UCLouvain

École polytechnique de Louvain

Adversarial Tool for Breaking Static Detection of Executable Packing

Author: **Jaber RAMHANI**
Supervisors: **Prof. Axel LEGAY, Alexandre D'HONDT, Charles-Henry BERTRAND VAN OUYTSEL**
Reader: **Prof. Wim MEES**
Academic year 2023–2024
Master in Cybersecurity

Abstract

Executable packing acts as a double-edged sword in the cyberspace, used to safeguard legitimate software but also for concealing malware. This technique changes how a computer program looks without affecting how it works, making it harder for security tools to detect when something is harmful. The ability to detect whether an executable is packed or not becomes capital in this scenario, as it signals the need for further investigation to understand the nature of the packing and the potential intentions behind it. Before any meaningful analysis of the executable behavior or code can be conducted, it is essential to determine if the executable has been packed and reverse the packing process for thorough investigations. Given that the authors of packed executables actively try to evade detection, analysis and reverse engineering of their code, it is imperative for a packing detection tool to be robust against adversarial attacks.

A great deal of research has focused on the static features of executables and their use with machine learning to either detect packing or maliciousness. In particular, some studies defined sets of such features, proving their relevance with many learning algorithms and a more recent study further identified the most significant of them, specifically for static detection of packing. In 2022, an experimental toolkit named Packing Box, presented by D'Hondt in the master thesis "*Experimental Toolkit for Studying Executable Packing – Analysis of the State-of-the-Art Packing Detection Techniques*" [1], was introduced for static analysis of packed executables. This aims to enable researchers to more effectively analyze and understand the detection of packed executables. Following this, an adversarial study, presented by Jennes in the master thesis "*Adversarial Learning on Static Detection Techniques for Executable Packing*" [2], targeted the vulnerabilities of static packing detectors and explored how specific binary alterations could evade static detection mechanisms.

This thesis aims to enhance the Packing Box, especially pushing the adversarial study of Jennes one step further by improving the current alterations set then expanding it and by studying their effects and impact on detection mechanisms. Our main contribution is thus to review and enhance existing alterations, to make new ones, to test them and to introduce a new tool based on them, **NotPacked++** . Furthermore, this attack tool is the weaponization of current adversarial studies and is designed to apply complex combinations of interleaved or interdependent alterations to packed executables that we call super-alterations, making them evade static detection by most common packing detectors. In this way, researchers in packing detection could use the tool to test the vulnerability of their model during the design phase, which is a key component to improve static detectors against malware.

Foreword

"True cybersecurity resilience is achieved by rigorously testing defenses with adversarial tools, revealing weaknesses and fortifying strengths."

ChatGPT4o

Executable packing is a well-known problematic in the field of reverse engineering and malware analysis. It often applies compression or encryption to hide parts of the target binary and prevent static detection. It also pertains to anti-reversing tricks that make reverse engineering a lot more difficult. Many static techniques – that is, not executing the target binary – have been developed in order to take advantage of the low overhead in computing characteristics, most often for deriving features for use with machine learning models like in the works of Biondi *et al.* [3] and D'Hondt [1], compared to running it with dynamic techniques – which requires execution. However, many static detectors rely on *significant* features, as identified by Bertrand Van Ouytsel *et al.* [4] in his work relying on the feature set of Biondi *et al.* [3].

Within this framework, a study was carried out by Jennes [2] in 2023 to extend the capabilities of the Packing Box [5] made by D'Hondt [1] in 2022. This thesis focused on the problem space by applying modifications that do not break executable's operation (e.g. section renaming, entropy decrease or signature replacement), sufficiently disturbing features to evade static detection to a partial or even complete extent. This adversarial study showed that the detection rate could drastically be decreased for some common detectors because their heuristics and underlying features were too simplistic. It was then shown by D'Hondt *et al.* [6] with some more advanced alterations – which are combinations of slight modifications – that some common and popular static detectors used in practical applications could be deeply impacted.

A typical scenario relying on this type of adversarial attack could be to evade mass-detection involved in a sample triage procedure prior to malware analysis. In this setting, we could for instance imagine a variant of the popular UPX packer, patched with new routines implementing the alterations of the Packing Box, to be applied after classical UPX-packing, so that this does not get detected anymore. Another solution is to use a dedicated attack tool that can be chained after packer's execution.

This master thesis reviews and enhances adversarial related capabilities of the Packing Box, improving existing alterations and finding new ones, and proposes such an adversarial tool, aptly named NotPacked++ (in reference to the popular text editor named Notepad++). Thanks to this work, Packing Box's adversarial capabilities reach a new level, bringing it one step closer to assessing feature robustness and finding even better features.

Alexandre D'Hondt, ir

Acknowledgments

I would like to express my deepest gratitude to those who have supported me throughout the journey of completing this master's thesis.

First and foremost, I would like to express my gratitude to my promotor, Professor Axel Legay, for his encouragement and support throughout this work, as well as for the engaging and pragmatic courses he presented.

I am also profoundly grateful for the guidance provided by my copromotors, Alexandre D'Hondt and Charles-Henry Bertrand Van Ouytsel. Their expertise, continuous encouragement, consistent support and insightful feedback were crucial in shaping the course of this research. Their close involvement and enthusiasm made this journey not only successful but also incredibly exciting.

I would also like to extend my sincere thanks to Professor Wim Mees, who acted as a reader for this thesis. I am especially grateful for the time and effort he dedicated to reviewing my work and for the valuable knowledge he shared during his courses. His teaching ignited my passion for reversing and malware analysis, which was a key motivator in my pursuit of this research.

Finally, I am deeply grateful to my friends and family for their enduring patience and consistent support over the past five years. Their belief in me has been a constant source of strength and motivation.

Table of Contents

List of Acronyms

AES	Advanced Encryption Standard
AML	Adversarial Machine Learning
API	Application Programmer Interface
APT	Advanced Package Tool
ASLR	address space layout randomization
AV	Anti-Virus
CFG	Control Flow Graph
CLI	Command Line Interface
COFF	Common Object File Format
CPU	Central Processing Unit
DEX	Dalvik EXecutable
DIE	Detect It Easy
DLL	Dynamic Link Library
ELF	Executable and Linkable Format
EP	Entry Point
ESCAPE	Entropy SCore Analysis of Packed Executable
FGM	Fast Gradient Method
GUI	Graphical User Interface
IAT	Import Address Table
IG	Information Gain
LIEF	Library to Instrument Executable Formats
LLM	Large Language Model
Mach-O	Mach Object
MITM	Man In The Middle
ML	Machine Learning

MZ (DOS)	Mark Zbikowski Disk Operating System
NN	Neural Network
OEP	Original Entry Point
OOP	Object Oriented Programming
OS	Operating System
PE	Portable Executable
PEAL	Packed Executable AnaLysis
PEiD	Packed Executable iDentifier
PHAD	PE Header Analysis-based packing Detection
RAM	Random Access Memory
RL	Reinforcement Learning
RNN	Recurrent Neural Network
RVA	Relative Virtual Address
SBA	Static Binary Analysis
UPX	Ultimate Packer for Executables
VA	Virtual Address
VM	Virtual Machine
Wine	Wine Is Not an Emulator
YAML	Yet Another Markup Language

List of Figures

List of Listings

List of Tables

1 Introduction

This chapter introduces the problematic and states the objectives of this thesis.

It first states the problem of packing in a general way. It then enumerates objectives for proposing our solution and provides some research questions we will focus on. Finally, it presents the structure of next chapters of this document.

Domain	Static Binary Analysis
Scope	Weaponization of evasion attacks against static packing detectors
Audience	Researchers, Malware Analysts
Purpose	Draw a state-of-the-art of static evasion techniques and tools available, propose an adversarial tool to evade static detection of packed executables

1.1 Problem Statement

Executable packing is a process of applying a transformation to a binary file through compression, encryption, or virtualization. It is used by legitimate software developers as a protective measure for intellectual property, license management, or as a technique for reducing file sizes. This use of packing is essential for maintaining the integrity and efficiency of software, providing a shield against piracy and unauthorized alterations. However, malicious actors also take advantage of packing to evade detection of malicious software by detection systems, using it to disguise malware as legitimate harmless executable files. This abuse of packing poses a significant challenge for security professionals, who strive to find the best way to differentiate between benign and malicious packed files. The paradox of executable packing — its role in both protecting legitimate software and facilitating cyber threats — creates a complex landscape for cybersecurity. This dual role requires advanced detection and analysis techniques to effectively manage the delicate balance between protection and evasion. Unpacking is necessary before analyzing the content of a packed executable. In order to do that we first need to determine if the executable have been packed or not, and also identify which packer was used.

Many static detection techniques have been developed in order to take advantage of the low computational power requirements and to avoid the need of executing files in a sandbox for dynamic analysis. Most often those static techniques are used to derive features from executables to be used with Machine Learning models, as demonstrated in the work of Biondi et al. [3]. However, many static detectors rely on a few common features as shown by Bertrand Van Ouytsel et al. [4], which seems to be vulnerable against an adversarial attack as the research of Jennes [2] shows. In 2022, D'Hondt [1] developed the Packing Box [7], an experimental toolkit for static analysis to efficiently study packed executables. Then using this toolkit, Jennes, in his adversarial study [2], focused on attacks against these packing detection tools and extended the set of binary alterations useful to evade static detection. Given that malware authors are perpetually refining their evasion tactics, the risk of these models being misled by deliberately altered binaries is high. This is especially true for systems relying on static features for detection, such as file structure and signatures, as these attributes are more straightforward to modify compared to dynamic features, which are analyzed during execution and include behavior patterns and system calls. Thus, incorporating an adversarial study into the initial design phase of detection models is essential for identifying and reinforcing potential weaknesses of the models.

NotPacked++ is developed to address this gap, by enhancing the adversarial landscape with a new tool, that can be used to test and assess static detection mechanisms. We introduce an easy-to-use and efficient tool based on recent adversarial studies in addition to our own adversarial research. This tool includes a wide range of efficient adversarial alterations to assess the vulnerability of static detection models and static detectors. It can later be integrated in the Packing Box framework and enhance the experimental toolkit, providing a more efficient tool for research and analysis. This tool serves as a vital tool for strengthening malware detection mechanisms, aligning with the pressing need for adaptive and robust cybersecurity defenses.

1.2 Objectives

This master thesis is a contribution to the Packing-Box framework and an introduction of a new adversarial tool. We implement a weaponized tool, *"**NotPacked++** "*, to alter packed PE executables in various ways that can fool packing detectors and evade detection while keeping the executable functional. The tool implements efficiently alteration and attack techniques existing in recent researches on packing detection, malware analysis and adversarial Machine Learning. We enhance the Packing-Box framework by optimizing existing alterations and introducing new ones, then we visualize their effect and study their impact on the detection of packed executables.

Our objectives are four-fold :

1. State the **background**.
   - A – Present general executables concept and fundamentals about the PE format and its structure.
   - B – Introduce the concept of runtime packing, including its detection and usage.
   - C – Provide an overview of static detection techniques.
   - D – Describe essential concepts for defining the adversarial setting of our work
2. Draw a **state-of-the-art**.
   - A – Give an overview of static packing detection field.
   - B – Provide a brief overview of adversarial and defense evasion research.
   - C – Present some known packers, detection tools and evasion tools.
3. Present the **framework**.
   - A – Define the adversarial setting of our research.
   - B – Give an overview of the Packing Box framework, its useful abstractions and building blocks.
   - C – Present our tool's development environment, including its architecture and how it works.
4. Conduct **experiments**.
   - A – Set the methodology used in our experiments.
   - B – Describe current alterations, validate their correctness and optimize them.
   - C – Show new alterations and study their impact on features and static packing detection.
   - D – Validate NotPacked++ by evaluating its speed performance and its impact on static detection.

1.3 Research Questions

Question 1	What is the state-of-the-art of alterations and adversarial tools ?
Question 2	How to optimize alterations and can we add new ones ?
Question 3	How to weaponize those alterations in an adversarial tool ?

1.4 Content

The following chapters are structured as follows :

- **Chapter 2 - Background** provides an explanation of the notions necessary to tackle the adversarial robustness of packing detection algorithms.
- **Chapter 3 - State of the art** summarizes the most relevant works from the literature concerning packing detection, malware analysis and adversarial tools.
- **Chapter 4 - Framework** describes the adversarial setting, presents the Packing Box framework and gives an overview of the development architecture of the NotPacked++ tool developed for this master thesis, including its implementation and usage.
- **Chapter 5 - Experiments** describes the methodology, details the existing set of alterations and their optimization, introduces new alterations, and studies attacks on static packing detectors using the NotPacked++ tool.
- **Conclusion** closes this document by presenting a general summary, parsing the achieved objectives, and providing ways ahead and ideas for future works.

1.5 Reading experience

This document is structured to be flexible and fast for readers, allowing to read targeted chapters or only going through summaries.

Already familiar with binary formats, binary analysis, runtime packing and static detection methods ?
Then you can safely skip the Chapter 2 as it presents basic definitions and overview of these topics. You can eventually get refreshed by reading the summary at the end of Chapter 2.

Already aware about the latest researches in the field of packing, static detection, adversarial ML and evasion tools ?
In this case, you can directly jump into the summary at the end of the Chapter 3.

Get started with the framework, the alterations and the tool's architecture ?
You can then hop to Chapter 4 to read about the Packing Box toolkit and building blocks used for our alterations.

Do you want to focus on experimental results ?
Then you can jump to Chapter 5 and review our experiments, observations and results.

2 Background

THIS chapter provides an essential understanding of executables and packing detection.

It starts by explaining the basic concepts of executables and then goes into detail about the Portable Executable (PE) format and its structure. Next, it introduces runtime packing and how it is commonly used in malware. Finally, the chapter covers static detection techniques, which analyze executables without running them.

Objectives

A – Present general executables concept and fundamentals about the PE format and its structure.

B – Introduce the concept of runtime packing, including its detection and usage.

C – Provide an overview of static detection techniques.

D – Describe essential concepts for defining the adversarial setting of our work

2.1 Executables

Executable files are binary programs designed to run on a computer system. These files can serve benign purposes, such as running a browser software, or malicious purposes, in which case they are commonly referred to as "Malware" (short for malicious software). Various format exist depending on the target Operating System, most common ones are Portable Executable (PE) for Windows, Executable and Linkable Format (ELF) for Linux, and Mach Object (Mach-O) for OSX. They all have different and unique architectures and contain all the execution requirements necessary to execute them on the target host. In this section we present some fundamental concepts about the most popular formats.

In this study, we will focus on the Portable Executable (PE) format, the main format for Windows, the most popular operating system. This format has the largest number of applications, offering a wide range of potential targets for analysis. Additionally, the PE format has the most resources available, including tools and research, making it an ideal choice for our examination

2.1.1 Generalities

An executable represents a compiled program in a binary format that is ready for execution. This binary format, specific to an operating system, encapsulates the necessary instructions in machine code for the Central Processing Unit (CPU) and data for the program's operation. When a user or system initiates an executable, the Operating System proceeds to allocate memory space, creating a process structure that serves as the program's operational context, loading the executable's binary content from disk storage into this allocated memory in the Random Access Memory (RAM). Then the Operating System (OS) hands over control to the loaded executable by giving to the CPU the position of the first instruction to execute. This address is called the **Entry Point** (EP).

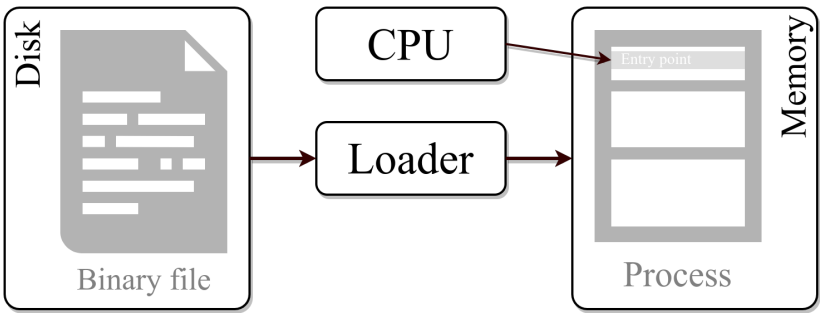


Figure 2.1: An executable binary file loaded into memory [1]

When stored on disk, the executable is kept as small as possible for storage and transportation, and when loaded into memory it expands to meet execution requirements of the system. This process involves allocating memory for the program's code, data, and stack segments, Dynamic Link Libraries (DLLs). The

transitioning from a compact static state to a fully operational state optimized for execution means that the layout of the binary file on disk and the layout in memory are quite different, making addressing a complex task. Therefore, understanding addressing terminology becomes essential for correctness of the program's execution and memory mapping.

Furthermore, the process of mapping the program into memory requires the executable to be in a specific format recognizable by the loader. Despite their differences, many executable file formats have a set of shared characteristics, layouts, and concepts crucial for this process. These shared elements are outlined as follows:

- **Header**: The header holds essential metadata about the executable required to be loaded and executed by the kernel, such as the program's Entry Point (EP), the type of machine it's designed to run on. In more recent executable formats, the header is often subdivided into several parts, including the main header, section headers, and optional headers.
- **Sections**: Sections divide the executable into logical parts, each holding different types of data such as code (*.text*), initialized data (*.data*), uninitialized data (*.bss*), and resources (*.rsrc*), structured for efficient loading and execution. It is often prefixed by a dot (e.g. : *.text*). Most formats use a Sections Table for referencing the start addresses and sizes of the sections.
- **Relocation**: Relocation involves adjusting addresses within the executable, ensuring that when the program is loaded into memory, references to variables and functions point to the correct locations. In many formats, this is managed through a Relocation Table.
- **Strings**: Strings are sequences of characters terminated by a null byte and used in the program, for example in error messages or interface text.
- **Imports**: The import table is used for identifying external dependencies, listing the functions and libraries required by an executable for its execution. This allows the loader to link these external resources at runtime, ensuring the executable has access to the necessary external functionalities. The export table is a similar concept but listing functions and variables that the executable makes available to other programs or libraries.
- Addressing:
  - **Physical address**: This is an address pointing to a position in the file at rest (on disk).
  - **Virtual Address (VA)**: This is an address in an executable, after it has been loaded into memory. It contains the image base and points to the absolute position in the virtual address space of a process, it is then mapped by the OS to a physical memory location in RAM when the program is executed.
  - **Relative Virtual Address (RVA)**: This is an address of an item after it was loaded into the memory, relative to the image base. Meaning it is a Virtual Address (VA) where we subtracted the image base address.

A Virtual Address (VA) is not as predictable as a Relative Virtual Address (RVA) because the loader might not load the executable image at its preferred location. The preferred location, known as the image base, is the default address where the executable expects to be loaded. However, due to address space layout randomization (ASLR) or conflicts with other loaded modules, the loader may choose a different address, making the actual VA less predictable. In contrast, an RVA remains consistent as it is relative to the image base, providing a stable layout independent address.

2.1.2 PE Format

The Portable Executable (PE) format, created by Microsoft [8], is the most used file format for executables (*.exe*) and DLLs (*.dll*). It is recognizable by its magic bytes ***0x00004550*** (*"PE\0\0"*) placed in the header. The PE file is designed to encapsulate all essential details needed for the Windows operating system to execute the program. In Figure 2.2, we show the generic structure of a binary.

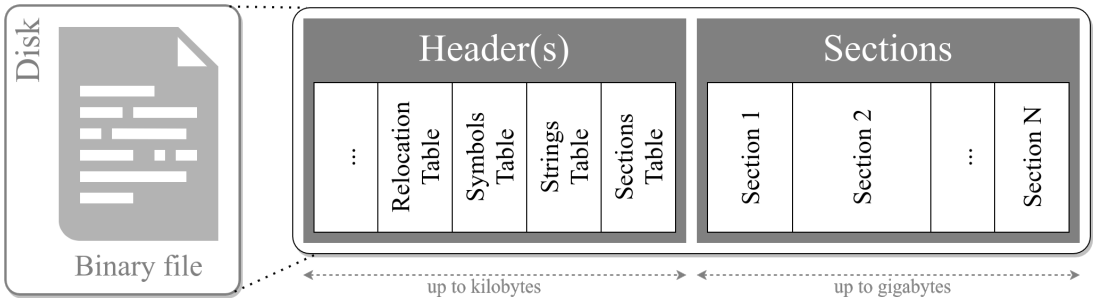


Figure 2.2: Basic structure of a binary file [1]

The PE file consists of two main components, as shown in Figure 2.3: the Header part and the Sections containing the code and data. It can also sometimes have additional data, called the overlay, and appended at the end of the file. It is not loaded in memory and is not part of the PE format structure.

Figure 2.3: Basic structure of a PE file [1]

Note that we only give an overview of the PE format and its key components to ensure a good understanding of subsequent chapters. For more details about the PE format, the interested reader can refer to the official Microsoft documentation [8].

PE Headers

- **DOS header** : This header preserves compatibility with the older MS-DOS executable format historically used on Windows. It contains a header starting with the magic bytes **0x5A4D** (*"MZ"*), followed by a "stub" which is an executable code that simply displays the message "*This program cannot be run in DOS mode*".
- **PE header** / NT header : This header starts with a signature or a magic bytes **0x00004550** (*"PE\0\0"*), and is composed by the following headers:
  - Common Object File Format (COFF) header (aka File header)
  - Optional header
- **Section table** : This contains the section header of each sections available in the PE file.

Table 2.1 presents the COFF header fields and a brief description of each fields.

Name	Description
Machine	Machine architecture to run on (Intel, AMD, ...)
NumberOfSections	Number of sections, size of the sections table
TimeDateStamp	A timestamp of the creation of the file
PointerToSymbolTable	File pointer to the table of symbols, if any
NumberOfSymbols	Number of entries in the symbol table
SizeOfOptionalHeader	Size of the optional header
Characteristics	Characteristics of the image

Table 2.1: COFF header fields

Table 2.2 highlights most important fields of the Optional Header.

Name	Description
Magic	Magic bytes to specify if the executable file is 32 bit or 64 bit
AddressOfEntryPoint	Address of the first instruction to start execution
ImageBase	Tells the windows loader the preferred loading address of the PE file in memory (default: `0x00400000` and `0x10000000` for DLLs)
SectionAlignment	The alignment of sections data when loaded in memory, equal to or greater than *FileAlignment* (default: the architecture's page size, `0x1000`)
SizeOfImage	Size of the entire image when loaded in memory, including all headers (must be a multiple of *SectionAlignment*)
SizeOfCode	The size of the code sections (*.text*), or the sum of all code sections if there are multiple sections.
BaseOfCode	The virtual address of the *beginning-of-code* section
SizeOfInitializedData	The size of the initialized data sections (*.data*)
SizeOfUninitializedData	The size of the uninitialized data sections (*.bss*)
SizeOfStackReserve	The space to reserve in memory for the stack
CheckSum	Checksum for file integrity

Table 2.2: Optional header fields

The section table is positioned right after the optional header, because there is no pointer to it. In the section table, each entry has a section header, and as shown in Table 2.3, each section header (section table entry) has a well-defined format.

Name	Description
Name	Is a string of 8 characters maximum, padded with null bytes. For longer names, the name starts a slash (/) and followed by the ASCII representation of a decimal number that is the offset of the long name in the string table. (e.g. "/42")
VirtualSize	This is the actual size of the section's data in memory
VirtualAddress	This is the address of the first byte of the section loaded in memory
SizeOfRawData	The size of the section in the file on the disk.
PointerToRawData	This is a file pointer to the section
PointerToRelocations	This is a file pointer to the beginning of the relocation entries for the section
PointerToLineNumbers	This is a file pointer to the beginning of the line-number entries for the section
NumberOfRelocations	The number of relocation entries for the section
NumberOfLinenumbers	Number of line-number entries for the section
Characteristics	This flag describes the characteristics of the section (readable-writable-executable, ..)

Table 2.3: Section header fields

PE Sections

Although section names within the PE format can be arbitrarily assigned without affecting the executable's functionality, there are standard section names predefined and each serves specific purposes. Typically, in most common scenarios, the name of a section is in itself a hint about its type and its content, which guides us in understanding the role and nature of the data it holds. Table 2.4 details the most common standard section names and their intended roles, giving us a good understanding of a typical structure of a PE file.

Name	Description	Comments
.text	Contains the executable code	Typically includes the program's entry point (EP).
.data	The initialized data	Holds global static variables defined with initial values
.bss	The uninitialized data	Holds reserved space for variables not initialized
.rdata	The read-only initialized data	Contains global constants and other read-only data
.edata	The export table	Lists functions that other programs can use (DLL only)
.idata	The import table	Lists functions the executable or DLL imports from other DLLs
.reloc	The relocation table	Provides a set of instructions to adjust addresses in memory, allowing the executable to be loaded at different memory addresses than its preferred ImageBase value. (only in executables)
.rsrc	The resources	Includes non-executable data like icons, images, sounds, ...

.tls	Thread Local Storage (TLS)	Contains data that is unique to each thread, facilitating thread-specific data management
.debug	Debugging information	Holds data for debugging, such as symbols and line numbers to map back with the original source code (should not be present in release version)

Table 2.4: Standard section names and purpose

2.2 Runtime Packing

A formal definition has been formulated in the work of D'Hondt [1] as follows: *"Packing, also sometimes referred to as runtime packing, is a set of transformations, applying to all or part of the sections of a target binary file, that modify its layout at rest while preserving its working at runtime"* . This includes compression, encryption and any transformation preserving its behavior. It is widely used to reduce the size of software for storage and transmission efficiencies, as well as to protect intellectual property and counteract reverse engineering efforts. Unfortunately, most malware authors also use packing to hide their code in order to bypass any static detection in place or prevent from reverse engineering.

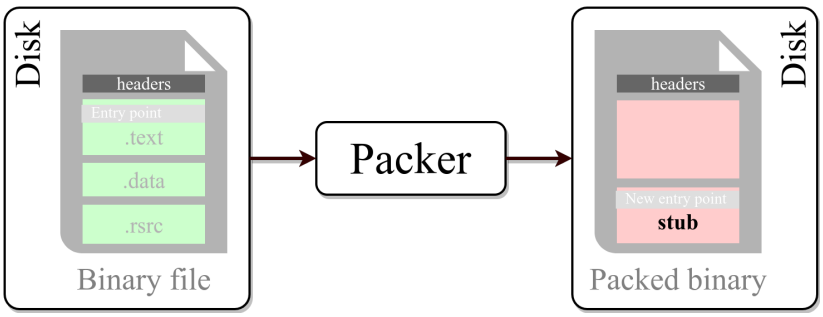


Figure 2.4: Simplified packing process [1]

In order to reverse the transformation applied during the packing process, a packed executable should embed a small piece of code, called the "stub", to unpack itself into its original form. This unpacking occurs seamlessly at runtime, ensuring the original code is restored in memory for execution, as depicted in Figure 2.5. Note that this unpacking process operates transparently, with no visible signs to the user. This ensures the application launches and runs smoothly without any noticeable delays or disruptions in its functionality.

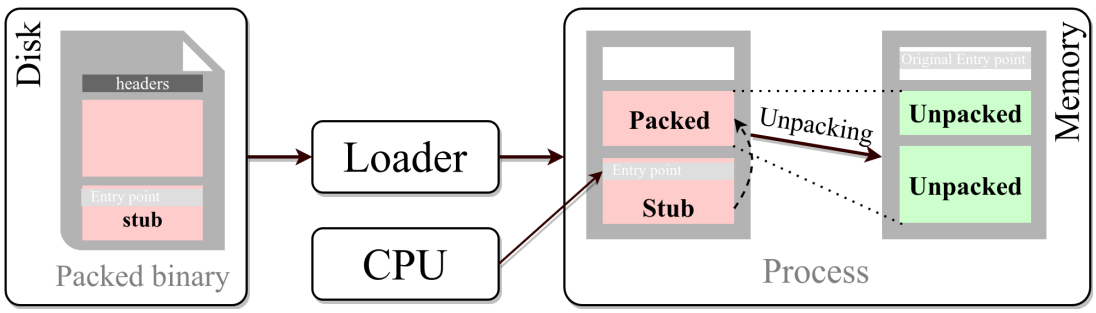


Figure 2.5: Simplified unpacking process [1]

During execution, the packed binary initiates at its altered Entry Point (EP) and begins the unpacking routine. It then reconstructs the layout of the original program in memory. It unpacks the content to a new writable section, then the execution control is transferred to the Original Entry Point (OEP).

2.2.1 Packing Operations

Packing is frequently reduced to compression techniques but it includes a broader range of operations. We state in Table 2.5 multiple operations used in packing as it has been highlighted by D'Hondt [1].

Name	Description	Comments
Compression	Reduces the binary's size	Probably the most common operation, many packers use classical compression algorithms such as LZ77, LZMA or even Deflate. It can be full or partial.
Encryption	Encrypts all or a part of a binary	It requires a key and generally uses symmetric encryption (e.g. AES), as it is lighter to embed in the stub. This makes the binary's data look random and thus increases the entropy. However, using the XOR encryption scheme with a large enough key could keep the entropy low.
Obfuscation	Makes the code unreadable	I.e. by masking API calls that may trigger detection rules or by deleting the import table in order to make reverse engineering more complex.
Virtualization	Embeds a Virtual Machine (VM) that executes code	It requires to embed a VM along with the original binary and may completely change its layout, possibly in a different architecture and instruction set than the host machine running it, making reverse engineering efforts tedious.
Anti-reversing tricks	Avoid the detection tools of reverse engineers	It includes anti-debugging, anti-virtualization, and anti-hooking techniques.

Table 2.5: List of operations for the packing process

2.2.2 Common Usage

Packing is widely used for various reasons, both positive and negative. Some packers can combine multiple operations, e.g. *Themida*, which combines compression and encryption of the executable file but also virtualization and anti-debugging mechanisms, making it a good candidate packer for protecting software against reverse engineering and piracy attempts.

We can distinguish three main categories of usage of packing :

- **Size reduction** : In order to reduce the file size and thus the transmission time, this compression technique is widely used by legitimate and malicious authors (e.g. UPX, MPRESS, ...).
- **Software license management & piracy prevention** : In order to protect intellectual property and prevent pirates from reversing a paid or protected software, developers tend to use packers that include encryption, virtualization or anti-debugging tricks (e.g. Enigma Protector, Themida, ...).
- **Hide malicious software** : In order to hide its malicious intent, malware authors often use packers. This can either be known packers (e.g. UPX, ...) or unknown ones created for that specific purpose.

Malware

A malware, or malicious software, is a "*software that fulfils the deliberately harmful intent of an attacker*". This is the definition introduced by Moser et al. [9]. Many malwares use common packers to hide their malicious intent in order to bypass any static detection in place, or obstruct potential reverse engineering efforts.

MITRE ATT&CK framework [10], a detailed knowledge base that tracks adversary tactics and techniques used by threat actors, includes a detailed list [11] of malware that used the "Software Packing" technique (T1027.002) [11] to conceal their malicious intent, some of which are detailed in Table 2.6.

Name	Description
2016 Ukraine Electric Power Attack	During the 2016 Ukraine Electric Power Attack, Sandworm Team used **UPX** to pack a copy of Mimikatz [12].
APT29	They used **UPX** to pack their files
Daserf	A version of Daserf backdoor uses the **MPRESS** packer.
H1N1	H1N1 uses a custom packing algorithm
Operation Dream Job	During their campaign "Operation Dream Job", the Lazarus Group packed malicious `.db` files with **Themida** to evade detection

Table 2.6: Example of malware using packing [11]

According to Brosch and Morgenstern [13] (2006) and a report published in 2009 by McAfee [14], over 80% of computer malware appear to be using packing techniques. The majority of packed samples tested by Brosch and Morgenstern in 2006 were packed with the famous, publicly available packer *UPX*.

In 2014, in the work of Ugarte-Pedrero et al. [15], authors found out that 47% of the 13,173 malware samples collected were identified by PEiD [16] [17] as packed by known packers, and the remaining 53% were not detected by PEiD but could be using a custom packer.

In addition to the usage of known packers, some threat actors used a custom packer that targets a specific heuristics. We can observe some malware using a low entropy packing technique, such as the Zeus botnet (Zbot), which can evade static detection that relies on entropy based heuristics.

2.3 Static Detection

This section provides an essential understanding of the methodologies used in static detection of malware, setting the stage for discussing evasion techniques in subsequent sections.
Note that we only focus on static detection, therefore dynamic detection will not be detailed.

Static detection refers to the process of analyzing executable files and identifying potentially malicious behavior without executing the binary. This method relies on examining the raw binaries to find indicators of malware or suspicious patterns. Dynamic detection refers to analyzing the behavior of an executable file by executing it in a virtual and controlled environment. Static detection techniques are essential for early detection of malware and packing, providing a first line of defense against malicious executables. One of its main advantage is that it requires less computational power than a dynamic detection.

This section explores two primary techniques used in static detection: signature-based and heuristics-based detection.

2.3.1 Signature-based Detection

Signature-based detection involves scanning executable files for specific sequences of bytes or patterns, known as signatures, which are unique to certain types of packers, malware, or malicious behaviors. These signatures can consist of byte sequences, recognizable code patterns, or even hashes of known malware. Antivirus software contains a database of signatures, against which it compares segments of the executable file. If a match is found, the file is flagged as potentially malicious.

The main advantage of signature-based detection is its simplicity and effectiveness in detecting known malware or known packers and it requires relatively low computational resources compared to more complex detection methods. However it is ineffective against new, unknown malware or variants of existing malware that have been altered through techniques such as obfuscation, encryption, or packing.

Those signatures are derived from analysis of known malware samples. This process involves disassembling or reverse engineering malware to identify unique code sequences or data that can serve as reliable indicators of malicious intent. For the definition of those signatures, the famous YARA tool is used in most anti-viruses and detection tools, it is a pattern/signature matching swiss knife for malware researchers. It is used to define signature rules and detect matching signature in a binary file. An example of a YARA rule definition is given in Figure 2.6, this rule defines the signature for the detection UPX packed binary files.

```
rule upx {
    meta:
        description = "UPX packed file"
        block = false
        quarantine = false
    strings:
        $mz = "MZ"
        $upx1 = {55505830000000}
        $upx2 = {55505831000000}
        $upx_sig = "UPX!"
    condition:
        $mz at 0 and $upx1 in (0..1024) and $upx2 in (0..1024) and $upx_sig in (0..1024)
}
```

Figure 2.6: Example of a YARA rule definition for UPX signature detection

2.3.2 Heuristics-based Detection

Heuristics-based detection is a method used to identify malware by looking at suspicious features, behaviors and structures in files, instead of searching for known malware signatures. For example, a heuristic rule might detect malware by identifying executables with unusually high entropy, which is often indicative of packing or obfuscation techniques. This approach includes simple heuristics, which identify general signs of malware, and ad hoc heuristics, which target specific types of malware or threats. By using both types of heuristics, this method improves the ability to detect different kinds of malicious executables, making it a strong addition to signature-based detection. In recent researches, those extracted features, behaviors and structures of malwares are often used as training data for Machine Learning (ML) models.

Ad hoc heuristics These are more precise techniques developed to target the detection of specific malware families or tactics, by extracting features commonly found in a particular malware family. They are crafted after analyzing specific malware samples or attack vectors, allowing for the detection of nuances that simple heuristics might miss. For example, identifying a unique sequence of operations or behavior that is characteristic of a known malware family is an application of ad hoc heuristics.

Simple heuristics This kind of analysis involves the application of rules to identify suspicious characteristics or behavior patterns that are commonly associated with malware. Unlike signature-based detection or ad-hoc heuristics, it does not rely on known malware samples but on the features or actions typically seen in malwares. Static heuristic analysis might involve disassembling the code to look for suspicious instruction sequences, such as those commonly used in exploits, or the presence of code typically associated with unpacking routines. Additionally, this analysis method involves detecting structural irregularities within executables that could indicate a malicious intent.

In their work, Arora et al. [18], defined and classified heuristics into 5 subcategories for PE files, this classification is shown in Figure 2.7.

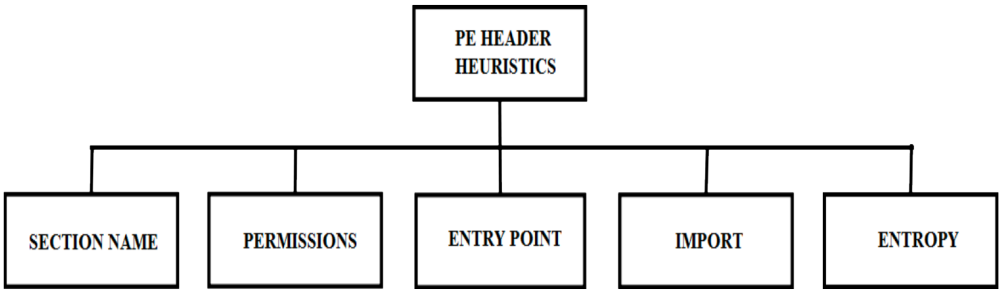


Figure 2.7: PE Heuristics classification [18]

Many static packing detectors uses the entropy of the executable or a specific portion of it to distinguish packed from non-packed executables. The entropy often refers to the Shannon entropy, which is a formula that measures the randomness of the information given or how unpredictable the information is. This formula is typically defined as follows:

$$H(x) = -\sum_{i=1}^{N} p(x_i) \cdot \log_2 p(x_i) \tag{2.1}$$

Where $H(X)$ is the entropy, N the number of outcomes and $p(x_i)$ is the probability of the occurrence of a byte in that particular portion of the file. A high entropy value reflects more unpredictable data and indicates that it is probably encrypted.

The heuristic approach is more adaptable than signature-based detection, because it is capable of identifying new and evolving threats without requiring prior knowledge of specific malware samples. However this can also lead to higher false positives, as legitimate software may also contain features or characteristics that are considered suspicious. This also requires more computational power than the signature-based detection, as we need to compute or extract those feature from the executable file.

Our goal in this work is to evade static packing detection by targeting heuristics through the alteration of packed binary files, to impact their extracted features while keeping the executable functional.

2.4 Adversarial Analysis

Adversarial analysis explores how attackers can manipulate inputs to deceive machine learning models and other detection mechanisms. These adversarial inputs cause models to make incorrect predictions. This section outlines the essential concepts needed to grasp the adversarial context in which our work operates.

2.4.1 Attack Space

There exist two primary approaches to study evasion techniques in packing and malware detection : perturbation in the feature space and alteration in the problem space.

In the **feature space**, researchers perturb the features extracted from executables to study the impact and evade detection. Many studies focused on the feature space [19] [20] [21]. While feature perturbation can reveal vulnerabilities in detection models, it has a significant drawback: if an evasion is successful, it is sometimes impossible to trace back and produce a functional executable that outputs those manipulated feature values. This means that successful evasions in the feature space do not necessarily translate to real-world executable files that can run without errors.

On the other hand, the **problem space** involves altering the executable itself rather than its features. By directly targeting and modifying executables, we can study the impact on features and detection while ensuring there exist such functional executable that evade detection. This approach provides a more realistic and practical evaluation of detection models, as it guarantees that the modified executables are both operational and capable of evading detection. This ensures that the models and detectors are tested under realistic conditions.

Figure 2.8 illustrates the distinction between both approaches.

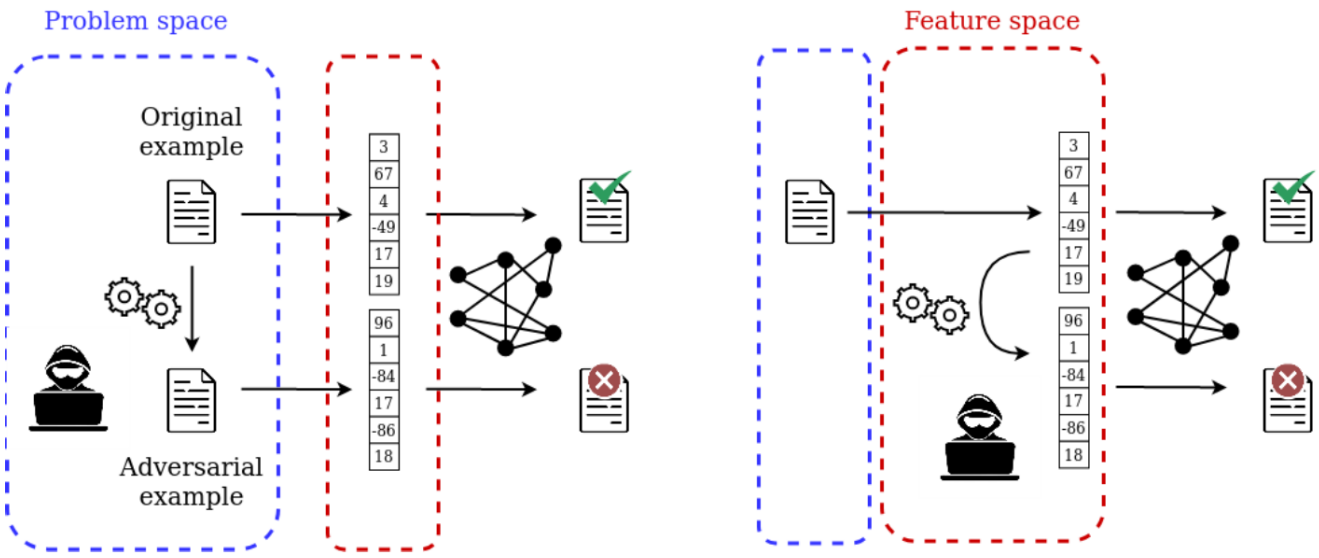


Figure 2.8: Problem space and feature space illustration [2]

In this diagram, the left side represents the problem space, where adversarial samples are created by directly altering an original executable. The extracted features of the adversarial samples are then tested against detection models to evaluate their effectiveness. The right side represents the feature space, where features are first extracted from original samples and then features are perturbed, and the resulting adversarial features are used to test the detection models. The key difference is that in the problem space, the altered executable remains functional, whereas in the feature space, the focus is solely on feature manipulation, which may not translate to a real-world functional executable. This challenge of inverse feature-mapping problem, has been studied in detail in the work of Pierazzi et al. [22].

2.4.2 Threat Model

The threat model of Carlini et al. [23] is a widely accepted framework in the literature that categorizes adversarial attacks based on the attacker's knowledge, capabilities, and goals:

Attacker's Knowledge: This refers to the amount of information the attacker has about the model. There exist two main adversarial settings for attacking ML models:

- **White-Box:** In this setting, the attacker has full knowledge of the detection model, including its architecture, parameters, and features. This allows the attacker to craft precise alterations to evade detection.
- **Black-Box:** Here, the attacker has no access to the internal functioning of the model and can only observe its output. The attacker relies on querying the model to gather information about its behavior and adapt their alterations accordingly.

Attacker's Capabilities: This describes what the attacker can do, often limited by the space in which perturbations can occur. In realistic scenarios, an attacker's ability to modify examples is typically constrained. For example, the attacker could have a restriction on the number of queries allowed. We distinguish two main categories of actions, that have been described in details in section 2.4.1 :

- **Feature Space attacks:** These attacks focuses on perturbing extracted features to evade detection. However, the modified features does not necessarily translate back into a real functioning executable.
- **Problem Space attacks:** These attacks involves transforming a binary file and indirectly impact the features used for detection. This ensures that such executable that evade detection while maintaining its functioning exists.

Attacker's Goals: This defines the attacker's objective when launching an adversarial attack. We distinguish two main objectives for an adversary :

- **Targeted attacks:** In a targeted attack, the goal is to produce a specific misclassification output. It increases the model's confidence in the prediction of the target class.
- **Untargeted attacks:** In these attacks the goal is to simply cause misclassification, thus any input being misclassified to any other class represents a successful attack. It reduces the model's confidence in the prediction of the correct class.

SUMMARY

Executables are binary programs that contains the necessary instructions in machine code to run on computer systems. They can be benign, like browser software, or malicious, known as malware. In this study, we focus on the PE format which is the main format for Windows, the most popular operating system. The PE format contain all components necessary for execution on the target host and is split into two main parts: the header part, with all metadata and the section part, with the program's code and data.

Runtime Packing is a set of transformation applied to a binary file that modifies its layout at rest while preserving its functionality at runtime. When the file is executed, it first unpacks itself into its original form before transferring control to the original executable. Different packing techniques exist, such as encryption, virtualization, and obfuscation. Packing is quite useful to protect intellectual property, reduce the size of an executable and prevent reverse engineering of legitimate software. Unfortunately, most malware authors also uses packing to hide their code in order to bypass static detection. Multiple report shows that over 80% of malware uses a form of packing technique, and many of those use known packers such as UPX.

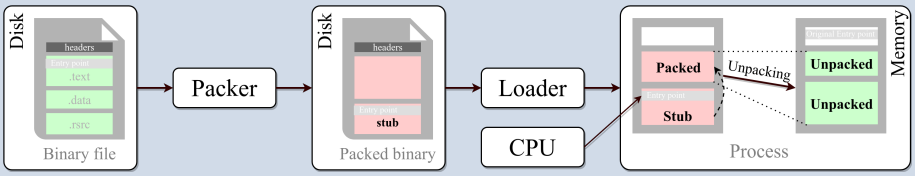


Figure 2.9: Packing and unpacking a binary file [1]

Static Detection consists of two main techniques: signature-based detection and heuristic-based detection. Signature-based detection relies on existing malware samples and searches for similar patterns. In contrast, heuristic-based detection generalizes by extracting structural or behavioral static features that are commonly found in malware or packed software. An example of simple heuristics used to distinguish packed from non-packed executables, is the Shannon entropy of the executable or a specific section of the file. Those heuristics of PE files have been classified by Arora et al. [18] into 5 distinct categories: *Section names*, *Permissions*, *Entry Point*, *Import*, *Entropy*.

Adversarial Analysis defines how attackers can manipulate inputs to evade detection. This section defines the attack space and threat model used in our work using the Carlini et al. [23] threat model framework. It categorizes adversarial attacks based on the attacker's knowledge, capabilities, and goals.

- **Knowledge**: In a **White-box** scenario, the attacker has a full knowledge of the model's internal configuration, features and parameters. In a **Black-box** scenario, the attacker only relies on input-output observations by querying the model.
- **Capabilities**: Two categories of actions defines the attacker's capabilities. Actions in the **Feature space**, where the attacker directly manipulates the internal features that the model uses to make predictions, and actions in the **Problem space**, where the attacker alters the executable directly.
- **Goals**: Two main objectives are defined. The first is to simply produce a misclassification of any sort, which is known as **Untargeted Attacks**. The second objective is to produce a specific, incorrect classification, known as a **Targeted Attack**.

3 State-of-the-art

THIS chapter explores the latest advancements in the field of packing detection and evasion.

It begins with an overview of the static packing detection field, outlining current methods and approaches. It then provides a brief overview of research on adversarial defense evasion tactics. The chapter also presents some well-known packers, detection tools, and evasion tools, highlighting the tools and techniques used in both attack and defense scenarios.

Objectives

A – Give an overview of static packing detection field.

B – Provide a brief overview of adversarial and defense evasion research.

C – Present some known packers, detection tools and evasion tools.

3.1 Static Packing Detection

The problem of packing detection has been a subject of many research due to its critical importance in the domain of malware analysis. Malware authors frequently use packing as a technique to obfuscate their malicious software's code, and evade static detection by antivirus programs. This problem has been discussed in many researches [24] [25], such as in the work of *Biondi et al.* [3], or in the BlackHat presentation *"Runtime Packers: The Hidden Problem?"* [13].

For identifying packed malware, many effective methods have been developed from basic signature matching to advanced machine learning algorithms. Initially, the primary method for identifying packed malware was through signature matching. This technique, while effective against known threats, proved inadequate against new threats or known malware obfuscated. As a response, heuristic-based approaches were introduced, extracting typical characteristics to predict unknown or modified packers. More recently, the focus shifted towards machine learning methods, which adapt more dynamically to emerging packing techniques by learning from large datasets of benign and malicious executables.

Hereafter, we delve into most notable researches used for packing detection, starting from signature-based detection, following with heuristic-based analysis, then moving to machine learning approach.

3.1.1 Signature-Based Approaches

Signature-based detection has been one of the earliest technique used for packing detection. This approach involves creating unique identifiers or signatures for known packing algorithms by analyzing specific patterns or sequences in binary files. These signatures are then stored in a database and used to scan and identify similar patterns in other executables. This method relies on matching these predefined signatures with observed data, which is fast for the identification of known packers but it requires regular updates to stay efficient with new packing methods.

2011	The New Signature Generation Method Based on an Unpacking Algorithm and Procedure for a Packer Detection [26]	Shin et al.
2012	SPADE: Signature based PAcker DEtection [27]	Naval et al.
2017	Packer identification based on metadata signature [28]	Hai et al.

Table 3.1: Signature based detection

Shin et al. [26] proposed a method for packing detection where they generate signatures based on unpacking algorithms. Each version of a packer (e.g. *UPX* version 3.94) uses the same unpacking routine to unpack the original executable into memory, meaning that all executables packed with it will share the same unpacking routine. This allows to generate reliable signatures that can effectively identify packed executables.

In their paper in 2012 [27], *Naval et al.* developed a method for automatically generating signatures for various packers and their different versions. Their technique involves comparing executables by pair, aligning byte sequences to identify common patterns, and byte sequences that appear in more than 50% of

samples are kept to create a signature. During the testing phase, a similarity score is computed between the signature and a test sample to define a threshold which will be used to determine if the sample is packed.

Hai et al. [28] introduce a metadata signature method based on a Control Flow Graphs generated by the BE-PUM [29] (Binary Emulator for PUshdown Model generation) tool. Their method uses the frequency vector of occurrences of classified obfuscation techniques. They categorized 14 obfuscation techniques into 6 groups, as listed below. On the resulting metadata signature, they identify packers via a chi-square test. Their experiments showed that this method was able to detect 607 of the 608 malware samples incorrectly identified by PEiD [16] [17].

1. **Code layout obfuscation**: *overlapping functions, overlapping blocks and code chunking.*
2. **Self-modification code (Dynamic code)**: *overwriting and packing/unpacking.*
3. **Instruction obfuscation**: *Indirect jump.*
4. **Anti-tracing**: *SEH (structural exception handler) and 2API (use of special APIs, LoadLibrary and GetProcAddress in kernel32.dll).*
5. **Arithmetic operation**: *Obfuscated constants and checksumming.*
6. **Anti-tampering**: *Timing check, anti-debugging, anti rewriting and hardware breakpoints.*

3.1.2 Entropy-Based Methods

Heuristic methods do not rely on signatures but instead analyze and extract various features of the executable to detect anomalies that may indicate packing. One particularly significant heuristic used for this purpose is the analysis of the entropy levels within the file's data. High entropy is often indicative of compression or encryption, common traits of packed executables. This heuristic-based approach allows for a more nuanced detection of packed files, especially when traditional methods fail to recognize new packing algorithms or heavily modified packed malware.

Entropy-based detection is a pivotal heuristic in identifying packed malware. This subsection will explore fundamental studies and key developments in entropy analysis applied to packing detection. By examining how various researchers have leveraged entropy to discern between benign and malicious executables, we can understand the effectiveness and limitations of this approach.

Entropy is the measure of randomness in some data, and is commonly used for identifying encrypted or obfuscated content. To quantify the entropy of data, we use the Shannon entropy formula given by the equation (2.1) in Chapter 2. High entropy, as calculated by this equation, suggests a higher level of unpredictability in the content of the file, which is a typical characteristic of encrypted or compressed data within a binary.

2007	Using Entropy Analysis to Find Encrypted and Packed Malware (Bintropy) [30]	Lyda and Hamrock
2009	Packed PE File Detection for Malware Forensics (REMINDer) [31]	Han et al.
2012	Entropy Score Analysis of Packed Executable (ESCAPE) [32]	Naval et al.
2016	Entropy analysis to classify unknown packing algorithms for malware detection [33]	Bat-Erdene et al.
2020	Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem [34]	Mantovani et al.

Table 3.2: Entropy only heuristics studies

Lyda and Hamrock's study [30] is one of the first to introduce entropy analysis to detect packed executables, they focused on the entropy of the entire file. They identified optimal threshold values that resulted in a detection accuracy of 99.99%. For instance, an entropy value falling in the range of 6.677 to 6.926 is indicative of a packed executable, and an encrypted file's entropy falls in a range between 7.174 and 7.177. They also introduced the **Bintropy** tool, for packing detection based on entropy, which later has been re-implemented and improved by D'Hondt in Python [35].

While initial research such as *Lyda and Hamrock*'s study [30] focused on classifying executables based on their average entropy, researchers quickly progressed to a more nuanced entropy analysis. This involved techniques using sliding windows or computing the entropy of individual sections within the executables. These approaches resulted in a more detailed and accurate detection of packing, and were frequently reported as highly effective in identifying packed malware. For instance, in 2009, *Han et al.* achieved significant results in their work [31], reporting a high accuracy by analyzing the entropy of the EP section instead of the average entropy of the entire file. They detected packed executables by calculating the entropy of the section containing the EP. If the entropy falls within the range of 6.85 to 7.99, the file is considered packed, as opposed to a normal PE file where the entropy should range from 4.55 to 6.82.

Naval et al. introduced "**ESCAPE**" [32], a method to quickly analyze packed executables using block-based entropy calculations. This technique cuts the hexadecimal disassembled code into blocks of 500 bytes and incrementally increased by 500B up to 20 blocks (500B, 1000B, 1500B, ...). Then it evaluates the combined entropy scores and average entropy per block, significantly reducing the processing time and effectively distinguishing between packed and not packed samples from a dataset of benign and malicious executables. They also identified which blocks were most critical for distinguishing between packed and not packed samples.

Bat-Erdene et al. [33] proposed an entropy-based approach to classify unknown packing algorithms used in malware. Their method involves scaling entropy values of executables, converting them into a symbolic representation using Symbolic Aggregate Approximation (SAX), and then applying machine learning algorithms, such as Naive Bayes and Support Vector Machines (SVM), to accurately classify packing algorithms.

In this paper [34], *Mantovani et al.* showed that the approximation *high entropy* $\approx$ *packed* is not correct in many cases. They highlighted a significant increase in low-entropy packing schemes that evade traditional high-entropy-based detection systems. They analyzed over 50,000 malware samples to show that over 30% uses low-entropy packing and suggested that relying only on entropy metrics is insufficient.

3.1.3 Other Heuristics

As suggested by *Mantovani et al.* [34], effective detection systems require more than just high-entropy based approaches to counter evasion techniques using low-entropy packing schemes. Many studies have focused on heuristic analysis, extracting and using relevant static features of packed PE files for detection.

2008	PE File Header Analysis-Based Packed PE File Detection Technique (PHAD) [36]	Choi et al.
2011	Structural Feature Based Anomaly Detection for Packed Executable Identification [37]	Ugarte-Pedrero et al.
2013	A Heuristics-based Static Analysis Approach for Detecting Packed PE Binaries [18]	Arora et al.
2015	A static heuristic approach to detecting malware targets [25]	Zakeri et al.
2022	Packer classification based on association rule mining [24]	Dam et al.

Table 3.3: Multiple heuristics-based detection research

Choi et al. [36] introduced a new technique called PHAD (PE Header Analysis-based Packed PE File Detection) to identify packed PE files. They extracted 8 specific unusual characteristics from the PE headers of executable files, and defined a Characteristic Vector with those distinct values. Then, PHAD calculates the Euclidean distance of the Characteristic Vector to distinguish between packed and non-packed files. The 8 characteristics chosen by authors are mainly related to section permissions and section names, they are listed in Table 3.4 along with other significant features of other researches.

Ugarte-Pedrero et al. [37] introduced an anomaly-based method where they first identify 125 raw header characteristics, extract 39 most significant ones based on the Information Gain (IG) of each characteristic and assigned a weight to each feature. Then they model each executable as a vector with values normalized between 0 and 1. The final step involves applying the calculated weights from Information Gain (IG) by multiplying them with each corresponding value in the normalized vector. Using this method, authors managed to obtain a high accuracy of 99% on unknown packed executable.

Arora et al.[18] employ a heuristic-based static analysis approach to detect packed PE binaries. They first define 14 features of the PE file, then organize those features into five categories — *Section Names, Permissions, Entry Point, Import Table, and Entropy* — as depicted in figure 2.7 in Chapter 2, then assigning to each feature a weight and risk. This approach involves analysing and extracting these features from the binary file, and then calculating a score to classify the executables as packed or unpacked based on a defined threshold. Using a dataset of 6755 samples, with 2078 packed and 4677 unpacked, this approach achieved a 99% detection rate.

In this paper [25], *Zakeri et al.* highlights that obfuscations techniques used by malware authors, including packing, creates anomalies in PE files. They extracts those anomalies from the PE file headers and other structural anomalies into 70 different heuristic features, after removing anomaly exceptions present in benign files. Then they compute the Information Gain (IG) and the frequency rate of each feature in malware, packed and benign samples to highlight most significant features. For the classification step, they use Machine Learning (ML) with a fuzzy classification algorithm called Fuzzy Unordered Rule Induction Algorithm (FURIA). From a dataset of over 63,000 file samples, labelled using PEiD and additional manual analysis, their experiment resulted in a high accuracy and a low false positive rate in the detection of packed files and malware.

Dam et al. [24] introduced an approach for packer classification using associative classification (AC) that combines association rules and classification. They tested different AC algorithms, including CBA, CMAR and PAM. They also introduced a new variation of CMAR and PAM algorithms called *Best Rule (BR)*. The effectiveness of these classifiers was evaluated on a large dataset of ~ 240000 samples, and resulted in a high accuracy.

In Table 3.4, the most significant features for packing detection found in the literature are listed. Features are sorted by category and are presented with a brief description along with the studies highlighting them as significant.

Category	Description	Paper
Entropy	The number of sections with high entropy	[25] [37] [18]
Entropy	Entry Point Section Entropy	[25] [37] [24] [3] [4]
Entropy	File entropy	[25] [37] [24] [3] [4]
Header	Number of sections	[25] [37] [24]
Header	Sum of sections raw sizes is bigger than file size	[37] [36] [25] [4]
Header	The number of sections with zero raw size	[25] [37] [3] [4]
Header	Section's raw data size per virtual size ratio	[37] [24] [4]
Header	Size of the code section	[37]

Header	File's Dll_characteristics value	[25] [4]
Header	Address of Entry point section (or not in Code section)	[37] [25]
Header	The number of resources	[25] [3] [4]
Header	No debug directory	[25] [3] [4]
Header	Checksum value	[25] [3] [4]
Header	Size of (un)initialized data	[25] [4]
Section name	The number of the section which name is not printable	[36] [37] [18] [25]
Section name	Entry point's section name non-standard	[37] [25] [4]
Section name	Non-standard section name	[37] [25] [18] [24] [3]
Section name	Presence of known packer section name	[18] [37]
Permission	The number of executable sections	[36] [37] [18] [3] [4]
Permission	The number of executable & writable sections	[36] [25] [37] [24] [3] [4]
Permission	The number of writable sections	[25] [3] [4]
Permission	The number of readable and executable sections	[37] [18] [4]
Permission	The number of section executable but not a code, or not executable but a code	[36] [25] [37] [18] [4]
Permission	If the section of entry point is not executable	[36] [25] [37] [4]
Permission	Entrypoint's section is not code	[36] [25] [37] [4]
Permission	Entrypoint's section is writable	[37]
Permission	No code section present (or Number of code section)	[36] [25] [37] [18] [4]
Import table	Number of import table entries (IAT's size)	[37] [18] [25] [24] [3] [4]
Import table	IAT address is in a non-standard section	[37] [18] [25]
Import table	The number of malicious API in IAT	[25] [24] [3] [4]
Import table	The ratio of malicious API calls to all API calls	[25] [24] [3] [4]

Table 3.4: List of most significant features

3.1.4 Machine Learning

A more recent technique used in the literature is Machine Learning based detection. It is a branch of artificial intelligence and involves developing algorithms that can process input data and use statistical analysis to predict an output based on previously learned patterns. Machine learning enables computers to perform tasks without explicit instructions by relying on patterns and inference instead. It involves training a model on a dataset to learn patterns and features that can then be used to make predictions or classifications on new, unseen data.

There exist two main categories in ML. The first is *supervised learning* where the dataset used for learning is labelled with the correct expected output given for each input. The goal of supervised learning is for the model to learn a mapping from inputs to outputs, to predict the output for new, unseen examples based on the learned relationships. The second one is *unsupervised learning* where the dataset is not labelled and the goal of the model is to learn from similarities in inputs to group similar patterns in a cluster for example.

2019	Effective, efficient, and robust packing detection and classification [3]	Biondi et al.
2021	Analysis of Machine Learning Approaches to Packing Detection [4]	Bertrand Van Ouytsel et al.
2023	Experimental Toolkit for Manipulating Executable Packing [5]	D'Hondt et al.

Table 3.5: Machine learning based detection

Biondi et al. [3] propose a machine learning-based method to enhance the detection and classification of packers in malware. They first extract 119 static features from the binary and categorize them into 6 groups (Table 3.6). They prioritize static features to lower the computational cost of feature extraction. Then they conduct various experiments using different ML algorithms and feature sets to balance between performance and feature extraction costs. The result of this work is a model that maintains a high accuracy while significantly reducing the computational time compared to existing methods in the literature.

Bertrand Van Ouytsel et al. [4] explored 11 different ML algorithms and 119 features in order to identify the most suitable models and most significant features for packing detection. They discovered that kNN, DT, MLP and SVM were the best algorithms in term of performance and accuracy, and the most significant features for packing detection were : *"the stack reserve, number of readable and writable sections, non-standard entry point section, raw and virtual data size of sections, byte entropy of .text sections, 12th entry point byte, and malicious API calls"* [4]. They also evaluated the long-term robustness of each model, identifying those 4 models as highly accurate over time by training on samples up to a certain date and testing on more recent samples.

D'Hondt et al. [5] introduce an experimental toolkit named "Packing Box" [7] designed to enhance the research and analysis of executable packing detection using a Machine Learning (ML) approach. It facilitates the entire ML workflow, including dataset preparation, model training and results visualization. It incorporates a variety of packers, static detectors, and analyzers. Also including executable alteration tool for adversarial learning studies. This open-source toolkit helps researchers to easily conduct studies within a single integrated environment, and enhances the accessibility and repeatability of research in this domain.

Name	Description
(BE) Byte Entropy	The Shannon entropy of different part of the binary or the entire file
(EB) Entry Bytes	Extract the first 64 bytes after the Entry Point
(IF) Import Functions	Features related to the import table, imported DLLs and functions (number of imported functions, ...)
(ME) Metadata	Features extracted from the PE header (checksum, baseAddress, ...)
(RE) Resource	Two features related to the resources (number of resources, and presence of debug directory)
(SC) Section	21 features related to sections (non-standard names, executable and writable section, ...)

Table 3.6: Categories of features from Biondi et al. *et al.* [3]

3.2 Adversarial Studies

As the domain of malware and packing detection overlap, the techniques developed for one are often applicable in the other. The objective within this overlapping problem space is to identify tools and techniques from malware detection that can be transposed to enhance packing detection strategies. Despite the critical importance of this area, many of the existing literature on adversarial attacks tends to focus on malware detection, researches specifically targeting packing detection remains relatively rare. This gap highlights the need for more focused adversarial studies to assess and improve detection mechanisms against sophisticated evasion techniques in both packing and malware detection domains, as this two domain overlap.

3.2.1 Adversarial Learning

Following the adoption of Machine Learning (ML) approaches to tackle the problem of packing detection, there has been an increasing need for adversarial research. Such studies are essential to evaluate the robustness and performance of these ML models against more sophisticated and up-to-date threats that continuously evolve. This ensures the model's effectiveness in a real-world scenario.

The goal of adversarial studies is to apply adversarial modifications to an executable to impact features and thus the detection mechanisms. In this context, it is necessary to understand the two primary types of adversarial attacks: black-box and white-box. Black-box attacks occur when the attacker has no knowledge of the underlying model's parameters, or features used during training. In white-box attacks the attacker knows those parameters and features used in training, and can thus target specific features or parameters to fool the detection.

2017	Black-Box Attacks against RNN based Malware Detection Algorithms [38]	Hu and Tan
2018	Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables [39]	Kolosnjaji et al.
2018	Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning [40]	Anderson et al.
2019	Exploring Adversarial Examples in Malware Detection [21]	Suciu et al.
2019	Explaining Vulnerabilities of Deep Learning to Adversarial Malware Binaries [20]	Demetrio et al.
2021	Functionality-preserving Black-box Optimization of Adversarial Windows Malware [41]	Demetrio et al.
2021	Adversarial EXEmples: A Survey and Experimental Evaluation of Practical Attacks on Machine Learning for Windows Malware Detection [19]	Demetrio et al.
2023	Certified Robustness of Static Deep Learning-based Malware Detectors against Patch and Append Attacks [42]	Gibert et al.

Table 3.7: Adversarial learning studies

The approach proposed by *Hu and Tan* [38] in 2017 consist of adding API calls in the Import Address Table (IAT) to evade detection of Recurrent Neural Network (RNN) based detection, without changing the functionality of the malware.

Kolosnjaji et al. [39] develop a gradient-based adversarial attack to evade detection and evaluate the robustness of deep learning models used for malware detection. They focus on manipulating only a few bytes

in the overlay at the end of malware samples, ensuring that the malware keeps its functionality while successfully evading the detection. Their results shows that this method can successfully bypass the targeted neural network, even with minimal modifications to the file.

Anderson et al. [40] present a black-box attack against machine learning malware detection models using Reinforcement Learning (RL). They develop a RL-based method where an agent learns to manipulate executable files through functionality-preserving transformations, to reach its ultimate goal of evading static detection. They published their development as an open-source project called "*gym-malware*" [43]. The agent operates in a completely black-box environment, and fine-tunes its manipulations of executable files through a trial-and-error process, learning from each interaction to improve its evasion techniques. This approach results in the generation of an adversarial modified malware samples that maintain its functionality and successfully bypassing machine learning malware detection models. The actions that can be performed by the agent on a malware sample, also called the action space, contains the following manipulation: append random bytes/strings/zeros , remove signature, remove debug information, change section names from a list/randomly, modify import/export API, break checksum and pack/unpack the file with UPX. An important notice highlighted by authors, is that files modified by LIEF [44], can have uncommon section names added like `.l1` and `.l2`. This can be a danger for adversarial training because the model learns the difference between "*modified by LIEF*" or not, instead of the intended classification. They also noticed that 8 out of 10 modified samples tested in a virtual machine executed successfully, highlighting the fact that even with functionality-preserving manipulation they still observe non-working output samples.

Suciu et al. [21] introduced a technique to evade malware detectors by altering the bytes within the slack space and appending bytes in the overlay at the end of a file. Their results shows that, the Fast Gradient Method (FGM) Append attack achieved a higher success rate, but it requires a much larger number of byte modifications to be successful.

Demetrio et al. [20] use a ML algorithm called "gradient descent" to generate bytes in order to modify DOS header bytes of the executables. They notice that header bytes are quite significant for classification but are easy to manipulate.

They later, in 2021, add new function-preserving alterations in their method [41]. Their approach focuses on functionality-preserving manipulations that inject benign content into malicious programs to evade detection while maintaining its original functionality. They define 8 structural file manipulation, and 5 behavioral manipulation maintaining the intended functionality of the executable. The study shows that these attacks can effectively evade detection by targeted models and commercial antivirus detectors.

The following list displays the 5 different behavioral manipulation defined by the authors:

1. **Packing** : Encrypts or encapsulate the binary inside another binary.
2. **Direct** : Rewrites specific portions of the code (e.g. replacing ADD x instructions with SUB $-x$)
3. **Minimal invasive** : Sets the entry-point to a new executable section that jumps back to the original code.
4. **Full translation** : Translates all the code to a higher representation then back to assembly language.
5. **Dropper** : Stores the code as a resource of another binary, which is then loaded at runtime.

In 2021, *Demetrio et al.* [19] propose 3 functionality-preserving manipulations, named *Full DOS*, *Extend* and *Shift*, which improves the probability of evasion and the amount of bytes manipulated in a white-box and a black-box attack settings. They also publish "*secml_malware*"[45] as an open-source framework that helps researchers to study white-box and black-box attacks. This framework includes the following attacks: DOS Header manipulation [20], Padding attack [39], GAMMA [41], FGSM padding + slack [21], Content shifting + DOS header extension [19], Header Fields manipulation [46].

Gibert et al. [42] introduce a certified robustness method for static deep learning-based malware detectors against patch and append attacks. Then they propose a chunk-based smoothing classification scheme,

where a classifier is trained to make predictions on separate chunks of a malware executable. This method maintains the model's accuracy even when parts of the malware have been altered, because, as they noticed, patch and append attacks can only modify a limited number of chunks. They assess the certified robustness of the proposed smoothed classier against five different evasion attacks : Slack+Padding [21], Full DOS [19], Content shifting [19], GAMMA [41] (black-box attack), Optimization of code caves [47] (black-box attack). A code cave attack, evolves using the unused space of a binary file to store malicious code to execute.

3.2.2 Attacks on Heuristics

Many studies focus on attacking Machine Learning (ML) based detection, but these models depend on an accurate ground truth, which is derived from a correctly labeled dataset. The labeling often relies on static detectors, which in turn rely on static heuristics.

However, only a few studies specifically targets these static heuristics. The researches presented below operate in the problem space, aiming to find binary transformations that significantly impact the most significant heuristics used by detectors while preserving the functionality of the binary. In contrast, feature space researches focuses on tweaking feature values to evade detection, but this approach does not ensure that the modified features correspond to a real, functional binary.

2021	Lost in the Loader: The Many Faces of the Windows PE File Format [46]	Nisi et al.
2023	Adversarial Learning on Static Detection Techniques for Executable Packing [2]	Jennes
2024	Evading Packing Detection: Breaking Heuristic-Based Static Detectors [6]	D'Hondt et al.

Table 3.8: Research attacking static heuristics

In their study [46], *Nisi et al.* present an analysis of different PE parser, they focus on the diverse ways Windows (XP, 7, and 10) and various software tools (e.g. ClamAV, Radare2 and Yara) handle the loading and parsing of the Portable Executable (PE) file format. They compare how each system processes malformed, transformed or non-standard PE files. Their analysis reveals that certain erroneous values in PE files are not checked and does not affect the operation of loaders, allowing these files to execute on target machines despite being malformed. To conduct this analysis, they create a framework that systematically tests and records the parsing behaviors of different loaders, including variations across different versions of Windows.

Jennes [2] first studied the effect of multiple basic binary manipulation, then defined more complex alterations based on those predefined building blocks. They targeted the most significant static features identified by Bertrand Van Ouytsel et al. [4], the ones that are often used to detect packed executables. They introduced 5 practical alterations to target those features, including adding 20 common API, adding a low entropy text section, filling section with zeros to match its virtual size, moving entry point to a new section with a standard name and renaming all non-standard sections names. This work studied attack on detectors but also on ML models.

In 2024, D'Hondt et al. [6] extended the Packing Box framework by defining, implementing, and evaluating five different alterations to study their effects on features and detection systems. They used the Packing Box toolkit to conduct their experiments and tested these alterations against a set of open-source packing detectors included in the framework. They showed that three out of the five alterations impacted most significant features defined in the literature [4]. This research demonstrates the effectiveness of these alterations in evading detection.

In Table 3.9, we summarize the functionality-preserving transformations found in the presented literature along with a brief description. The most observed transformation are alterations filling the slack space at the end of sections, renaming sections, injecting a new section and moving the entrypoint to a new location. We notice that none of the studies targets the section permissions, even-though it is listed in the most significant features of Bertrand Van Ouytsel et al. [4].

Name	Description	Paper
Filling Slack Space	Fills the slack space at the end of a section, with arbitrary unused data	[40] [21] [41] [2] [42] [6]
Rename sections	Modifies the names of sections within the PE file	[40] [41] [2] [42] [6]
Section Injection	Inserts a new section into the file by creating additional entries in the section table	[40] [41] [2] [42] [6]
Move entry point	Sets the entry-point to a new executable section (or slack space) and jumps back to the Original Entry Point (OEP)	[40] [41] [2] [42] [6]
Import Injection	Adds new import functions or library to the Import Address Table (IAT)	[40] [41] [2] [42] [6]
Padding	Adds additional bytes (in the overlay) at the end of the file, beyond its original content	[40] [39] [21] [41] [42]
Set checksum	Edits the original checksum in the file header	[40] [41] [2] [42]
Edit debug info	Changes debug metadata of the executable	[40] [41] [42]
DOS header edit	Modifies bytes in the DOS Header, which are unused by modern operating systems.	[20] [41] [42]
Content Shifting	Creates space before the beginning of a section by shifting content forward and injecting new content in the created space	[41] [19] [42]
Extend DOS Header	Injects content before the PE header, extending the DOS header.	[41] [42]
Edit signature	Removing/Adding the signature information of the file	[40]
Header edit	PE Header fields manipulation	[46]
Edit section permissions	Edit the section's permissions	NONE

Table 3.9: Functionality-preserving manipulations

3.3 Tools

This section presents some useful tools for binary analysis and packing detection, including common packers, static detectors, and adversarial detection evasion tools. We first review a few packing tools used to shield legitimate software and disguise malicious ones by employing various techniques including compression and obfuscation. Then we explore tools that analyze executable files without executing them, highlighting methods that detect malware and packing based on static analysis of the file's code and structure. Finally, we present tools and techniques employed to evade static detection.

3.3.1 Common Packers

Some of the packers listed below, are freely available and open source, this helps researchers to conduct efficient experiments, but it also allows attackers to use those to hide their malware and modify the packer to make a custom one from an open source base.

Name	Techniques	Comments
UPX	Compression	UPX is the most common packer, it is free and open-source.
ASPack	Compression, obfuscation	ASPack is used to protect against reverse engineering.
ASProtect	Encryption, anti-tampering	ASProtect is used to prevent disassembly by unauthorized sources.
Amber	Compression, obfuscation, encryption	Amber offers a combination of compression, obfuscation, and encryption to protect binaries from reverse engineering and analysis. It is open-source and designed to bypass AV.
VMProtect	Virtualization, code conversion	VMProtect prevents disassembly and reverse engineering.
ZProtect	Obfuscation, encryption	ZProtect provides protection against cracking and reverse engineering.
Themida	Virtualization, encryption, anti-reversing	Themida has been designed for software protection, against tampering and reverse engineering.
Armadillo	Encryption, compression	Armadillo is a commercial protector and licensing system.
MPRESS	Compression	MPRESS is a high-performance executable packer for PE32/PE32+ files, free and open-source.
NSPack	Compression, obfuscation	NSPack provides strong compression and protection features.
RLPack	Compression, obfuscation	RLPack compresses and protects executable files.
UPack	Compression	UPack is a packer for Windows executables, emphasizing size reduction.
MEW	Compression	MEW is an older executable packer known for high compression.
FSG	Compression	FSG (Fast Small Good) is a free packer designed to reduce the size of executable files.
Telock	Encryption, obfuscation	Telock is known for its complex obfuscation and anti-debugging techniques.
BoxedApp	Virtualization	BoxedApp is a commercial virtualization library, an executable packer, and a developer API to create custom packers. They create a new process and inject the payload into the memory of the remote process, this way the payload is never written on disk.

Table 3.10: Packing tools

3.3.2 Static Detectors

A common method for identifying packed executables not only utilizes custom signatures but also incorporates heuristic analysis. While the popular detector called PEiD [16] solely relies on signatures, another popular detector called Detect It Easy [48] uses heuristic rules in addition to signatures to enhance detection capabilities. Additionally, statistical methods are preferred in tools like BinStat and Bintropy, offering another layer of analysis by evaluating the numerical data patterns within binaries. Moreover, static detection tools are crucial for accurately creating labeled datasets for training Machine Learning models, an essential step as many modern ML approaches rely on these tools for accurate data labeling.

2007	**`Bintropy`** performs block-based entropy calculations to assess whether binaries might be packed or encrypted.	[35]
2008	**`PEid`** is a widely used tool that relies on signatures to identify packers and compilers in PE files.	[16]
2009	**`REMINDer`** offers detailed static analysis to detect anomalies in PE files, helping to unearth obfuscated code within binaries.	[49]
2010	**`BinStat`** utilizes statistical methods to analyze binary files, providing insights into their composition.	[50]
2014	**`Manalyze`** is an open-source tool that applies heuristic and signature checks to analyze PE files for security assessments.	[51]
2018	**`PyPacker-Detect`** is a Python-based tool that leverages heuristics to identify whether a PE file is packed.	[52]
2019	**`PEFrame`** helps in identifying potential malware and extract artifacts from PE files using static analysis.	[53]
2019	**`Unprotect`** is a python tool for parsing PE malware and extract evasion techniques.	[54]
2021	**`Detect It Easy`** (**`DIE`**) is a tool for determining file types and associated packers or compilers through heuristics and signatures.	[48]

Table 3.11: Static detector tools

Bintropy [35] is an analysis tool that calculates the block entropy of a binary file for estimating the likelihood that it contains compressed or encrypted data, which is indicative of packing or obfuscation. This tool is essential for initial scans in environments where high entropy is a predictor of potentially harmful software behavior. Lyda and Hamrock presented this tool in 2007 in the scope of their research [30].

PEiD [16] is the most common open source tool to detect PE file's packer and compiler. This signature based approach can currently detect PE files based on more than 600 different signatures. It provides OEP finder, crypto analyzer, generic unpacker, and it also provides function to inspect multiple files at the same time by downloading an entire directory.

In 2021 D'Hondt released a reimplementation of PEiD in Python [17] with the combination of multiple signatures databases, totalling more than 5.000 signatures. It extends the capabilities of the original tool, allowing researchers and analysts to use its signature-based detection within Python scripts and to find new signatures from datasets with a basic pattern similarity algorithm. Some other reimplementations of PEiD have been released, such as *"Yet another implementation of PEiD with yara"* [55].

REMINDer (REsponse tool for Malware INDication) [49] uses a simplistic algorithm to detect packed PE files. It first determines if the EP section is writable and then uses a threshold on the entropy of this EP section.

BinStat [50] applies statistical analysis to provide a comprehensive overview of a binary's structure, identifying statistical anomalies that may indicate packing, encryption, or other modifications intended to obfuscate its malicious intent.

Manalyze [51] is an open-source tool designed to improve the security analysis of PE files by applying heuristic and signature checks. It helps identify potential malware and extract artifacts from PE files using static analysis techniques. It includes a variety of plugins for performing checksum validations, detecting anomalies, analyzing headers, and checking for malicious behavior using both heuristics and signatures. It relies on PEiD's signatures DB, Yara rules and some heuristics. Manalyze is designed to assist security researchers in identifying malicious code and understanding the behavior of PE files.

PyPackerDetect [52] is a python tool utilizing heuristic analysis to identify whether a PE file is packed, analyzing features such as entropy and specific section characteristics commonly altered by packers. It's designed to be a lightweight tool that can be easily integrated into larger Python-based analysis workflows.

PEFrame [53] exposes hidden information in PE files to detect suspicious anomalies, malware, and extracts artifacts like URLs, IP addresses, and suspicious strings through static analysis. It is particularly useful for forensic analysts and malware researchers looking to quickly assess the threat level of unknown binaries. It can help malware researchers to detect packers, XOR encryption, digital signatures, mutex, anti debugging tricks, anti virtual machine, suspicious sections and functions, and more.

Unprotect [54] is a Python tool designed to analyze malware, focusing on parsing a PE file and extracting evasion techniques used within the malicious binaries. It provides a classification of evasion techniques and is a relevant resource for malware analysts to identify and understand the evasion tactics employed by malware to bypass detection. This tool includes a packing detection mechanism relying on PEiD detector, the entropy of section and the bytes after EP.

Detect It Easy, or *DIE*, [48] provides a versatile interface for file type detection along with detailed analysis of the content such as packers, compilers, and more using a mixture of heuristic and signature-based methods. It includes a plugin architecture that enables users to extend its functionality according to specific needs, making it highly customizable.

A few useful tools for packing detection but not directly qualified as static detectors can be spotted. We listed a few of them in the following timeline 3.12.

2010	**`Fast Universal Unpacker (FUU)`** aids in unpacking a wide range of packed files to facilitate deeper static analysis.	[56]
2020	**`RetDec`** is a retargetable machine-code decompiler that can decompile binary files into a higher-level language, useful in static analysis.	[57]
2023	**`IATelligence`** is a Python tool that will extract the IAT of a PE file and request GPT model [58] to get more information about the API and the ATT&CK [10] matrix related.	[59]

Table 3.12: Other useful tools for static analysis

Fast Universal Unpacker (FUU) [56] facilitates the extraction and analysis of packed files. It is a tool equipped with a set of plugins designed to unpack, decompress, and decrypt programs protected with popular packer software such as UPX, ASPack, FSG, and ACProtect. It is particularly useful in malware analysis, for its ability to handle multiple packing algorithms.

RetDec (Retargetable Decompiler) [57] translates machine code into a higher-level, human-readable C-like or Python language. It includes many features, such as packer detection, control flow graph generation, and call graphs. Its ability to decompile binaries into understandable code is crucial for static analysis, as it

allows researchers to examine a program's functionality clearly and comprehensively without running the actual binary.

IATelligence [59] is a Python tool that extracts the Import Address Table (IAT) from a PE file and uses OpenAI's GPT model [58] to provide details about each imported API entry in the binary. It also searches for related MITRE ATT&CK [10] techniques and explains how the API could potentially be used by attackers. This tool is useful for packing detection as it can identify unusual patterns or the presence of functions commonly associated with packing.

3.3.3 Evasion Tools

Multiple evasion tools have been developed to target ML-based and heuristic-based static detectors. Many of those focus on evasion of malware, but as the domain of packing is pretty close to malware analysis, we selected the tools that are the most relevant for packing detection evasion.

2015	**`EvadeML`** : is a Python library for creating adversarial attacks against machine learning models	[60]
2017	**`SigThief`** : is a python script that steals signatures from a PE file and injecting an invalid signature to another file	[61]
2018	**`AVET`**: (Anti-Virus Evasion Tool) is a tool that automates the process of evading antivirus detection	[62]
2018	**`MalwareRL`** : is a Reinforcement Learning-based approach to evade static detection of PE malware by Machine Learning models	[63]
2019	**`CarbonCopy`** : is a tool used to create a spoofed SSL certificate from any given website and sign an input executable to evade detection	[64]
2021	**`SecML Malware`** : is a Python library for creating adversarial attacks against Windows Malware detectors	[45]
2022	**`BitCamo`**: An adversarial Adversarial Machine Learning (AML) tool for modifying Windows PE files to evade detection by malware classifiers.	[65]

Table 3.13: Evasion tools

EvadeML [60] is a Python library for creating adversarial attacks against ML models. It is designed to generate adversarial examples that can evade Machine Learning-based malware detection systems.

SigThief [61] is a Python tool that steals a valid signature from a file and inject it in another PE file. This injected signature is not valid, but author noticed that some Anti-Virus (AV) vendors accept files without checking that the signature is actually valid.

AVET [62] is an AV Evasion Tool that automates the process of evading antivirus detection. It is a tool that can be used to test the effectiveness of AV software against malware. This project is also open-source [62].

MalwareRL [63] is a RL-based approach to evade static detection of PE malware by ML models. In their paper [66], authors defined 15 manipulations in the action space of the RL agent :

Name	Description
modify_machine_type	Modify the machine type in the PE header
pad_overlay	Pad the overlay with random bytes

append_benign_data_overlay	Append benign data to the overlay
append_benign_binary_overlay	Append a benign binary to the overlay
add_bytes_to_section_cave	Add bytes to the section cave
add_section_strings	Add strings to a section
add_section_benign_data	Add benign data to a section
add_strings_to_overlay	Add strings to the overlay
add_imports	Add imports to the import table
rename_section	Rename a section
remove_debug	Remove the debug information from the PE header
modify_optional_header	Modify the optional header
modify_timestamp	Modify the timestamp in the PE header
break_optional_header_checksum	Break the checksum in the optional header
upx_unpack	Unpack the file using UPX
upx_pack	Pack the file using UPX

Table 3.14: MalwareRL agent action space

CarbonCopy [64] is a Python tool that creates a valid signature from a spoofed SSL certificate of a given website and injects it into a PE file. This makes the executable appear as if it was legitimately signed.

SecML Malware [45] is a Python library for creating adversarial attacks against Windows Malware detectors. It is designed to generate adversarial examples that can evade machine learning-based malware detection systems. This tool includes 6 different attacks in its action space, as listed in Table 3.15.

Name	Description
Partial DOS Header manipulation	Formulated by Demetrio et al. [20]
Padding attack	Formulated by Kolosnjaji et al. [39]
GAMMA	Formulated by Demetrio et al. [41]
FGSM padding + slack	Formulated by Suciu et al. [21]
Content shifting and DOS header extension	Formulated by Demetrio et al. [19]
Header Fields	Inspired by Nisi et al. [46]

Table 3.15: Included attacks in SecML Malware [45]

BitCamo [65] is an Adversarial Machine Learning (AML) tool for modifying PE files to evade detection by malware classifiers. This tool focuses on overlays of PE files. It is an open-source project available on GitHub [65].

SUMMARY

The field of **static packing detection** has evolved significantly over the past 17 years, beginning with the outstanding work of Lyda and Hamrock [30] that introduced the *Bintropy* tool. Early research developed **simple heuristics** to detect packing, such as using the entropy of the file as criterion. Subsequently, **signature-based** methods were introduced and allowed to identify packed executables by recognizing known patterns found in packed binaries. However, this approach requires constant updates and is inefficient on custom unknown packers.

More recently researchers explored the usage of **Machine Learning (ML)** models for packing detection. These models can adopt two primary approaches: unsupervised learning, where the training dataset is unlabeled and the model is tasked to find the best ways to classify the data, and supervised learning, where the dataset is labeled. Supervised learning, while effective, depends on the accuracy of the labeled data, and many ML models in the literature rely on static detectors to label their training datasets.

Along with the emergence of ML-based detection, various **adversarial learning** studies have been published, targeting these detection models to assess their robustness and efficiency. To date, only a few adversarial studies focused on **attacking heuristics** used in static detectors. These attacks are mainly operated in the *problem space* by transforming a binary file to target heuristics, in contrast to the *feature space* where the transformation is done on features used by a ML model.

Many freely available and open source **packing tool** have been developed, serving different purposes and employing various methods including compression, encryption, virtualization and more. This facilitates researchers to build detection mechanisms but also enables attackers to use it for malware concealment and even customize it. One of the earliest **static detector** tool only uses the entropy of a file with threshold to detect packing. Since then, many tools emerged using different approaches such as detection using signatures, heuristics, anomalies and statistical methods. In 2015, a new open-source **evasion tool** named *EvadeML* [60] was introduced to create adversarial attacks to target ML models. Since then, various tools appeared and targeted ML models but also static detectors.

4 Framework

THIS chapter outlines the framework of our research.

It starts by defining the adversarial setting and the threat model of our study. Next, it provides an overview of the Packing Box framework, describing its components, functionality and abstractions. It also details how alterations work within this framework, including their integration and the building blocks required to define new alterations. Finally, this chapter presents the development environment of our tool, including its architecture, implementation and usage.

Objectives

A – Define the adversarial setting of our research.

B – Give an overview of the Packing Box framework, its useful abstractions and building blocks.

C – Present our tool's development environment, including its architecture and how it works.

4.1 Adversarial Setting

In this section, we outline the threat model of this work, it is designed to assess the capabilities of adversaries in evading static detection of PE packed executables by both static detectors and Machine Learning (ML) models. We use the threat model framework presented in the work of Carlini et al. [23], as detailed in Section 2.4.2.

4.1.1 Threat model

Adversary Knowledge

The focus is on adversaries operating under a **Black-Box** setting, where the internal configurations and features of the ML models are not accessible, but the adversary can interact with the model as an oracle, by submitting inputs and receiving outputs.

Note that the internal functioning of static detectors (i.e. features and heuristics used) is considered part of the public knowledge. This is either available in published papers or by analysis of the tool itself and its source code when an implementation is available. We consider that an attacker would have this knowledge.

Adversary Capabilities

The adversary is assumed to have the following capabilities:

- **Query access**: The ability to interact with the ML model by submitting a PE file and receiving the classification decisions (malicious or benign, packed or not packed) without direct access to the underlying model parameters or training data. Static detection tools can also be queried.
- **Problem Space Actions**: The capability to modify executables in a way that is likely to mislead ML models or static detectors, to make it appear as cleanware.

Adversary Goals

The primary goal of the adversary is to carefully modify packed PE executables in the problem space such that it reaches the following goals:

- **Evade detection**: Evade static detection by most common detectors and also ML models by performing a **targeted attack** that tricks the detection mechanism into incorrectly classifying the modified packed sample as unpacked.
- **Preserve functionality**: Maintain the functionality of the original executable, ensuring that it keeps its intended behavior after modifications.

In a real-world scenario this packed PE file could be a malware packed to hide its malicious intent. Because packing could be detected easily and is often used by malware authors, an attacker would also want to make it appear as not packed.

Our work thus focuses on an adversary having a **Black-Box** access to detection mechanisms, performing actions in the **Problem Space**, with the goal of using a **Targeted Attack** to trick detectors to classify modified attack samples as unpacked.

4.1.2 Scenario

In this work, we consider an adversary operating in a scenario where the primary objective is to bypass static packing detection. Specifically, the adversary seeks to manipulate a packed executable, e.g. packed with a popular free packer like UPX, to evade detection by making it appear as not packed.

A typical scenario for this type of adversarial attack involves evading mass-detection systems used during the sample triage process before thorough malware analysis. In this context, an adversary might modify a commonly used packer like UPX by integrating new routines that implement specific alterations. These alterations would be designed to defeat static detection mechanisms, causing the packed executable to be misclassified as unpacked. Alternatively, the adversary could use a specialized tool in conjunction with the packer, applying these evasive techniques after packing is completed to achieve the same goal.

The tool developed in this work is designed to be used by an adversary immediately after the packing process, as per this scenario.

4.2 Experimental Toolkit

This section introduces the toolkit that is the foundation on which we build our tool. It presents its toolset and the manipulations we can use to elaborate the logic of this tool.

4.2.1 Packing Box

Developed by D'Hondt in 2022, the Packing-Box framework is a Docker container based on a Linux (Ubuntu) environment, offering a wide range of tools for researchers to perform experiments on packing detection. This open-source framework is available on GitHub[1], where we obtained the Docker architecture depicted in Figure 4.1. The main point of this architecture is that it is designed in three layers ; the lower layer is the OS based on Linux for its broad support for multiple executable formats, the middle layer relates to support for binary formats and the libraries that are required for computations, and the upper layer holds the custom mechanics implementing abstractions of entities (i.e. executables, datasets, models, packers, ...) some of which have been made configurable and/or interactive thanks to a Python CLI tool.

This framework supports tasks such as executable parsing, dataset manipulation, detection, binary alterations, ML model training, and visualization. This Docker container serves as a Command Line Interface environment equipped with a comprehensive toolset. It brings together executable analyzers, packing

[1] https://github.com/packing-box/docker-packing-box

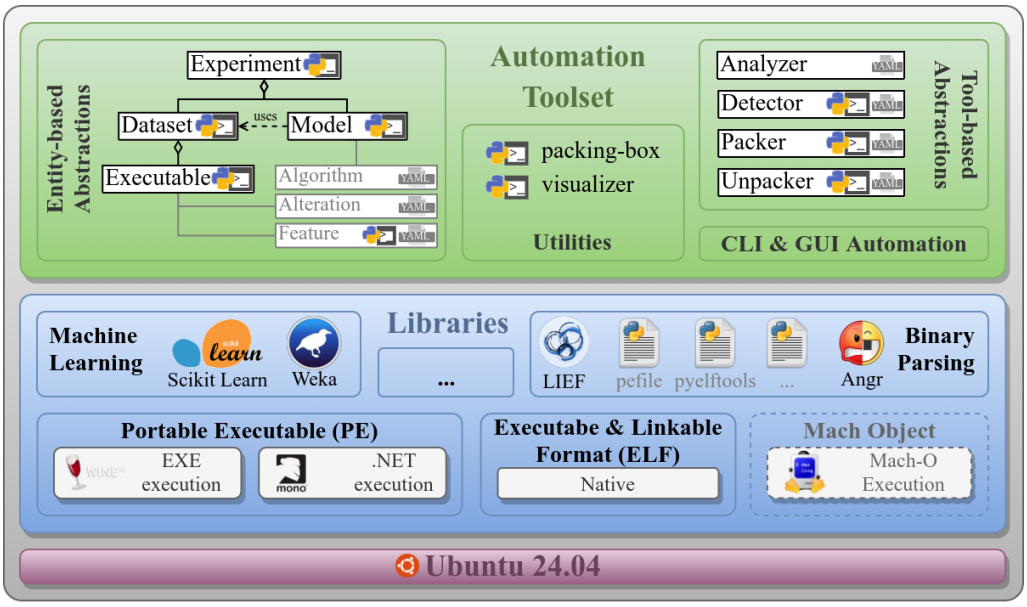


Figure 4.1: Packing-Box architecture [7]

detectors, packers, and unpackers, along with many tools for generating and manipulating datasets of executables across various formats, including PE, ELF, and Mach-O. It aims to provide a single platform for studying packing and automate Machine Learning pipelines with support of many algorithms.

A major part of our experiments are conducted using the Packing Box framework [7], it is thus important to understand the capabilities and abstractions of the framework.

Framework overview

Packing-Box includes abstractions and tools for the entire Machine Learning pipeline. This pipeline, as presented in Figure 4.2, is split into 4 main steps, where each step is backed by one or more abstractions and related tools. For instance, step `PREPARE` uses the `dataset` tool that relies on the `Dataset` entity consists of a set of `Executable`'s that has itself `Alterations` and `Features`.

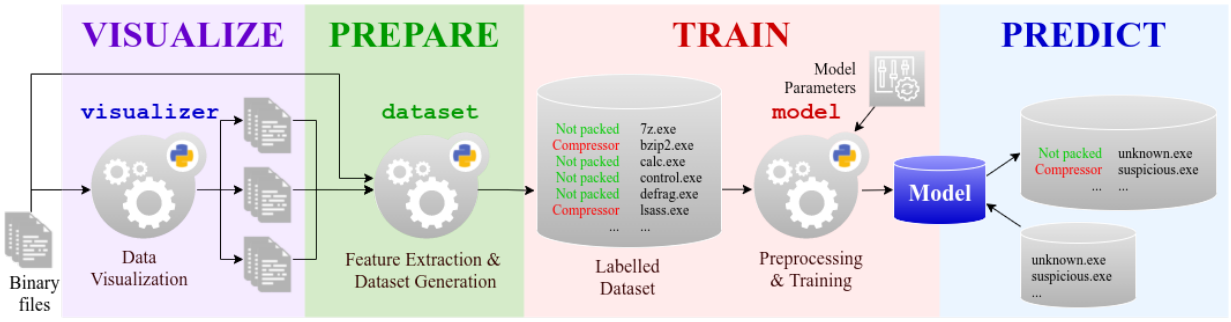


Figure 4.2: Packing-Box Machine Learning pipeline [7]

Visualizer is a tool that implements many visualizations of binary files from a single file or a dataset for comparison. It allows to distinguish and sort samples and helps to spot outliers.

Dataset is a tool to manipulate a `Dataset`, a set of binaries enriched with metadata that is used to train and test ML models. Samples can be stored in their original form or in a `FilelessDataset`, meaning that extracted features are stored instead of the entire file. This reduces the space used and the computing time of the model as those features are precomputed before training.

Model is a tool for ML model manipulation, such as training and prediction, but also for manipulating the model itself, including its metadata. It automates the `TRAIN` and `PREDICT` steps of the ML pipeline.

Packer can utilize installed packers program to a whole dataset or a single file. Below is an example for packing a single file with UPX :

```
┌──[user@packing-box]──[~]─────────
$ packer upx 7z.exe
```

Detector uses static packing detectors to launch a detection with one or many detectors at the same time.

```
┌──[user@packing-box]──[~]─────────
$ detector upx_7z.exe -d peid
upx
```

Note that packers and static detectors can both be used directly through the command line, without necessarily using the generic tool. But the tool provides seamless integration with other abstractions of the Packing Box, such as datasets.

Experiment is another useful abstraction that is used to create a dedicated workspace environment for each experiment. Each workspace experiment has its own parameters, datasets and models. The great advantage of this is that experiments can then be shared between researchers and results can be reproduced in an easy way for the sake of repeatability [5].

4.2.2 Dataset Manipulation

One of the first steps in the Machine Learning pipeline is dataset preparation. In Packing-Box, all dataset manipulations are done via the tool `dataset` and takes a command as the first positional argument. Some of the most useful commands for such manipulations are listed below.

- `alter` : alters the target dataset with a set of transformations defined in the experiment workspace (typically in a file called `alterations.yml`)
- `list` : lists existing datasets
- `make` : creates a new dataset with the specified parameters (e.g. format, number of files, packer)
- `ingest` : ingests samples from an external folder with a well-defined structure into new datasets
- `merge` : merges the second input dataset given into the first one
- `select` : select a subset of the dataset matching a given pattern
- `convert` : converts the target dataset into a fileless one, meaning it computes the selected features and removes the files from the dataset
- `export` : exports a dataset to a standard format (e.g. CSV, ..) and executable files into a folder.
- `plot` : draws a plot about a given dataset, it can for example plot distribution of features, information gain, labels and more.

The interested reader may refer to the official documentation [7] for other commands. A brief overview of how to use Packing Box is provided in Appendix B, including examples with the `dataset` tool.

In the following example, we present a basic usage of this abstraction to manage datasets inside the Packing-Box. We simply use the '`dataset select`' command with a custom query to extract from the dataset only UPX-packed samples. This selected subset is placed in a new dataset named '`upx_example`'.

A complete and realistic usage of the '`dataset`' abstraction will be detailed in the chapter 5.1.

```
┌──[user@packing-box]──[~]
└─$ dataset select baseline upx_example --query "label == 'upx'"
100% ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100/100 samples • 0:00:00 •
   0:00:00 • baseline
```

Dataset plot is used to visualize information about a dataset and the distribution of its samples. For instance, the plot shown in Figure 4.3 is generated by the command below and displays the distribution of the feature "`number_wx_sections`" (*number of sections that are writable and executable*) within the dataset.

```
┌──[user@packing-box]──[~]
└─$ dataset plot features baseline_np number_wx_sections
00:00:01.218 [INFO] Computing features...
<<snipped>>
00:00:04.387 [INFO] Saving to <<snipped>>/number_wx_sections.png...
```

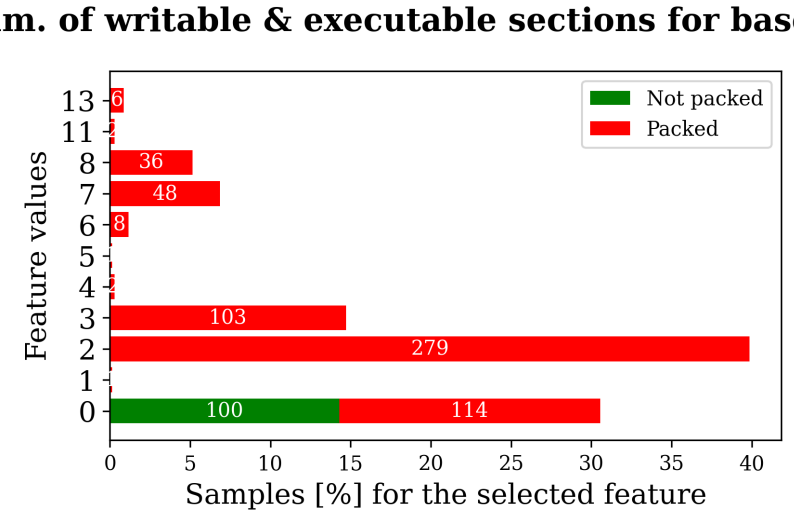


Figure 4.3: Distribution of the feature `number_wx_sections` in a dataset

Two other notable '`dataset plot`' subcommands are '`infogain`' and '`infogain-compare`'. Examples of each are provided below. Figure 4.4 illustrates the result of the `infogain-compare` subcommand, while Figure 4.5 displays the result of the `infogain` subcommand.

```
┌──[user@packing-box]──[~]
└─$ dataset plot infogain-compare dataset_not-packed --datasets dataset_amber
   dataset_jdpack dataset_upx_baseline --max-features 20
```

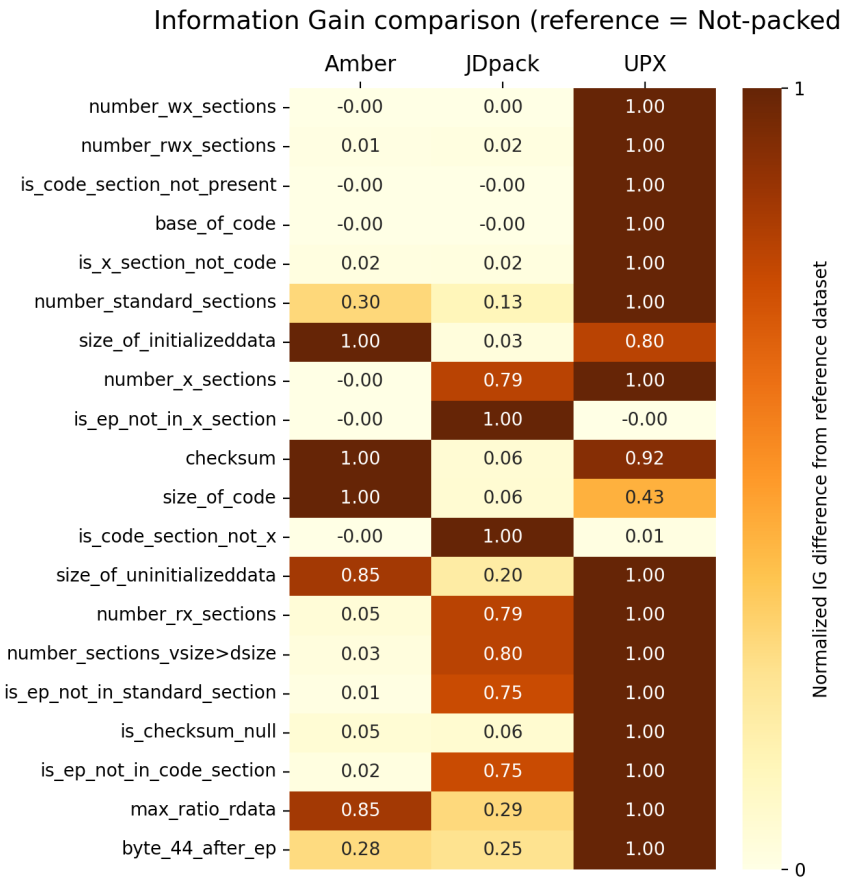


Figure 4.4: Example of IG comparison of Amber, JDpack and UPX with a reference dataset of not packed samples

Figure 4.4 provides a comparison of the Information Gain (IG) of 3 datasets packed with 3 different packers (i.e. Amber, JDpack, UPX) against a reference dataset of not packed samples. It shows the normalized difference between the IG of the reference dataset and that of each packed dataset.

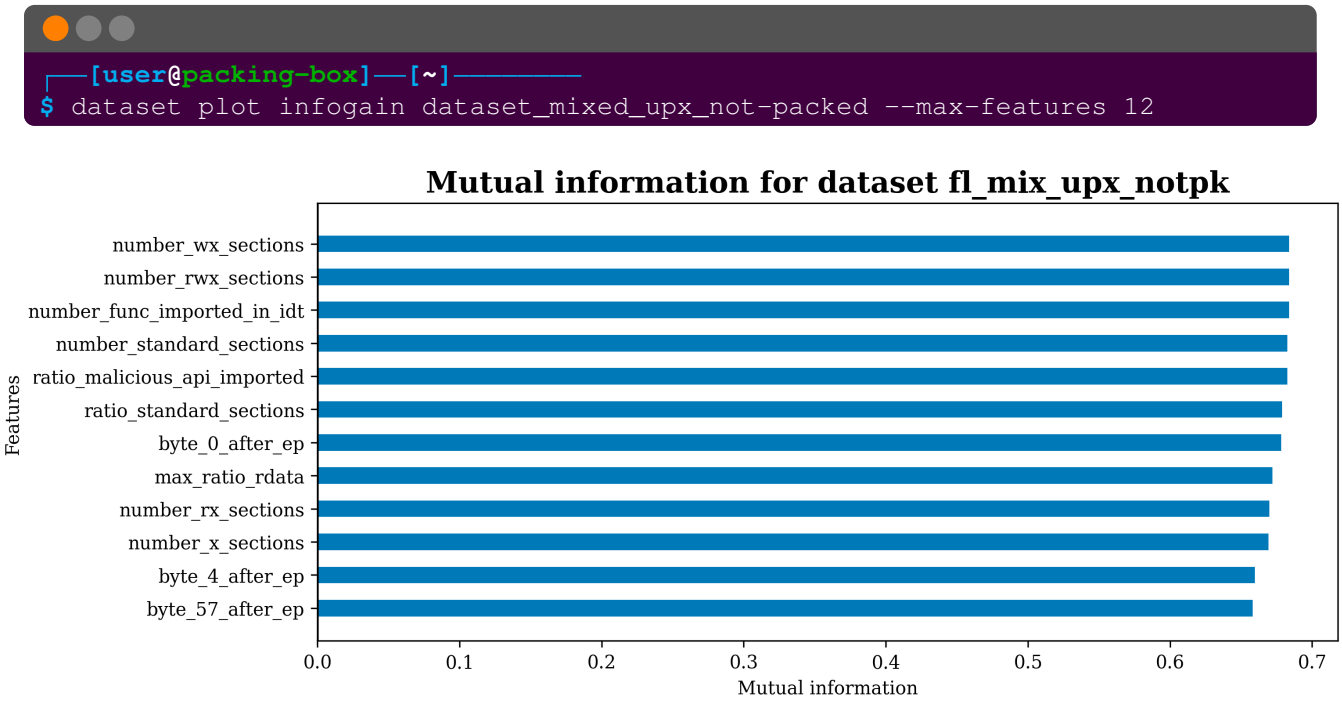


Figure 4.5: Example of IG of a dataset mixing not packed and UPX packed samples

The plot in Figure 4.5 shows the Information Gain in a mixed dataset of UPX packed samples and not packed samples, offering insights about the features that are the most useful to distinguish between UPX and not packed files.

4.2.3 Visualization

During experimentation, it is quite important to visualize the structure of files to study the effect of alterations or packing on a binary and interpret the results of the experiment. In the Packing-Box framework, the visualization process is done via the `visualizer` tool.

This tool requires a command as second positional argument, including :

1. `compare` : shows a byte-per-byte comparison plot of 2 given binaries.
2. `features` : displays a table listing the computed features of a given file.
3. `find` : searches for the presence of a given file in each dataset.
4. `plot` : gives a visual representation of multiple versions of a given sample side by side.
5. `remove` : removes the specified file from the dataset folder.

Visualizer compare allows us to perform a byte-wise comparison of multiple files, an example of usage is given below and its result is shown in Figure 4.6. Note that the pattern arguments are given in Regex format. When the `--text` option is used, the result is returned as a text.

```
┌──[user@packing-box]──[~]─────────
$ visualizer compare ^ 7z.exe$ ^ upx_7z.exe$

┌──[user@packing-box]──[~]─────────
$ visualizer compare ^ 7z.exe$ ^ upx_7z.exe$ --text
<<snipped>>
-0xEE       0x2   NumberOfSections:              0x4
+0xEE       0x2   NumberOfSections:              0x3
@@ -47,6 +55,6 @@
<<snipped>>
```

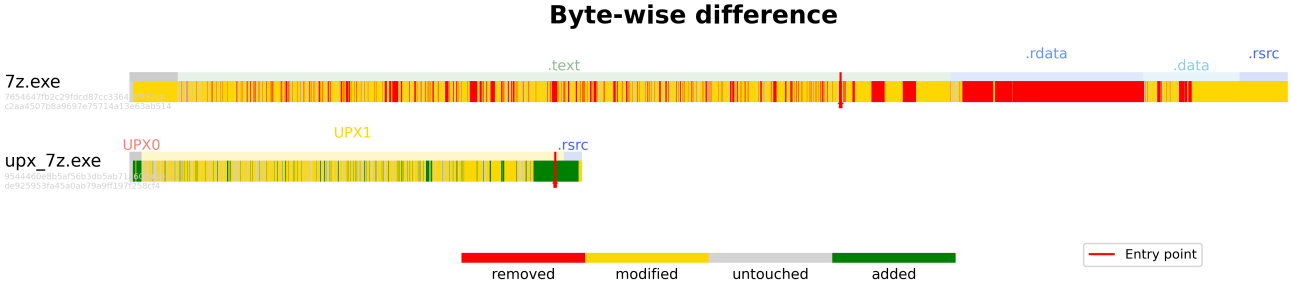


Figure 4.6: Example of a visualization comparing a file with its UPX packed version

Visualizer Plot produces a visual representation of a file alongside with its packed or altered versions within a dataset. It enables the comparison of samples with their modified versions. This feature is particularly useful for identifying differences across multiple packers or alterations, such as section names, entrypoint and entropy (using *Bintropy* [35]). To use it, we need to specify the filename of the sample we wish to plot and the folder containing the different datasets. The `-l` option allows us to specify labels from

the dataset, such as the packer used for the packed version. For example, we can plot the original `7z.exe` alongside its packed versions by different packers using the command given below, the resulting plot is given in Figure 4.7.

```
┌──[user@packing-box]─[~]──────
$ visualizer plot 7z.exe dataset-packed-pe --label not-packed --label jdpack
   --label upx --label telock --scale
```

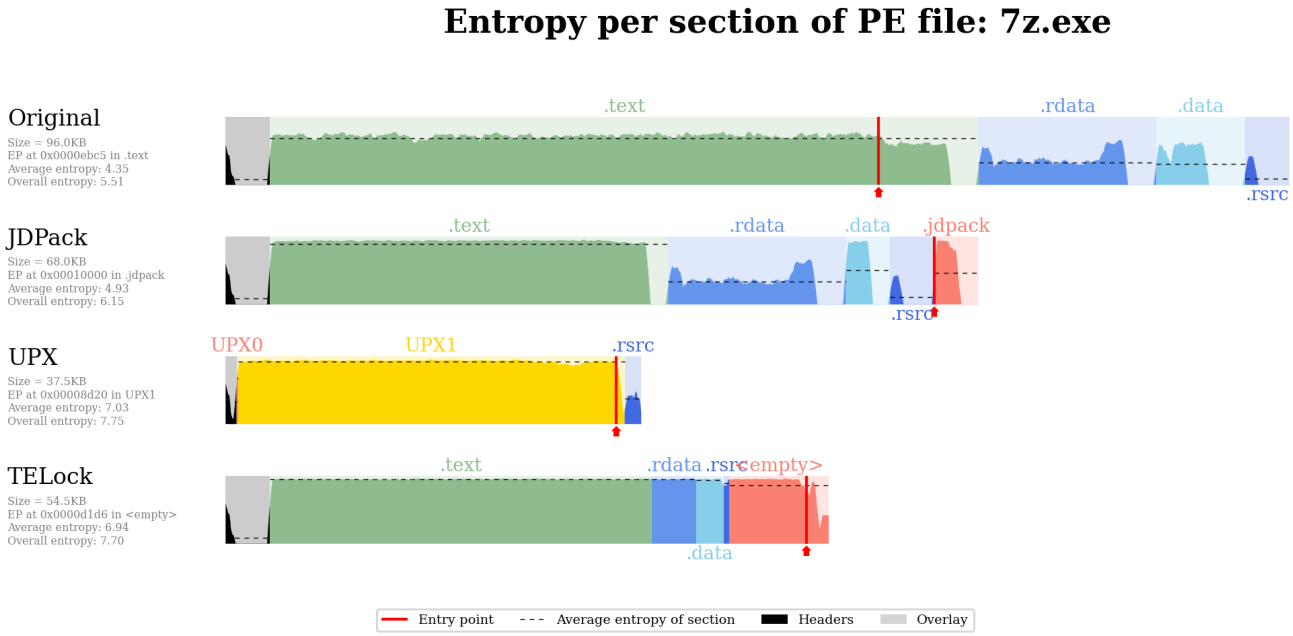


Figure 4.7: Example of plot comparing the structure and the entropy of a sample across many packers

4.2.4 Alterations

Binary alterations are defined in a YAML formatted file named "`alterations.yml`". This structured format allows for multiple alterations to be specified in a single file, providing enhanced flexibility. Researchers can easily add, edit, or remove alterations as needed for their experiments. The structure of the definitions within the file is as follows :

```
1 add_common_api_import:
2   apply: True
3   description: Add a randomly-chosen common API call to the Import Address Table
4   result:
5     PE: add_API_to_IAT(choice(COMMON_API_IMPORTS))
```

Listing 4.1: Example of `alterations.yml` definition

In Listing 4.1, `add_common_api_import` is the name of the alteration. It includes a description and a result section, which contains an entry `PE`, indicating that the specified alteration is intended for PE files.

The alteration pipeline, as illustrated in figure 4.8, defines the structure of the process of defining and applying alterations on a dataset. It uses a combination of predefined functions, lists, and objects as building blocks to define practical alterations that will alter a given dataset.

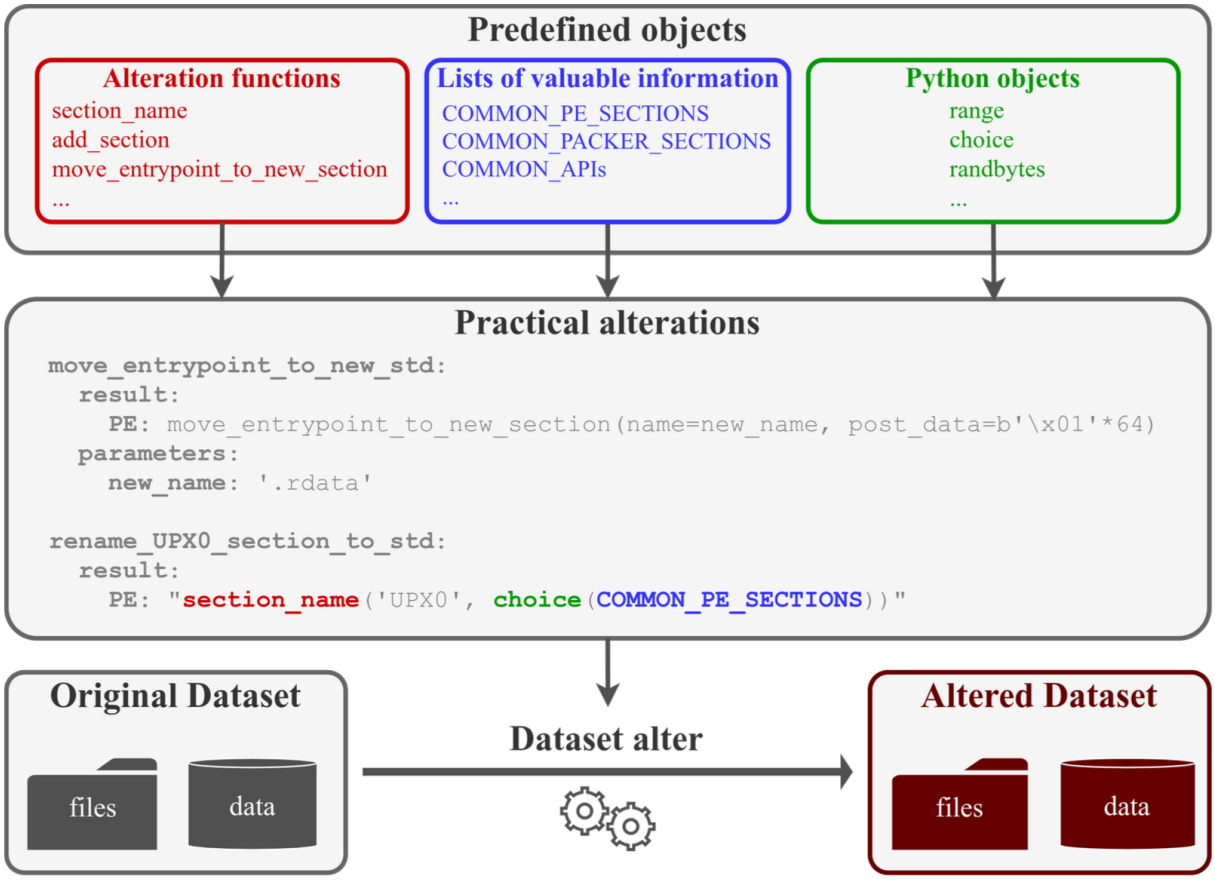


Figure 4.8: Alteration pipeline structure [2]

Predefined Objects

Alteration functions: These are the fundamental functions that implements specific modifications of binary files. These functions form the core of the alteration process, providing the necessary operations to apply changes in the binaries. The following functions known as modifiers are listed here after :

- `add_API_to_IAT` : This adds an API function call to the Import Address Table of the sample.
- `add_lib_to_IAT` : This adds a DLL library to the Import Address Table (IAT) of the sample.
- `add_section` : This function will add a new section in the file based on the given name and content. It can receive as parameter the section type, characteristics and data content.
- `append_to_section` : Function that takes an array of bytes as input and appends the data at the end of the section specified in parameters.
- `move_entrypoint_to_new_section` : This adds a new section with a code called a trampoline that jumps back to the original Entry Point (EP). It also accept 'pre_data' and 'post_data' as parameter, to insert custom bytes before and after the trampoline code.
- `move_entrypoint_to_slack_space`: This alteration moves the Entry Point (EP) to the slack space of the specified section by injecting trampoline code that jumps back to the original Entry Point (EP). Slack space refers to the unused space at the end of a section, which may not always be available. This function also accepts 'pre_data' and 'post_data' as parameters to insert content before and after the trampoline. It is important to note that the larger the inserted code, the less likely there will be enough slack space, as this function is not intended to increase the section size.
- `set_checksum` : It will replace or set the checksum in the header with the provided value.

In addition to modifiers, we also have extractors and parsers. One of the extractors is the 'pefeats' function that extracts a vector of 119 features. Those are defined in the Packing-Box framework, and have been introduced in the work of *Biondi et al.* [3] and further used by *Bertrand Van Ouytsel et al.* [4].

At this time, the main parser is LIEF and a parsed executable gets abstracted so that it exposes standard attribute names for use in the YAML definitions of alterations. Some of these attributes are listed hereafter, as called when using a parsed PE file.

- `pe['empty_name_sections']` : is a helper function that gives a list of sections of the parsed executable that have an empty name.
- `pe['known_packer_sections']` : is a helper function that gives a list of sections of the parsed executable whose name are included in the `COMMON_PACKER_SECTION_NAMES` list.
- `pe['(non_)standard_sections']` : this helper function is similar but for section names included (or not) in the `STANDARD_SECTION_NAMES` list.
- `pe['section_names']` : is used to get the list of section names of an executable.
- `pe['section_average_entropy']` : computes the average entropy across all section's entropy.
- `pe['real_section_names']` : extracts the real section names of sections having names longer than 8 characters. Because when using longer names, the PE format uses as section name the offset to the long name in a string table (e.g. '`/42`').
- `section['entropy']` : uses Bintropy to compute the section's entropy. This function is available via section parsed objects.

Lists of valuable information: These lists provide predefined constant values that can be used in the alterations, this allows researchers to easily add more values to a list and this can even be embedded in an experiment. Example of lists are given below :

- `STANDARD_SECTION_NAMES` : contains standard section names commonly found in PE formatted files (not packed).
- `COMMON_PACKER_SECTION_NAMES` : contains section names commonly found in packed executables.
- `COMMON_API_IMPORTS` : contains API function names commonly found in non packed PE files.
- `COMMON_DLL_IMPORTS` : contains Dynamic Link Library (DLL) names commonly found in non packed PE files.
- `COMMON_MALICIOUS_APIS` : contains API function names commonly found in malware or packed executable.
- `COMMON_DEAD_CODE` : for our alteration improvement we added this definition of dead code that are instructions in assembly that does nothing interesting.

Python objects: These are standard Python objects like `range`, `choice`, and `randbytes`, and can also be incorporated in the alteration definition. These objects enhance the flexibility and variability of the alterations, allowing researchers to define more dynamic and diverse modification scenarios.

Practical Alterations

Practical alterations are defined through the integration of predefined objects. For instance, the simple example on Figure 4.8 (`rename_UPX0_section_to_std`) involves changing the section name of 'UPX0' to a randomly selected section from the list `STANDARD_SECTION_NAMES`.

These practical alterations are structured to be easily customizable and applicable, providing a flexible framework for conducting diverse experiments on binary files.

In order to define a new alteration, we simply need to add an entry in the YAML file, then we need to use available predefined objects given by the Packing-Box framework. The figure 4.8 shows an example by adding the definition of a new alteration named "`rename_UPX0_section_to_std`".

4.3 Development Environment

This section outlines the environment and workflow involved in the development of our tool, detailing its architecture, implementation, and testing processes. We begin by discussing the overall architecture of the tool, explaining its core components and how they interact. Following this, we delve into the specifics of the implementation, describing the development environment, programming languages, libraries, and frameworks used. Finally, we cover the testing methodologies employed to ensure the tool's effectiveness and reliability. This overview provides a clear understanding of the development lifecycle and the processes involved in creating a robust adversarial tool for evading static packing detection.

4.3.1 Attack Tool's Architecture

NotPacked++ is designed as an adversarial tool to alter packed executables and evade static packing detection, useful for both malware analysts and red teamers. It focuses on the PE file format, with a modular and extensible design for future expansion. Primarily designed for the Linux environment in ELF format, this tool is intended for use in typical attack settings like Kali Linux. In its initial version, it exclusively supports the PE format.

The tool interacts with the user through a **Command Line Interface (CLI)**, allowing them to easily provide input executables. The CLI is designed to be user-friendly and supports various customization options.

Our architecture is defined by three main structural components, each playing a crucial role in the overall functionality of the tool. These components are designed to work seamlessly together, ensuring that the tool is both user-friendly and efficient. The three main components are illustrated in Figure 4.9.

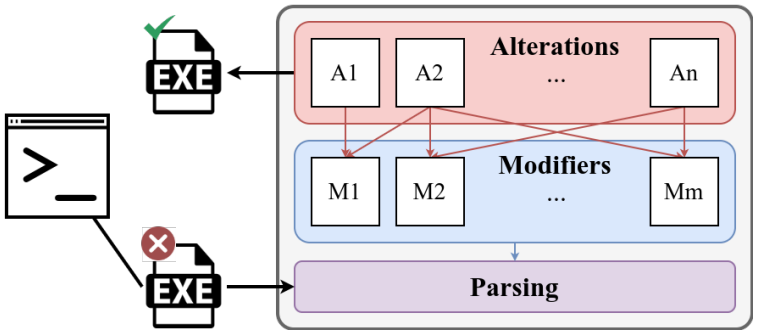


Figure 4.9: NotPacked++ architecture

- **Parsing**: This component handles the initial loading and parsing of the input packed executable, preparing the binary for alterations. It includes the parsing library LIEF but also additional parsing capabilities.

- **Modifiers**: The core module responsible for applying various functionality preserving transformations to the binary. This includes operations like section name changes, entry point modifications, and filling sections with zeros.
- **Alterations**: This module applies one or more modifications to the parsed binary file through the usage of the Modifiers module.

4.3.2 LIEF parsing library

Library to Instrument Executable Formats (LIEF) [44] is an open source library designed to handle executable formats across different platforms and architectures. It provides a comprehensive abstraction for parsing, modifying, and analyzing binary files, in various formats. LIEF is particularly useful for researchers, developers, and security professionals who need to work with executable formats at a low level. Our tool is developed using LIEF, taking advantage of its parsing and rebuilding capabilities.

Supported Formats

LIEF supports a wide range of executable formats, including but not limited to :

- **Portable Executable (PE)**: The PE format is used on Windows operating systems, the most used OS in the world.
- **Executable and Linkable Format (ELF)**: This format is used on Unix-like systems, including Linux.
- **Mach Object (Mach-O)**: This Mach-O file format is typically used on macOS and iOS platforms.

Core Features

LIEF offers a robust set of features that facilitate various binary manipulation tasks :

- **Parsing**: LIEF can parse executable files and extract detailed information about their structure, including headers, entrypoint, sections, symbols, and more.
- **Modifying**: Users can modify different parts of an executable file, such as altering headers, adding or removing sections, and patching binaries.
- **Rebuilding**: After modifications, LIEF can rebuild the executable, ensuring that it remains functional and correctly formatted.
- **Cross-platform Compatibility**: LIEF is designed to work across multiple platforms, providing a consistent interface regardless of the underlying operating system.
- **Binding for Multiple Languages**: LIEF provides bindings for several programming languages, including Python, C++, and C, making it accessible to a large range of developers.

Applications

This cross platform library is suitable for a variety of applications:

- **Reverse engineering**: LIEF is useful in the reverse engineering of binaries by providing detailed insights into their structure and content.
- **Malware analysis**: Security researchers use this library to analyze and understand malicious software, identifying modifications and unusual patterns within executables.

- **Binary modification**: Developers can use LIEF to alter binaries, adding custom code or modifications for performance monitoring, debugging, or other purposes.
- **Software patching**: LIEF enables the creation of patches for existing software, allowing updates or bug fixes to be applied directly to binary files.

Usage

A simple example of usage parsing and modifying a PE file using Python is provided in Listing 4.2. This shows parsing, modification and rebuilding of a binary file, in 5 different steps described below :

1. **Load** the PE file
2. **Access** the parsed value of the address of the entrypoint via LIEF's abstraction and print the address.
3. **Modify** the address of the entrypoint in the PE header.
4. **Add** a new section called "`.new_section`" having `0x90` bytes (`NOP` instructions) as content.
5. **Save**, rebuild the edited binary file into a new valid PE formatted file.

```python
import lief
# 1. Load the PE file
binary = lief.parse("sample.exe")
# 2. Print the original entry point
print(f"Original Entry Point: {hex(binary.entrypoint)}")
# 3. Modify the entry point
binary.optional_header.addressof_entrypoint = 0x12345678
# 4. Add a new section
new_section = lief.PE.Section(".new_section")
new_section.content = [0x90] * 100 # NOP instructions
binary.add_section(new_section)
# 5. Save the modified binary
binary.write("modified_sample.exe")
```

Listing 4.2: Example LIEF usage in Python

During the experiment phase, we will use the Packing-Box framework which uses the Python API. For efficiency purposes, the C++ API will be used for our adversarial tool. The C API is not being used because it offers a limited set of features and does not meet our requirements.

In order to compile the C++ file, we need to add an additional parameter "`-lLIEF`" to the compilation command to link the LIEF library. An example of command is shown below:

```
┌──[user@host]──[~]──────────
$ g++ -o modify_binary modify_binary.cpp -lLIEF
```

This command will link the library dynamically, but the same command using `-l:libLIEF.a` parameter instead compiles the file statically, ensuring that all necessary files of the library are included in the resulting binary.

4.3.3 Implementation

The development of NotPacked++ involved several strategic decisions and a methodical approach to ensure its effectiveness, reliability and portability. Below, we outline the key aspects of the implementation process, detailing the choices made for the programming language, parsing technique, framework usage, code implementation, and testing methodologies. This approach was essential to create from an experimental framework to a robust tool capable of altering packed executables and evading static packing detection, while being easy to use and easily portable.

Programming Language and Parsing The tool is written in C++ to ensure a fast execution time, which is crucial for handling complex binary alterations and large number of files. We chose C++ because the parsing library we use, LIEF, has been written in C++ and it offers a robust C++ API. This API provides all the necessary functions to modify and rebuild executables, unlike the more limited C API. The choice of the LIEF library was made because it offers robust parsing and rebuilding capabilities, it is already used in the Packing-Box framework's implementation and it is well-documented.

Object-Oriented Design To enhance the modularity and maintainability of the tool, we adopted an Object Oriented Programming (OOP) approach. Key components such as `Modifiers`, `Alterations`, `Utilities`, and `PEBinary` classes were designed to encapsulate specific functionalities. The Modifiers class handles the logic for altering binary files, the Alterations class defines the types of modifications that can be applied, and the `PEBinary` class manages the PE file structure and its attributes.

Packing-Box Framework The Packing-Box framework served as our starting point and baseline. We studied its current implementation of modifiers and alterations, to be transposed to the C++ tool. We also leverage its capabilities to visualize and verify the correctness of our modifications to ensure that the altered executables maintains their intended structure and functionality. Figure 4.10 illustrates the workflow of using the Packing Box for the development of the tool.

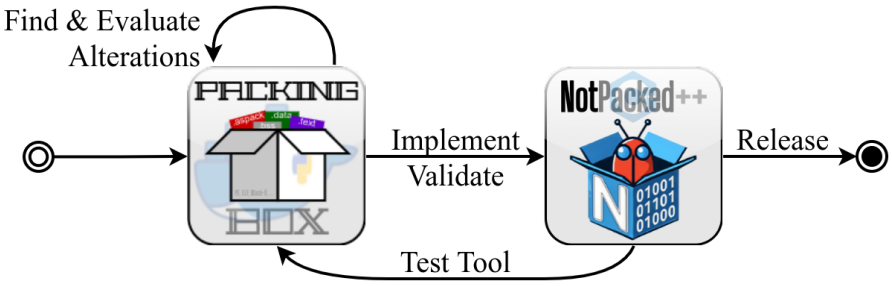


Figure 4.10: Workflow of the tool using Packing Box

Code Conversion The process of converting Python code alterations into working C++ code was done by taking advantage of a Large Language Model (LLM), such as OpenAI's ChatGPT [2]. This model assisted us in quickly transforming Python code into C++ code. Each modifier function was converted one by one, followed by meticulous assessment and debugging to ensure the resulting C++ code works as intended.

Testing and Validation After conversion, the code was built and compiled into an executable tool, as an ELF formatted executable. We conducted thorough testing for each alteration function to ensure effective modification of example binaries. The Packing-Box framework was then used to visualize these altered binaries, allowing us to assess whether the alterations produced the desired structural changes.

[2] `https://openai.com/chatgpt`

4.3.4 NotPacked++

NotPacked++ [67] is an adversarial tool designed to modify packed executable files to evade static packing detection mechanisms. In its initial version, the tool only focuses on the PE file format, the most common file format for binary analysis. It is meant to be used by malware analysts to test the effectiveness of their detection mechanisms and enhance their detection capabilities. It may also be useful for red teamers for testing the effectiveness of their evasion techniques, and highlight potential weaknesses of a target's security mechanisms.

The name "NotPacked++ " has been chosen based on multiple references that highlights the tool's advanced capabilities and core purpose :

- It honors to the popular text editor "Notepad++", implying a tool that offers both advanced features and a user-friendly usage.
- The "++" symbol indicates that the tool is written in C++, a programming language known for its efficiency in handling low-level operations.
- The name encapsulates the tool's core functionality of transforming packed executables to make them appear "not packed," thereby evading detection mechanisms.

This clever naming encapsulates the tool's mission to make executables look "not packed" in a straightforward, yet powerful manner.

Installation

To install NotPacked++ , we either download the latest release from the GitHub releases page [3], or build our own version by following these steps :

1. First we start by cloning the repository from the public Github page.
2. Next we need to install dependencies :
   - We simply run our prepared installation script `install_lief.sh` (it may require root privilege) :
   - Alternatively, we can use the **Dockerfile** provided to create a ready to use docker environment with all dependencies installed.
3. Finally, we compile and build the tool using the `make` command.

```
┌──[user@host]──[~]─────────
$ git clone https://github.com/packing-box/packer-masking-tool.git && cd
   packer-masking-tool
┌──[user@host]──[~]─────────
$ ./install_lief.sh
┌──[user@host]──[~]─────────
$ make
```

[3] https://github.com/packing-box/packer-masking-tool/releases

By default, the `Makefile` is defined such that the resulting compiled tool links the LIEF library statically, which means it embeds the necessary files from the library in the tool itself. This design is for the tool to be easily portable, but it increases the tool's size significantly. In order to use dynamic linking instead and thus reduce the size of the resulting tool, we can edit the Makefile of the project. In that file the compilation argument specified by default is "`-l:libLIEF.a`" and has to be replaced by the dynamic linking argument "`-lLIEF`".

Usage

The tool can be executed using various options to alter the packed executable in many ways:

```
┌──[user@host]──[~]──────────
$ ./notpacked++ <input_file> [OPTIONS]
```

Available options include:

- `--help`: Display a help message.
- `-o <output_file>`: Specify the output file name of the resulting altered binary.
- `--fill-sections`: Fill sections with zeros up to its virtual size.
- `--rename-sections`: Rename packer sections to standard names.
- `--permissions`: Update section permissions, move the entry point and rename sections.
- `--raw-size`: Edit the raw size value for zero sized sections in section headers without impacting the file size.

If at least one alteration option is specified, only the selected alterations will apply. The default behavior if no options are specified is as follows :

```
┌──[user@host]──[~]──────────
$ ./notpacked++ <input_file> -o <input_file>_out.exe --permissions --raw-size
```

It is important to keep in mind that alterations are applied in the same order as the provided options.

Demonstration

To demonstrate the practical application of NotPacked++ , consider the following usage example. This showcases how the tool can be used to transform a packed executable to make it evade static detection of packing.

Below is an example of the default behavior of the tool where no option are given.

```
┌──[user@host]──[~]─────────
$ ./notpacked++ sample.exe
<<snipped>>
[INFO]   Filling sections with zeros from their raw size to their virtual
   size...
[.] Truncating data to fit available space: 110592 bytes
[.] Truncating data to fit available space: 2560 bytes
[.] Truncating data to fit available space: 2048 bytes

[INFO]   Updating the permissions of all sections to standard ones
   (rwx/rw-/..), moving the entry point to a new section and Renaming
   sections to standard ones...
[+] Renaming section UPX0 to .data
[+] Renaming section UPX1 to .rdata

[SUCCESS]  File saved as: sample_out.exe
```

As we can notice, the alterations `--fill-sections` and '`--permissions`' have been applied. First, we see the output of the `--fill-sections` alteration. Then, we observe that the section `UPX0` have been renamed `.data` and `UPX1` renamed `.rdata`, their permissions have been changed from `rwx` to their standard permissions `rw-` and '`r-`'.

Another usage example is shown below, where we specify the output filename and 3 different alterations:

```
┌──[user@host]──[~]─────────
$ ./notpacked++ sample.exe -o altered_sample.exe --move-ep --rename-sections
<<snipped>>
```

This command takes `sample.exe` as input, moves the entry point to a new low entropy section and renames sections to standard section names. It outputs the resulted binary file as `altered_sample.exe`.

For more detailed usage instructions, the `--help` option can be used. This command provides a comprehensive list of all available options and their descriptions.

The interested reader can find more information on NotPacked++ in Appendix D.1, including the output of the `--help` option showing available alterations.

SUMMARY

In the first part of this chapter, we defined the adversarial **scenario** and **threat model**. The adversary is supposed to be able to query the ML model without knowing its underlying configuration and features. Additionally, the adversary can alter binaries to evade static detectors. We consider the internal features and heuristics of static detectors to be public knowledge, either through published documents or tool analysis. The adversary's objective is to avoid detection while preserving the functionality of the original binary. We've seen two approaches to study evasion techniques in packing detection : attacks in the feature space and attacks in the problem space. In this study we will work in the problem space.

In a second part, we presented the **Packing Box**, an experimental toolkit for studying executable packing. It offers a wide range of tools for researchers to perform experiments on packing detection and automate machine learning pipeline. We introduced each tool relevant to our study, as the majority of our experiments are conducted using the Packing Box framework. These abstractions include `dataset`, `visualizer`, `packer`, and `detector`.

In the last part of this chapter, we defined the development environment of our tool **NotPacked++**. Its initial version only supports the PE format. The architecture is divided in 3 main modules: the Parser, Modifier and Alteration module. The tool is used through a Command Line Interface (CLI), and allow multiple options. We briefly presented the LIEF library that is used to parse input PE files. Our development starts with the Packing Box framework, where we test and experiment with alterations. Then we use Large Language Model (LLM) for quickly converting Python code into C++ code, followed by the implementation into our tool and validation through visualization of the effects in the Packing Box. The name "NotPacked++" is a reference to the popular text editor "Notepad++". The "++" indicates that it was developed in C++, which is known for its efficiency. This name also reflects the tool's main functionality: transforming packed executables to make them look as "not packed" thereby evading static detection.

5 Experiments

THIS chapter details the experimental processes and findings of our research.

It starts by detailing the current alterations, verifying their correctness, and optimizing them. Then, it introduces new alterations and analyzes their effects on targeted features through visualizations. The chapter then demonstrates the evasion capabilities of our alterations against common static packing detectors and compares their detection accuracy. Finally, it demonstrates how the NotPacked++ tool was used to study and evaluate the impact of its implemented alterations on features and detectors, and evaluate its speed performance.

Objectives

A – Set the methodology used in our experiments.

B – Describe current alterations, validate their correctness and optimize them.

C – Show new alterations and study their impact on features and static packing detection.

D – Validate NotPacked++ by evaluating its speed performance and its impact on static detection.

5.1 Methodology

In this chapter, we detail the methodology and approach of our experiments. Our goal is to evaluate the effectiveness of alterations and our tool in altering packed executables to evade static packing detection. We describe the datasets used for our experiments, verify the functionality of files altered by existing alterations and analyze their effects on features. Additionally, we explore improvements, introduce new alterations and present the usage of the NotPacked++ tool.

5.1.1 Approach

In the following Table 5.1 we list our approach for each steps of our experiments, starting by existing alterations, then diving into new alterations to be added and finally our experiment using our new tools.

	Name	Description	Outcome
5.2.1	Functional verification	Verify that altered executable remain functional after applying current alterations.	Visual representation and assessment
5.2.2	Effect on features	Visualize the impact of each alteration on features	Visual analysis
5.2.3	Alteration improvement	Optimize, fix and improve current state alterations of the Packing-Box	Optimized alterations
5.3	New alterations	Add new alterations to the current state	New alterations implementation, visualize its effect
5.4	NotPacked++	Alter executable files using the tool and visualize its impact on the detection.	Visualization and impact on detection

Table 5.1: Experiments approach

Functional Verification: We begin by verifying the functionality of each executable that has been altered by our alterations. This step ensures that alterations do not break the executable's operation. Each altered executable is tested in a controlled environment to confirm that it still works as expected. This is an important step as we don't want our tool to corrupt input executables. To facilitate our verification we choose and run executable that runs on the console, this enable us to quickly verify it is functional. The chosen executables are 32-bit binaries from the reference datasets [68]. We then use Wine to execute PE files and analyze the expected output on the console. Wine [1], initially standing for "*Wine Is Not an Emulator*", is defined by authors as a compatibility layer that allows Windows applications to run on POSIX systems, including Linux and macOS. Instead of emulating or virtualizing Windows internals, Wine converts Windows API calls into POSIX calls on-the-fly, thus avoiding the performance and memory overhead associated with other methods and enabling smooth integration of Windows applications into our environment.

[1] `https://www.winehq.org`

Effect on Features : Next, we discuss and visualize the targeted features of each alteration. This involves analyzing how each alteration affects key features that are commonly used in static detection models. By visualizing these feature changes using the commands `dataset plot features(-compare)` or `dataset plot infogain(-compare)` of the Packing-Box framework, we can better understand the impact of our alterations on the detection capability.

Alteration Improvement : We then present improvements made to existing alterations. Based on our analysis and testing, we identify opportunities to enhance the effectiveness of these alterations or fix the implementation. The goal of these improvements is to increase the evasion capabilities of alterations while ensuring that executables remain functional. We start by finding solutions to alterations breaking the executable's functionality, then we explore improvements for working alterations.

New Alterations : Afterwards, we also introduce new alterations specifically designed to target significant features for packing detection. We visualize the effect of these new alterations on the features and evaluate their impact on the detection rate. This step involves a thorough analysis to determine how well these new alterations evade static detection by evaluating how the detectors' accuracy is affected. This includes the usage of the following commands : `dataset plot features(-compare)`, `dataset plot infogain(-compare)`, `detector` and a custom script to build a nice table of the detection rates.

NotPacked++ : In the final part of this chapter, we present NotPacked++ , our adversarial tool. We provide an overview of its effectiveness and usage. We then test the tool on a variety of executables, visualizing and assessing the effects of the tool's alterations. Finally, we demonstrate how the detection rate drops when these altered executables are given to static packing detection systems. We finish this chapter by a small speed comparison of our tool compared to the experimental toolkit Packing Box.

5.1.2 Datasets

By taking advantage of the `dataset` abstraction, we created several datasets for our experiments. We began by importing labeled samples from the external source of the *dataset-packed-pe* GitHub repository [68]. This resource provides 432 not packed PE executables along with numerous samples packed using 25 different packers. Next, we gather packed samples from the imported datasets, selecting a few executables into a baseline dataset in order to obtain a distributed dataset.

The resulting datasets are listed as follows:

- **`baseline`** : this baseline dataset contains 600 packed samples, each 100 packed with different packer (ASpack, FSG, MoleBox, UPX, PEtite, PECompact)
- **`baseline_np`** : this is the same as *baseline* with the addition of 100 samples of non-packed binaries.
- **`not-packed`** : this is a dataset containing 432 not packed samples.
- **`fl_*`** : Every dataset starting with this prefix are fileless dataset and contains computed features of the dataset mentioned after the prefix.

The selection of these 6 packers for our baseline dataset is motivated by the need to cover a diverse range of packing techniques and levels of complexity, while maintaining a manageable and balanced dataset for in-depth analysis.

In the following example, we demonstrate the basic usage of the dataset abstraction for managing datasets within the Packing-Box framework. Using this abstraction, we create the previously described datasets. We start by ingesting the entire *dataset-packed-pe* resource, which includes 432 unpacked samples, 123 UPX-packed samples, and various other packed executables.

```
┌──[user@packing-box]──[~]─────────
$ git clone https://github.com/packing-box/dataset-packed-pe
┌──[user@packing-box]──[~]─────────
$ dataset ingest dataset-packed-pe --labels dataset-packed-pe/labels.json
   --exclude outliers
<<snipped>>
```

After ingesting the datasets, we create a new `baseline` dataset. This dataset consists of 600 packed samples, extracted from the *dataset-packed-pe* resource, and contains 100 samples for each packer. We create our baseline dataset by selecting 100 samples from each chosen packed dataset and importing them into the baseline dataset.

```
┌──[user@packing-box]──[~]─────────
$ dataset select upx baseline -n 100
100% ━━━━━━━━━━━━━━━━━━━━━━━━ 100/100 samples • 0:00:17 • 0:00:00 • baseline
┌──[user@packing-box]──[~]─────────
$ dataset select aspack baseline -n 100
100% ━━━━━━━━━━━━━━━━━━━━━━━━ 100/100 samples • 0:00:17 • 0:00:00 • baseline
┌──[user@packing-box]──[~]─────────
$ dataset select petite baseline -n 100
100% ━━━━━━━━━━━━━━━━━━━━━━━━ 100/100 samples • 0:00:17 • 0:00:00 • baseline
<<snipped>>
```

The reason behind the choice of getting 100 samples is to obtain a distributed dataset as the resource used only contains around 100-150 packed samples for each packer.

To plot and analyze features of a given dataset, they need to be computed first. To avoid computing features every time we generate a plot, we can precompute the features and create a fileless dataset using the `dataset convert` command. This approach enhances efficiency and saves time in our experiments. Below is an example of creating a fileless dataset from the `not-packed` dataset:

```
┌──[user@packing-box]──[~]─────────
$ dataset convert not-packed -n fl_not-packed
100% ━━━━━━━━━━━━━━━━━━━━━━━━ 432/432 samples • 0:00:17 • 0:00:00 • fl_not-packed
```

5.2 Existing Alterations

In this section, we will verify that the altered executables maintain their functionality. Although Jennes [2] also conducted a similar verification, their focus was limited to visualizing the binary to check the functionality of the alteration functions rather than the functionality of the executable itself.

5.2.1 Functional Verification

The current `alterations.yml` file definition is the result of the work conducted by D'Hondt [1] in 2022 for the creation of the Packing-Box and the work of Jennes [2] in 2023 to advance the adversarial study further and identify new useful alterations for evading packing detection. We define below each alteration, its

purpose and visualize its effect on a binary file, then we run an altered executable to validate the functionality preserving feature of the alteration. We use Wine to run PE files, and we specifically chose samples that allows us to easily view the output on the console for a quick confirmation of its functionality, without the need of a GUI. For our assessment, the most frequently used samples are the UPX packed versions of `du.exe` (Directory disk usage reporter) and `7z.exe` (7-Zip).

add_20_common_api_imports : Adds common API imports to the IAT and without duplicates. This alteration tries to affect the feature related to import functions, such as the number of imported functions and the ratio of malicious imported API. Many packers have only a limited amount of imported functions as their only purpose is to unpack the original binary and continue execution. This alteration will loop and add 20 common API by using the predefined modifier `add_API_to_IAT`.

In order to visualize the effect of this alteration on a binary, we can use the command shown below. This command compares the `upx_calc.exe` file in the base dataset `upx_baseline` with the altered dataset named `add_20_common_api_imports`. The result of this command is displayed in figure 5.1.

```
┌──[user@packing-box]──[~]─────────
$ export hash_calc_exe=<<snipped>>
┌──[user@packing-box]──[~]─────────
$ visualizer compare "datasets/upx_baseline/files/$hash_calc_exe"
   "datasets/add_20_common_api_imports/files/$hash_calc_exe" -l1 before -l2
   after -t add_20_common_api_imports -o figures/add_20_common_api_imports.png
<<snipped>>
```

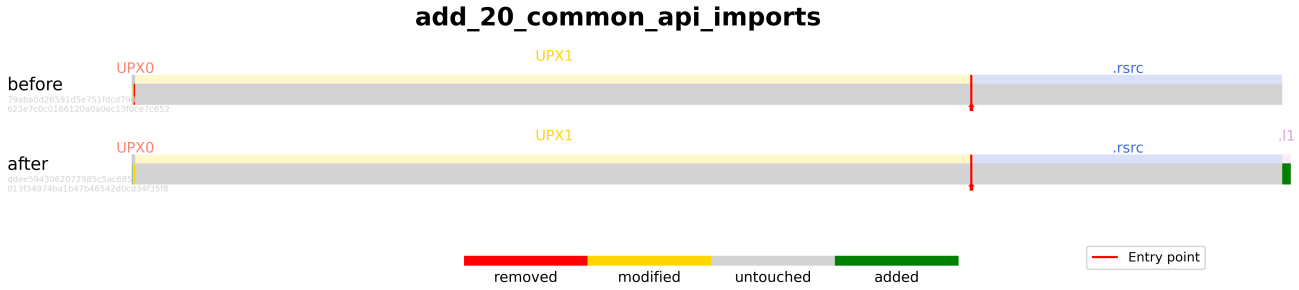


Figure 5.1: Visualization of the effect of `add_20_common_api_imports` alteration

As observed, when adding an API function or a DLL library, LIEF automatically creates a new section named `.l1` to place the new IAT. This effect was also identified by Jennes [2]. When checking the functionality of the altered executable, we encountered an error in Wine, saying "`Library <<snipped>>.DLL not found`". This error occurred because the injected API imports reference DLL that are not installed in the current Wine setup. To address this, we limited the list of common API functions to those in the `Kernel32.dll` library. However, we found that the resulting executable was still **not functional**. Upon further investigation, we discovered that the main cause of this issue is LIEF's rebuilding process. The library fails to patch the original IAT to point to the new location in the freshly added section named `.l1`.

add_low_entropy_text_section : This alteration adds a new code section with low-entropy data and a common name for code sections. If a name is not available, it will use a randomly selected common section name. The purpose of this alteration is to target the entropy features. Many packers have high-entropy sections that can be a clear indicator of encrypted or obfuscated data, which is often associated with packing. By adding a new low entropy `.text` section, we can target the `entropy_of_code_section` feature and affect the overall file entropy as well.

A visualization of the effect on an example PE file is displayed in figure 5.2 where we can spot a new section named `.text` as expected and a modification in the header to add the new entry in the section header.

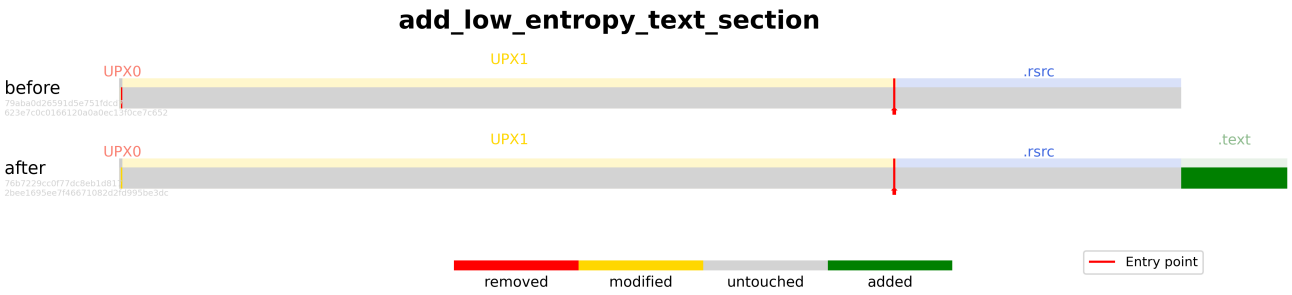


Figure 5.2: Comparison of an executable after applying the `add_low_entropy_text_section` alteration

The resulting executable remains functional as we can notice below.

```
┌──[user@packing-box]──[~]─────────
$ wine upx_du.exe
DU v1.61 - Directory disk usage reporter
Copyright (C) 2005-2016 Mark Russinovich<<snipped>>
```

fill_sections_with_zeros : Appends `0` bytes to sections to stretch their sizes from their raw size to their virtual size. This has been introduced because some packer include small or empty section where the unpacked original binary will be placed, and this unusual size of section also has a bigger virtual size than raw size. This alteration is defined such that it loops through all sections of the input file and fills all sections with 3 randomly chosen bytes repeated up to their virtual size.

It is important to note that the file size increases significantly as shown in Figure 5.3 where we see the effect of this alteration and assess its correctness.

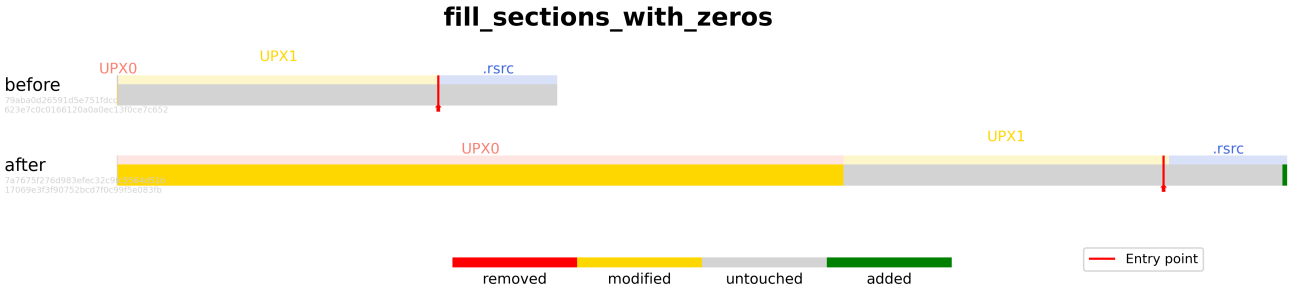


Figure 5.3: Shows the effect of `fill_sections_with_zeros` alteration

The current implementation of the `fill_sections_with_zeros` alteration does not always guarantee that the executable remains functional. Below is an example running the `upx_du.exe` file that have been altered by this alteration. In this example, we observe that the altered executable `upx_du.exe` packed with UPX could not be executed properly, while the altered version of `upx_7z.exe` runs successfully.

```
┌──[user@packing-box]──[~]─────────
$ wine upx_du.exe
<<snipped>>
Application could not be started.
ShellExecuteEx failed: Bad EXE format for upx_du.exe.
┌──[user@packing-box]──[~]─────────
$ wine upx_7z.exe
7-Zip 3.12  Copyright<<snipped>>
```

Further analysis revealed a corrupted row in the section header of the resulting `upx_du.exe` file. This issue led us to investigate the implementation more closely and re-implement the alteration, which we successfully fixed. We thus consider this alteration valid for maintaining the functionality of the binaries.

move_entrypoint_to_new_low_entropy_section : Similar to the `add_low_entropy_text_section` alteration, it will create a new low entropy code section, but also move the Entry Point (EP) to this new section named with a standard name.

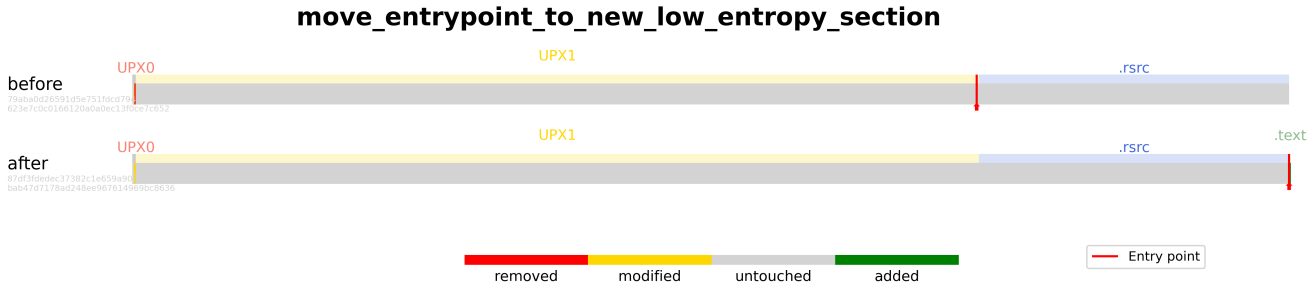


Figure 5.4: The effect of `move_entrypoint_to_new_low_entropy_section` alteration

In figure 5.4, is illustrated the effect of this alteration on a binary file, showing that the entry point has been relocated to a new section named "`.text`".

Note that, currently, this alteration is not resistant to further signature creation. Thus the trampoline code injected in the new Entry Point could be used to generate a signature for static detection.

```
┌──[user@packing-box]──[~]─────────
$ wine upx_du.exe
wine: Unhandled page fault on execute access to 000204A0 at address 000204A0
   (thread 0424), starting debugger...
<<snipped>>
```

In order to test whether the functionality has been maintained, we ran one of the altered executables of the dataset. The tested file loaded successfully but failed to run properly and an error is shown at runtime. This suggests that the injected trampoline code might be erroneous. After thorough analysis of the alteration's internal implementation, we identified the cause of this issue. This will be detailed in the next section.

rename_packer_sections : This alteration renames up to 50 sections, beginning with the entry point section, which is renamed "`.text`" if this name is not already in use. It then targets sections with empty names followed by those with commonly known packer section names, such as `UPX0` or `.aspack`.

In the example shown in Figure 5.5, we observe that section names commonly associated with packers have been replaced with standard section names, and the section already named with a standard name is left as-is. For instance, `UPX0` has been renamed to `.data`, `UPX1` to `.text` and `.rsrc` is left untouched.

To ensure that the executable still runs correctly after this alteration, we execute it as given below. The following console output shows that the executable, after being altered, continues to work as expected:

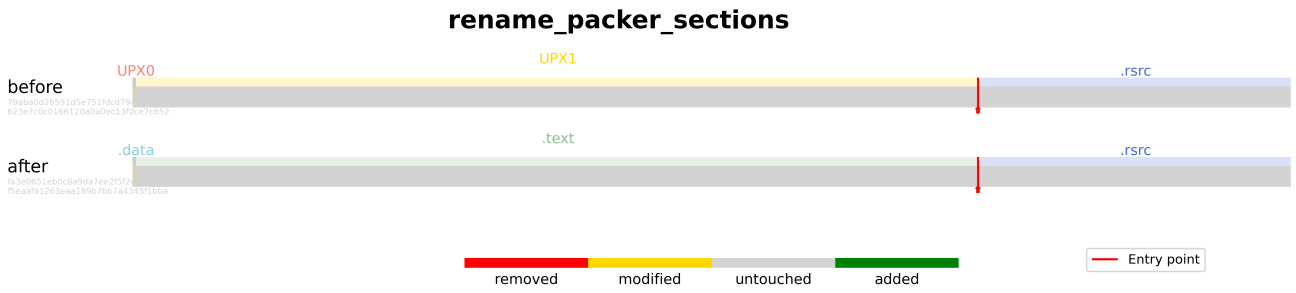


Figure 5.5: Plot of the effect of `rename_packer_sections` alteration

```
┌──[user@packing-box]──[~]──────────
$ wine upx_du.exe
DU v1.61 - Directory disk usage reporter
Copyright (C) 2005-2016 Mark Russinovich<<snipped>>
```

5.2.2 Effect on Features

After verifying the functionality of executables altered by each alterations prior to our research, we now examine the impact of these alterations on the targeted features. To achieve this, we use the information gain plot, generated using the command `dataset plot infogain`.

Information gain is a measure to quantify how much information a feature provides about the given dataset compared to a reference dataset. It allows us to quantify the effect of each alteration on specific features defined by Biondi et al. [3].

This method is essential for understanding how well the alterations perform and identifying areas of improvements.

As highlighted by Jennes [2], the `size_of_header` feature is always affected by alterations. This happens because the original dataset contains incorrect size of header value. When LIEF rebuilds the files, it fixes this value, resulting in a noticeable impact on this feature.

add_20_common_api_imports : When adding 20 common API functions to PE files, we observe the expected impact on features such as the number of functions imported, the number of addresses in the IAT, and the ratio of malicious API imports.

As we can see in Figure 5.6, we also notice several other impacted features related to section permissions and section names. This is due to the added `.l1` section that LIEF creates when rebuilding the IAT. This new section has writable and executable permissions and a non-standard name, which explains the broader effects of this alteration.

add_low_entropy_text_section : This alteration successfully reduces the overall entropy of the file. One of the feature also impacted is the `entropy_code_section` even though the added section does not have the code characteristics, this is because the computation of this feature consider a section named `.text` as a code section without consulting the characteristics of the section. As shown in Figure 5.7, the added section

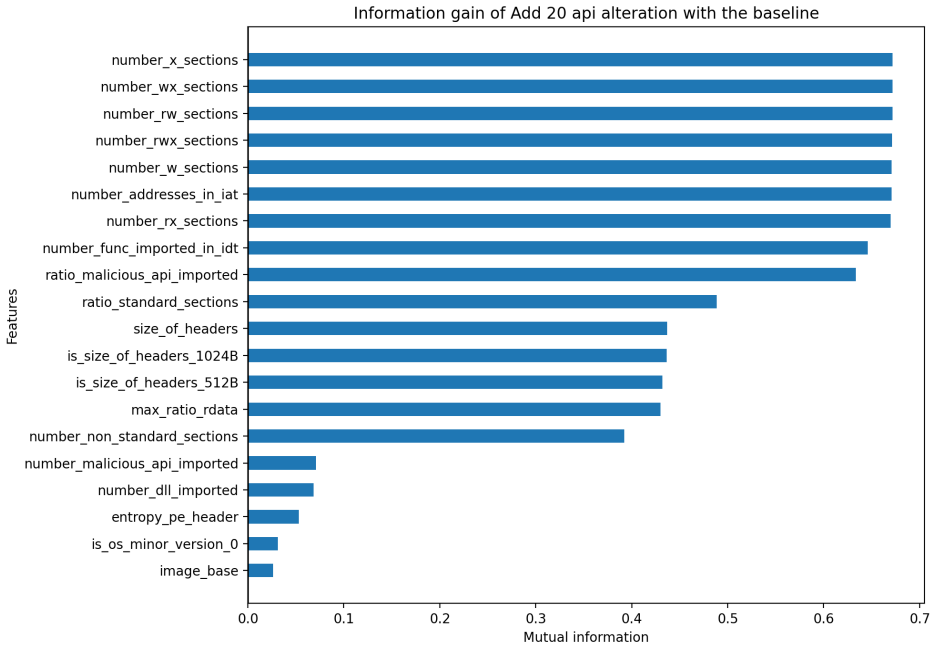


Figure 5.6: Information gain for `add_20_common_api_imports` alteration

has a very low entropy compared to others, and the average entropy drops from 7.03 to 6.51 and the overall entropy drops from 7.75 to 7.56. This would impact the Bintropy detector from *Lyda and Hamrock* [30] as they consider an executable as packed if the average entropy falls in the range between 6.677 and 6.926, and now we have 6.51. However, the entropy threshold of *Han et al.* [31] would still consider it as packed because they use the entropy of the EP section.

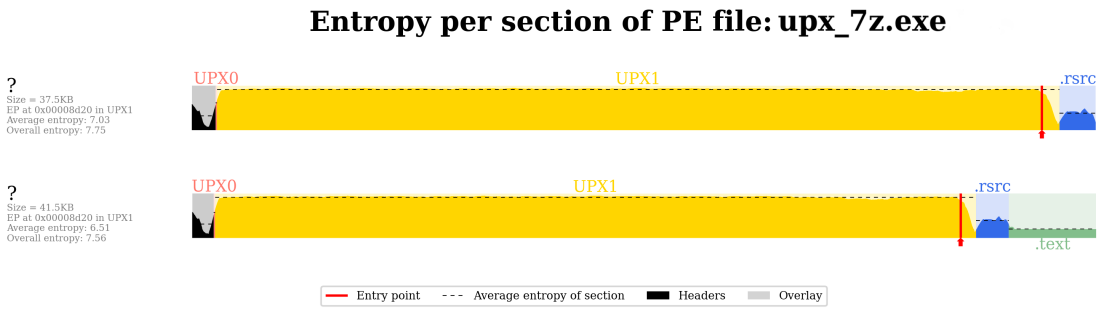


Figure 5.7: Entropy of an executable after applying `add_low_entropy_text_section` alteration

fill_sections_with_zeros : As Figure 5.8 illustrates, this alteration targets mainly the following features :

- **number_sections_size_0** : Number of sections having their physical size (on disk) equal to zero
- **number_sections_vsize>dsize** : Number of sections having their virtual size greater than their raw data size
- **ep_ratio** : Ratio between raw data and virtual size for the entry point section
- **(max/min)_ratio_rdata** : Maximum/Minimum ratio raw data per virtual size among all the sections

This alteration also causes a noticeable increase in file size. We can see this difference by comparing the original executable with the altered version using the `du` (Disk Usage) command. In the following example, the file size increased by +160% of its initial size.

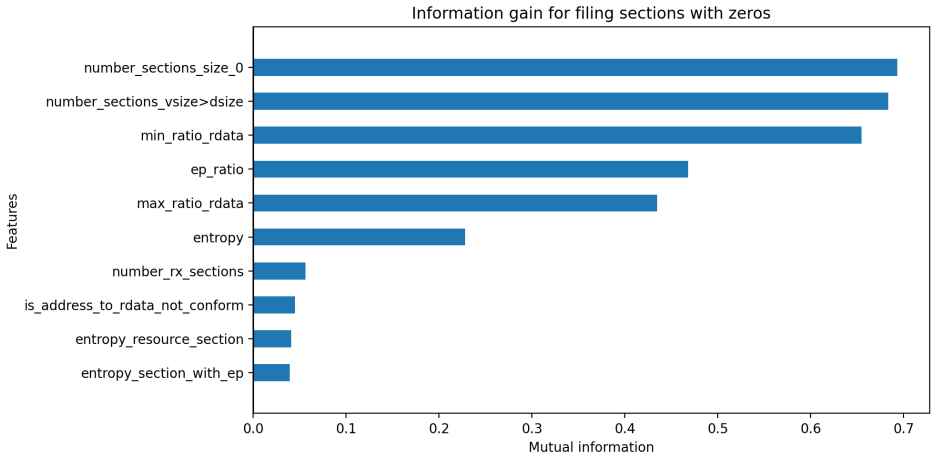


Figure 5.8: Information gain for `fill_sections_with_zeros` alteration

```
┌──[user@packing-box]──[~]──────────
$ du -h upx_7z.exe
40K upx_7z.exe
┌──[user@packing-box]──[~]──────────
$ du -h upx_7z_altered.exe
104K    upx_7z_altered.exe
```

move_entrypoint_to_new_low_entropy_section : This alteration targets mostly features related to the first bytes after the Entry Point and the EP related features such as `is_ep_not_in_standard_section` and `is_ep_not_in_code_section`.

As we can visualize in figure 5.9, the alteration produces samples having all their first byte after EP equal to `0xE9`. This byte correspond to the opcode of the jump instruction (`JMP`) found in the trampoline code injected to jump back from the current Entry Point (EP) to the original EP. The effect on this feature is problematic because it creates a recognizable pattern that allows the `byte_0_after_ep` feature to be used to detect our altered samples easily.

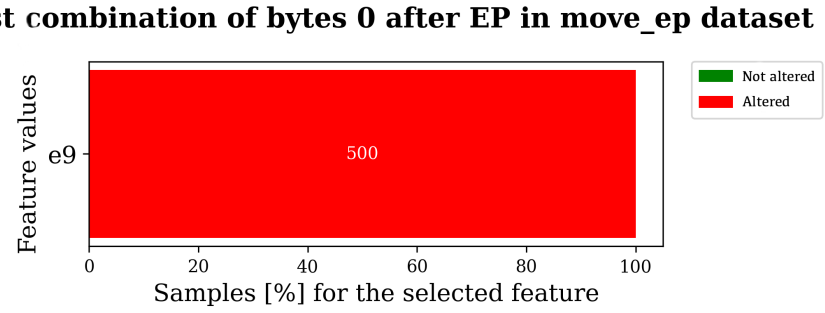


Figure 5.9: Byte after EP for the Move EP alteration

rename_packer_sections : The most impacted features by this alteration are the number of standard and non-standard sections, as well as the ratio of standard sections in the file. This is because we rename all sections, causing all previously non-standard section names to become standard ones. The other features, shown in Figure 5.10, are also affected due to feature computations being based on section names rather than the characteristics of the code itself.

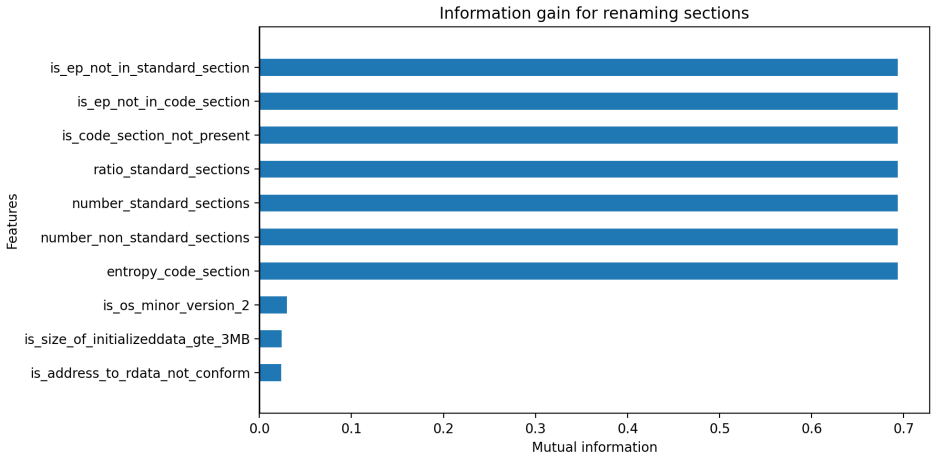


Figure 5.10: Information gain for `rename_packer_sections` alteration

Overall Overview To visualize the overall impact of all alterations on the targeted features, we use the command `dataset plot infogain-compare` to generate a side-by-side comparison of the normalized information gain for each altered dataset. The resulting heatmap, shown in Figure 5.11, illustrates the significance of each feature: darker colors indicate features that have a significant impact on detecting and identifying the dataset, while lighter colors indicate features that are less significant or not significant at all. Zeros across all features would mean the samples are indistinguishable from unpacked samples.

We observe in this Figure 5.11 that the alteration that has the most impact on a wide range of features is the "`move_entrypoint_to_new_low_entropy_section`".

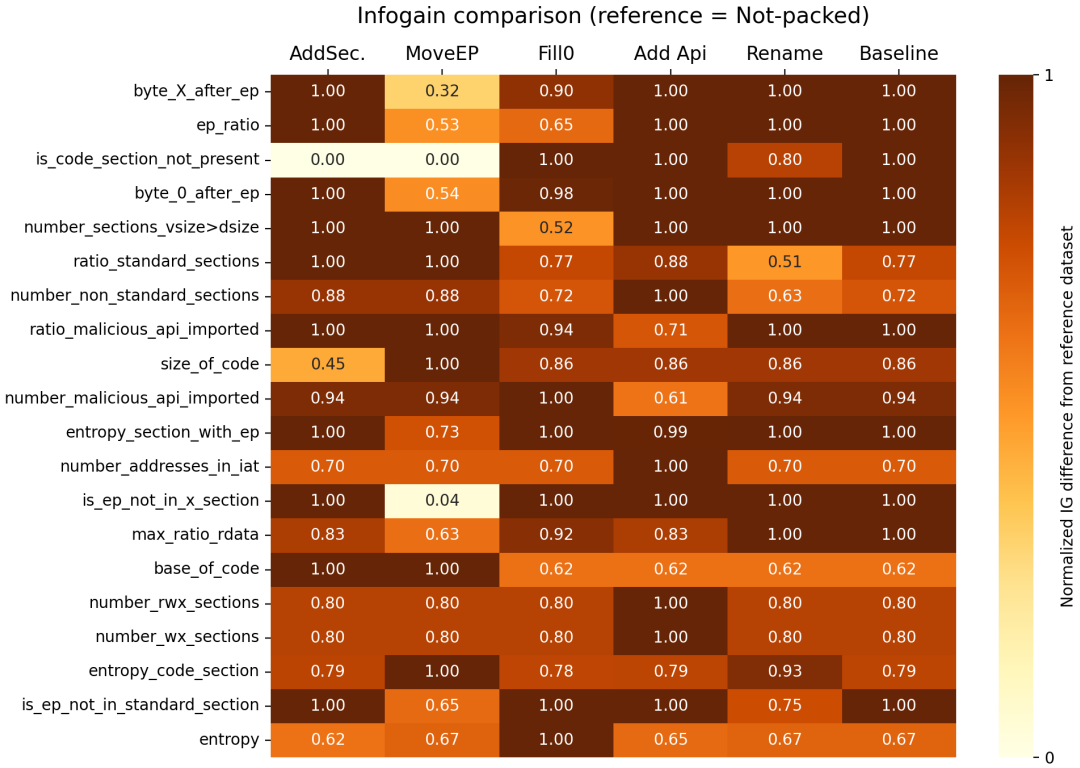


Figure 5.11: Information gain comparison for all existing alterations

Table 5.2 presents an overview of the targeted features and whether the alteration maintain the executable functional. From the five existing alterations, *adding api imports* and *moving the Entry Point (EP)* broke the executable functionality, where the 3 others keeps binary functional.

Alteration	Functionality	Features
add_20_common _api_imports	✘ Broken	Number of `RWX` sections, number of addresses in IAT and imported functions, and ratio of malicious API imported
add_low_entropy _text_section	✔ Functional	File entropy, entropy of code section, number of executable sections and ratio of standard sections
fill_sections _with_zeros	✔ Functional	Number of sections with size 0, number of section having virtual size bigger than raw size and (min/max) ratio of raw data.
move_ep_to_new _low_entropy_section	✘ Broken	Byte 0 after EP, bytes X after EP, EP (not) in standard/code section
rename_packer_sections	✔ Functional	Number and ratio of (non) standard sections, EP (not) in standard section and EP not in code section

Table 5.2: Existing alterations overview of targeted features and functionality preserving validation

5.2.3 Alteration Improvement

In section 1.3, our RQ2 posed the question: "*How to optimize alterations and can we add new ones?*". This research question focuses on improving the existing alterations and introducing new ones to enhance the current findings. In this section, we will explore possible improvement of the current implementation of alterations.

Before improving the `move_entrypoint_to_new_section` alteration, we first fixed it as it did not maintain the executable functionality. The interested reader can find more details about the fix of this alteration in the Appendix A.

Improving move_entrypoint_to_new_section In the newly fixed version, we now have an alteration that maintain functionality of 32-bit executables as well as 64-bit. However, this injected code might be used as a signature because the bytes after Entry Point are fixed, as shown previously in Figure 5.9.

To handle this potential weakness of the alteration, we introduce randomization into the first few bytes after the Entry Point, this would impact features such as `byte_X_after_ep` and perturb the effect seen in Figure 5.9 and Figure 5.13 (a).

This approach involves injecting dead code before the jump of the trampoline. Dead codes are instructions that does not affect the program's functionality. Examples of dead code, as shown in Listing 5.1, include operations like XOR-ing a value with itself, or pushing and popping values from the stack without any purpose.

```
0x90                    # NOP
0x33, 0xDB              # XOR EBX, EBX
0x83, 0xC0, 0x00        # ADD EAX, 0
0x87, 0xC0              # XCHG EAX, EAX
0x89, 0xC2              # MOV EDX, EDX
<<snipped>>
```

Listing 5.1: Example of instructions `DEAD_CODE`

In our new version, we start with the same implementation as the fixed `move_entrypoint_to_new_section` alteration described in Listing A.2 of the Appendix A.1, except that we inject a random number of dead code instructions after the EP and before the trampoline jump. Furthermore, we identified a recurring pattern in non-packed executables, as illustrated in Figure 5.12, which displays the most common combinations of values for bytes 0, 1, and 2 following the Entry Point in a dataset of non-packed binaries. Over 60% of the selected top ten have their first 4 bytes equal to `0x83 0XEC 0x0C`.

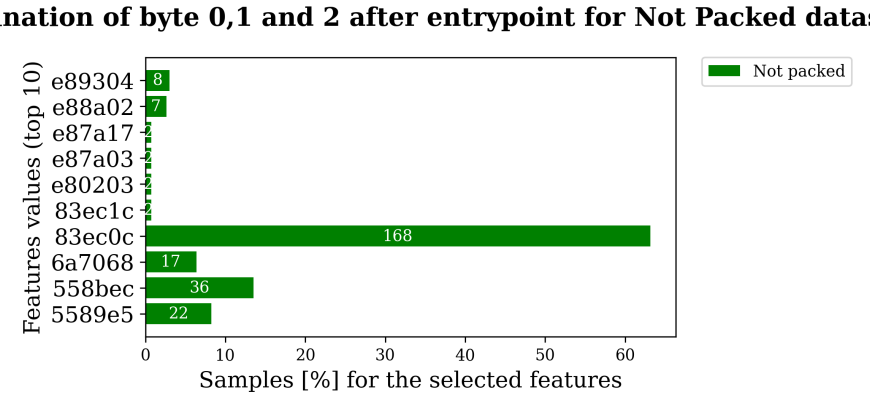


Figure 5.12: Most combination of bytes 0,1 and 2 after the EP for not packed dataset

The bytes shown in Figure 5.12 were analyzed and studied to understand their purpose. We identified that they correspond to a procedure that reserves space on the stack using the instruction `SUB esp, 0xc`. This instruction subtracts 12 bytes from the stack pointer (`esp`), effectively allocating space for local variables or other data needed during the executable's execution. Additionally, we observed that around 20% of the bytes are composed by `0x55`, followed by either `8b ec` or `89 e5`. The first byte corresponds to a `PUSH ebp` instruction, and both subsequent combinations are opcodes for `MOV ebp, esp`. These two instructions form a typical prologue at the start of a function in assembly language. This prologue may also include stack reservation for local variables. A complete example is provided in Listing 5.2. This sequence saves the base pointer (`ebp`) and sets it to the current stack pointer (`esp`), establishing a stable stack frame and ensuring the correct execution flow.

```
1 push ebp              # 55
2 mov ebp, esp          # 89 e5 / 8B EC
3 sub esp, 0xc          # 83 ec 0c
```

Listing 5.2: Assembly prologue instructions

We might also observe an epilogue procedure at various positions in the same binary files. This epilogue restores the stack pointer to free the space reserved in the prologue for local variables. It does that by doing the opposite operation, illustrated in Listing 5.3, it starts by moving `ebp` into `esp` to restore the stack pointer to its original value and POP-ing from the stack the original value of `ebp`.

```
1 mov esp, ebp          # 89 EC
2 pop ebp               # 5D
3 ret                   # C3
```

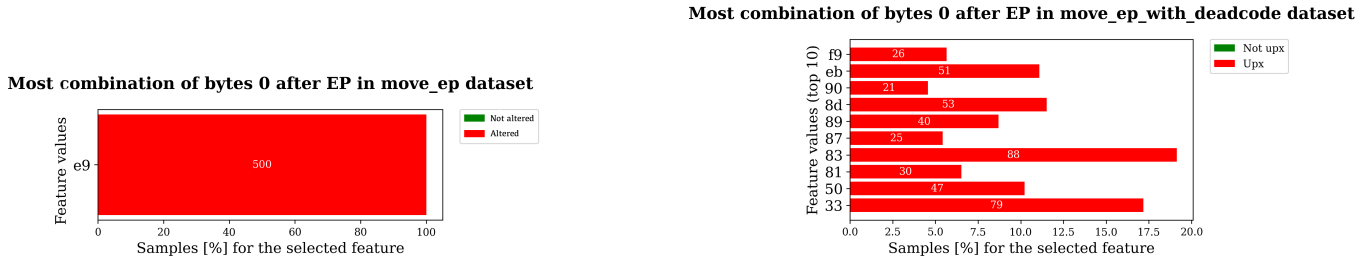
Listing 5.3: Assembly epilogue instructions

As these bytes are often found in not packed executables, we thus introduce these typical bytes on the beginning of the EP to mimic not packed binaries. We include the `SUB esp, 0xc` bytes by default, and we randomly choose whether to add in front of it the prologue instructions, this introduces randomized pattern, but typical for not packed executables.

An example of the Entry Point content in an altered binary file is presented below, illustrating 4 injected dead code instructions followed by the trampoline jump instruction. We can observe that the jump instruction uses the new offset jump technique discussed previously. The bytes of the jump instruction include `0xE9` for the jump opcode, followed by a signed 32-bit offset in little endian format. In the example below, the offset is "`09 e9 ff ff`", which corresponds to `0xfffe90e` and converts to **-5874** in decimal. The *objdump* tool automatically converts this to the estimated address during decompilation, allowing for a quick assessment of the offset's correctness and the added dead code instructions.

```
┌──[user@packing-box]─[~]──────────
$ objdump upx_7z.exe -d | grep ".text" -A 5
0041b000 <.text>:
  41b000:       81 e6 ff ff ff ff       and    \$0xffffffff,\%esi
  41b006:       05 01 00 00 00          add    \$0x1,\%eax
  41b00b:       2d 01 00 00 00          sub    \$0x1,\%eax
  41b010:       87 c0                   xchg   \%eax,\%eax
  41b012:       e9 09 e9 ff ff          jmp    0x419920
```

To validate the effectiveness of this improvement on the targeted features, we illustrate in Figure 5.13 the feature values of byte 0 after EP. The plot (b) clearly shows that we have successfully perturbed this feature, resulting in a much more distributed set of values compared to the plot (a) before improvement.



(a) Before

(b) After improvement with dead code injection

Figure 5.13: Byte after EP for `move_ep` before and after improvement

The second variant of adding randomly prologue and epilogue code as first bytes also successfully mimic the distribution of the not packed dataset. This is observed when comparing Figure 5.14 to the non packed dataset distribution shown in Figure 5.12.

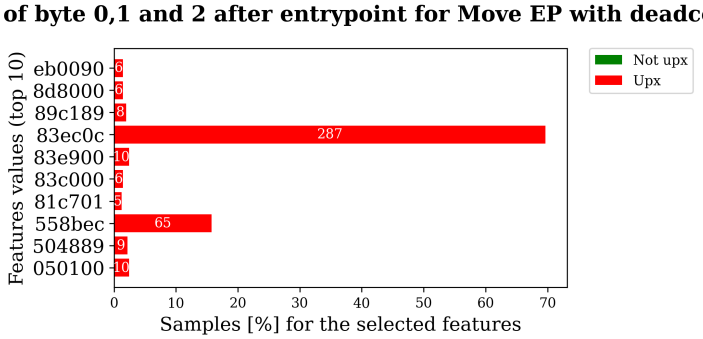


Figure 5.14: Most combination of bytes 0,1 and 2 after the EP for altered dataset

An essential step of our experiments after introducing new elements into alterations is to ensure the executable altered remains functional. To validate this, we tested an executable modified by the new version and confirmed it runs perfectly and remains thus functional as shows this correct output below. We also validated the alteration on a x64 executable and it remains functional as well.

```
┌──[user@packing-box]──[~]──────────
$ wine upx_7z.exe
7-Zip 3.12  Copyright<<snipped>>
┌──[user@packing-box]──[~]──────────
$ wine upx_SimpleTest.exe
SUCCESS
```

We also validated our alterations maintain the executable functionality in an ASLR environment. To do that we first need to add the DLL characteristic named "**DYNAMIC_BASE**" to the PE binary file. This flag tells the loader to ignore the executable's preferred address and load it at a random address instead. Next, we executed the file on a Windows machine and observed a successful run.

Our improved `move_entrypoint_to_new_section` alteration successfully maintained the file's functionality for packers such as Eronana, UPX, and PEtite. However, we discovered that TELock fails to run correctly, displaying before quitting an integrity check warning in a popup GUI as shown in Figure 5.15. Even after repacking it with `UPX`, which kept it functional, applying our alteration caused it to fail again. This is because **`TELock`** includes anti-debugging techniques and integrity checks.

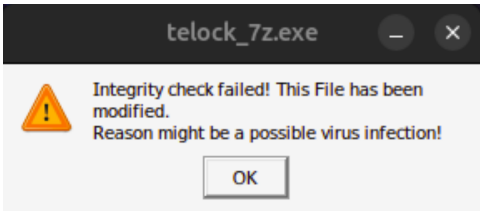


Figure 5.15: TELock integrity check warning message after alteration

FINDINGS

- Alteration *move_entrypoint_to_new_section* was breaking the executable functionality and was fixed with a more reliable solution compatible with x64 files and valid in an ASLR enabled environment.
- Alteration *move_entrypoint_to_new_section* had its first bytes fixed to a single value and thus makes it recognizable. We thus added some **randomization** in the first X bytes after the EP using dead code instructions, which successfully perturbed this recognizable pattern.
- Some packers, TELock for example, runs integrity checks so any alteration applied on it will break the functionality. This issue rises a new area of research to bypass this integrity check.

5.3 New Alterations

After parsing all previously defined alterations, we explore new possibilities to enhance and extend the alteration set already existing by adding new alterations.

5.3.1 Change Section Raw Size

Alteration `fill_sections_with_zeros` aims to target features such as `number_sections_size_0`, `number_sections_vsize>dsize`, `ep_ratio`, `(min/max)_ratio_rdata` and slightly the `entropy` of the file. However, as demonstrated in previous section, the size on disk of the altered file is significantly increased by more than 150% of its original size.

In order to introduce a new alteration and ensure it makes packed executables look like a non-packed ones, we first studied the targeted features and typical values for non-packed binaries. Figure 5.16 shows the number of section having their virtual size greater than its raw size. We observed that the majority of non-packed executables have a value of 1 for this feature. This section is typically the `.data` section, which reserves space in memory for uninitialized variables. Additionally, a few samples from the PECompact packer exhibit the same value as the non-packed dataset, but this might just be an error in parsing because manual analysis showed us 100 PECompact samples with a value of 2.

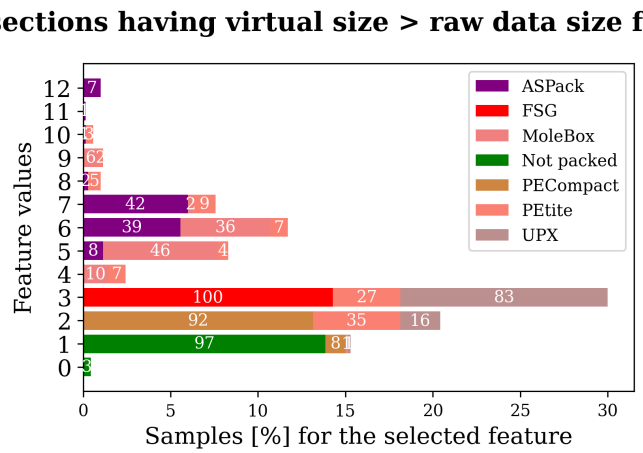


Figure 5.16: Number of sections with virtual size greater than raw size for mixed dataset

For the feature `number_sections_size_0`, Figure 5.17 illustrates the values for a mixed dataset consisting of 600 samples, equally distributed with 100 samples from each of 6 different packers, along with 100 non-packed samples. We observe that the non-packed dataset has values equally distributed between 0 and 1 section having a raw size equal to zero.

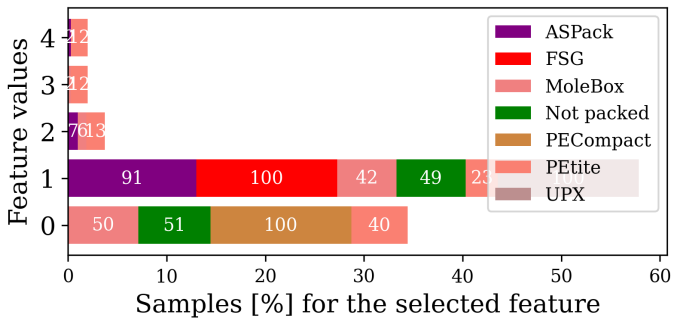


Figure 5.17: Number of sections with raw size equal to 0 for mixed dataset

In order to target these features while keeping the file size relatively low, we were inspired by the work of Nisi et al. [46]. Their research explores erroneous header values that are still accepted by OS loaders. This approach provided a valuable idea for our alteration, and we discovered that the section's raw size value in the section header is also not well checked by loaders. Next we verified our intuition by modifying these values by slightly increasing them and validating that the functionality remains unchanged. We continued this process until the executable is not functional anymore to study the maximum allowed values and the checks performed by loaders.

We found that the maximum raw size of a section is determined by the space between the start of the section and the end of the file, rather than up to the end of the next section as expected. Therefore, to add more raw size, we need to increase the file size by extending the next sections or by adding bytes in the overlay of the file. This ensures that the theoretical end of the section does not fall outside the raw size of the file, maintaining the executable's functionality.

This new observation highlights that the resulting executable has overlapping sections. We also discovered that this overlap causes the overlapping data to be duplicated in memory, with one copy for each section.

This leads us to introduce the alteration **`change_section_raw_size`**, that changes the header value 'raw size' of sections having a raw size equal to zero, to match the maximum allowed value. Another approach, not selected for our alteration, consist of setting the value to the virtual size of that section by adding data to the overlay to keep the executable functional.

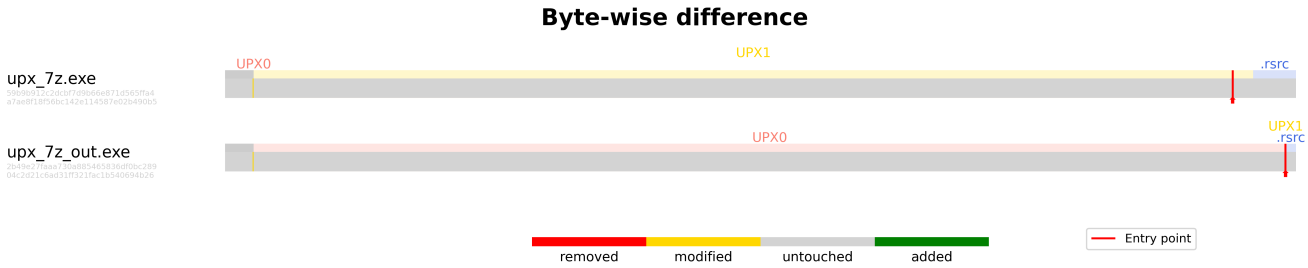


Figure 5.18: Visualization of an executable after altering raw size of 0 sized sections

Figure 5.18 illustrates the effect of this alteration on a binary file. A tiny yellow line is visible on the left part representing the modification made to the header. Interestingly, the visualization tool was tricked by this alteration and its overlapping sections, perceiving the `UPX0` section as having a large size, displaying other sections collapsed together and moving the entry point where in reality it did not move. A more reliable visualization should display sections positions based on their raw address rather than one after the other. This observation is interesting as parser used to parse PE files in detectors might also be tricked by this alteration.

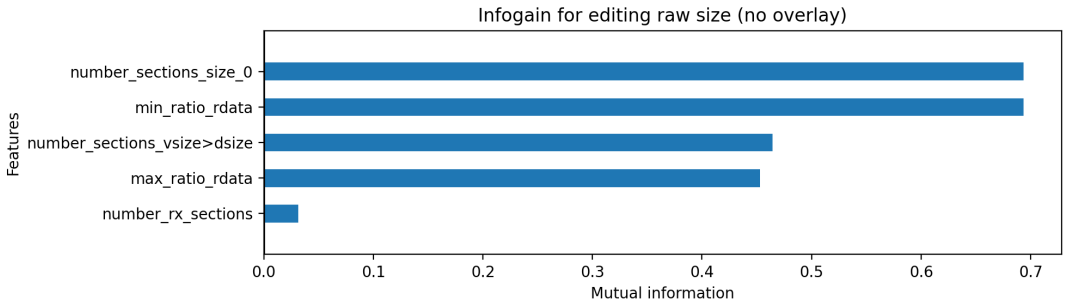


Figure 5.19: Information gain of editing the raw size

We observe in Figure 5.19 that we successfully target the same features as `fill_sections_with_zeros`. However, we found out that this alteration might trigger another feature from the work of Choi et al. [36], `is_sum_of_all_sections>file_size`, and this feature it is not present in the 119 most significant features selected by Biondi et al. [3]. This feature sums all sections' sizes and triggers if this sum is bigger than the file size. In order to counteract that feature and keep the file functional, we only edit the raw size up to the maximum allowed. Another approach consist of adding to the overlay of the file the the same amount of the difference between the maximum allowed raw size value and the actual new value edited.

We altered an example executable by only editing its raw size value from 0 to the maximum allowed to remain functional, and we successfully validated the functionality preserving feature of our alteration.

```
┌──[user@packing-box]──[~]──────────
$ wine upx_7z_raw_size_edit.exe
7-Zip 3.12  Copyright<<snipped>>
```

The LIEF library was not able to correctly perform this alteration, because when changing the raw size value, LIEF automatically adds bytes to fill and fit the new section size. Because of this we built our own implementation of a simple PE parser to edit the section header value without impacting the size of the file and breaking the executable. This implementation is detailed in Appendix D.3.1.

5.3.2 Super-Alteration

In the previously defined alterations, the affected features did not include the number of writable and executable sections, except for the `add_20_common_api` alteration, which adds a new `rwx` section and increases this feature value. This feature is one of the 119 most significant features identified by Biondi et al. [3] and Bertrand Van Ouytsel et al. [4]. As shown in Figure 5.20, the features `number_(r)wx_sections` are still significant for distinguishing between non-packed binaries and packed samples altered by existing alterations.

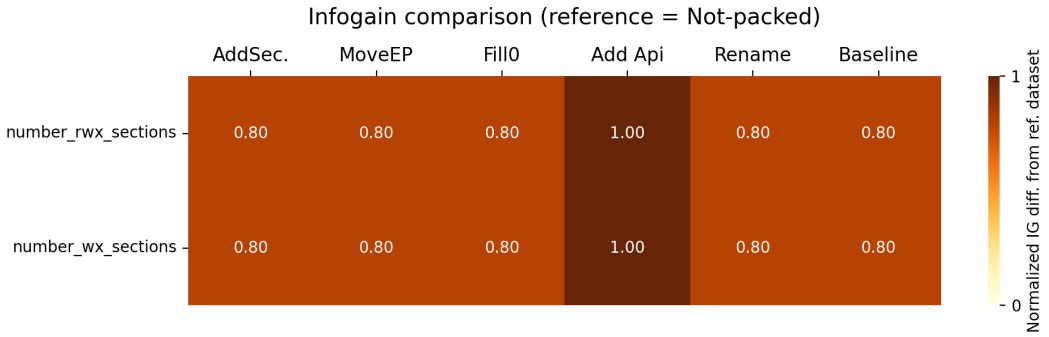


Figure 5.20: Information gain comparison of feature `number_(r)wx_sections` for existing alterations

As illustrated in Figure 5.21, non-packed binaries have zero writable and executable sections. Interestingly, the Amber packer and a few samples from PEtite also show this feature value as zero, while other packers represented in the figure have higher values. This is a significant feature for detecting packing since the unpacking process typically requires a writable section to store the original binary and execute it.

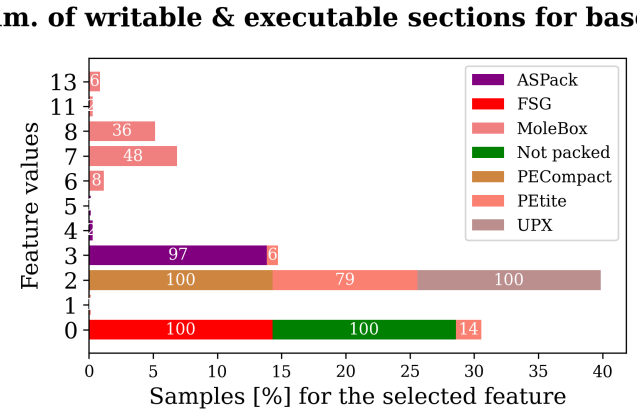


Figure 5.21: Number of writable and executable sections in a mixed dataset

Since non-packed binaries typically have zero sections with both writable and executable permissions, we have introduced a new super alteration `change_sections_permission`, targeting this feature. The term of super alteration is used as the transformations included are multiple, and involves moving the entry point to a new section, renaming the sections and editing sections permission. Our main objective is to also achieve a value of zero writable and executable sections in the resulting altered files.

In order to target that feature, we performed analysis of Amber packed executables using a decompiler such as *Ghidra*, and uncovered their approach to keeping the feature value at zero. This led us to the intuition that we need to change the permissions of these sections dynamically during execution to ensure the unpacking stub remains functional.

Our approach involves using the `VirtualProtect` function to adjust the memory protection dynamically during execution. To perform this modification of permissions in memory, we thus use the function `VirtualProtect` [2] from the `Kernel32.dll` library. The first step of our alteration was to add this function to the Import Address Table (IAT) if not already present. We then we referred to the documentation [2] to learn more about the usage of this function and its definition, as listed below 5.4.

```
BOOL VirtualProtect(
  [in]  LPVOID lpAddress,
  [in]  SIZE_T dwSize,
  [in]  DWORD  flNewProtect,
  [out] PDWORD lpflOldProtect
);
```

Listing 5.4: VirtualProtect function definition

In the definition given in listing 5.4, the parameter **`lpAddress`** specifies the starting address of the region to be changed. **`dwSize`** indicates the size of the region to be changed. **`flNewProtect`** sets the new memory protection option to be applied on that region. The most important value we will use is `PAGE_EXECUTE_READWRITE` (`0x40`) which enables *execute*, *read-only*, or *read/write* access to the region. Other possible values are given in the documentation of Microsoft [3].

The last parameter **`lpflOldProtect`** is the address of a variable that will receive the previous access protection value of the specified region. Note that if this parameter is `NULL` or not a valid variable, the function will fail.

To provide the Virtual Address (VA) of the target section, we need the to know the image base where the executable have been loaded into memory. This step is necessary because address space layout randomization (ASLR) causes the executable to load at a random address rather than the preferred image base value. This is achieved by making a function call in assembly. The `CALL` instruction automatically pushes the current address onto the stack to be used as a return address. Within the called function, we retrieve this return address from the stack using the `POP` instruction and subtract the known RVA of the calling instruction. This calculation gives us the image base, which is the actual address of the first byte of the image loaded into memory.

While the CALL/POP technique is efficient, it may also be flagged as typical malware behavior. Since addressing this issue is beyond the scope of our research, we leave it for future exploration.

[2] https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotect
[3] https://learn.microsoft.com/en-us/windows/win32/Memory/memory-protection-constants

We still need the address of `VirtualProtect` function to be called, this can be found as a fixed value in the IAT, and be easily retrieved by LIEF. We also move the entrypoint to a new section to edit target sections with this modification of permissions, and we use the same trampoline approach used in the alteration `move_entrypoint_to_new_section` (JMP <offset>).

Now that we have defined all the components of our new alteration, we can assemble them together.

1. Parse all sections and update permissions:
   - For sections with standard names, we verify and update their permissions to match the typical permissions of standard that section name.
   - For sections with non-standard names, we rename them to a standard section name with matching permissions. If no matching permissions are found, we simply rename the section and update the permissions accordingly.
2. Append assembly bytes for `VirtualProtect` call:
   - For all sections whose permissions have been updated, we append the assembly bytes for the `VirtualProtect` call to the data to be injected. This call will restore each section to its original permissions.
3. Prepare the trampoline code and assembly calls:
   - We first insert the `POP` instruction to retrieve the image base.
   - Next, we append the assembly bytes with all the `VirtualProtect` calls, that restores sections permission.
   - Then we add the trampoline code with the JUMP using the offset technique as described in the `move_entrypoint_to_new_section` alteration.
4. Insert prepared bytes:
   - Insert all the prepared bytes at the start of a new `.text` section, and before the entry point.
5. Inject the actual entry point bytes:
   - Finally, we inject the actual entry point, which contains dead code instructions and the `CALL` instruction. This calls the start of the section to execute the code that restores the permissions and jumps back to the original entry point of the file before alteration.

Here is an example of the bytes injected by this alteration, presented in Listing 5.5. The entry point shows dead code followed by a CALL instruction using an offset as an operand. At address `0x41b076`, the CALL instruction calls the function at address `0x41b000`, which uses POP to get the caller's return address (`0x41b076` + `0x5`) and then subtracts `0x1b07b` to successfully set `EDI` to the image base `0x400000`.

After retrieving the image base, the code initiates the `VirtualProtect` calls. This involves pushing the parameters onto the stack in reverse order (last parameter first) and calling the function by using the image base to calculate its Virtual Address. Finally, the code jumps to the original entry point.

Note that here because we have retrieved the image base dynamically, we could use the PUSH/RET technique discussed previously. For simplicity, we kept the trampoline code as an offset jump.

```
Disassembly of section .text:

0041b000 <.text>:
  41b000:        5b                           pop    ebx          # 0x41b07b
  41b001:        b8 7b b0 01 00               mov    0x1b07b,eax
  41b006:        29 c3                        sub    eax,ebx
  41b008:        8b fb                        mov    ebx,edi
# VirtualProtect call
  41b00a:        83 ec 04                     sub    0x4,esp      # local var.
  41b00d:        54                           push   esp          # lpflOldProtect
  41b00e:        68 40 00 00 00               push   0x40         # flNewProtect (rwx)
  41b013:        68 00 00 01 00               push   0x10000      # dwSize
  41b018:        b8 00 10 00 00               mov    0x1000,eax   # RVA of target
  ↪  section
  41b01d:        01 f8                        add    edi,eax      # Add imagebase
  41b01f:        50                           push   eax          # lpAddress (VA)
  41b020:        b8 50 a4 01 00               mov    0x1a450,eax  # VirtualProtect RVA
  41b025:        01 f8                        add    edi,eax      # Add imagebase
  41b027:        ff 10                        call   *(eax)       # CALL
# ... <<snipped>>
# Jumping to the OEP
  41b067:        e9 b4 e8 ff ff               jmp    0x419920     # Trampoline
# >>> Current EP  # (Dead code until CALL)
  41b06c:        90                           nop
  41b06d:        90                           nop
# ... <<snipped>>
  41b076:        e8 85 ff ff ff               call   0x41b000
```

Listing 5.5: Disassembly of section text section after applying the new alteration

We can visualize the effect of this alteration as shown in Figure 5.22, which depicts the comparison of the visualization of an executable before and after applying the alteration. We observe that a new section named `.text` have been added and contains the moved entrypoint. We also notice a change in section names and a modification in the header, because the renaming of sections and the displacement of the entrypoint is done by modifying the header.

We can visualize the effect of this alteration in Figure 5.22, which compares an executable before and after the alteration. We observe that a new section named `.text` has been added and contains the moved entry point. Additionally, we notice changes in section names and modifications in the header due to the renaming of sections and the relocation of the entry point.

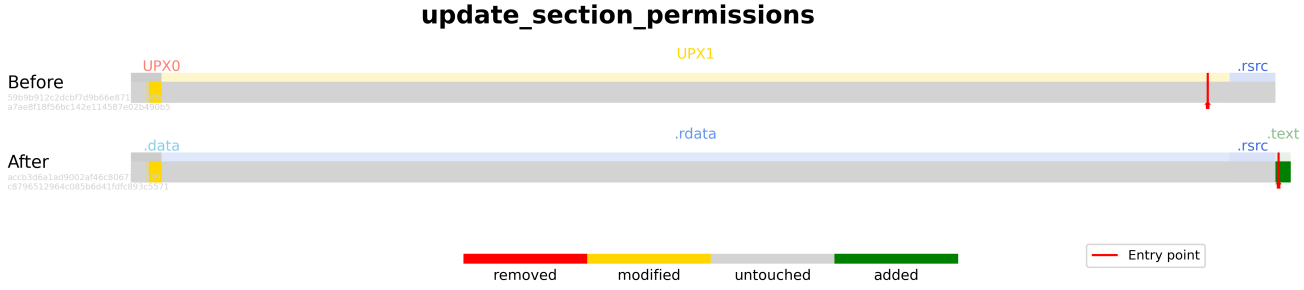


Figure 5.22: Visualization of the effect of the alteration update_permissions on binaries

Our alteration was validated by ensuring the altered executables remained functional and affected the target features. We observed that the resulting binaries remained functional as expected, and the permissions were correctly restored dynamically, as demonstrated in Figure 5.23, where we used the debugger XDBG to visualize the sections permission during execution.

Address	Size	Info	Protection
00400000	00001000	upx_7z_out.exe	-R---
00401000	00010000	".data"	-RWC-
00411000	00009000	".rdata"	-R---
0041A000	00001000	".rsrc"	-R---
0041B000	00002000	".text"	ER---

(a) Before (State at entrypoint)

Address	Size	Info	Protection
00400000	00001000	upx_7z_out.exe	-R---
00401000	00010000	".data"	ERWC-
00411000	00009000	".rdata"	ERWC-
0041A000	00001000	".rsrc"	-RWC-
0041B000	00002000	".text"	ER---

(b) After (State after trampoline)

Figure 5.23: Visualization on a debugger of section permissions changes during a single run

```
┌──[user@packing-box]──[~]
└─$ wine upx_7z_out.exe
7-Zip 3.12  Copyright<<snipped>>
```

To analyze the features affected by this alteration, we compute the information gain of the altered dataset against a non-packed reference dataset. This allows us to identify features that are significant for distinguishing altered packed binaries from non-packed ones. In Figure 5.24, we compare the information gain of this new alteration with the baseline dataset and the `move_entrypoint_to_new_section` alteration (both with and without dead code injection). The results show that our alteration successfully impacts the targeted features, especially the `number_(r)wx_sections` feature, reducing the normalized information gain from 1 to 0.

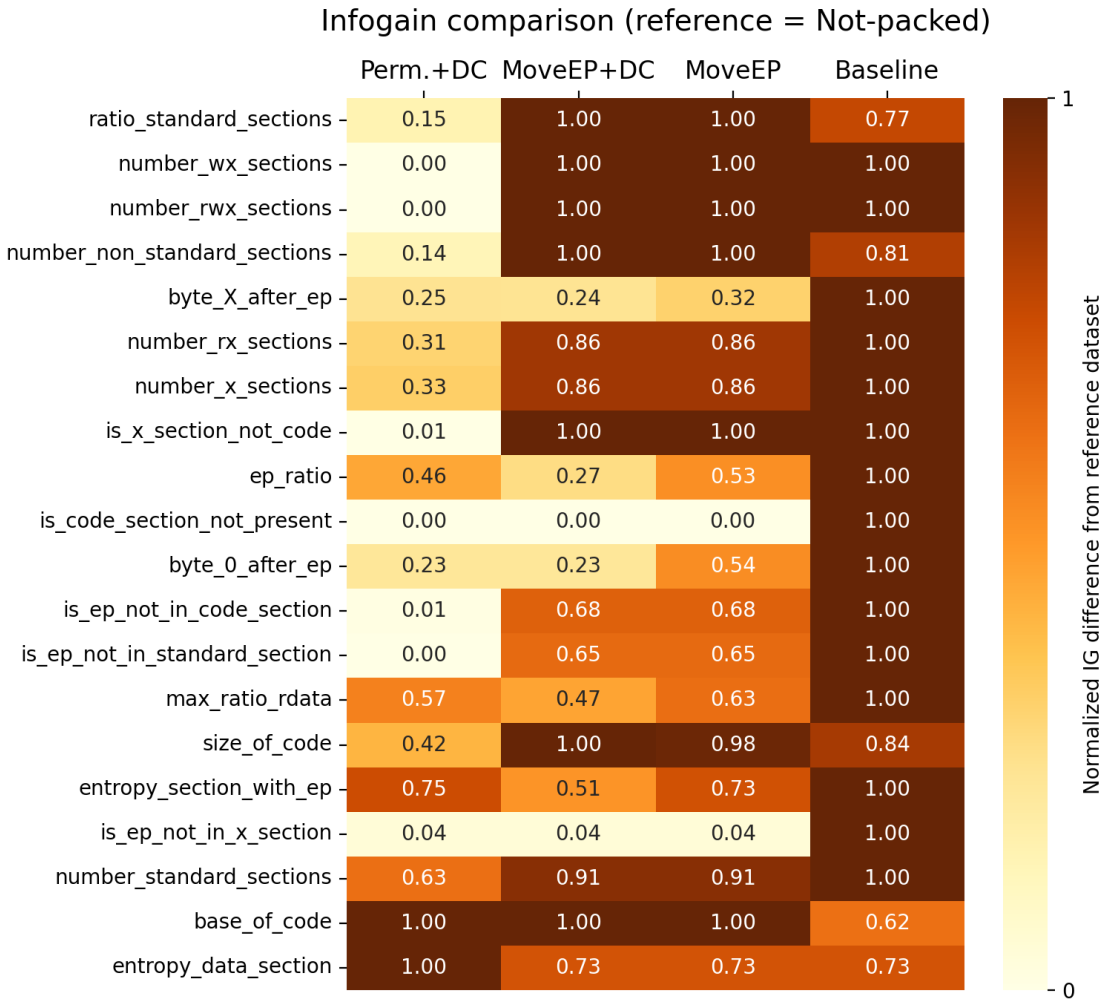


Figure 5.24: Infogain comparison of changing section permissions and move EP with randomization

FINDINGS

- Our alteration `edit_raw_size` successfully targets the expected features without increasing the file size, unlike the `fill_sections_with_zeros` alteration, which significantly increases the file size.
- LIEF automatically increases the section size by adding real bytes when changing the section header size value.
- Previous alterations did not target features related to section permissions. We introduced a new **super alteration** to successfully target these features without breaking the executable's functionality.

5.4 NotPacked++

Our adversarial tool implements the original alterations with the introduced improvements and the new alterations presented previously. We thus used the tool to efficiently apply all alterations to samples in a baseline dataset. To constitute our baseline dataset, we gathered 600 samples, where each 100 samples are packed with different packer, as detailed in Section 5.1.2 Next, we studied and compared the effect of all alterations on features. Additionally, we studied the effect on static detectors and assess the effectiveness of our tool.

5.4.1 Setup

NotPacked++ is used by providing the file path of the input file followed by various options to select the alterations to be applied. The command used for one single file is given below.

```
┌──[user@host]──[~]─────────
$ notpacked++ upx_7z.exe --fill-sections --permissions -o upx_7z.exe
<<snipped>>
```

However, the tool in its initial version does not accept a directory as input. Because of that, in order to apply alterations on a dataset of samples, we define an external script that automates the usage of the tool for each file in a given folder. The implementation of this script is presented in Listing 5.6.

```
#!/bin/bash
# apply the 'notpacked++' tool (command) to all files in the folder
if [ -z "$1" ]; then
    echo "Usage: $0 <directory> [additional arguments for notpacked++]"
    exit 1
fi

for file in $(ls $1); do
    echo "Editing file: $file"
    sha256sum $1/$file | cut -d ' ' -f 1
    ./notpacked++ $1/$file -o $1/$file "${@:2}"
    sha256sum $1/$file | cut -d ' ' -f 1
done
```

Listing 5.6: Implementation of a script to mass apply notpacked++

Next, we apply all alterations on a dataset copied from the baseline dataset. The command shown below applies the alterations `fill_sections_with_zeros` and `change_sections_permissions`, our super-alteration. This is because the alteration modifying the permissions already includes the relocation of the entry point and the renaming of sections.

```
┌──[user@host]──[~]──────────
$ apply_notpacked++.sh copy_baseline --fill-sections --permissions
<<snipped>>
```

5.4.2 Effect on features

In Figure 5.25, we illustrate the affected features when combining all alterations and comparing with using one alteration at a time. In this result, we clearly see an improvement when using our super alteration that *updates section permissions* and an even better result when applying all alterations. The last two columns on the left of the figure shows the application of the super alteration `change_permissions` and `add_api`, combined with `edit_raw_size` and the other combined with `fill_sections_with_zeros`. We also notice a difference in those two as they target the same features but `fill_sections_with_zeros` increases the file size and thus features like `min_ratio_rdata` and `number_sections_vsize>dsize` are impacted negatively.

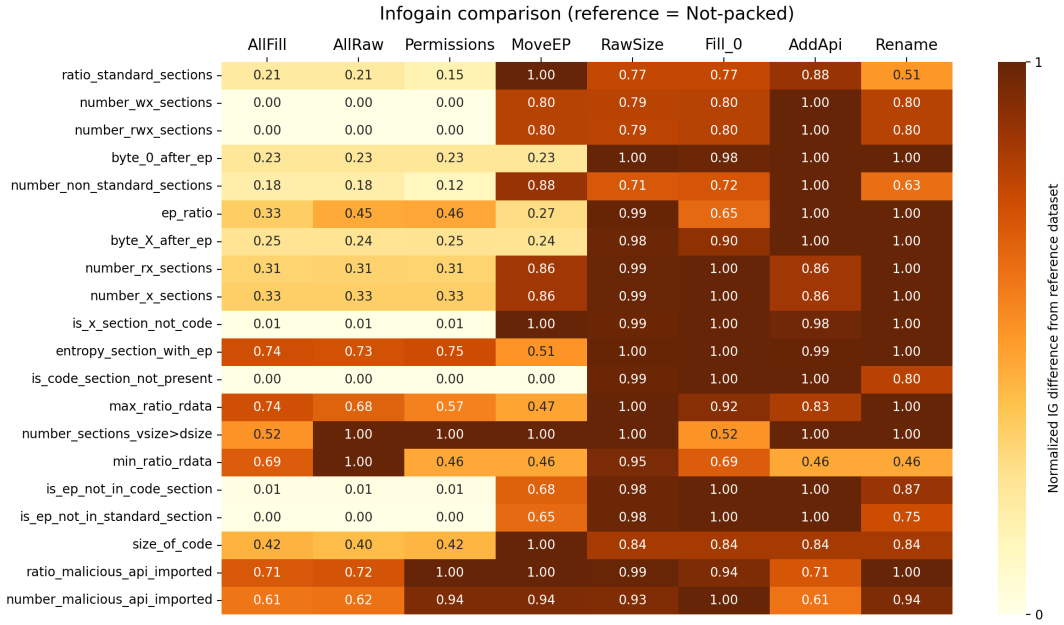


Figure 5.25: Comparison of information gain for all alterations

5.4.3 Impact on detectors

To verify the effectiveness on freely available static detectors, we prepared one dataset per alteration including the reference baseline dataset and an additional dataset for all alterations combined. In Table 5.3, we present the accuracy of each detector on the given dataset.

The accuracy metric evaluates how close a set of observations are from their true values, reflecting the overall effectiveness of a detector. To calculate the accuracy, we use the formula shown in Equation (5.1), where TP stands for true positives, TN for true negatives, FP for false positives, and FN for false negatives.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.1}$$

We compute these accuracy values by running the following command inside the Packing Box :

```
┌──[user@packing-box]──[~]──────────
$ detector baseline_upx --binary -d die -m
100.00%,100.00%,100.00%,100.00%
```

The `-m` metric argument ensures the output shows metrics. The first of the 4 values given is the accuracy of the detector on the given dataset, which is the value displayed in Table 5.3. In this table, baseline values are highlighted in **black**, values that remain unchanged after alterations are indicated in **gray**, small impacts on accuracy are shown in **green**, significant impacts are marked in **orange**, and accuracy droping below 10% are displayed in **red**.

Dataset	Bintropy	DIE	Manalyze	PEiD	PePack	PyPackerDetect	Reminder	RetDec
not-packed	100.0%	100.0%	100.0%	99.5%	98.6%	83.8%	100.0%	99.5%
baseline	69.5%	100.0%	47.7%	100.0%	100.0%	100.0%	50.5%	100.0%
add_api	66.3%	99.8%	47.7%	100.0%	100.0%	100.0%	50.5%	100.0%
rename_sections	70.0%	99.8%	11.0%	100.0%	100.0%	97.2%	50.5%	100.0%
add_section	0.0%	99.8%	47.7%	100.0%	100.0%	100.0%	50.5%	100.0%
fill_sections	56.5%	81.6%	47.7%	82.2%	82.8%	100.0%	18.2%	83.50%
edit_raw_size	70.4%	97.7%	47.7%	97.8%	100.0%	100.0%	50.3%	97.8%
move_ep_original	70.0%	17.0%	47.7%	0.0%	28.8%	99.5%	0.0%	65.5%
move_ep_deadcode	68.3%	17.0%	47.7%	0.0%	28.8%	99.5%	0.0%	50.0%
change_permissions	11.2%	0.3%	0.0%	0.2%	68.8%	49.0%	0.0%	19.2%
perm_fill_api	8.2%	0.2%	0.0%	0.0%	38.3%	0.0%	0.0%	19.5%

Table 5.3: Accuracy of various detectors on our altered dataset

Note that some accuracy values shown in Table 5.3, may show unexpected values such as 0.2% or 99.8%, this may indicate that a few samples were corrupted and failed. In our dataset of 600 samples, 0.2% would correspond to approximately one sample being accurately detected.

In Table 5.3, we first display the accuracy of each detectors on not packed dataset to identify detectors that may output false positives. We observe a lower accuracy of 83% for PyPackerDetect, meaning that this detector may produce false positives. Next, we see the accuracy of each detectors on the baseline dataset, and as we can see without any alteration applied the accuracy of Bintropy, Manalyze and and Reminder are quite low ranging from 47% to 69%, while other detectors displays an accuracy of 100%.

- **`add_api`**: This alteration barely affects the selected detectors, we observe only a tiny accuracy drop of -3.2% for the Bintropy detector and a negligible accuracy drop for DIE that may be due to a failing sample.
- **`rename_sections`**: This alteration mainly affects the Manalyze detector as it relies on the presence of known section section names. The accuracy of the PyPackerDetect also drops by -2.8%.
- **`add_low_entropy_section`**: The Bintropy detector is clearly impacted, reaching 0% accuracy, as it relies on entropy only and the alteration successfully reduces the entropy.
- **`fill_sections_with_zeros`**: This impacts the Reminder detector significantly, as it uses the entropy of the entry point section as indicator. Manalyze and PyPackerDetect remain unchanged and other detectors shows a little drop of the accuracy ranging from 13% and 18%.
- **`raw_size_edit`**: With this alteration we notice only a very small effect on the detection by Detect It Easy (DIE), PEiD, RetDec and a negligible effect on Reminder.
- **`mode_ep_original`**: Moving the entry point to a new section has a significant impact on many detectors. Some, like PEiD and Reminder, experience a complete drop of the accuracy to **0%**, while others, such as DIE, PePack, and RetDec, show a significant reduction of their accuracy.
- **`mode_ep_deadcode`**: The optimized version with injected deadcode affects the same detectors as `mode_ep_original`. We notice a small improvement for the Bintropy detector and a significant 15% decrease in accuracy for the RetDec detector.
- **`change_permissions`**: This super-alteration drops half of all detectors accuracy to 0%. Next, we observe a high impact on Bintropy and RetDec with an accuracy below 20%. The accuracy of PePack and PyPackerDetect is significantly affected dropping from 100% to respectively 68% and 49%.
- **`permissions`** + **`fill_sections`** + **`add_api`**: When combining our super alteration with `add_api` and `fill_sections_with_zeros`, we notice a significant accuracy drop from 49% to **0%** for PyPackerDetect. **This is an important impact as other alterations were not able to evade that specific detector.** We also observe an improvement for Bintropy and PePack.

In order to have a more nuanced understanding of how each alteration impacts the effectiveness of static detection, we introduce the **Evasion Score**. This metric is defined as the average of the absolute differences between the altered accuracy and the baseline accuracy across all detectors. The Evasion Score provides insight into how much each altered dataset deviates from the baseline, helping to identify which alterations are more effective in evading detection. This formula is presented in Equation (5.2).

$$\text{Evasion Score} = \frac{1}{n}\sum_{i=1}^{n}\left(\text{Accuracy}_{\text{Baseline}} - \text{Accuracy}_{\text{Altered Set i}}\right) \tag{5.2}$$

Alterations	Evasion Score
perm_fill_api	78.6%
change_permissions	61.2%
move_ep_deadcode	45.8%
move_ep_original	42.6%
add_section	25.7%
add_api	24.9%

rename_sections	21.9%
raw_size	16.2%
fill_sections	15.1%
baseline	0.0%

Table 5.4: Evasion Scores across all detectors used

In Table 5.4, we have a clear overview of the evasion capabilities of each alterations, and we can notice that the super alteration introduced displays the best evasion score, meaning it successfully evades the detection by dropping the detection by 78.6% combined across all selected detectors.

5.4.4 Speed Performance

In addition to its portability advantage over the Packing Box experimental toolkit, our tool also exhibit significantly faster performance. To compare the speed of our tool with the Packing Box framework, we ran the same alteration on the same binary example and measured the execution time.

```
┌──[user@packing-box]──[~]─────────
$ time executable alter upx_7z.exe
real    0m1.583s
user    0m1.600s
sys 0m0.525s
┌──[user@host]──[~]─────────
$ time ./notpacked++ upx_7z.exe -rename-sections
<<snipped>>
real    0m0.007s
user    0m0.003s
sys 0m0.003s
```

The results clearly show that the Packing Box framework took **1.583 seconds** to execute the alteration that *renames sections*, while our tool, NotPacked++ , completed the same task in just **0.007 seconds**, which is almost instantaneous to the human perspective.

FINDINGS

- We successfully attacked many significant features with our tool.
- Our new alteration **raw_size_edit** affects the detectors less than the existing **fill_sections_with_zeros**, and is accurately detected by all detectors.
- Many freely available detectors fail to detect altered executables and have their accuracy dropped to 0% for most detectors.
- One of the most impacting alteration is our super-alteration `change_permissions` and it is even more impacting when combining it with `add_api` and `fill_sections_with_zeros` alterations.
- Our super-alteration `change_permissions` is the first alteration introduced to impact the detector PyPackerDetect significantly.
- Our adversarial tool, NotPacked++ , alters an executable faster than the Packing Box experimental toolkit.

SUMMARY

In this chapter, we described the datasets used, verified the functionality of altered binaries, and analyzed the impact of alterations on features. Additionally, we explored improvements, introduced new alterations, and presented the usage of the NotPacked++ tool.

First, we described **existing alterations**, analyzed their **effect on features** and their effect on binary files using visualization tools inside the Packing Box. We then conducted a **functional verification** to verify that each alteration maintains the executable's functionality, and we discovered that two alterations were breaking the functionality of the binary. Based on the observations, we proposed **improvements** by adding **randomization** to the *move entry point* alteration that was vulnerable to signature based detection.

Next, we introduced two **new alterations**, studied their effect on features, and verified their effectiveness in maintaining the file's functionality. The first alteration involves *editing the raw size* of sections, which affects the same features as *filling sections with zeros* without increasing the file size. The second is a **super-alteration** combining multiple alterations, including *editing section permissions*, which affects the majority of the most significant features.

Finally, we validated our NotPacked++ tool, which includes the most relevant alterations discussed above, by analyzing the **effect on features** using a plot of the normalized information gain (IG) compared with a reference dataset of unpacked samples. This analysis highlighted the most significant features for distinguishing between the two datasets. Additionally, we studied the **effect of our tool against static packing detectors** and observed a **highly effective evasion capability** for our super-alteration. Lastly, we compared the speed performance of our tool with the experimental toolkit and observed significantly faster results, as expected.

6 Conclusion

THIS chapter wraps up this work, stating the objectives achieved throughout this document, providing a summary of the main contributions of this work and concluding with potential directions for future work.

6.1 Summary

Executable packing is the process of transforming binary files without affecting their behavior at runtime. It is widely used by legitimate software developers to protect intellectual property, manage licenses, and reduce file size, but also by malicious actors to disguise malware as legitimate harmless executable files to evade static detection. Static detection is a more efficient approach as it inspects files without execution, resulting in faster and safer analysis. In contrast, dynamic detection requires running the executable to study its behavior, which is more resource-intensive and riskier. Unpacking the packed executable is required to examine the original binary's content within a packed file. This highlights the importance of robust detection methods for malware analysis to identify packed files before unpacking them.

Many static detection techniques have been developed, and recent ones involve deriving features from executables to be used with Machine Learning models. A majority of static detectors rely on a few common features, which can be vulnerable to adversarial attacks, as highlighted by Biondi et al. [3] and Bertrand Van Ouytsel et al. [4]. In 2022, D'Hondt [1] created the Packing Box [7], an experimental toolkit for static analysis of executable packing. Using this toolkit, Jennes's adversarial study [2] focused on attacks against packing detectors and extended the set of executable alterations efficient for evasion. Incorporating an adversarial study into the initial design phase of detection models is essential for identifying and reinforcing potential weaknesses. Very few studies focus on attacking those static detectors and ML models.

To address this gap, we developed and presented a new adversarial tool capable of evading the static detection of packed executables. NotPacked++ is a weaponized tool designed to enhance the adversarial landscape by efficiently implementing the most relevant alterations from recent studies. It can be applied to evaluate and develop more reliable static detection mechanisms. Additionally, this tool will be incorporated into the Packing Box framework, adding a valuable and portable tool to the toolkit.

6.2 Achievements

We describe in this section how we were able to accomplish the objectives we established for this thesis. We first outline our objectives and how we met them, then provide summarized answers to the research questions, and finally highlight our contributions to the field of packing detection and malware analysis.

6.2.1 Objectives

The objectives predefined in Section 1.2 were reached as summarized in this subsection.

1. Chapter 2 provided the essential knowledge needed to understand the concepts and methodologies explored in this thesis.
   - A – We presented the general concept of executable files and the structure of the Portable Executable (PE) format.
   - B – Runtime packing was defined, along with its detection and usage.
   - C – We stated the main static detection techniques existing for packing detection.
   - D – We defined the essential concepts, including the threat model and attack space, for the adversarial setting of our work.
2. Chapter 3 presented the state-of-the-art of packing detection, adversarial studies and tools.
   - A – We provided an overview of the static packing detection field.
   - B – We summarized research on adversarial attacks and defense evasion
   - C – We compiled and listed some well-known packers, detection tools, and evasion tools.

3. Chapter 4 introduced the framework of our research.
   A – We described the adversarial setting for our research by detailing the adversary's capabilities and goals.
   B – We provided an overview of the Packing Box framework, including its essential abstractions and building blocks.
   C – Our tool's development environment was described, including its architecture, its components, and usage.
4. Chapter 5 described practical experiments that were carried out.
   A – The methodology used in our experiments was established.
   B – We described current alterations, validated their correctness, and optimized them to better evade detection.
   C – We introduced new alterations, visualized their effects and studied their impact on features and detection tools.
   D – We successfully validated our tool, NotPacked++ , by analyzing its impact on static detectors and comparing its speed performance with the Packing Box experimental toolkit.

6.2.2 Research Questions

In Section 1.3, we introduced three key research questions, which we can now briefly answer.

Question 1 **What is the state-of-the-art of alterations and adversarial tools ?**
Many studies focus on the feature space, which does not guarantee a realistic evasion capability. In the problem space, realistic alterations targeting ML models are often limited to renaming sections or perturbing a few bytes in the overlay or slack space. Only a few adversarial studies have focused on attacking heuristics used in static detectors. A majority of alterations are developed to attack a subset of significant features identified by Biondi et al. [3]. Existing *adversarial tools* provide a theoretical/experimental alterations that are not easily applicable in a real scenario, as opposed to our proposed tool. Some of the adversarial tools uses ML to attack the detection mechanism, this is also often limited to padding, appending bytes, renaming sections or adding api imports. However, all tools are oriented to AV malware evasion and no available tool focuses on packing detection evasion. Very few tools combines multiple alterations in the problem space without breaking executable functionality.

Question 2 **How to optimize alterations and can we add new ones ?**
We optimize them by analyzing the effect of existing alterations on features and verifying that they maintain the executable's functionality to identify potential area of improvement. Two alterations were fixed to maintain the executable functionality, and one was optimized by adding randomization to perturb a specific feature to evade signature based detection. Yes, we managed to add 2 new alterations and study their impact on features and detection tools. This is done by analyzing targeted features of current alterations and identifying less targeted features that are relevant for packing detection, and introducing a new alteration inspired from recent research in malware analysis and adversarial learning.

Question 3 **How to weaponize those alterations in an adversarial tool ?**
We start by selecting functional and relevant alterations existing in recent literature, then we validate their effectiveness using an experimental toolkit, and take advantage of LLMs to quickly convert Python code to C++ code and implement the alterations in the new tool. The result is a console-based tool that allows users to specify alterations using parameters

like `--rename-sections`. This design ensures the tool is user-friendly and flexible, with a simple interface that enables users to easily alter an executable by selecting the most suitable command-line arguments.

6.2.3 Contributions

Below is listed the main contributions of our thesis.

- ✓ We gathered and studied the effectiveness of existing alterations and verified their functionality-preserving property, ensuring that the altered binary remains operational.
- ✓ The alteration "*move entry point*" was fixed and optimized with randomization to better evade static detection, as it had a flaw allowing a simple signature-based detection to be successful.
- ✓ We explored the effectiveness of a new alteration to *edit the raw size* of sections without increasing file size.
- ✓ A super-alteration was introduced for updating the section permissions to match standard ones. We call it "*super-alteration*" because besides updating section permissions, this alteration also includes renaming sections, moving the entry point and randomization.
- ✓ We developed and published a new open source adversarial tool named NotPacked++ , that includes the most effective alterations and is meant to be used by researchers to assess their packing detection mechanism against adversaries.

6.3 Future Works

This last section provides ideas for further studies and improvement in the field of static packing detection.

6.3.1 Research Ideas

During this work, we gathered multiple research ideas along the way that could be valuable to explore for further optimizing the effectiveness of our alterations in all scenarios and improve the packing detection field and evasion techniques. These ideas of research are proposed hereafter:

- **Leveraging the relocation table for entry point relocation :** Explore the use of the relocation table (`.reloc`) in the PE format to develop new approaches for moving the entry point, and potentially enhancing the effectiveness of alterations.
- **Impact on malware behavior detection :** Investigate how our alterations affect typical malware behavior detection, such as the use of `VirtualProtect` or CALL/POP techniques, to understand any unintended consequences on detection systems.
- **Obfuscation of unpacking stubs :** Explore methods to obfuscate known unpacking stubs, aiming to evade signature-based detection systems that rely on recognizing these stubs.
- **Parametric study on dead code injection :** Conduct a parametric study to determine the minimum amount of dead code needed to significantly impact detection rates, optimizing the balance between evasion and performance.
- **Bypassing integrity checks in packers :** Adversarial study to bypass integrity checks implemented by certain packers, such as TELock, to ensure our alterations maintain the functionality of the executable.

- **Targeting entropy-based detection :** Investigate techniques to evade entropy-based detection by splitting high-entropy sections into multiple chunks and adding low-entropy data in the slack space, which won't be loaded into memory, thus preserving the executable's functionality.
- **Extending impact analysis to dynamic detection :** Expand the study of current alterations' impact by assessing their effectiveness against dynamic analysis, which could reveal new insights into evasion techniques.

6.3.2 Framework Improvement

We also suggest a few improvements of the experimental environment and the attack tool developed for this thesis.

- **Extending dataset resources:** Expand the current dataset resource to include x64 binaries and analyze how these files impact features and detectors.
- **Supporting additional formats:** Extend our adversarial tool to support other formats, such as ELF and Mach-O, broadening its applicability and effectiveness.

References

[1] Alexandre D'Hondt. "Experimental Toolkit for Studying Executable Packing - Analysis of the State-of-the-Art Packing Detection Techniques". UCL - Ecole polytechnique de Louvain, 2022. URL: http://hdl.handle.net/2078.1/thesis:35692.

[2] Romain Jennes. "Adversarial Learning on Static Detection Techniques for Executable Packing". UCL - Ecole polytechnique de Louvain, 2023. URL: http://hdl.handle.net/2078.1/thesis:40178.

[3] Fabrizio Biondi et al. "Effective, Efficient, and Robust Packing Detection and Classification". In: *Computers & Security* 85 (May 2019), pp. 436–451. ISSN: 0167-4048. DOI: 10.1016/j.cose.2019.05.007. URL: http://www.sciencedirect.com/science/article/pii/S0167404818311040.

[4] Charles-Henry Bertrand Van Ouytsel et al. "Analysis of Machine Learning Approaches to Packing Detection". In: *CoRR* abs/2105.00473 (May 2021). URL: https://arxiv.org/abs/2105.00473.

[5] Alexandre D'Hondt, Charles Henry Bertrand Van Ouytsel, and Axel Legay. "Experimental Toolkit for Manipulating Executable Packing". In: *Risks and Security of Internet and Systems*. Ed. by Abderrahim Ait Wakrime et al. Cham: Springer Nature Switzerland, June 2024, pp. 263–279. ISBN: 978-3-031-61231-2. DOI: 10.1007/978-3-031-61231-2_17.

[6] Alexandre D'Hondt, Charles-Henry Bertrand Van Ouytsel, and Axel Legay. "Evading Packing Detection: Breaking Heuristic-Based Static Detectors". In: *21st Conference on Detection of Intrusions and Malware & Vulnerability Assessment*. Springer Nature Switzerland, July 2024, pp. 174–183. ISBN: 978-3-031-64171-8. DOI: 10.1007/978-3-031-64171-8_9.

[7] Alexandre D'Hondt. *Packing-Box*. 2022. URL: https://github.com/packing-box/docker-packing-box.

[8] *PE Format (Microsoft Docs)*. PE Format - Win32 apps | Microsoft Docs. Mar. 12, 2024. URL: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format (visited on 03/12/2024).

[9] Andreas Moser, Christopher Kruegel, and Engin Kirda. "Exploring Multiple Execution Paths for Malware Analysis". In: *2007 IEEE Symposium on Security and Privacy (SP '07)*. 2007, pp. 231–245. DOI: 10.1109/SP.2007.17.

[10] MITRE Corporation. *MITRE ATT&CK*. https://attack.mitre.org/. 2013.

[11] URL: https://attack.mitre.org/techniques/T1027/002/.

[12] ParrotSec. *Mimikatz*. 2020. URL: https://github.com/ParrotSec/mimikatz.

[13] Tom Brosch and Maik Morgenstern. *Runtime Packers: The Hidden Problem?* Presentation at BlackHat USA. Available online. 2006. URL: https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf.

[14] McAfee. *The Good, the Bad, and the Unknown*. https://www.techdata.com/mcafee/files/MCAFEE_wp_appcontrol-good-bad-unknown.pdf. 2009.

[15] Xabier Ugarte-Pedrero et al. "On the adoption of anomaly detection for packed executable filtering". In: *Computers & Security* 43 (June 2014), 126–144. ISSN: 0167-4048. DOI: 10.1016/j.cose.2014.03.012. URL: http://dx.doi.org/10.1016/j.cose.2014.03.012.

[16] snaker. *PEiD*. Version 0.95. https://github.com/wolfram77web/app-peid, 2008. URL: https://github.com/wolfram77web/app-peid/.

[17] Alexandre D'Hondt. *PEiD (CLI)*. Version 1.2.3. 2021. URL: https://github.com/dhondta/peid.

[18] Rohit Arora et al. "A Heuristics-based Static Analysis Approach for Detecting Packed PE Binaries". In: *International Journal of Security and Its Applications* 7.5 (Sept. 2013), pp. 257–268. ISSN: 17389976, 17389976. DOI: 10.14257/ijsia.2013.7.5.24. URL: http://article.nadiapub.com/IJSIA/vol7_no5/24.pdf.

[19] Luca Demetrio et al. "Adversarial EXEmples: A Survey and Experimental Evaluation of Practical Attacks on Machine Learning for Windows Malware Detection". In: *ACM Transactions on Privacy and Security* 24.4 (Sept. 2021), 1–31. ISSN: 2471-2574. DOI: 10.1145/3473039. URL: http://dx.doi.org/10.1145/3473039.

[20] Luca Demetrio et al. *Explaining Vulnerabilities of Deep Learning to Adversarial Malware Binaries*. 2019. DOI: 10.48550/arXiv.1901.03583. URL: https://doi.org/10.48550/arXiv.1901.03583.

[21] Octavian Suciu, Scott E. Coull, and Jeffrey Johns. "Exploring Adversarial Examples in Malware Detection". In: *2019 IEEE Security and Privacy Workshops (SPW)*. IEEE, May 2019. DOI: 10.1109/spw.2019.00015. URL: http://dx.doi.org/10.1109/SPW.2019.00015.

[22] Fabio Pierazzi et al. "Intriguing Properties of Adversarial ML Attacks in the Problem Space". In: *CoRR* abs/1911.02142 (2019). arXiv: 1911.02142. URL: http://arxiv.org/abs/1911.02142.

[23] Nicholas Carlini et al. *On Evaluating Adversarial Robustness*. 2019. arXiv: 1902.06705 [cs.LG]. URL: https://arxiv.org/abs/1902.06705.

[24] Khanh Huu The Dam et al. "Packer classification based on association rule mining". In: *Applied Soft Computing* 127 (Sept. 2022), p. 109373. ISSN: 1568-4946. DOI: 10.1016/j.asoc.2022.109373. URL: http://dx.doi.org/10.1016/j.asoc.2022.109373.

[25] Mohaddeseh Zakeri, Fatemeh Faraji Daneshgar, and Maghsoud Abbaspour. "A static heuristic approach to detecting malware targets: Static malware detection". en. In: *Security and Communication Networks* 8.17 (Nov. 2015), pp. 3015–3027. ISSN: 19390114. DOI: 10.1002/sec.1228. URL: https://onlinelibrary.wiley.com/doi/10.1002/sec.1228.

[26] Donghwi Shin et al. "The New Signature Generation Method Based on an Unpacking Algorithm and Procedure for a Packer Detection". In: *International Journal of Advanced Science and Technology*. Vol. 27. IJAST, Feb. 2011, pp. 59–78. URL: https://www.earticle.net/Article/A147420.

[27] Smita Naval et al. "SPADE: Signature Based PAcker DEtection". In: *Proceedings of the First International Conference on Security of Internet of Things*. SecurIT '12. New York, NY, USA: ACM, Aug. 2012, pp. 96–101. ISBN: 978-1-4503-1822-8. DOI: 10.1145/2490428.2490442. URL: https://dl.acm.org/doi/10.1145/2490428.2490442.

[28] Nguyen Minh Hai, Mizuhito Ogawa, and Quan Thanh Tho. "Packer Identification Based on Metadata Signature". In: *Proceedings of the 7th Software Security, Protection, and Reverse Engineering / Software Security and Protection Workshop* (Orlando, FL, USA). SSPREW-7. New York, NY, USA: ACM, Dec. 2017, p. 11. ISBN: 978-1-4503-5387-8. DOI: 10.1145/3151137.3160687. URL: https://dl.acm.org/doi/10.1145/3151137.3160687.

[29] Nguyen Minh Hai. *BE-PUM*. Version 1. 2016. URL: https://github.com/NMHai/BE-PUM.

[30] Robert Lyda and James Hamrock. "Using Entropy Analysis to Find Encrypted and Packed Malware". In: *IEEE Security & Privacy* 5.2 (2007), pp. 40–45. DOI: 10.1109/MSP.2007.48.

[31] Seungwon Han, Keungi Lee, and Sangjin Lee. "Packed PE File Detection for Malware Forensics". In: *2009 2nd International Conference on Computer Science and its Applications*. IEEE, 2009. DOI: 10.1109/csa.2009.5404211. URL: http://dx.doi.org/10.1109/csa.2009.5404211.

[32] Smita Naval et al. "ESCAPE: Entropy Score Analysis of Packed Executable". In: *Proceedings of the Fifth International Conference on Security of Information and Networks*. SIN '12. New York, NY, USA: ACM, Oct. 2012, pp. 197–200. ISBN: 978-1-4503-1668-2. DOI: 10.1145/2388576.2388607. URL: https://dl.acm.org/doi/10.1145/2388576.2388607.

[33] Munkhbayar Bat-Erdene et al. "Entropy Analysis to Classify Unknown Packing Algorithms for Malware Detection". In: *International Journal of Information Security*. Vol. 16. 3. Springer, May 2016, pp. 227–248. DOI: 10.1007/s10207-016-0330-4. URL: https://link.springer.com/article/10.1007/s10207-016-0330-4.

[34] Alessandro Mantovani et al. "Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem". In: *Network and Distributed Systems Security (NDSS) Symposium 2020*. NDSS, Feb. 2020, p. 15. ISBN: 1-891562-61-4. DOI: 10.14722/NDSS.2020.24297. URL: https://www.ndss-symposium.org/wp-content/uploads/2020/02/24297.pdf.

[35] Alexandre D'Hondt. *Bintropy*. Version 1.2.3. 2021. URL: https://github.com/dhondta/bintropy.

[36] Yang-Seo Choi et al. "PE File Header Analysis-Based Packed PE File Detection Technique (PHAD)". In: *International Symposium on Computer Science and Its Applications*. IEEE, Oct. 2008, pp. 28–31. DOI: 10.1109/CSA.2008.28. URL: https://ieeexplore.ieee.org/document/4654055.

[37] Xabier Ugarte-Pedrero, Igor Santos, and Pablo Garcia Bringas. "Structural Feature Based Anomaly Detection for Packed Executable Identification". In: *Computational Intelligence in Security for Information Systems*. Ed. by Alvaro Herrero and Emilio Corchado. Springer, 2011, pp. 230–237. ISBN: 978-3-642-21323-6. DOI: 10.1007/978-3-642-21323-6_29. URL: https://link.springer.com/chapter/10.1007/978-3-642-21323-6_29.

[38] Weiwei Hu and Ying Tan. *Black-Box Attacks against RNN based Malware Detection Algorithms*. 2017. DOI: 10.48550/ARXIV.1705.08131. URL: https://arxiv.org/abs/1705.08131.

[39] Bojan Kolosnjaji et al. *Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables*. 2018. DOI: 10.48550/ARXIV.1803.04173. URL: https://arxiv.org/abs/1803.04173.

[40] Hyrum S. Anderson et al. *Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning*. 2018. DOI: 10.48550/ARXIV.1801.08917. URL: https://arxiv.org/abs/1801.08917.

[41] Luca Demetrio et al. "Functionality-Preserving Black-Box Optimization of Adversarial Windows Malware". In: *IEEE Transactions on Information Forensics and Security* 16 (2021), 3469–3478. ISSN: 1556-6021. DOI: 10.1109/tifs.2021.3082330. URL: http://dx.doi.org/10.1109/TIFS.2021.3082330.

[42] Daniel Gibert, Giulio Zizzo, and Quan Le. "Certified Robustness of Static Deep Learning-based Malware Detectors against Patch and Append Attacks". In: *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*. CCS '23. ACM, Nov. 2023. DOI: 10.1145/3605764.3623914. URL: http://dx.doi.org/10.1145/3605764.3623914.

[43] Hyrum S. Anderson et al. *Gym-malware*. Version 1. 2018. URL: https://github.com/endgameinc/gym-malware.

[44] Romain Thomas. *LIEF*. Version 0.12.0. QuarksLab, 2022. URL: https://github.com/lief-project/LIEF.

[45] Luca Demetrio and Battista Biggio. *secml-malware: A Python Library for Adversarial Robustness Evaluation of Windows Malware Classifiers*. 2021. arXiv: 2104.12848 [cs.CR]. URL: https://github.com/pralab/secml_malware.

[46] Dario Nisi et al. "Lost in the Loader:The Many Faces of the Windows PE File Format". In: *24th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID '21. ACM, Oct. 2021. DOI: 10.1145/3471621.3471848. URL: http://dx.doi.org/10.1145/3471621.3471848.

[47] Javier Yuste, Eduardo G. Pardo, and Juan Tapiador. "Optimization of code caves in malware binaries to evade machine learning detectors". In: *Computers & Security* 116 (May 2022), p. 102643. ISSN: 0167-4048. DOI: 10.1016/j.cose.2022.102643. URL: http://dx.doi.org/10.1016/j.cose.2022.102643.

[48] horsicq. *DIE*. Version 3.03. https://github.com/horsicq/Detect-It-Easy, 2021. URL: https://github.com/horsicq/DIE-engine/releases.

[49] Seungwon Han, Keungi Lee, and Sangjin Lee. *REMINDer*. 2009. URL: https://doi.org/10.1109/CSA.2009.5404211.

[50] Kil Jin et al. "BinStat Tool for Recognition of Packed Executables". In: *ijofcs.org* 6.1 (Sept. 2010), pp. 44–58. DOI: 10.5769/J201101003. URL: http://www.ijofcs.org/abstract-v06n1-pp03.html.

[51] Ivan Kwiatkowski. *Manalyze*. https://github.com/JusticeRage/Manalyze, 2014. URL: https://github.com/JusticeRage/Manalyze.

[52] cylance. *PyPackerDetect*. 2018. URL: https://github.com/cylance/PyPackerDetect.

[53] guelfoweb. *PEFrame*. Version 6.0.2. 2019. URL: https://github.com/guelfoweb/peframe.

[54] Thomas Roccia. *Unprotect*. Version 1. 2019. URL: https://github.com/fr0gger/unprotect.

[55] K-atc. *PEiD (Yara)*. Version 0.1.1. 2017. URL: https://github.com/K-atc/PEiD.

[56] nahuelriva. *FUU*. Version 0.1.1b. https://code.google.com/archive/p/fuu/downloads, 2010. URL: https://github.com/crackinglandia/fuu.

[57] Avast. *RetDec*. Version 4.0. Avast, 2020. URL: https://github.com/avast/retdec.

[58] OpenAI. *OpenAI's ChatGPT*. 2023. URL: https://openai.com/index/gpt-4/.

[59] Thomas Roccia. *IATelligence*. 2023. URL: https://github.com/fr0gger/IATelligence.

[60] Weilin Xu, Yanjun Qi, and David Evans. *EvadeML*. 2015. URL: https://github.com/uvasrg/EvadeML.

[61] Josh Pitts. *SigThief*. 2017. URL: https://github.com/secretsquirrel/SigThief.

[62] Govolution. *AVET*. 2018. URL: https://github.com/govolution/avet.

[63] Hyrum S. Anderson et al. *MalwareRL*. 2018. URL: https://github.com/bfilar/malware_rl.

[64] paranoidninja. *CarbonCopy*. 2019. URL: https://github.com/paranoidninja/CarbonCopy.

[65] Justin Burr. *BitCamo*. 2022. URL: https://github.com/juburr/bitcamo.

[66] Hyrum S. Anderson et al. *Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning*. 2018. DOI: 10.48550/ARXIV.1801.08917. URL: https://arxiv.org/abs/1801.08917.

[67] Alexandre D'Hondt and Jaber Ramhani. *NotPacked++*. 2024. URL: https://github.com/packing-box/packer-masking-tool.

[68] Alexandre D'Hondt. *Dataset of Packed PE*. Dataset. 2021. URL: https://github.com/dhondta/dataset-packed-pe.

[69] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Section: INSTRUCTION SET REFERENCE, M-U. Intel Corporation. 2024, p. 1243. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

A Fix Alterations

A.1 Move Entrypoint To New Section

As discovered in Section 5.2, the alteration `move_entrypoint_to_new_section` did not maintain the functionality of the target binary file. To understand the cause of this issue, we closely examine the implementation of the modifier function where we find the piece of code causing the issue.

```
1  def _move_entrypoint_to_new_section(parsed, logger):
2      old_entry = parsed.entrypoint + 0x10000
3      #  push current_entrypoint
4      #  ret
5      entrypoint_data = [0x68] + list(old_entry.to_bytes([4, 8][parsed.path.format[-2:]
       ↪  == "64"], "little")) + [0xc3]
6      # other possibility:
7      #  mov eax current_entrypoint
8      #  jmp eax
9      #  entrypoint_data = [0xb8] + list(old_entry.to_bytes(4, 'little')) + [0xff, 0xe0]
10     <<snipped>>
```

Listing A.1: Initial implementation of the `move_entrypoint_to_new_section` modifier

We observed a line in the implementation (line 2 of listing A.1) that calculates the address of the current Entry Point (EP). This address is intended to be used to jump back from the inserted trampoline code to the Original Entry Point (OEP). The code snippet responsible for this jump, given in line 3 and 4 of Listing A.1, consists of using the `PSH/RET` technique. Here, the address of the old EP is first pushed onto the stack using the `PUSH` instruction, followed by a `RET` instruction to initiate the return procedure. This procedure retrieves the address from the stack using `POP` and jumps back to that address. This simple jump of the return procedure requires thus the address to be a VA. While a simple direct jump to the address could be sufficient, this technique is employed to obfuscate the code and confuse static analyzers and decompilers.

However, if the provided address is invalid or does not point to the correct location, the procedure will fail, causing the executable to stop its execution. In this implementation, the original Entry Point (EP) is calculated using the RVA of the EP and adding a fixed constant `0x10000`. This causes the trampoline code, when executed, to jump to an invalid VA. Instead, the image base address should be added to calculate the VA of the OEP. Nevertheless, this approach rely on the position where the executable is loaded which is not always at the preferred image base value. In an environment where address space layout randomization (ASLR) is enabled, executables are loaded at a random base address, this would cause our altered binary to not function correctly as the computed address is not valid or points to an unexpected position.

Additionally, this alteration's trampoline is also invalid for **x64** executables, this is because in the `PSH/RET` technique used, the `PUSH` instruction does not accept immediate address in 64 bit mode. As per the official

documentation [1] of Intel's architecture [69] shown in Figure A.1, this instruction allows immediate address for 8,16 and 32 bit, but not for 64 bit addresses.

50+rw	PUSH r16	0	Valid	Valid	Push r16.
50+rd	PUSH r32	0	N.E.	Valid	Push r32.
50+rd	PUSH r64	0	Valid	N.E.	Push r64.
6A ib	PUSH imm8	I	Valid	Valid	Push imm8.
68 iw	PUSH imm16	I	Valid	Valid	Push imm16.
68 id	PUSH imm32	I	Valid	Valid	Push imm32.

Figure A.1: Intel's documentation for PUSH instruction

To solve these issues, we explored multiple paths that includes ideas for getting the image base dynamically from different sources and using various techniques, such as taking advantage of the PEB or using a `CALL/POP` technique. More details on the explored approaches are given in Appendix A.2. For this alteration's improvement, we selected the less complex technique that does not rely on the image base and does not include a long recognizable pattern that can be used as signature.

After extensive exploration, we adjusted the implementation to compute the address as an offset from the next instruction to the original entry point. We calculate this offset and inject the `JMP` instruction with the relative 32-bit signed offset, which can be either positive or negative. The implementation of the new approach is shown in Listing A.2.

```
1  <<snipped>>
2  _trampoline_code = lambda oep_or_offset: [0xE9, *list(oep_or_offset.to_bytes(4,
   ↪  "little", signed=True))] # JMP -1234
3
4  def _move_entrypoint_to_new_section(parsed, logger):
5      oep = parsed.optional_header.addressof_entrypoint
6      # add trampoline code
7      ep_data = _trampoline_code(oep) # oep used as placeholder here
8      # create new section
9      add_section(name, section_type or parsed.SECTION_TYPES['TEXT'], characteristics,
10                 list(pre_data) + ep_data + list(post_data))(parsed, logger)
11     # update content and trampoline offset (do it after to know the address of the new
       ↪  section)
12     s = parsed.section(name)
13     offset = oep - (s.virtual_address + len(pre_data) + len(ep_data))
14     s.content = list(pre_data) + _trampoline(offset) + list(post_data)
15     # update EP
16     parsed.optional_header.addressof_entrypoint = s.virtual_address + len(pre_data)
```

Listing A.2: New implementation of the `move_entrypoint_to_new_section` modifier

This new implementation is accomplished in three steps:

1. First, we create the new section with trampoline code containing a placeholder address for jumping back. This step is necessary to determine the address of the newly inserted section. (Line 5-10)

2. Next, we compute the actual offset to the original entry point. This is done by subtracting from the entry point address the sum of the address of the inserted section, the length of the data preceding the trampoline code, and the length of the trampoline code itself. (Line 13)

3. Finally, we update the section's content to replace the placeholder address with the correct offset address. We also update the header's `addressof_entrypoint` value to point to the new entry point at the start of the trampoline code. (Line 14-16)

[1] `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`

The primary advantage of this approach is its effectiveness in ASLR-enabled environments and its compatibility with PE file operating in 64-bit mode, while keeping the trampoline code extremely small to avoid signature based detection.

This new alteration is sensitive to modification of section sizes because it relies on offsets. Therefore, when combining alterations, we must ensure that we do not modify section raw sizes after applying this one. For instance, if we apply the "*move entrypoint*" alteration before "*fill sections with zeros*", the resulting executable won't work because the section sizes changed after the offset calculation.

A.1.1 Implementation new move_ep with deadcode

The new implementation of the `move_ep` alteration is given in Listing A.3. We have set 3 useful constants to control the number of occurrence for the dead code injection, including `MAX_DEAD_CODE`, `MIN_DEAD_CODE` and `MAX_REPEATED_DEAD_CODE`. By default we have set to respectively 42, 21 and 4. However, a specific fine-tuning is needed to find the perfect amount of random deadcode injection that would impact the most features and detection.

```
_trampoline_code = lambda oep_or_offset: [0xE9, *list(oep_or_offset.to_bytes(4,
↪  "little", signed=oep_or_offset < 0))]
def _move_entrypoint_to_new_section(parsed, logger):
    # <<snipped>>
    random_number = random.randint(0, 100)
    inject_typical_byte_infront, using_prolog = random_number < 70, random_number < 15
    # Randomly add prolog (mostly found in NotPacked binaries)
    #   push ebp / mov ebp, esp /&/ sub esp, 0xc
    code_data = ([0x83, 0xec, 0x0c] if random.randint(0, 1) == 1 else [0x55, 0x8b,
    ↪  0xec, 0x83, 0xec, 0x0c]) if inject_typical_byte_infront else []

    if with_dead_code:
        dead_code = get_pe_data()["DEAD_CODE"]
        # Select k random elements from the dead_code list (With random number of
        ↪  repetitions for each element (allowing for repetition) )
        _deadcode_data = random.sample(dead_code, k=random.randint(MIN_DEAD_CODE,
        ↪  MAX_DEAD_CODE), counts=[random.randint(MAX_REPEATED_DEAD_CODE -
        ↪  (MAX_REPEATED_DEAD_CODE // 2), MAX_REPEATED_DEAD_CODE) for _ in
        ↪  range(len(dead_code))])
        # <<snipped>>
        code_data += _deadcode_data

        if inject_typical_byte_infront:
            # restore stack (move esp, ebp / pop ebp) or (add esp, 0xc)
            code_data += [0x89, 0xec, 0x5d] if using_prolog else [0x83, 0xc4, 0x0c]
        # add dead code to be executed before the jump back to the original EP
        ep_data += code_data

    # add trampoline code
    ep_data += _trampoline_code(oep)
    # <<snipped>>
    offset = oep - (s.virtual_address + len(pre_data) + len(ep_data))
    s.content = list(pre_data) + code_data + _trampoline_code(offset) + list(post_data)
    # <<snipped>>
```

Listing A.3: Implementation of move_ep using dead code

A full list of our deadcode instructions used in our attack tool and our experiments are listed in Listing A.4.

```
0x90                                # NOP
0x33, 0xC0                          # XOR EAX, EAX
0x33, 0xC9                          # XOR ECX, ECX
0x33, 0xDB                          # XOR EBX, EBX
0x83, 0xC0, 0x00                    # ADD EAX, 0
0x87, 0xC0                          # XCHG EAX, EAX
0x83, 0xE9, 0x00                    # SUB ECX, 0
0x89, 0xC2                          # MOV EDX, EDX
0x50, 0x58                          # PUSH EAX + POP EAX
0x8D, 0x1B                          # LEA EBX, [EBX] (load effective address of EBX
↪ into EBX)
0x81, 0xC7, 0x01, 0x00, 0x00, 0x00, 0x81, 0xEF, 0x01, 0x00, 0x00, 0x00  # ADD EDI, 1 +
↪ SUB EDI, 1
0x81, 0xE6, 0xFF, 0xFF, 0xFF, 0xFF  # AND ESI, 0xFFFFFFFF (bitwise AND with immediate
↪ value)
0x83, 0xCA, 0x00                    # OR EDX, 0 (bitwise OR with immediate value)
0xEB, 0x00                          # JMP SHORT \$+2 (jump short 2 bytes, same as NOP)
0x33, 0xD2                          # XOR EDX, EDX
0xD1, 0xC0                          # ROL EAX, 1 (rotate left)
0x50, 0x58                          # PUSH EAX + POP EAX
0x05, 0x01, 0x00, 0x00, 0x00, 0x2D, 0x01, 0x00, 0x00, # ADD EAX, 1 + SUB EAX, 1
0xF9, 0xF8                          # STC + CLC (set carry flag + clear carry flag)
0x89, 0xC1, 0x89, 0xC8              # MOV ECX, EAX + MOV EAX, ECX
0x8D, 0x80, 0x00, 0x00, 0x00, 0x00  # LEA EAX, [EAX]
0x83, 0xC8, 0x00                    # OR EAX, 0
0xEB, 0x00, 0x90                    # JMP SHORT 2 bytes over NOP
0x50, 0x48, 0x89, 0xC1, 0x40, 0x58  # PUSH EAX + DEC EAX + MOV ECX, EAX + INC EAX + POP
↪ EAX
```

Listing A.4: Example of instructions `DEAD_CODE`

A.2 Other Explored Ideas

During our experiments, we explored many approaches to fix the trampoline code of alteration `move_entrypoint_`
while keeping the executable functional in an ASLR environement.

A.2.1 Process Environment Block

One of the approach consist of get dynamically the image base from the *Process Environment Block* (PEB).
The PEB is a data structure in Windows operating systems that contains information about the currently
running process, including the base address of the executable image in memory.

The following code snippet demonstrates how this is achieved:

```
# >> move_entrypoint_to_new_section
# ============ Trampoline ================
if is_64bits:
  # ========== 64-bit =============
  # mov rbx, gs:[0x60h]              # Get the PEB
  # mov rdi, [rbx+0x10h]             # Get image base address at offset 0x10
  # mov rbx, original_entrypoint
  # add rbx, rdi
  # jmp rbx
  # ================================
  # <<snipped>>
else:
  # ========== 32-bit =============
  # mov ebx, fs:[0x30]               # Get the PEB
  # mov eax, [ebx+8]                 # Get image base address
  # add eax, original_entrypoint
  # push eax; ret
  # ================================
  # <<snipped>>
```

Listing A.5: Implementation of `move_ep` alteration using the PEB

In this code A.5, the image base is obtained by loading the PEB address into the EBX/RBX register and then accessing the image base stored at offset +8 (+0x10 for x64 architecture) from the PEB. The original entry point is then computed by adding the base address to the original entry point RVA. Finally, the calculated address is pushed onto the stack, and the function returns to continue execution at the original entry point, ensuring that the executable remains functional.

This approach resulted in a functional alteration that successfully preserves the executable's functionality, even in environments where ASLR is enabled. However, this complex unusual approach involve multiple fixed instructions bytes that can be used as a signature and might also trigger typical malware behavior heuristics, for that reason we continued exploring new approaches.

A.2.2 CALL/POP

Another approach explored involves using a CALL instruction followed by a POP instruction. When the CALL instruction is executed, it pushes the address of the next instruction (the return address) onto the stack. By retrieving this address from the stack and subtracting the RVA of the instruction following the caller, we can calculate the current image base. This image base is then used for the trampoline jump by adding it to the hardcoded RVA of the original entry point.

This technique was not chosen as we found an simpler approach involving a simple jump with an offset instead of an address, this was explained in Appendix A.1.

It's important to note that the `CALL/POP` technique was used in our super-alteration to make the call to the `VirtualProtect` function. This step was essential because we needed the Virtual Address (VA) of the target section to change its permissions. By using the `CALL/POP` method, we were able to retrieve the image base and calculate the Virtual Address (VA) needed.

A.3 Fill Section With Zeros

We observed that the executable resulting from the alteration `fill_sections_with_zeros` was sometimes broken and sometimes working perfectly fine. We thus delve into the analysis of the broken executable, and we discovered that the file had corrupted values in the section header of the first section, as we showed in Figure A.2. We thus re-implemented the alteration and successfully fixed it.

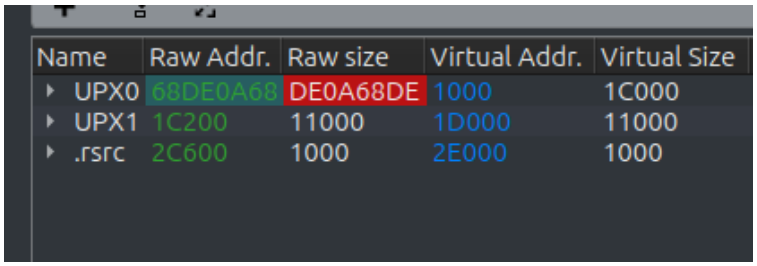


Figure A.2: Corrupted section header value after alteration

B PBox usage

B.1 Packing Box Framework

B.1.1 Experiment

Experiment is a useful abstraction that is used to create a dedicated workspace environment for each experiment. Each workspace experiment has its own parameters, datasets and models. The greatest advantage of such a workspace is being able to easily share it with other researchers, e.g. for the sake of validation by reproducing results. The most important commands available are listed in the following help message:

```
┌──[user@host]──[~]─────────
$ experiment --help
<<snipped>>
usage: experiment [-h] [--help] [-v] CMD ...

positional argument:
  CMD         command to be executed
    close     close the current experiment
    open      open the specified experiment
    <<snipped>>
    import    import a YAML configuration, data or script into the experiment
    list      list the existing experiments
    <<snipped>>
    show      get an overview of the experiment
<<snipped>>
```

B.1.2 Dataset

One of the first steps in the Machine Learning pipeline is the dataset preparation. In the Packing-Box framework, all dataset manipulations are done via the abstraction tool `dataset` and takes a command as the second positional argument. This tool allows to execute many different manipulations as the help message hereafter shows.

```
┌──[user@host]──[~]─────────
$ dataset --help
<<snipped>>
This tool aims to manipulate a dataset in many ways including its creation,
   enlargement, update, alteration, selection, merging, export or purge.
<<snipped>>
CMD             command to be executed
  [create/modify/delete]
    alter       alter the target dataset with a set of transformations
    <<snipped>>
    ingest      ingest samples from a folder into new dataset(s)
    make        add n randomly chosen executables from input sources to the
   dataset
    purge       purge a dataset
    remove      remove executables from a dataset
    select      select a subset of the dataset
    update      update a dataset with new executables
  [read]
    browse      compute features and browse the resulting data
    <<snipped>>
    show        get an overview of the dataset
    view        view executables filtered from a dataset
<<snipped>>
```

Another useful positional argument is the `dataset plot` that is used to plot and compare features or information gain in a given input dataset. The available commands are displayed below, along with an example of usage to plot the feature `number_wx_sections` of a mixed dataset. The resulting figure was shown in Figure 5.21.

```
┌──[user@host]──[~]─────────
$ dataset plot --help
<<snipped>>
positional arguments:
  {characteristic,features,features-compare,infogain,infogain-compare,
   labels,samples}
                        command to be executed
    characteristic      plot reduced data highlighting a characteristic
    features            distribution of one or multiple features (bar chart)
    features-compare    compare feature difference with other datasets
    infogain            sorted distribution of information gains for the
   selected features (bar chart)
    infogain-compare    compare information gain difference between this
   dataset and others
    labels              distribution of labels in the dataset (pie chart)
    samples             plot each executable from the target dataset
<<snipped>>

┌──[user@host]──[~]─────────
$ dataset plot features mix_bl number_wx_sections
00:00:01.218 [INFO] Computing features...
<<snipped>>
00:00:04.387 [INFO] Saving to <<snipped>>/number_wx_sections.png...
```

B.1.3 Vizualizer

This abstraction is useful for generating visualizations of binaries to compare an original PE file and its packed versions, it highlights their sections and plots their entropy and features comparing them side by side. An overview of the most useful commands used in our work are shown in the help message below.

```
┌──[user@host]──[~]─────────
$ visualizer --help
<<snipped>>
  [visualization]
    compare   compare files from two sources
    features  compute features for files matching the regex with the input
   labels
    plot      plot files matching the regex given the selected labels
<<snipped>>
Usage examples:
  visualizer compare "PsExec.exe\$" samples-folder "PsExec.exe\$"
   altered-samples-folder
  visualizer features . dataset-packed-pe --label Exe32Pack
  visualizer find . dataset-packed-pe --max-not-matching 2 --exclude outliers
   --do-not-display
  visualizer plot "PsExec.exe\$" PackingData -l not-packed -l MEW -l NSPack -l
   RLPack -l UPX
```

B.1.4 Model

This utility purpose is to train Machine Learning models based on a given dataset. It supports various algorithms, Grid Search cross-validation and allows to select a data scaler for the preprocessing.

```
┌──[user@host]──[~]─────────
$ model -help
<<snipped>>
positional argument:
  CMD           command to be executed
  [create/modify/delete]
    edit        edit the performance log file
    purge       purge the selected model
    rename      rename the selected model
    train       train a model on the given dataset
  [read]
    browse      browse input data (including predictions) based on the
   selected criteria
    compare     compare the selected model with others
    list        list all the models from the workspace
    preprocess  preprocess features and visualize the result
    show        get an overview of the model
    test        test the model on a given input
    visualize   visualize the model
<<snipped>>
```

Model train is a command used to train a model using an input dataset, it allows to select a ML algorithm and many parameters for fine tuning.

```
┌──[user@host]──[~]─────────
$ model train -help
<<snipped>>
positional arguments:
  dataset    dataset for training the model

options:
<<snipped>>
  -A ALGO, --algorithm ALGO
                        machine learning algorithm to be used
                        - ab      : Adaptive Boosting
                        - bn      : Bayesian Network
                        - bnb     : Bernoulli Naive Bayes
                        - d       : Decorate
                        - dl85    : DL8.5 (optimal decision tree)*
                        - dt      : Decision Tree*
                        - gb      : Gradient Boosting*
                        - gnb     : Gaussian Naive Bayes
                        - j48     : Decision Tree (J48)
                        - knn     : k-Nearest Neighbors*
                        - lp      : Linear Perceptron
                        - lr      : Logistic Regression
                        - lsvc    : Linear Support Vector Classification*
                        - mlp     : Multi-Layer Perceptron
                        - mnb     : Multinomial Naive Bayes
                        - rf      : Random Forest*
                        - svm     : Support Vector Machine*
                        - ac      : Agglomerative Clustering
                        - ap      : Affinity Propagation clustering
                        - birch   : BIRCH clustering
                        - dbscan  : Density-Based Spatial Clustering of
    Applications with Noise
                        - kmeans  : K-Means clustering
                        - mbkmeans: Mini-Batch K-Means clustering
                        - ms      : Mean Shift
                        - optics  : Ordering Points To Identify the Clustering
    Structure
                        - sc      : Spectral Clustering
                         * supports Grid Search cross-validation
                         (default: dt)
<<snipped>>
```

A simple example is presented below where a model is trained using the Gaussian Naive Bayes algorithm on a dataset of 50 PE formatted samples from which 26 are packed with the UPX software. Then the model is tested on another dataset, after which the accuracy, precision and other values are printed out.

```
┌──[user@host]──[~]─────────
$ model train upx50 -A gnb
<<snipped>>
00:00:04.557 [INFO] Training model...
00:00:04.558 [INFO] [1/2] MinMaxScaler
00:00:04.569 [INFO] [2/2] Gaussian Naive Bayes

Name: upx50_pe32-pe64_50_gnb_f118

Classification metrics
    .      Accuracy   Precision   Recall    F-Measure     MCC        AUC
 ──────────────────────────────────────────────────────────────────────────
  Train   100.00%    100.00%     100.00%   100.00%     100.00%   100.00%
  Test    100.00%    100.00%     100.00%   100.00%     100.00%   100.00%
<<snipped>>
┌──[user@host]──[~]─────────
$ model list
Models (1)

             Name                Algorithm       Description        Dataset
   Size     Formats     Packers
 ──────────────────────────────────────────────────────────────────────────
   ──────────────────────────
  upx50_pe32-pe64_50_gnb_f118   GNB         Gaussian Naive Bayes   upx50
   50      PE32,PE64    upx{26}

┌──[user@host]──[~]─────────
$ model show upx50_pe32-pe64_50_gnb_f118
Model characteristics

 · Path:
   /mnt/share/experiments/test_exp/models/upx50_pe32-pe64_50_gnb_f118
 · Size:           13KB
 · Algorithm:      Gaussian Naive Bayes (GNB)
<<snipped>>

 ┌──[user@host]──[~]─────────
$ model test upx50_pe32-pe64_50_gnb_f118 upx
00:00:01.615 [INFO] Test dataset:  upx
00:00:01.616 [INFO] Computing features...
100% ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 123/123 samples • 0:00:05 •
   0:00:00

Classification metrics
  Accuracy   Precision   Recall   F-Measure    MCC    AUC   Processing Time
 ──────────────────────────────────────────────────────────────────────────
  5.69%      0.00%       0.00%    0.00%       0.00%   -     6.573ms
```

B.2 Real Section Names

In the PE format, section names longer than 8 bytes are replaced by an offset in the String Table where the **real section names** are stored. In the existing implementation inside the Packing-Box, the framework relied on the `objdump` tool to retrieve them. To avoid relying on an external tool that may output unexpected values, we improved the parsing of real section names, by implementing the parsing without relying on

"objdump" anymore. Listing B.1 shows the original implementation relying on objdump.

```
_____________________________________________________________________
OLD APPROACH USING OBJDUMP
_____________________________________________________________________
def real_section_names(self):
    """ This only applies to PE as section names are limited to 8 characters for image
    ↪  files ; when using longer
         names, they are mapped into a string table that 'objdump' can read to recover
↪  the real section names. """
    if self.path.group != "PE":
        raise AttributeError(f"'{self.__class__.__name__}' object has no attribute
        ↪  'real_section_names'")
    if not hasattr(self, "_real_section_names"):
        names = [ensure_str(s.name) for s in self]
        from re import match
        if all(match(r"/\d+\$", n) is None for n in names):
            self._real_section_names = {}
            return self._real_section_names
        real_names = []
        out, _ = execute(["objdump", "-h", str(self.path)])
        for l in out.decode("latin-1").split("\n"):
            m = match(r"\s+\d+\s(.*?)\s+", l)
            if m:
                real_names.append(m.group(1))
        self._real_section_names = {n: rn for n, rn in zip(names, real_names) if
        ↪  match(r"/\d+\$", n)}
    return self._real_section_names
```

Listing B.1: Implementation to parse real section names from the string table

Listing B.2 presents the updated implementation that parses the String Table to retrieve real section names. In this code snippet, we first calculate the string_table_offset by determining the address of the Symbol Table and then adding the product of the number of symbols and 18, which is the size of each entry in the Symbol Table. This calculation gives us the offset to the String Table, which, according to the official PE format specifications, is always located immediately after the Symbol Table.

```
    ____________________________________________________________________
    NEW : WITHOUT OBJDUMP
    ____________________________________________________________________
def __decode_section_name(self, pe, encoded_name, file):
        if encoded_name.startswith('/'):
            offset = int(encoded_name[1:])
            string_table_offset = pe.header.pointerto_symbol_table +
            ↪  pe.header.numberof_symbols * 18
            real_name_offset = string_table_offset + offset

            # Seek to the calculated offset
            file.seek(real_name_offset)

            # Read the null-terminated string from the file
            real_name = b''.join(iter(lambda: file.read(1), b'\x00')).decode('utf-8',
            ↪  errors='ignore')

            return real_name
        else:
            return encoded_name

@property
def real_section_names(self):
    """ This only applies to PE as section names are limited to 8 characters for image
    ↪  files ; when using longer
         names, they are mapped into a string table that 'objdump' can read to recover
↪  the real section names. """
    if self.path.group != "PE":
        raise AttributeError(f"'{self.__class__.__name__}' object has no attribute
        ↪  'real_section_names'")
    if not hasattr(self, "_real_section_names"):
        names = [ensure_str(s.name) for s in self]
        from re import match
        if all(match(r"/\d+\$", n) is None for n in names):
            self._real_section_names = {}
            return self._real_section_names
        real_names = []
        with open(self.path, "rb") as f:
            for encoded_name in names:
                real_name = self.__decode_section_name(self, encoded_name, f)
                real_names.append(real_name)

        self._real_section_names = {n: rn for n, rn in zip(names, real_names) if
        ↪  match(r"/\d+\$", n)}
    return self._real_section_names
```

Listing B.2: Implementation to parse real section names from the string table

C YARA DETECTION

C.1 Additional detection example

Virustotal

To test our adversarial tool, we uploaded to Virustotal an UPX packed example executable and an altered version, then we compared the results. As we can see in Figure C.1, the UPX packed executable was flagged as packed and correctly labeled as UPX. In the figure in the right side, we observe that the detection was not able to label the altered executable as packed. This shows our attack tool successfully evades most static packing detection tools.

(a) UPX packed executable (b) Altered executable

Figure C.1: Executable packing detection using Virustotal

Interestingly, we also notice that their version of Detect It Easy (DIE) believed our altered sample was packed with VMProtect. After further analysis of the DIE rule that triggers, we discovered the reason for that behavior. A condition in this rule was triggered whenever a CALL instruction is made using the 0xFF opcode within the first 200 bytes after the entrypoint. This instruction is a `CALL <eax/ebx/..>` procedure using a register as operand.

This rule in Detect It Easy (DIE) was not triggering in the DIE inside the Packing Box, this probably means that these rules were updated since the implementation of the Packing Box.

We thus updated the alteration to inject more Dead Code instructions after the entry point, this workaround was successful to bypass that detection.

(a) UPX packed executable (b) Altered executable

Figure C.2: (V2) Executable packing detection using Virustotal

Figure C.2 shows the packing detection of Virustotal using the updated alteration. We observe now that no packing flag has been triggered at all, successfully evading the detection.

C.2 Advanced YARA detection

Interestingly, we discovered that our altered binaries are still detectable by advanced signature-based detection methods that does not only rely on the bytes after the EP but also search for sequences of known bytes throughout the entire executable. This approach is **very slow and resource intensive**, because it searches for a pattern across the entire file, which has a $O(n)$ complexity. We found a YARA rule for the detection of UPX in an open source project on Github [1], as shown in Figure C.3. Using this rule with the YARA tool on our altered executable, we confirmed that it is indeed detected. However, this rule includes not only one but eight patterns to search in the entire file, making it even more resource intensive.

```
┌──[user@host]──[~]──────────
$ yara upx2.yara upx_7z_out.exe
UPX upx_7z_out.exe
```

```
rule UPX {
    strings:
        $noep1 = { B8 ?? ?? ?? ?? B9 ?? ?? ?? ?? 33 D2 EB 01 0F 56 EB 01 0F E8 03 00 00 00 EB 01 0F EB
01 0F 5E EB 01 }
        $noep2 = { 5E 89 F7 B9 ?? ?? ?? ?? 8A 07 47 2C E8 3C 01 77 F7 80 3F ?? 75 F2 8B 07 8A 5F 04 66
C1 E8 08 C1 C0 10 86 C4 29 F8 80 EB E8 01 F0 89 07 83 C7 }
        $noep3 = { 01 DB [0-1] 07 8B 1E 83 EE FC 11 DB [1-4] B8 01 00 00 00 01 DB }
        $noep4 = { 9C 60 E8 00 00 00 00 5D B8 B3 85 40 00 2D AC 85 40 00 2B E8 8D B5 D5 FE FF FF 8B 06
83 F8 00 74 11 8D B5 E1 FE FF FF 8B 06 83 F8 01 0F 84 F1 }
        $noep5 = { 8A 06 46 88 07 47 01 DB 75 07 8B 1E 83 EE FC 11 DB }
        $noep6 = { FF D5 80 A7 ?? ?? ?? ?? ?? 58 50 54 50 53 57 FF D5 58 61 8D 44 24 ?? 6A 00 39 C4 75
FA 83 EC 80 E9 }
        $noep7 = { 55 FF 96 ?? ?? ?? ?? 09 C0 74 07 89 03 83 C3 04 EB ?? FF 96 ?? ?? ?? ?? 8B AE ?? ??
?? ?? 8D BE 00 F0 FF FF BB 00 10 00 00 50 54 6A 04 53 57 }
        $noep8 = { FF D5 8D 87 ?? ?? ?? ?? 80 20 ?? 80 60 ?? ?? 58 50 54 50 53 57 FF D5 58 61 8D 44 24
?? 6A 00 39 C4 75 FA 83 EC 80 E9 }
        $ep1 = { 60 E8 00 00 00 00 58 83 E8 3D }
        $ep2 = { 60 E8 00 00 00 00 83 CD FF 31 DB 5E }
        $ep3 = { 50 BE ?? ?? ?? ?? 8D BE ?? ?? ?? ?? 57 83 CD }

    condition: any of ($noep*) or for any of ($ep*) : ($ at entrypoint)
  }
```

Figure C.3: Advanced YARA rule for UPX detection

[1] https://github.com/SpiderLabs/yara-ruby/blob/master/spec/samples/upx.yara

D NotPacked++

D.1 Attack Tool

This section outlines multiple usage scenarios of our adversarial tool. Available options are shown in the help message below:

```
┌──[user@host]─[~]─────────
$ ./notpacked++ --help
    _   __       __  ____                 __                __
   / | / ____   / /_/ __ \____ _______/ /_____  ____/ / __    __
  /  |/ / __ \/ __/ /_/ / __ '/ ___/ //_/ _ \/ __  __/ /___/ /_
 / /|  / /_/ / /_/ ____/ /_/ / /__/ ,< /  __/ /_/ /_  __/_  __/
/_/ |_/\____/\__/_/    \__,_/\___/_/|_|\___/\__,_/ /_/   /_/


 Authors      :   Jaber RAMHANI, Alexandre D'Hondt
 Version      :   0.1
 Copyright    :   (c) 2021-2024 Alexandre D'Hondt, Jaber Ramhani
 License      :   GNU General Public License v3.0
=========================================================

Description: This program applies some alterations to a PE file.
 Note that when no alteration is specified ALL of them will be applied, if at
   least one is specified only selected ones will be applied

Usage: ./notpacked++ <input_file> [OPTIONS]

    -o <output_file>  : Set the output file name.
   (default:<input_file>_out.exe)
    --help            : Display this help message.

Other options: (by default the behavior is --fill-sections --permissions)
    --fill-sections   : Fill sections with zeros from their raw size to their
   virtual size.
    --rename-sections : Rename packer sections to standard section names.
    --permissions     : Update the permissions of all sections to standard
   ones (rwx/rw-/..), moves the EP to a new section and renames sections.
    --raw-size    : Edit the raw size value in the header of sections having a
   0 raw size (without adding real data bytes).
```

D.1.1 Scenario 1

The first scenario of usage of NotPacked++ is to simply use default alterations defined in the tool. This can be done by simply providing the filename as input without any parameters after. An example of this command is shown below where the super-alteration that changes the section's permissions is applied with the alteration that fills sections with zeros.

```
┌──[user@host]──[~]─────────
$ ./notpacked++ example.exe
<<snipped>>
[INFO]   Filling sections with zeros from their raw size to their virtual
   size...
[.] Truncating data to fit available space: 110592 bytes
[.] Truncating data to fit available space: 2560 bytes
[.] Truncating data to fit available space: 2048 bytes

[INFO]   Updating the permissions of all sections to standard ones
   (rwx/rw-/..), moving the entry point to a new section and Renaming
   sections to standard ones...
[+] Renaming section UPX0 to .data
[+] Renaming section UPX1 to .rdata

[SUCCESS]  File saved as: example_out.exe
```

D.1.2 Scenario 2

Another scenario of usage would be to perform a specific alteration to a binary file, by selecting the argument in the command line. An example is given below where we *rename sections* and *fill sections* of the input file "example.exe" and output the file as "just_a_normal_pe_file.exe".

```
┌──[user@host]──[~]─────────
$ ./notpacked++ example.exe --rename-sections --fill-sections -o
   just_a_normal_pe_file.exe
<<snipped>>
[INFO]   Renaming packer sections to standard section names...

[INFO]   Filling sections with zeros from their raw size to their virtual
   size...
[.] Truncating data to fit available space: 110592 bytes
[.] Truncating data to fit available space: 2560 bytes
[.] Truncating data to fit available space: 2048 bytes

[SUCCESS]  File saved as: just_a_normal_pe_file.exe
```

The alterations selected using the tool are executed in the order in which the options are provided.

D.2 Dataset x64

During our experiment we made sure our alterations maintains the executable's functionality for both x86 and x64 formats. However, as shown below, the reference datasets [68] did not include any x64 console applications. To temporally address this, we created our own executable, which imports some unused libraries and prints "**SUCCESS**" to the console

```
┌──[user@host]──[~]─────────
$ file dataset-packed-pe/**/* | grep PE32+
dataset-packed-pe/not-packed/SmartDefragBootTime.exe:
                  PE32+ executable (native) x86-64, for MS Windows, 5 sections
dataset-packed-pe/packed/Amber/amber_SmartDefragBootTime.exe:
                  PE32+ executable (native) x86-64 (stripped to external PDB),
   for MS Windows, 11 sections
```

```cpp
#include <iostream>

int main() {
    std::cout << "SUCCESS" << std::endl;
    return 0;
}
```

Listing D.1: Implementation of a simple x64 executable

D.3 Edit Raw Size

This alteration edits in the header the raw size value of section having a size of 0, without impacting the real file size. This section discuses the limitation found in the parsing library used and the solution to address this limitation. In a second part, it presents some interesting observation made during our experiments with this alteration.

D.3.1 Parsing

Our attack tool uses the LIEF library to parse PE executables. However, the newly added alteration that edits the raw size of a section without growing its real size, could not be performed using LIEF as it automatically adds bytes to fill the space according to the changes raw size value.

Because of this limitation of LIEF, we built our own parsing implementation for our adversarial tool.

```cpp
struct DOSHeader {
    uint16_t e_magic; // Magic number: 0x5A4D
    //<<snipped>>
    uint32_t e_lfanew;
};
struct PEHeader {
    uint32_t Signature; // PE\0\0
    //<<snipped>>
    uint32_t PointerToSymbolTable;
    uint32_t NumberOfSymbols;
    //<<snipped>>
};
struct OptionalHeader {
    uint16_t Magic; // 0x10B - PE32, 0x20B - PE32+
    //<<snipped>>
};
struct SectionHeader {
    uint8_t Name[8];
    union {
        uint32_t PhysicalAddress;
        uint32_t VirtualSize;
    } Misc;
    uint32_t VirtualAddress;
    uint32_t SizeOfRawData;     // Our Target for Raw-Size-Edit
    uint32_t PointerToRawData;
    //<<snipped>>
};

int parser(){
    // Example
    std::ifstream file(filePath, std::ios::in | std::ios::binary);

    DOSHeader dosHeader;
    file.read(reinterpret_cast<char*>(&dosHeader), sizeof(DOSHeader));
    // verify magic number
    if (dosHeader.e_magic != 0x5A4D) {return 0;}

    file.seekg(dosHeader.e_lfanew, std::ios::beg);

    PEHeader peHeader;
    file.read(reinterpret_cast<char*>(&peHeader), sizeof(PEHeader));

    // verify magic number PE\0\0
    if (peHeader.Signature != 0x4550) {return 0;}

    std::vector<uint8_t> optionalHeaderData(peHeader.SizeOfOptionalHeader);
    file.read(reinterpret_cast<char*>(optionalHeaderData.data()),
    ↪  peHeader.SizeOfOptionalHeader);

    int numberOfSections = peHeader.NumberOfSections;
    int sectionHeadersOffset = dosHeader.e_lfanew + sizeof(PEHeader) +
    ↪  peHeader.SizeOfOptionalHeader;

    std::vector<SectionHeader> sectionHeaders(numberOfSections);
    file.seekg(sectionHeadersOffset, std::ios::beg);
    file.read(reinterpret_cast<char*>(sectionHeaders.data()), sizeof(SectionHeader) *
    ↪  numberOfSections);

    file.close();
    // <<snipped>>
}
```

Listing D.2: Implementation of our PE parser

D.3.2 Overlap

This alteration that edits in the header the raw size value of section maintains the executables functionality. Interestingly, it makes the section overlap with others and some parsers may be tricked into perceiving the edited section as the section containing the entry point. As shown in Figure D.1, the visualization tool in the Packing-Box failed to display the overlapped section correctly, and instead it shows all sections collapsed in the end of the file.

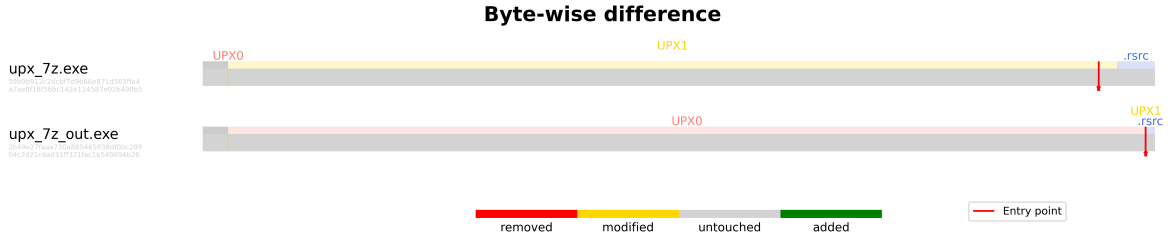
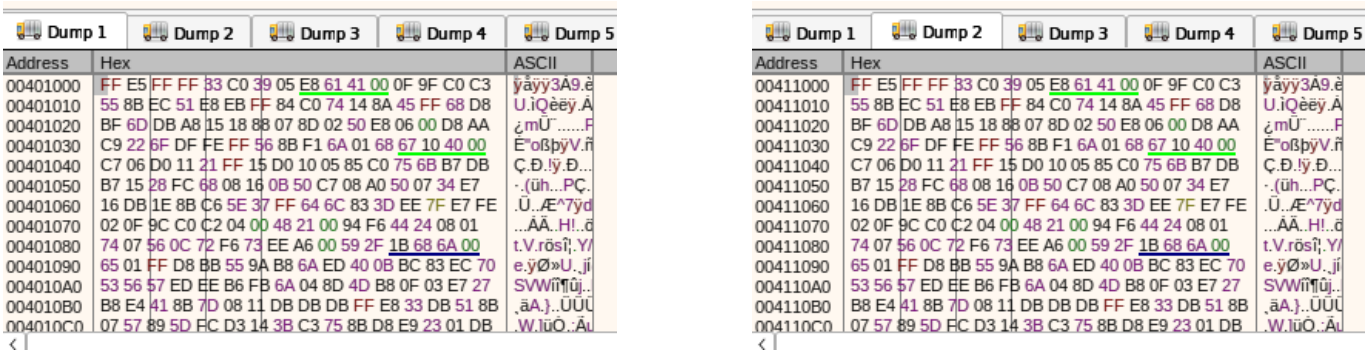


Figure D.1: Comparison of an executable before and after altering the raw size value

Another interesting observation is that the overlapping data is duplicated when loaded into memory, with each section containing a copy. This duplication is illustrated in Figure D.2, where the same data appears in both locations: at the start of section `UPX0` and also in `UPX1`.



(a) `UPX0` section

(b) `UPX1` section

Figure D.2: Memory dump of two overlapping sections