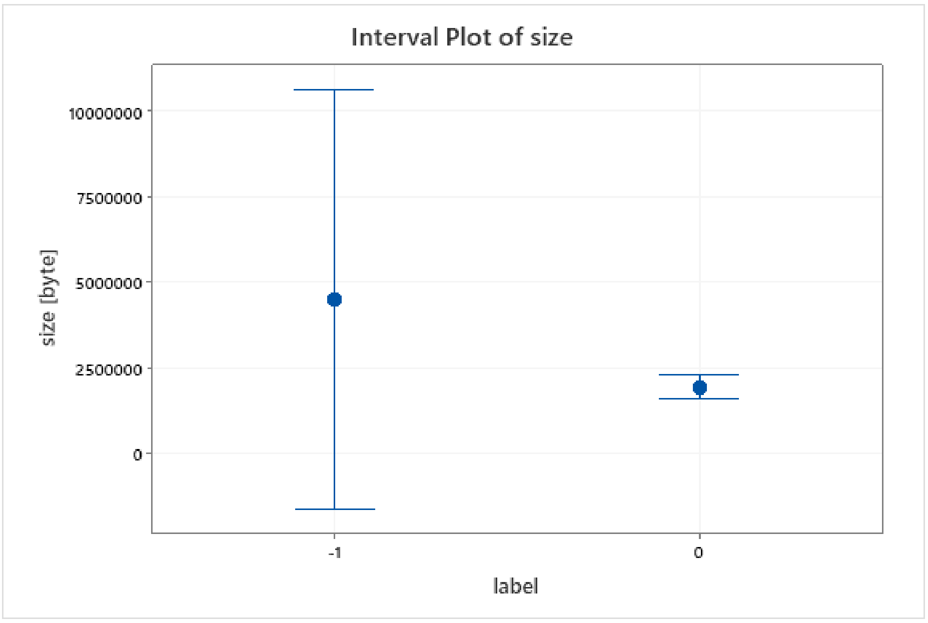
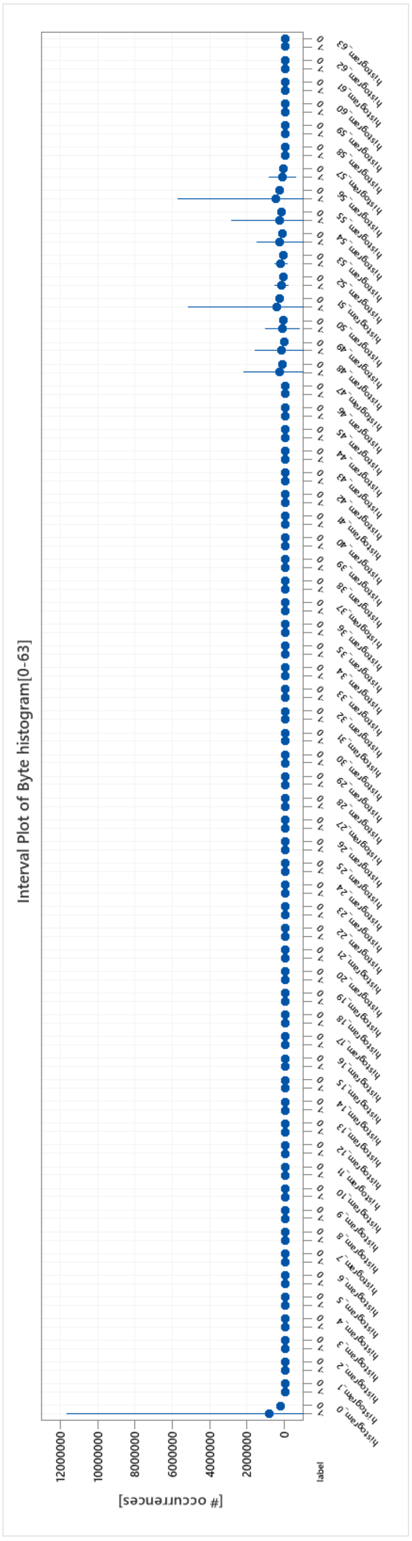


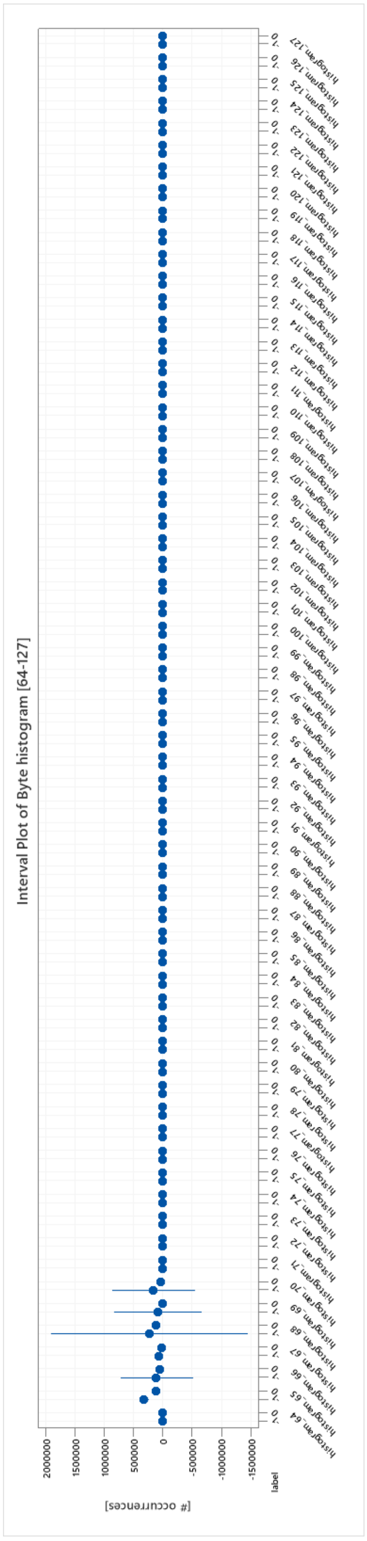
Figure 3.5: Interval Plot of Size field

It is no coincidence that these instructions refer to operations of `ADD`, `JMP`, `PUSH`, `MOV`, and so on. It is coherent with [21], where preliminary experiments revealed, at a first sight analysis, that the byte distribution (byte histogram) is among the most sensitive features. This finding could suggest that machine and deep learning based malware detectors, which work on static and surface features, could be fooled by adversarial malicious samples that are able to reach a bytes distribution with a high level of likelihood with the goodware bytes distribution.

Finally, the latest features to be analyzed refer to the Characteristics and the DLL Characteristics of the File Header and the Optional Header of the PE, respectively. The graph in Figure 3.8 is quite clear and shows how the Characteristics 3 and 12 and the DLL Characteristic 0, respectively `IMAGE_FILE_LOCAL_SYMS_STRIPPED`, `IMAGE_FILE_DLL` and `IMAGE_DLLCHARACTERISTICS_HIGH_ENTROPY_VA`, are discriminating for the distinction between the two types of software.

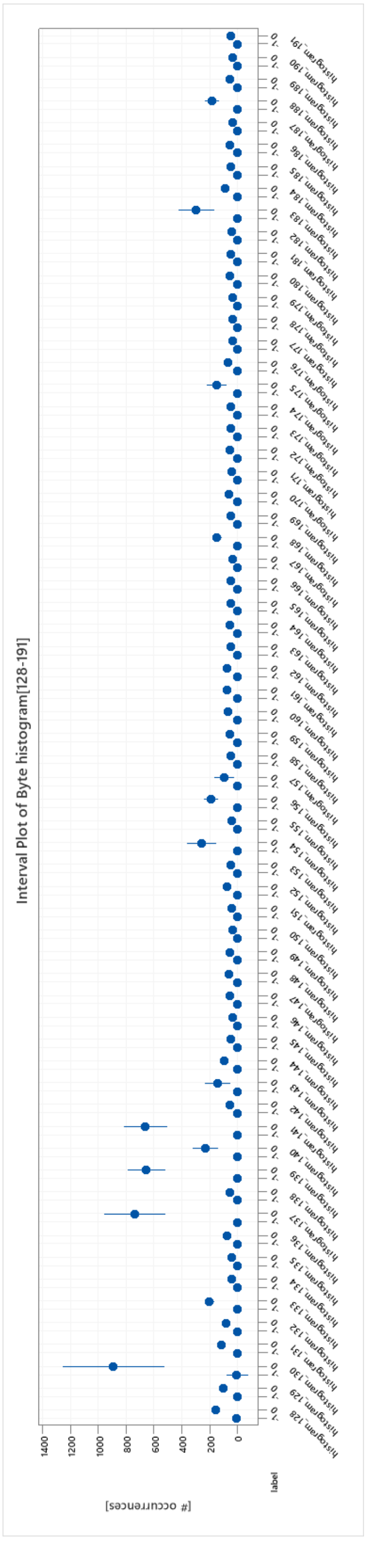


(a)

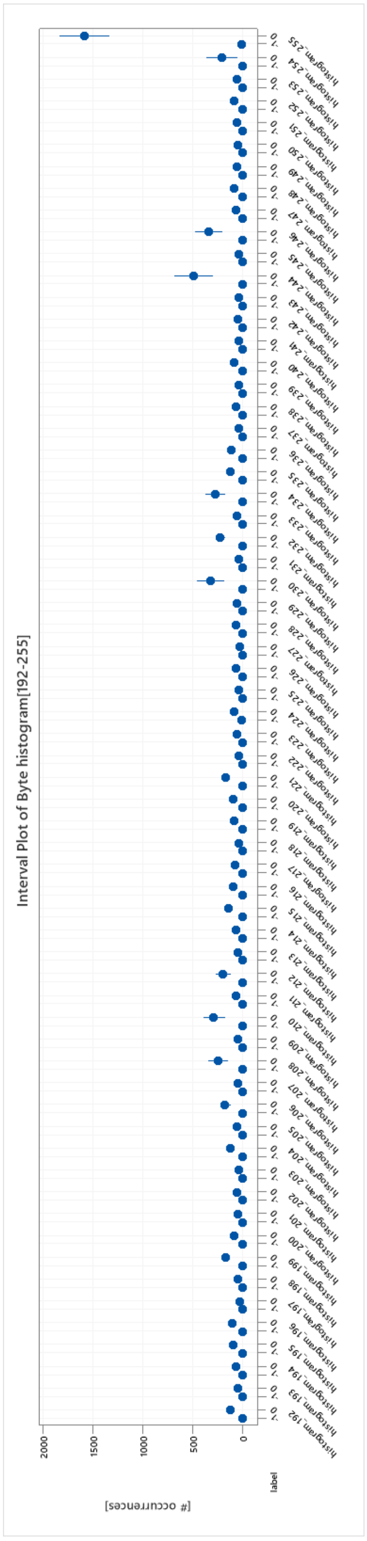


(b)

Figure 3.6: Interval plot of Byte Histogram [0-127]



(a)



(b)

Figure 3.7: Interval plot of Byte Histogram [128-255]

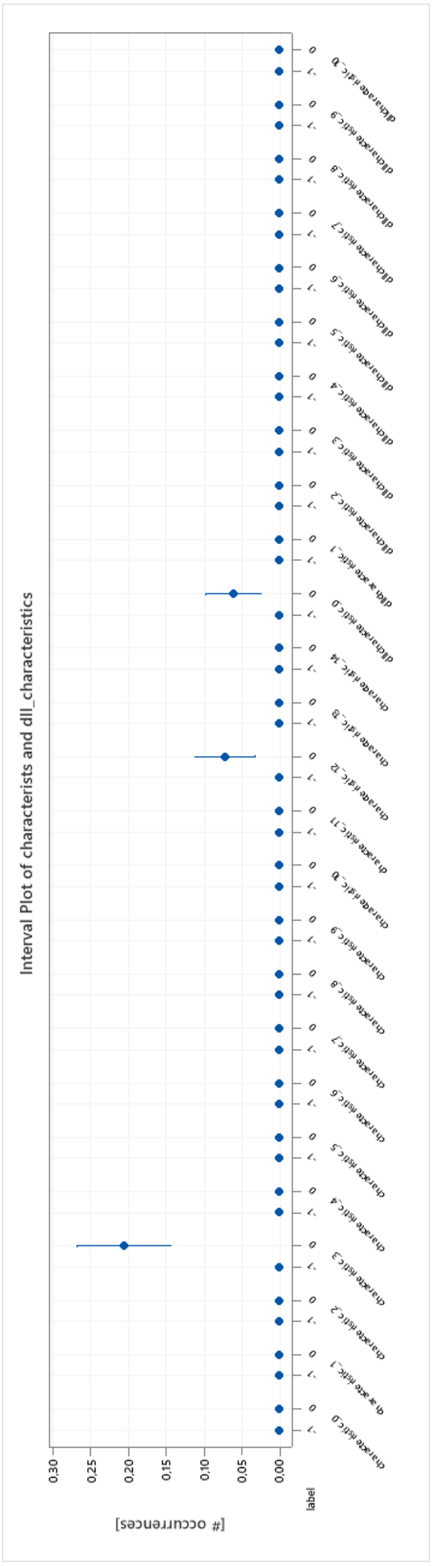


Figure 3.8: Interval plot of File Header Characteristics and Optional Header DLL Characteristics

Conclusions

Presence of adversarial samples indicates the vulnerability of the machine learning models. Learning robust models and measuring their susceptibility against adversarial attacks are need of the day [45].

Attacks and defenses on adversarial examples draw great attention. The vulnerability to adversarial examples becomes one of the major risks for applying DNNs in safety critical environments. Adversarial perturbations can easily fool Deep Neural Networks (DNNs) in the testing/deploying stage exploiting blind spots in the ML engine. The effectiveness of an adversarial system is measured in terms of what is commonly called Evasion Rate. As in [21], the evasion rate represents a measure of the success rate obtained by generators networks in fooling their opponent discriminators; it can be computed as the ratio between the number of adversarial examples that were misclassified as benign samples (also referred in the following as "goodware") by each detector, over the total amount of adversarial samples submitted to the discriminators. It depends upon a specific group of features considered for the input set.

Applied to the creation of malware, GANs are able to generate a new instance of a malware family without knowing an explicit model of the initial distribution of the data. So an attacker could use GANs to fool detection systems, just by sampling the provided data. On the other hand, GANs are also useful to build more robust machine learning models helping in the development of a better training set.

Real defense technologies such as AV or EDR must take into account an acceptable trade-off among the detection accuracy, short learning times and limit the size of data obtain able by selecting a convenient combination of the sensitive feature. The effectiveness of an attack on the ML model also depends on the knowledge of the system by the attacker. In this case study, we conducted a grey-box attack in which the features of the training set are known: this permits us to reach a very high evasion rate (about 99%).

Future research should consider, as first step, to integrate the engine in a GAN for the generation of adversarial samples. As a second step, you can think of the potential effects of other features in executable files more carefully, such as the Section Header and the content of the sections themselves. This is because very often these are customized, but if generated by well-known packers they could somehow experience some major implications. Furthermore, one might think of trying to work on the EAT (Export Address Table of PE file), to try to understand if there are patterns present in the exported functions and, if so, how they affect the evasion rate. Finally, it would be desirable to conduct a validation on a much larger data set of malware and to consider all decision trees that have been generated by the classification algorithm.

Bibliography

[1] *Microsoft Portable Executable and Common Object File Format Specification* (revision 6.0, .doc format), 1999.

[2] Matt Pietrek. *Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format.* In MSDN Magazine & MSJ, Vol.17 (No. 2, 3), 2002.

[3] M. Brand, C. Valli, and A. Woodward. *Malware forensics: Discovery of the intent of deception.* In The Journal of Digital Forensics, Security and Law: JDFSL, 5(4):31, 2010.

[4] D. Devi and S. Nandi. *Pe file features in detection of packed executables.* In International Journal of Computer Theory and Engineering, 4(3):476, 2012.

[5] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville and Yoshua Bengio. *Generative Adversarial Nets.* arXiv preprint arXiv:1406.2661, 2014.

[6] Bengio, Y. *Learning deep architectures for AI.* Now Publishers, 2009.

[7] Hinton, G., Deng, L., Dahl, G. E., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., and Kingsbury, B. *Deep neural networks for acoustic modeling in speech recognition.* In IEEE Signal Processing Magazine, 29(6), 82–97, 2012.

[8] Krizhevsky, A., Sutskever, I., and Hinton, G. *ImageNet classification with deep convolutional neural networks.* In NIPS', 2012.

[9] Glorot, X., Bordes, A., and Bengio, Y. *Deep sparse rectifier neural networks.* In AISTATS, 2011.

[10] Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. *Maxout networks.* In ICML', 2013.

[11] Jarrett, K., Kavukcuoglu, K., Ranzato, M., and LeCun, Y. *What is the best multi-stage architecture for object recognition?* In Proc. International Conference on Computer Vision (ICCV'09), pages 2146–2153., IEEE, 2009.

[12] Hyrum S. Anderson and Phil Roth. *EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models.* arXiv preprint arXiv:1804.04637, 2018.

[13] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. *Survey of machine learning techniques for malware analysis.* In Computers & Security, 81:123–147, 2019.

[14] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H Austin, and Mark Stamp. *A comparison of static, dynamic, and hybrid analysis for malware detection.* In Journal of Computer Virology and Hacking Techniques, 13(1):1–12, 2017.

[15] Ivan Firdausi, Alva Erwin, Anto Satriyo Nugroho, et al. *Analysis of machine learning techniques used in behavior-based malware detection.* In 2010 second international conference on advances in computing, control, and telecommunication technologies, pages 201–203. IEEE, 2010.

[16] Ahmad Azab, Mamoun Alazab, and Mahdi Aiash. *Machine learning based botnet identification traffic.* In 2016 IEEE Trustcom/BigDataSE/ISPA, pages 1788–1794. IEEE, 2016.

[17] Ahmad Azab, Robert Layton, Mamoun Alazab, and Jonathan Oliver. *Mining malware to detect variants.* In 2014 Fifth Cybercrime and Trustworthy Computing Conference, pages 44–53. IEEE, 2014.

[18] Mamoun Alazab, SitalakshmiVenkatraman, PaulWatters, and Moutaz Alazab. *Information security governance: the art of detecting hidden malware.* In IT security governance innovations: theory and research, pages 293–315. IGI Global, 2013.

[19] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. *{TESSERACT}: Eliminating experimental bias in malware classification across space and time.* In 28th fUSENIXg Security Symposium (fUSENIXg Security 19), pages 729–746, 2019.

[20] Giovanni Apruzzese, Michele Colajanni, Luca Ferretti, Alessandro Guido, and Mirco Marchetti. *On the effectiveness of machine and deep learning for cyber security.* In 2018 10th International Conference on Cyber Conflict (CyCon), pages 371–390. IEEE, 2018.

[21] Corrado Aaron Visaggio, Fiammetta Marulli, Sonia Laudanna, Benedetta La Zazzera and Antonio Pirozzi. *A Comparative Study of Adversarial Attacks to Malware Detectors Based on Deep Learning.* In Malware Analysis Using Artificial Intelligence and Deep Learning, Springer International Publishing, 2021.

[22] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and J Doug Tygar. *Adversarial machine learning.* In Proceedings of the 4th ACM workshop on Security and artificial intelligence, pages 43–58, 2011.

[23] Jason Brownlee. *Generative Adversarial Networks with Python: Deep Learning Generative Models for Image Synthesis and Image Translation.* In Machine Learning Mastery, 2019.

[24] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective.* In Adaptive Computation and Machine Learning series, 2012.

[25] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* 2006.

[26] Alec Radford and Luke Metz and Soumith Chintala. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.* arXiv preprint arXiv:1511.06434, 2016.

[27] Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning.* In Adaptive Computation and Machine Learning series, 2016.

[28] Ian Goodfellow. *NIPS 2016 Tutorial: Generative Adversarial Networks.* arXiv preprint arXiv:1701.00160, 2017.

[29] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. *Adversarial examples: Attacks and defenses for deep learning.* In IEEE transactions on neural networks and learning systems, 30(9):2805–2824, 2019.

[30] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. *Adversarial malware binaries: Evading deep learning for malware detection in executables.* In 2018 26th European Signal Processing Conference (EUSIPCO), pages 533–537, IEEE, 2018.

[31] Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. *Deceiving end-to-end deep learning malware detectors using adversarial examples.* arXiv preprint arXiv:1802.04528, 2018.

[32] Jinlan Zhang, Qiao Yan, and Mingde Wang. *Evasion attacks based on wasserstein generative adversarial network.* In 2019 Computing, Communications and IoT Applications (ComComAp), pages 454–459, IEEE, 2019.

[33] Babak Bashari Rad, Maslin Masrom and Suhaimi Ibrahim. (2012). *Opcodes Histogram for Classifying Metamorphic Portable Executables Malware.* In International Conference on E-Learning and E-Technologies in Education, ICEEE, 2012.

[34] Ayan Dey, Sukriti Bhattacharya and Nabendu Chaki. *Byte Label Malware Classification using Image Entropy.* In ACSS, 2019

[35] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. *Intriguing properties of neural networks.* arXiv preprint arXiv:1312.6199, 2013.

[36] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. *Explaining and harnessing adversarial examples.* arXiv preprint arXiv:1412.6572, 2014.

[37] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. *Adversarial perturbations against deep neural networks for malware classification.* arXiv preprint arXiv:1606.04435, 2016

[38] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. *The limitations of deep learning in adversarial settings.* In Security and Privacy (EuroS&P), 2016 IEEE European Symposium on, pages 372–387. IEEE, 2016.

[39] Weiwei Hu and Ying Tan. *Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN.* arXiv preprint arXiv:1702.05983, 2017.

[40] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye and Tie-Yan Liu. *Lightgbm: A highly efficient gradient boosting decision tree.* In Advances in neural information processing systems, pages 3146–3154, 2017.

[41] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro and Charles Nicholas. *Malware detection by eating a whole exe.* arXiv preprint arXiv:1710.09435, 2017.

[42] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. emphDelving into transferable adversarial examples and black-box attacks. arXiv preprint arXiv:1611.02770, 2016.

[43] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. *Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models.* In Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security, pages 15–26. ACM, 2017.

[44] Amir Nazemi and Paul Fieguth. *Potential adversarial samples for white-box attacks.* arXiv preprint arXiv:1912.06409, 2019.

[45] Vivek B.S., Konda Reddy Mopuri and R. Venkatesh Babu. *Gray-box Adversarial Training.* arXiv preprint arXiv:1808.01753, 2018.

[46] Masataka Kawai, Kaoru Ota, and Mianxing Dong. *Improved malgan: Avoiding malware detector by leaning cleanware features.* In 2019 International Conference on Artificial Intelligence in Information and Communication (ICAIIC), pages 040–045. IEEE, 2019.

[47] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndic, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. *Evasion attacks against machine learning at test time.* In Joint European conference on machine learning and knowledge discovery in databases, pages 387–402. Springer, 2013.