Generic Malware Unpacking: Existing Solutions, Requirements, New Approach for Windows Malware

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Thorsten Jenke

aus

Euskirchen

Bonn, 2025

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn.

Gutachter/Betreuer:	Prof. Dr. Peter Martini Rheinische Friedrich-Wilhelms-Universität Bonn
Gutachter:	Prof. Dr. Elmar Padilla Hochschule Bonn-Rhein-Sieg
Tag der Promotion:	6.10.2025
Erscheinungsjahr:	2025

Summary

Malware continues to be a substantial threat to cybersecurity, amplified by the widespread use of malicious executable packing, so-called packers. These packers inflate the number of unique samples in the wild by introducing polymorphism and hinder and delay in-depth malware analysis, making unpacking an essential first step.

Security researchers are developing countermeasures against these packers, called unpackers. Early unpackers targeted specific packer types, but the variety of different packers prompted the development of generic malware unpackers. Such tools aim to unpack the original binaries without prior knowledge of used packer's properties and capabilities, relying instead on broadly applicable assumptions about packer behavior. The more generic the assumptions are made, the more generic the unpacker. However, the lack of empirical understanding of packer capabilities has forced researchers to rely on subjective experience in practical malware analysis when defining these assumptions.

To remedy this, this dissertation defines scientifically sound prerequisites for generic malware unpackers and demonstrates their application in a proof-of-concept tool. This is accomplished by conducting studies on packer capabilities, deriving unpacker requirements from those insights, and using these requirements as the basis to develop a generic malware unpacker.

Since Windows malware running on x86 and x86_64 processors is the most common type of malware, it is the main focus in this work. Two studies have been conducted to explore the unpacking behavior of Windows malware: one in a singular process and the second across multiple processes. The results of these studies have been used to formulate requirements for a Windows-focused generic malware unpacker. These requirements were then applied to evaluate the genericity of previously proposed solutions. No previously proposed unpacker meets all the requirements. The final steps demonstrate how a generic malware unpacker can be implemented based on the previously identified requirements. A new generic malware unpacker called GeMU is proposed and implemented into a proof of concept. Evaluation on three Windows malware data sets used in the unpacking behavior studies confirms that GeMU achieves high coverage across diverse samples.

Publications

This thesis is based on the following peer-reviewed publications:

- Jenke, Thorsten, Simon Liessem, Elmar Padilla, and Lilli Bruckschen. "A Measurement Study on Interprocess Code Propagation of Malicious Software." International Conference on Digital Forensics and Cyber Crime. Cham: Springer Nature Switzerland, 2023. [https://doi.org/10.1007/978-3-031-56583-0_18]
- Jenke, Thorsten, Elmar Padilla, and Lilli Bruckschen. "Towards Generic Malware Unpacking: A Comprehensive Study on the Unpacking Behavior of Malicious Run-Time Packers." Nordic Conference on Secure IT Systems. Cham: Springer Nature Switzerland, 2023. [https://doi.org/10.1007/978-3-031-47748-5_14]
- Jenke, Thorsten, Max Ufer, Manuel Blatt, Leander Kohler, Elmar Padilla, and Lilli Bruckschen. "Democratizing Generic Malware Unpacking." 2025 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). IEEE, 2025. [https://doi.org/10.1109/EuroSPW67616.2025.00010]

Contents

1 Introduction

Despite significant advances in computer system defense, the persistent threat of malware continues to pose a substantial risk to all digital systems. In April 2025, AV-Test found 952,604,358 unique malware samples for Microsoft Windows and an increase of 73,063,458 unique samples in 2024 [1]. To defend against this threat, automated tools are developed for early detection and mitigation. However, malware authors use tools to protect their software against detection and analysis to increase its longevity. These tools are called packers and obfuscate or compress code that is brought into an executable state during the run time of the malicious binary.

In turn, malware analysts responded by creating unpackers, most of which are tailored to specific off-the-shelf packers. Packer manufacturers and malicious actors countered this development by constantly changing their packers and developing custom packers, resulting in a high number of custom packers only used by singular actors or adjacent groups [2]. Analysts tried to counteract this development with generic malware unpackers. This arms race is known as the packer problem [2, 3].

The goal of a generic malware unpacker is to create a lasting solution to the packer problem by being able to remedy every kind of packer. Achieving this objective requires abstraction away from the individual packer implementations toward a common feature that underlies the unpacking methodology. Such an approach requires a deep understanding of the vast landscape of different packers. Moreover, the unpacker must be implemented in a way that is applicable in real-life scenarios.

Despite the development of generic malware unpackers since at least 2005 [4], the challenges posed by packers spread and persist, indicating that no lasting solution has been achieved. This dissertation investigates a method for constructing a generic malware unpacker from a blueprint that requires only a set of packed malware as input. This blueprint is exemplified by applying it to malware developed for Microsoft Windows that runs natively on x86 and x86_64 processor architectures. First, the capabilities and implementations of packers are explored and studied to identify their shared features [5, 6], allowing the categorization of packers along their shared characteristics. The resulting categories serve as an abstraction from the individual designs and implementation details of the different packers. Each category encapsulates a distinct capability, thereby providing the leverage to perform unpacking. Consequently, the ability to tackle each of these categories yields a list of requirements that a generic malware unpacker must fulfill to unpack every packer present in the data set used to explore the capabilities of packers. These requirements are then used to evaluate previous unpacking methodologies targeting x86 and x86_64 Windows malware. The assessment reveals that no previously developed generic malware unpacker is capable of unpacking all types of implementations discovered in the studies.

To fully address all aspects of the requirements, a new generic malware unpacker was developed and evaluated. This unpacker is called GeMU and is intended as a starting point for generic malware unpackers based on the lessons learned in this work.

1.1 Research Questions

This section describes the research questions that drive this thesis. The main driving motivation of this dissertation is to provide a blueprint for creating a generic malware unpacker.

The first step involves identifying the requirements a malware unpacker must adhere to be considered generic. This is explored in the first research question of this dissertation:

Research Question 1:
What are the requirements for a malware unpacker that it must meet to be generic?

The answer to this research question is a set of requirements, enabling comparison and evaluation of previous generic malware unpackers that are focused on the same packer ecosystem from which the requirements were derived. As such, the next research question concerns previous academic work in the field of generic malware unpacking to determine which, if any, generic malware unpackers are able to solve the packer problem. Therefore, the second research question is:

Research Question 2:
Do the previously proposed generic malware unpacker meet the requirements of a generic malware unpacker?

These requirements also inform the creation of a generic malware unpacking methodology, as it abstracts away individual packer designs, thereby enabling a generic solution to the packer problem. Therefore, the last research question is:

Research Question 3:
Is it possible to develop a malware unpacker that fulfills all the requirements of a generic unpacker and works perfectly on the data set used to create the requirements?

1.2 Contributions

This section summarizes the contributions of this dissertation, which have been published in peer-reviewed papers. Rather than reiterating those papers, this dissertation transfers their findings to a different context and improves them. The peer-reviewed papers are:

- Towards Generic Malware Unpacking: A Comprehensive Study on the Unpacking Behavior of Malicious Run-Time Packers [5]
- A Measurement Study on Interprocess Code Propagation of Malicious Software [6]
- Democratizing Generic Malware Unpacking [7]

1.2.1 Analysis of Past Research

To reveal the shortcomings of the related work, a thorough analysis of past generic malware unpackers is carried out. Terminological inconsistencies emerged: the term *unpacking* is applied variably, and some publications conflate it with other nomenclatures. Therefore, we propose a new definition for *unpacking* to reflect new developments in the field of generic malware unpacking. The unpacking heuristics in related work are also gathered and clearly stated. Lastly, the data sets used for evaluating generic malware unpackers are gathered and evaluated against the prudent practices defined by Rossow et al. [8]. The results reveal that no previous generic malware unpacker has been evaluated using a sufficient data set.

In short, the following contributions are provided:

- A detailed analysis of the nomenclatures used in literature.
- An overview of the unpacking heuristics used in previous generic malware unpackers.
- An analysis of the evaluation data sets against the prudent practices defined by Rossow et al. [8].
- A new definition for *unpacking*.

The analysis of the nomenclature and heuristics of the paper *Democratizing Generic Malware Unpacking* [7] are presented in Chapter 3.

1.2.2 Malware Unpacking Within a Single Process

To fill the gap in the general field of malicious run-time packers, the behavior of malware packers within a single process was explored in a study. A mathematical representation of malware unpacking is presented, and a PANDA-based malware packer measurement system [9] is implemented for the technical realization of our study. With this system, a study is performed using two data sets: a real-world malware data set called Malpedia [10], and a data set called dataset-packed-pe [11] containing binaries packed with off-the-shelf packers.

The following contributions are provided:

- A model for displaying the unpacking behavior of malware.
- A technical implementation for a packer measurement system as a plugin for PANDA.
- A study to examine the intra-process unpacking behavior of malware.

The findings of the paper *Towards Generic Malware Unpacking: A Comprehensive Study on the Unpacking Behavior of Malicious Run-Time Packers* [5] are presented in Chapter 4.

1.2.3 Multi-process Malware Unpacking

An often underappreciated topic in dynamic analysis and especially malware unpacking is the ability of malware to inject other processes. To address this gap, research was undertaken to investigate code propagations, which refers to the phenomenon of distributing code outside the current running process.

First, a definition of code propagations is provided, together with a mathematical definition for the topology of code propagations and the techniques employed. Using these definitions, a program is developed that relies on API-hooking to measure the code propagations of a given malware sample. Finally, this program is applied to Malpedia to identify the code propagation implementations observed in real-world scenarios and the corresponding techniques employed to achieve them.

In short, the following contributions are made:

- A model to display multi-process unpacking behavior of malware.
- A method for measuring code propagations.
- A study to examine multi-process unpacking behavior of malware.

The findings of the paper *A Measurement Study on Interprocess Code Propagation of Malicious Software* [6] are presented in Chapter 5.

1.2.4 Requirements for a Generic Malware Unpacker and new Malware Unpacker

Initially, the results of previous studies are used to abstract the implementation details of packers to formulate the requirements for a generic malware unpacker. These requirements provide a benchmark for evaluating the genericity of previous academic unpacking methodologies. Guided by this assessment, the most promising generic malware unpackers are combined to form GeMU, the generic malware unpacker. Implemented as a proof of concept, GeMU is evaluated using the three data sets, Malpedia, dataset-packed-pe, and one based on the samples uploaded to Malware Bazaar [12] in 2024. The evaluation demonstrates that deriving requirements for a generic malware unpacker from a set of packers enables the construction of an unpacker that works perfectly on these data sets.

The following contributions are made:

- A set of requirements for a generic malware unpacker.
- Evaluation of previous unpacking methodologies using these requirements.
- A methodology for a generic malware unpacker that meets all the requirements.
- A proof-of-concept implementation of this methodology.
- A study using two data sets to generate the requirements showing that the generic malware unpacker fulfills the requirements.

The remaining findings of the paper *Democratizing Generic Malware Unpacking* [7] are presented in Chapter 6 and Chapter 7.

All these contributions culminate in the central contribution of this dissertation:

Central Contribution:

This dissertation provides a blueprint for creating a generic malware unpacker based on a series of studies.

1.3 Outline

This dissertation illustrates that, despite extensive academic efforts, a generic malware unpacker remains unattainable with current knowledge. It addresses this knowledge gap through two different studies, whose results are used to assess previous generic malware unpackers and to create a new generic unpacker.

Chapter 2 establishes the required foundations. It elaborates on the background needed to grasp the themes, questions, and results presented in this dissertation, followed by an introduction to malware analysis, and concludes with a description of malware packing.

Chapter 3 examines prior research on past generic malware unpackers to reveal their shortcomings, which shall be remedied in this thesis. It collates and analyzes their terminology for malware unpacking, the assumptions and heuristics used in their methodologies, and their data sets, then proposes a plan to address these shortcomings and improve previous research.

Chapter 4 explores how malware writes and runs code within a single process. A definition of the malware unpacking model is given, followed by the development of a measurement system, which implements the unpacking model and enables observation of the unpacking behavior. Using this measurement system, a study is conducted showing the various ways malware writes and executes code within a single process.

Chapter 5 details a measurement study to explore code propagations in malware, i.e., how malware writes and executes code across multiple processes. A model for this type of behavior and the measurement method are introduced, followed by a study using real-life malware and an implementation of the measurement method.

Chapter 6 synthesizes these two studies to derive the requirements for a generic malware unpacker. These requirements are used to show that previous unpacking methodologies do not meet the requirements and are therefore incompatible with some packers encountered in the wild.

Equipped with this new knowledge, Chapter 7 presents GeMU, an implementation of a generic malware unpacker that meets all the requirements. This chapter describes GeMU's methodology and implementation, followed by an evaluation and discussion.

Chapter 8 concludes with a discussion of the dissertation's impact, acknowledges its limitations, and offers an outlook for possible future work.

2 Foundations

In this chapter, the technical foundations for understanding this dissertation are presented. To this end, malware analysis is introduced with a special emphasis on dynamic analysis. This is followed by a brief overview of different important terminologies from the Windows ecosystem, including the PE-Format, the Virtual Address Space, and the Windows API. After that, the foundations for executable unpacking are laid.

2.1 Malware Analysis

Malware analysis denotes the act of analyzing a malicious program to obtain information about its functionalities, capabilities, and behavior. The methods used in this process can be divided into two different fields of dynamic and static analysis. Dynamic analysis describes analysis techniques in which the entire code or pieces of it is executed, while static analysis omits execution. A mix of static and dynamic analysis is also a valid option for malware analysis, called a hybrid approach. The remainder of this chapter briefly talks about static analysis, as it is imperative to understand the goal of this thesis, followed by a description of dynamic analysis with a focus on virtualization and emulation [13].

2.1.1 Static Analysis

Static analysis describes the process of analyzing malware without executing any code. Instead, it is characterized by manual or automated assembly analysis, as well as signature-based analysis methods [13].

Therefore, code that is only revealed during the execution of the program thwarts static analysis [2]. There is very little work on static malware unpacking [14] and as a result a dynamic analysis step is often needed to reveal hidden code. This step can be performed manually using various tools, such as debuggers, to find the exact point in time when the unpacking has been performed [15].

After successfully revealing the hidden code layer, i.e. unpacking, static analysis on the entire code becomes possible. Since the malware is now in its decrypted form, the machine code can be disassembled and analyzed. This analysis may include the attribution of the sample to an actor or group of actors [16].

2.1.2 Dynamic Analysis

Dynamic analysis describes the process of analyzing malware by executing pieces of or all of the malware's code. Executing the malware's code allows observation of effects, possible return values, and other results of the executed functionality without trying to understand the effects by thoroughly studying the code [13, 15].

A crucial aspect to consider is that the malicious nature of malware requires a careful and responsible approach to dynamic analysis. Therefore, precautions must be taken when performing dynamic malware analysis. One way to prevent harm is by using isolated environments, such as virtual machines or emulation, which leads to the next section [13].

2.1.2.1 Virtualization and Emulation

Since the subjects of malware analysis should be considered dangerous programs that can cause great harm, several precautions should be taken to prevent malware from doing harm during analysis. Therefore, the lab setup should include a way to execute malware in an isolated and controlled environment, which also enables rapid disinfection of the machine. This level of control can be achieved using guest systems in which a full operating system is executed. In this dissertation, two different products are used to run a guest system.

The first is called VirtualBox and is provided by Oracle [17]. This tool provides a virtual machine, meaning a guest system that runs in a different processor mode which isolates the execution of the guest from the host system, enabling high performance.

The other approach used in this dissertation is the emulation of a guest system. This is provided by QEMU [18]. QEMU is a full-system emulator which, in contrast to VirtualBox, does not use a specific processor mode to isolate the guest. Instead, it implements hardware in software to run the guest system. As a consequence, emulation is much slower than virtualization but enables complete introspection in the execution of the guest, enabling more complex analysis.

2.1.2.2 Semantic Gap

A challenge of introspection is to bridge the semantic gap. The semantic gap describes the missing interpretation of the raw bytes of guest memory to obtain their high-level interpretation. To gain meaningful information about the state of the machine, the correct bytes at the correct addresses must be read and interpreted. The location and meaning of these bytes depend greatly on the operating system and the architecture of the hardware [19].

2.1.3 Evasion Techniques

Malware authors aim to thwart malware analysis to conceal their goals and motivations or hinder the development of countermeasures. This is done using so-called evasion techniques, which is an umbrella term for techniques that are not part of the malware's main functionality but instead serve the purpose of protecting the malware from detection, analysis, and scrutiny. Therefore, evasion techniques range from passive techniques that have been applied statically before malware is released to active techniques that only show their effect during execution [20].

One notable class of evasion techniques, which is encountered in this thesis, is based on detecting the environment in which the malware is executed. To protect the malicious code from analysis, the malware authors can anticipate the nature of the machines on which their malware shall not be executed. These types of machines can be virtual machines or emulations that the malware authors suspect are used in malware analysis [15, 20]. The evasion techniques encountered in this research alter the malware's behavior based on certain hardware configurations, specific strings encountered in the artifacts on the system, absent user interaction, or missing Internet connectivity.

In addition to this, malware frequently uses stalling code regions to thwart analysis. These evasion techniques use sleep functions, complex calculations, or repeated superfluous calls to the API to delay its execution [21]. This is done to exceed the sandboxing time of the analysis machine, overwhelm the machine with complex operations to halt its execution, or overwhelm the logging system of the analysis system with redundant information.

2.2 Microsoft Windows Foundations

This section outlines the basics of Microsoft Windows to improve the clarity of this dissertation. To fully grasp the technical background of malware unpacking, it is important to know some concepts of the portable executable format, how processes are handled in Microsoft Windows, and the Windows API.

2.2.1 PE-Format

Since 99% of Windows malware is encountered in the PE-format [1], this subsection highlights aspects of the PE-format that are needed to understand this dissertation. The PE-format is a specification to describe Windows files that contain executable code [22]. These files are called Portable Executables (PE) and Common Object File Format (COFF) files [22].

2.2.1.1 Sections

When a PE-file is loaded into memory, different sections of virtual memory need to be allocated for different parts of the image to be loaded [22]. Common compilers divide

00DD0000	00001000	System	calc.exe	IMG	-R---
00DD1000	00001000	System	".text"	IMG	ER---
00DD2000	00001000	System	".data"	IMG	-RW--
00DD3000	00001000	System	".idata"	IMG	-R---
00DD4000	00005000	System	".rsrc"	IMG	-R---
00DD9000	00001000	System	".reloc"	IMG	-R---

Figure 2.1: Screenshot of x64dbg showing the calc.exe of Windows being loaded into a process.

the PE-file data into different sections that have different permissions to protect the image from harmful or accidental manipulation. An example of this can be found in Figure 2.1. These sections are called image sections and are used by packers. UPX, for example, introduces the *.upx1* and *.upx2* sections to perform the unpacking [23].

2.2.1.2 Import Table

The import table details the symbols that are imported by the code contained in the PE-file. It also contains the library that shall be loaded and the symbols imported from that library. These symbols can be imported using either their name or an ordinal [22].

Hex

80 9D A2 77	50 E1 9F 77	30 C5 A2 77	20 43 A3 77
50 0B A0 77	00 00 00 00	90 3B A4 77	60 3D A4 77
80 42 A4 77	80 9D 9F 77	F0 3C A1 77	C0 C6 A1 77
90 86 A2 77	4C 69 AF 77	50 94 A3 77	80 1F A2 77
F0 3C A1 77	C0 56 A4 77	00 B1 A3 77	30 5B A3 77
E0 C7 A3 77	B0 10 A3 77	80 56 A4 77	F0 30 A9 77
F0 30 A4 77	A0 89 A2 77	90 83 A0 77	E0 A5 A0 77
C0 45 A4 77	30 33 A4 77	10 80 A4 77	50 BC 9F 77
90 11 A3 77	70 22 A3 77	40 FC 9F 77	50 03 A1 77
40 00 A1 77	B0 C6 A2 77	40 E8 A0 77	50 07 A0 77
40 46 A4 77	D0 ED A2 77	70 32 A4 77	10 1F A2 77
10 0C A5 77	80 0B A5 77	A0 0A A5 77	90 33 A4 77
90 34 A4 77	50 35 A4 77	D0 AE A2 77	20 30 A4 77
90 41 A4 77	20 CE A2 77	30 3A A4 77	F0 15 A2 77
60 33 A4 77	80 7B A0 77	10 70 A9 77	00 31 A4 77
60 1F A0 77	40 46 A4 77	30 21 A2 77	20 A7 A3 77
F0 99 A2 77	90 41 AB 77	60 44 A4 77	10 60 A9 77
E0 0C A3 77	B0 34 A4 77	20 D1 A3 77	80 34 A4 77
50 34 AB 77	90 C9 A9 77	30 67 A3 77	40 81 A4 77
40 AF A2 77	70 1E A0 77	80 AF A4 77	70 83 A2 77
50 31 A4 77	50 42 A4 77	30 7F A4 77	50 1F A2 77
00 32 A4 77	20 67 A3 77	00 3B A4 77	70 35 A0 77
D0 31 A4 77	00 36 A4 77	D0 0D A3 77	50 33 A4 77
30 62 A3 77	00 BC AA 77	90 C8 A3 77	80 20 A2 77
70 B9 AA 77	C0 BF AA 77	B0 AE A2 77	20 BC AA 77
00 BB AA 77	50 BF AA 77	60 8C A4 77	00 AD A4 77
60 AE A4 77	60 66 A3 77	50 46 A0 77	B0 F8 A2 77
00 05 A5 77	10 3D A0 77	80 3C A0 77	B0 FF A2 77
80 05 A5 77	80 7E A4 77	C0 32 A4 77	B0 90 A5 77
60 81 A5 77	C0 6E A4 77	30 A9 A3 77	20 33 A4 77
A0 3F A4 77	60 89 AB 77	60 26 A4 77	00 49 A4 77
80 86 A4 77	30 FA A4 77	20 AF A4 77	70 3D A4 77
C0 0B A3 77	30 5C A2 77	70 D8 9F 77	E0 5D A2 77
40 D9 9F 77	70 72 9F 77	90 BF 9F 77	10 31 A4 77
20 96 A4 77	20 44 A2 77	E0 52 A2 77	10 C7 9F 77

Figure 2.2: Screenshot of an import address table taken in x64dbg.

When an image is loaded into memory, the import table is resolved. To do so, the denoted PE-files are loaded into the memory space of the program. Each of these PE-files has a table of exported functions, i.e. functions that are meant to be called from outside its own image. This table is used to find references to the names or ordinals denoted in the import table. So, if the name or the ordinal refers to a function, then

the export table contains a reference to this function that can be used to call that function [22]. These references are gathered and put into the import address table, which is a buffer containing consecutive references that gives it an intuitive look, as can be seen in Figure 2.2.

2.2.2 Virtual Address Space

Since unpacking involves interactions with memory, this subsection presents an overview of how memory is managed in the x86 and x86-64 architectures. Therefore, this subsection introduces the virtual address space as implemented in Microsoft Windows and the fundamental data structures that make up and reside in the virtual address space [24].

The virtual address space (VAS) is a continuous virtual memory space that is made up of physical memory pages that are not continuous [24]. This space is addressed using virtual addresses by the CPU to directly read and write memory. The virtual addresses are mapped onto physical addresses, which point to the real physical memory on the hardware. Linear sequences of these addresses are fixed in length and are always mapped together. Such a sequence is called a memory page. The CR3 register holds a value that is used to identify the mapping between virtual and physical addresses. Therefore, whenever a virtual address is translated into a physical address, the value in CR3 is used. That means that it is an indicator of the running process and also provides a layer of isolation between the processes so that one process is unable to access the memory of another process [24]. So, the virtual address space enables the process to be executed in a predictable environment, since the dependency on volatile physical addresses is resolved in a step that is transparent to the process. For this reason, it is the main data structure for the execution of user code in Microsoft Windows and includes data structures that are vital for the execution of programs [24].

2.2.2.1 Section

The virtual address space is made up of sections that are related to the sections used in the PE-file as described above. A section is a continuous piece of memory that can be addressed and accessed by a program. These sections need to be explicitly allocated for them to exist and therefore to be accessed. However, they cannot be used freely as each section can be read-, write-, or execution-protected. These permissions can be changed using the Windows API and functions like *VirtualProtect* [25].

There are multiple ways in which a section can be allocated. As mentioned above, when a PE-file is loaded into memory, the sections listed in the header are allocated and filled according to the information in the PE-file. Another way is to use the Windows API with functions like *VirtualAlloc* [26]. However, some sections are allocated when the process is first spawned, i.e. the *Thread Environment Block* [27] and its substructures, i.e. the *Process Environment Block* [28].

2.2.2.2 Thread Environment Block and Process Environment Block

One of the main challenges in malware analysis is to bridge the semantic gap, as mentioned in Chapter 2.1.2.2. The *Thread Environment Block* (TEB) and the *Process Environment Block* (PEB) contain the information needed to identify the process that is currently being executed by the operating system. Therefore, this data structure can be used in dynamic analysis to identify the processes of interest.

The TEB can be located using the FS register for 32-bit or the GS register for 64-bit. The TEB in turn contains the PEB address, which contains the current unique process ID and the name of the process.

2.2.2.3 Stack

The stack is a fundamental data structure on the Intel architectures and in the Windows calling conventions. Each thread running in a process has its own stack. The stack is used to store local variables, function parameters, and the return address to reconstruct the original program flow after a function call.

2.2.2.4 Heap

Another central data structure is the heap, which is used for dynamic memory allocation. A process can have multiple heaps and is used for data that is needed in the context of multiple functions. Windows provides multiple functions to allocate memory on the heap and also to create new heaps [24].

2.2.3 API

The Windows API provides an interface to a plethora of functions that may or may not interact with the kernel. Before the Windows API can be called, the associated library must be loaded into the process memory, which is described in the previous section, Chapter 2.2.2. During the loading process, the library is read and mapped to memory. Therefore, the library code is part of the user mode. The kernel exists outside the user space and manages the interactions between processes and hardware components. The kernel can be extended with modules that can be utilized by malware to gain complete access to the operating system and hide its behavior and existence from the analysis system.

This dissertation focuses on Windows malware that does not contain a kernel component. Therefore, the malware analyzed in this dissertation performs parts of its functionality through the Windows API, making this behavior observable in the Windows API [15].

Malware may use the Windows API to achieve the same functionality in different ways. For example, to copy data from one buffer to another, malware may allocate a buffer and write data into that buffer by directly addressing it inside the code body of the

malware. It may also use functions such as *memcpy* [29] to copy data from one buffer to another. This function does not interact with the kernel, but instead the user-mode library performs the read and write operations. This makes these operations observable for a user-mode-focused dynamic analysis system. However, there is also the function *WriteProcessMemory* [30] that writes from one buffer to another. It offers the extended functionality that the copy operation may also target a process different from the calling one. Therefore, these write operations are performed from the kernel to the user space memory. The analysis system is evaded because the operations occur outside the user mode, although the call to *WriteProcessMemory* can still be observed.

The usage of functions can be observed dynamically or statically. Traditionally, imported functions can be found in the Import Table of the portable executable [31]. Modern dissemblers can attribute and annotate jumps in the Windows API directly in the disassembly [32, 33, 34]. In dynamic analysis, API calls are observed by comparing executed addresses with the addresses found in the import address table shown in Figure 2.2.

2.3 Runtime Packing

Runtime packing is the main focus of this dissertation, therefore, in this section, malware packing is described.

A packed executable refers to a program that has been compressed or modified. Upon execution of this program, this altered code is put into an executable state and executed. More generally speaking, a packed executable refers to a program that has one or more hidden code segments that are revealed during execution.

2.3.1 Types of Packers

There are different options for packers, ranging from open-source free unpackers [23] to closed source commercial products [35] with different implementations.

Historically [36], malware unpacking has been described as the act of restoring the original binary to its original form before it is packed. This stems from the assumption that packers are exclusively programs that take a standalone binary as input and produce a packed or compressed binary. The point in which the malware is fully unpacked, and the execution is handed to the unpacked code is called a tail-jump. However, several studies [2, 37, 5] have shown that the landscape of packed malware is much more complex. Therefore, for the remainder of this thesis, a packer is a piece of code creating hidden code that is revealed during execution. Code is hidden if it cannot be observed by simply reading the binary code of the packed executable. In turn, a packed executable is characterized by the presence of hidden code.

In generic malware unpacking, packers are interpreted from the artifacts and modifications inside the binary that is encountered in the wild. Ugarte-Pedrero et al. formalized six different types of packers [2]:

- Type I: Singular tail-jump at the end of the packer routines.
- Type II: Multiple tail-jumps in a row.
- Type III: All packers with more complex topologies.
- Type IV: The execution trace is intertwined between malware and packer stub. The malware triggers part of the packers, and the execution jumps back and forth.
- Type V: Malware and packer code are mangled together, and the malicious code is revealed successively.
- Type VI: Encrypt and decrypt malicious code on demand. These packers are further divided into different levels of granularity.

Type I is understood to be the simplest type of packer. It consists of exactly one packer with a single buffer, which directly contains the payload. A typical example of this type is the UPX packer [23] which is also well known because it is open source and available for many platforms.

Type II and Type III describe more complex packer structures. Type II is a series of packers in a straight line, i.e. the obfuscated buffer contains another packer and another obfuscated buffer. Type III is a catchall group that contains all other possible packer topologies [2].

The last three types are subtypes of Type III and describe the position of the payload code in relation to the packer code. Type IV means that the malware and packer code call each other, but they are two distinct entities. Type V describes binaries in which malware and packer code are mangled together, and Type VI is another special type of packer in which the malware code is decrypted before being called and encrypted after execution [2].

This taxonomy reinforces the notion that the packer code and the malware code are two different entities. However, the author of this dissertation conducted a study [5] that challenges this notion. Rather, packer functionality is a property that can be added to any kind of malware. The definition of packer and malware exists on a spectrum, which in turn means that packer should be understood as a functionality that is added to a given malware rather than two different entities.

2.3.2 Packing and Unpacking Steps

To understand how unpacking works, it is first shown how, in general, a packer packs a binary. There are a variety of ways to achieve this. However, the general steps are as follows:

- Step 1: Encrypt/Encode/Compress the buffer of protection-worthy code.
- Step 2: Include the code to decrypt/decode/decompress the buffer called packer stub in the binary that will be released.
- Step 3: Insert the packer stub and subsequent execution of the decrypted/decoded/decompressed code into the execution of the malware.

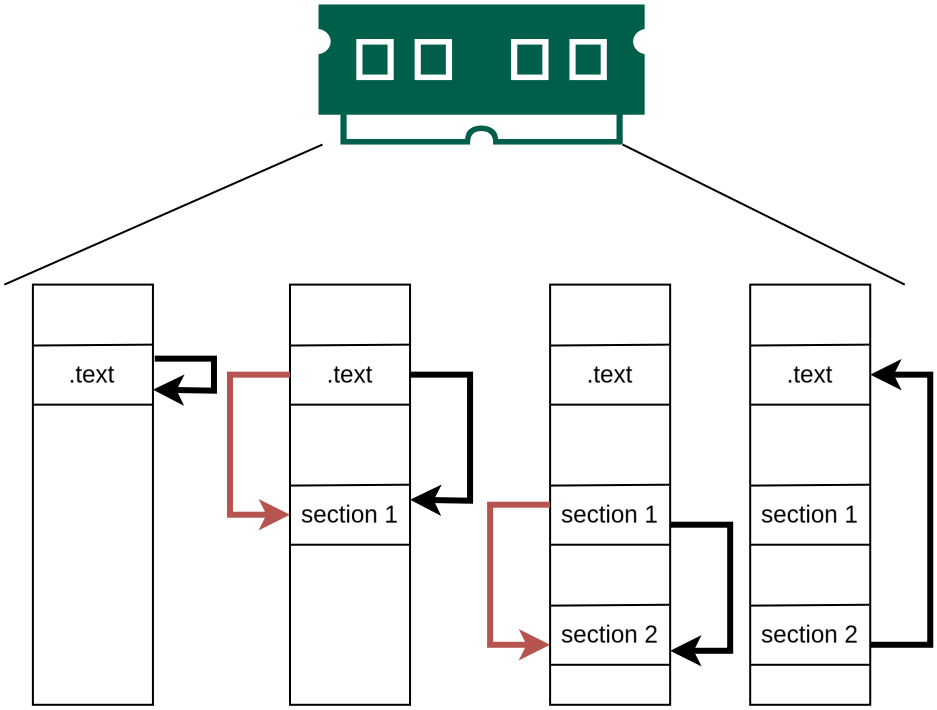

Figure 2.3: This figure shows the unpacking behavior of Opachki. The black arrows denote the writing and execution of code. The red arrows show a memory allocation using VirtualAlloc.

Step 1 is the most diverse step, as there is a multitude of different approaches to this. There are packers that take an unpacked binary as input and produce a functionally identical packed binary, such as UPX [23]. However, it is also possible to implement packing via compiler macros, which enable a more fine-grained obfuscation of functions that are especially worth protecting.

As previously established, the unpacking process of a given malware sample can include one or more unpacking steps. These unpacking steps consist of several steps.

- Step 1: Decrypt/Decode/Decompress the buffer containing the next unpacking layer or payload.
- Step 2: Write the unpacked code to a piece of memory.
- Step 3: Transfer the execution to the unpacked code.

Step 1 varies very strongly from packer to packer, due to the selection of readily available compression, encryption, or encoding schemes. This is the step that packer-specific malware unpackers take to recreate. The unpacking algorithm is identified and reimplemented as a countermeasure to this specific packer. However, generic malware unpackers aim to use the vulnerabilities exposed in Step 2 and Step 3 for their methodologies.

2.3.2.1 Real-World Example Opachki Malware

To provide the reader with a more intuitive understanding of what malware unpacking is, this section presents the unpacking behavior of a particular packed sample of the Opachki family. This process is depicted in Figure 2.3. This example has been chosen because it demonstrates how multiple unpacking layers are used during the unpacking, as well as the allocation of new memory and the overwriting of previously executed code.

When the sample is launched, the code inside the *.text* image section is executed. At this point, new code is generated, written into the same *.text* section, and executed. This is the first unpacking layer. This layer allocates a new section using VirtualAlloc [26] and fills it with data and code. The written buffer starts with a few magic bytes and the string "Mystic Compressor", which is a custom packer specialized in malware. It is suggested to have been developed in Russia [38] and appears to be used in the context of the Zeus and Opachki malware families [39, 40]. This is the second unpacking layer. Using this packer as a means to identify threat actors has only been possible because this intermediate unpacking layer is also analyzed. This means that these intermediate layers may also yield useful information, which will become important later in this thesis, when discussing the necessity of unpacking every stage of the unpacking process.

Afterwards, a new section is allocated again, using VirtualAlloc [26]. Now, the identical code from the last unpacking step is written in the new section, with a different piece of this code being executed. This is the third unpacking layer.

This third layer now overwrites the original binary code pieces, as well as the code unpacked in the first layer, which reside in the *.text* image section. Subsequently, this new code is executed.

2.4 Conclusion

This chapter discussed the foundations for understanding unpacker methodologies. First, the basics of malware analysis were given by introducing static and dynamic analysis. The dynamic analysis is further elaborated on through the lenses of virtualization and emulation, which are both central to this thesis. Secondly, the fundamental concepts of Microsoft Windows were introduced, which are integral to understanding malware unpacking. Lastly, malware packing was examined more closely by introducing the different types of packers and explaining the typical workflow of an unpacking step, which is further supported by a real-life example.

Now that the fundamentals for malware unpacking have been discussed, the next chapter will provide a study on previous generic malware unpackers.

3 Malware Unpacking in the Literature

The analysis of terminology, unpacker description, and the definition of unpacking presented in this chapter is based mainly on prior research carried out by the author of this dissertation. The material has been restructured from an independent article to align with the structure of this dissertation. The author of this dissertation has published the original work in the following paper:
Jenke, Thorsten, Max Ufer, Manuel Blatt, Leander Kohler, Elmar Padilla, and Lilli Bruckschen. "Democratizing Generic Malware Unpacking." 2025 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). IEEE, 2025. [7]

This chapter reviews generic malware unpackers published between 2005 and 2019, as no contributions were found outside this time frame. The purpose of this chapter is to highlight the shortcomings of previous research and to outline how this dissertation seeks to address them. To this end, the related work is analyzed in three aspects.

The first concerns the **terminology** used to discuss the contributions and findings of each study. To present their findings clearly, malware authors must define the specific behavior and obfuscation of the malware that their tool mitigates, along with their tool's intended function. This terminology varies between publications, and identical terms may carry different meanings depending on context. Therefore, this chapter also examines whether the contribution distinguishes between malware and the packer.

The second aspect concerns the **assumptions** and **heuristics** employed by generic malware unpackers. To tackle multiple packers at once, such unpackers must rely on assumptions about packer behavior. Broad assumptions enable broader applicability; in general, broader assumptions about packers lead to more generic unpackers. This necessity has led to the development of various heuristics designed to detect when unpacking occurs, thereby enabling the extraction of the maximum amount of unpacked code at the optimal time. By defining which behaviors belong to the unpacking process, the heuristic governs how unpacking is observed and measured. Therefore, this section discusses the heuristics and assumptions that previous work made to build their unpackers. The effectiveness and implementations of their heuristics are evaluated in Chapter 6.

The third aspect concerns the **data sets** used to evaluate the presented approaches. These evaluations serve to demonstrate the feasibility and correctness of the methodology and its implementation. Correctness means that the methodology produces the correct output, that is, the desired unpacked data. Since no formal requirements define what a generic malware unpacker must be capable of, theoretical evaluation is not feasible. Practical evaluation is therefore required, which involves implementing the unpacker as a

proof of concept and testing it on a data set. Consequently, the quality of the evaluation depends directly on the quality of the data set. Therefore, the data sets are analyzed in this chapter.

3.1 Generic Malware Unpackers in Literature

This section presents several works focused on the development of generic malware unpackers. Each description highlights the heuristics, technologies, and unpacking terminology used, as well as the data sets and evaluation methodologies. If the technologies used for the implementation are not mentioned, it means that this information could not be inferred from the respective publication.

Malware Normalization

Malware Normalization by Christodorescu et al. [4] (2005) introduces an unpacker that uses the write-then-execute heuristic also called WxE. Figure 3.1 displays an automaton that describes how the write-then-execute heuristic distinguishes any written data code from unpacked code. This heuristic relies on the assumption that during unpacking, code is written to memory and subsequently executed. Thus, if a piece of memory is written to and then executed, it is assumed to contain unpacked code. This is a very broad assumption about the behavior of packers and malware and, therefore, results in false positives [5]. Nevertheless, it has become widely used and has been adopted in numerous unpacking methodologies, with two established granularities: byte [41, 42, 2, 43, 44, 45, 46, 4] and page [3, 47] level. The byte-level approach is very fine-grained but introduces a substantial performance overhead due to the amount of information. This motivated the development of the page-level approach, where a write access to a page marks it as dirty, and subsequent execution indicates an unpacking step. Compared to the byte-level write-then-execute scenario, this approach yields an enormous performance improvement, as not every write operation needs to be tracked. Christodorescu et al. implemented their unpacker in QEMU and evaluated their approach using seven off-the-shelf packers and variants of two malware families. Anti-Virus software was used to confirm the correctness of their output. They refer to packed binaries as self-generating programs and describe unpacking as self-generating. Their tool aims to *normalize* or *unpack* the malware. They do not consider packer and malware to be two different entities.

Polyunpack

Polyunpack by Royal et al. [36] (2006) employs a heuristic called new instruction. It is based on the assumption that whenever an instruction is executed that was not present in the original malware binary, an unpacking has occurred. To this end, the packed binary is disassembled before execution, and all pieces of code are collected. Next, the malware is executed and halted regularly to check whether the executed instructions are in the disassembled packed binary. If not, an unpacking has occurred. The authors

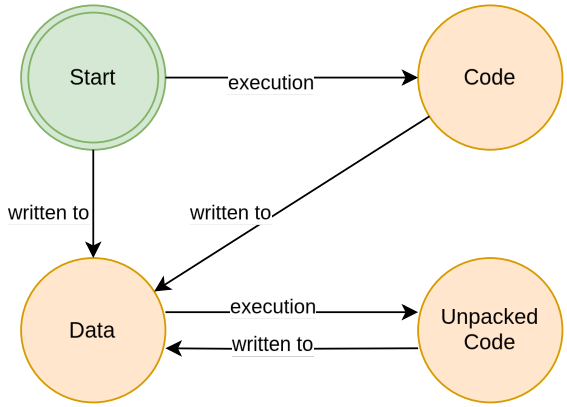

Figure 3.1: This automaton models the write-then-execute heuristic for a single piece of memory. If a piece of memory is executed, it contains code. If that piece of code is overwritten, then the piece of memory contains an unknown type of data. If that data is executed, the write-then-execute heuristic is triggered, and the piece of memory contains unpacked code.

implemented their unpacker into a debugger, which traces the malware and also extracts hidden code. They refer to this process as *hidden-code extraction* for the purpose of Polyunpack in their title and throughout their paper. However, they also use the term *generic unpacking* interchangeably. Additionally, they introduce the term *unpack-executing* to describe the unpacking behavior of malware. They consider malware and packer to be two different entities. For evaluation, they used 3,467 different malware samples and employed an antivirus scanner to confirm the correctness of the unpacking results.

Renovo

Renovo by Kang et al. [41] (2007) is another generic unpacker that relies on the byte-level write-then-execute heuristic. It works very similarly to the one proposed by Christodorescu et al. Renovo uses shadow memory to track write operations performed by the malware. When the memory to which has been written is subsequently executed, the extraction of this memory section is triggered. The authors refer to code that is hidden in an executable as *hidden code*, with packing being just one method of achieving this. Accordingly, the purpose of Renovo is described as hidden-code extraction. Their approach does not consider malware and packer to be two different entities. For evaluation, their data set contained 374 malware samples and one benign sample packed with 20 off-the-shelf packers, and they searched for suspicious strings to validate the correctness of the output.

Omniunpack

Omniunpack by Martignoni et al. [47] (2007) also implements the page-level write-then-execute heuristic. The malware's behavior is tracked via page-level access violations. To

determine the end of the unpacking routine, an antivirus scanner scans the memory each time a write-then-execute event triggers. To achieve this, the heuristic is implemented in a Microsoft Windows kernel module. The authors exclusively use *unpacking* to describe both the malware's behavior and the purpose of Omniunpack. In addition, they consider malware and packer to be two different entities. For evaluation, they used one toy example packed with 20 off-the-shelf packers.

Hump-and-Dump

Hump-and-Dump by Sun et al. [48] (2008) is less a generic malware unpacker than a heuristic. It is based on the assumption that unpacking occurs within a large loop with many iterations and is followed by code pieces that are executed only once. Sun et al. propose to construct a histogram with the number of executions of each piece of code, sorted by the last executed pieces of code. Two parameters are determined: one to identify the hump, i.e., the basic blocks with a large number of executions, and another to identify the sequence of code that is only executed once. The authors exclusively refer to this behavior and their approach as *unpacking*. Their proof-of-concept implementation is built primarily using IDA Pro. For evaluation of their proof of concept, they used eleven off-the-shelf packers on two benign known programs.

Etherunpack

Ether by Dinaburg et al. [42] (2008) is a hypervisor that aims to analyze malware. In their paper, the authors introduce EtherUnpack, an unpacker that uses their own hypervisor and is a recreation of Renovo. Unlike the terminology used in the Renovo paper, the authors omit the term *hidden-code* and instead exclusively use *unpacking*. Similarly to Renovo, they treat malware and packer as a single entity. For their evaluation, they recreate the study used in the Renovo paper.

Eureka

Eureka by Sharif et al. [49] (2008) is an unpacker that introduces a novel heuristic by measuring the entropy of the process' memory. The assumption is that packed malware has a very high entropy, whereas unpacked code has a very low entropy. Thus, a transition from high to low entropy of the memory indicates that unpacking is in progress or has finished. This approach is complemented by a second heuristic based on API calls. This heuristic is based on the assumption that malware and packers exhibit distinct patterns in their use of the Windows API. So, calls to API functions commonly associated with malware signify that the unpacking is complete. In their terminology, the term *unpacking* refers to the behavior of the malware, while *defeating obfuscation* is used to describe Eureka's purpose, although the two terms are used interchangeably. The authors treat malware and packer as two different entities. For evaluation, they used 914 malware samples and one benign program packed with 15 off-the-shelf packers. To validate the correctness of Eureka's output, a code-to-ratio analysis was used.

JUSTIN

JUSTIN by Guo et al. [3] (2008) implements the page-level write-then-execute, complemented by additional heuristics introduced by Isawa et al. [50]. These additional heuristics are disregarded in this dissertation, since they have not been added by the original author. Its implementation is very similar to Omniunpack and also accepts the dichotomy of malware and packer. Their approach was evaluated using 183 malware samples using six different packers and one sample packed with one off-the-shelf packer. In their discussion, the authors use the *unpacking* nomenclature.

Coogan

Coogan et al. [14] (2009) present a very different approach to unpacking. Their system is based on static analysis by analyzing the programs' binary and extracting the unpacking routine. This approach can be considered as an application of the write-then-execute heuristic: the analysis identifies the routines that would write to memory and detects when this written memory would be executed. The extracted routine is processed so that it can be executed outside the malicious binary to unpack the hidden code. *Unpacking* is the dominant nomenclature in their description, and they consider malware and packer to be two different entities. They tested their proof of concept with four different malware samples and one benign sample with an off-the-shelf packer.

Tracesurfer

Tracesurfer by Guizani et al. [51] (2009) implemented a write-then-execute-based unpacker in Intel PIN, an instrumentation program that can be used to trace a program. The authors use the term *unpacking* to discuss their approach and the behavior of malware, and do not consider malware and packer to be two different entities. Tracesurfer was evaluated using 59,544 samples. The correctness of their output is based on whether an unpacking has occurred.

Jeong

Jeong et al. [52] (2010) present an entropy-based unpacker similar to Eureka. In their paper, they use the term *unpacking*. They also assume that the packer and malware are distinct entities. Their evaluation was performed using ten benign programs packed with eleven off-the-shelf packers and six malware samples.

Malwise

Malwise by Cesare et al. [43] (2012) integrates the methodology of Renovo with the entropy-based analysis used in Eureka. The authors use the term *unpacking* to discuss their contribution. They also do not consider that malware and packer might be the same. To test their approach, they used 14 different off-the-shelf packers and 15,569 malware samples. The correctness of their output was validated using similarity scores against known malware.

Codisasm

Codisasm by Bonfante et al. [44] (2015) is another byte-level write-then-execute-based unpacker implemented using Intel PIN. However, the focus of their contribution lies not in the unpacking part, but in the proposition of a disassembler capable of correctly processing code with different entry points. Additionally, the authors provide a detailed description of multilayered packing. They mostly use the term *self-modifying code* to describe the unpacking behavior of malware and do not accept the dichotomy of malware and packer. The unpacking part was evaluated using notepad.exe packed with 28 off-the-shelf packers and 500 malware samples, however the output of the 500 samples does not seem to have been assessed.

Binunpack

Binunpack by Cheng et al. [53] (2018) introduces an unpacker that implements a novel heuristic called rebuilt-then-run. This approach is based on the assumption that unpacking has occurred when the import address table is being built, as the packer part of the program requires fewer imports than the unpacked malware. Thus, a sudden increase in imported symbols is interpreted as an indication that the original malware is running and the unpacking has concluded. This means that they consider malware and packer to be two different entities. The authors also use the *unpacker* term for their paper. Their evaluation used 238,835 malware samples and one known program packed with 28 off-the-shelf packers. The correctness of Binunpack's output was confirmed using a series of metrics similar to Eureka.

Malflux

Malflux by Lim et al. [54] (2019) proposes a unique generic unpacker that combines the write-then-execute heuristic with hump-and-dump. The hump is measured by counting sequential write operations in a memory region. If the number of write operations exceeds a certain threshold, it is referred to as flux. The subsequent execution of such a memory region means that the malware has unpacked itself. The authors describe this process *hidden code extraction* and consider malware and packer to be two different entities. They implemented their approach using PANDA [9], a QEMU fork that enables fine-grained analysis. The evaluation was performed using 18 benign samples packed with 12 off-the-shelf packers and 116 samples spanning ten different families. They used another unpacker to confirm the correctness of their output.

Roamer

Roamer by the author of this dissertation [55] (2019) introduces an unpacker based on the novel heuristic that focuses on memory allocation behavior. This heuristic assumes that malware must allocate new memory regions to unpack and execute itself. Consequently, newly allocated memory regions, including the mapped image of the input executable,

Work	Tool-Purpose	Behavior	MW != P
Normalization	Normalize / Unpacking	Self-Generating	False
Polyunpack	Hidden-Code Extraction	unpack-executing	True
Omniunpack	Unpacking	Unpacking	True
Renovo	Hidden-Code Extraction	Unpacking	False
Hump-And-Dump	Unpacking	Unpacking	True
Etherunpack	Unpacking	Unpacking	False
Eureka	Defeat obfusc.	auto deobfusc.	True
JUSTIN	Unpacking	Unpacking	True
Coogan	Unpacking	Unpacking	True
Tracesurfer	Unpacking	Unpacking	False
Jeong	Unpacking	Unpacking	True
Malwise	Unpacking	Unpacking	True
Codisasm	Extraction	Self-Modifying Code	False
Binunpack	Unpacking	Unpacking	True
Malflux	Hidden-Code Extraction	Unpacking	True
Roamer	Unpacking	Unpacking	False

Table 3.1: This table summarizes the terminology used by the authors to describe their contribution and indicates whether they consider malware and packer two different entities. Unpacking clearly emerges as the most prevalent used term.

are considered likely to contain malicious unpacked code. The unpacking agent was implemented in Python and runs inside VirtualBox. This publication uses the term *unpacking* and assumes that malware and packer can be the same entity. Malpedia was used for the evaluation and the correctness based on whether a new PE-Header has been found in the output.

3.2 Analysis

This section examines each of the mentioned aspects in detail. The assumptions about malware behavior made by the unpackers are addressed in Chapter 6.

3.2.1 Terminology

This subsection analyzes the terminology used in the related work, as summarized in Table 3.1. The table lists the reviewed contributions along with the terminology they use to describe the purpose of their tool and the behavior of the malware. Of the 16 works reviewed in this chapter, ten exclusively use the term *unpacking*, while the remaining six adopt a variety of alternative nomenclatures. Table 3.1 also indicates whether the authors consider malware and packer to be separate entities.

Notably, 70% of the publications that exclusively use the term *unpacking* also assume that malware and packer are distinct entities. In contrast, among those that do not

solely use the term *unpacking*, only 50% maintain this distinction. Those works that also introduce different terms still revert to the term *unpacking* in their discussions. This reflects the importance of the term *unpacking* in not only academic publications but also in conversations within the research community. However, its usage often implies that malware and packer are distinguishable (Table 3.1), a notion that, as shown by the authors of this dissertation [5] and by Ugarte-Pedrero et al. [2], does not universally apply. Moreover, *unpacking* is frequently used to describe the behavior of malware, which is not encapsulated in the definition.

3.2.2 Heuristics

Heuristic	**Number Of Unpackers**
Write-Then-Execute	10
Entropy Analysis	3
Hump-And-Dump	2
New Instruction	1
Rebuilt-Then-Execute	1
New Memory Sections	1

Table 3.2: This table presents which heuristics are employed by the malware unpackers to detect unpacking behavior and how many packers are using each heuristic. If an unpacker combines two heuristics, it is counted under both.

Table 3.2 presents the different heuristics found in the related work and the number of publications that employ each. If an unpacker combines two approaches, it is counted once in each approach. The write-then-execute heuristic emerges as the most prominent heuristic, appearing in ten of the sixteen reviewed publications. Other heuristics did not seem to catch on in academic research, whereas write-then-execute has been in use since at least 2005 and remained prevalent through 2019. A detailed evaluation of these heuristics is provided in Chapter 6.

3.2.3 Data Sets

As noted previously, all publications assessed in this chapter use data sets to perform practical evaluations of their approaches. These data sets include three different types of samples:

1. Unclassified malware encountered in the wild and used without prior analysis.
2. Analyzed real-world malware samples for which malware family and/or the applied packer has been identified.
3. Benign samples, such as open-source tools or native Windows tools that have been packed with off-the-shelf packers. The term *off-the-shelf packers* refers to commercially available or freely distributed packers.

Work	Data Set	Correctness
Normalization	7 OTS X two malware families	Anti-Virus
Polyunpack	3,467 MW	Anti-Virus
Omniunpack	20 OTS X 1 Benign	Ground Truth
Renovo	1 benign X 14 OTS & 374 MW	Search String
Hump-And-Dump	11 OTS X 2 Benign programs	Ground Truth
Etherunpack	Same as Renovo	Search String
Eureka	914 MW & 1 benign X 15 OTS	Code-To-Data Ratio
JUSTIN	183 MW X 6 OTS	Anti-Virus
Coogan	4 MW & 1 benign X 1 OTS	Ground Truth
Tracesurfer	59,544 MW	Unpacking Behavior
Jeong	10 benign X 11 OTS & 6 MW	Ground Truth
Malwise	3 benign X 14 OTS & 15,569 MW	Similarity Scores
Codisasm	1 benign X 28 OTS & 500 MW	Ground Truth
Binunpack	238,835 MW & 1 MW X 28 OTS	Entropy analysis
Malflux	18 benign X 12 OTS & 116 MW	Other Unpacker
Roamer	Malpedia	New PE-Header

Table 3.3: This table summarizes the data sets used in the evaluation of each work, along with the methodologies employed to determine the correctness of the unpackers. The abbreviations *OTS* and *MW* refer to off-the-shelf packers and malware samples, respectively. As the table shows, the data sets vary in quality. If one unpacker is tested with multiple data sets of the same type, then the number of samples are added.

Table 3.3 presents the data sets used in each approach along with the method used to assess the correctness of the unpacked output. *OTS* refers to off-the-shelf packers, and *MW* denotes real-world malware. Three publications use one or more antivirus scanners to verify the correctness of the unknown malware samples. This strategy has the drawback that malware must be known by the antivirus software vendor or conform to a specific format. Therefore, eight other publications apply heuristics to determine correctness. The remaining five constructed a ground truth by either selecting already analyzed malware or by using clinical samples, i.e. both input and packer are known.

The data sets used in the reviewed publications vary considerably. At one end of the spectrum, Coogan et al. evaluated their approach using just four malicious samples and one benign program packed with a single off-the-shelf packer. At the other end, Binunpack was tested on a data set comprised of 238,835 samples. Notably, even in earlier years, larger data sets were occasionally used. Polyunpack employed over 1,000 malware samples in 2006, Tracesurfer used 59,544 malware samples in 2009, and Binunpack 238,835 samples in 2018. Meanwhile, small clinical data sets prevailed from the first publication in our review in 2005, through Coogan et al. in 2009, up to Malflux in 2019. To assess the quality of these data sets, the contributions were evaluated against Rossow's Prudent Practices for Malware Experiments, which were formulated by Rossow et al. [8] as a set of prudent practices when designing malware data sets for malware experiments. Contributions are assessed on whether they are following prudent practices for the creation of data sets and their presentation, while practices regarding the setup

Work	Violated Prudent Practices
Normalization	A2, C1
Polyunpack	A2, B3, B5, C1
Omniunpack	A2, B1, B3, B5, C1
Renovo	A1, A2, B1, B3, B5, C1
Hump-And-Dump	No Malware
Ether	Same as Renovo
Eureka	A2, B1, B3, B5, C1
JUSTIN	A1, A2, B1, B3, C1
Coogan	B3, C1
Tracesurfer	A1, A2, B1, B3, C1
JEONG	A2, B3, B5, B6, C1
Malwise	A1, A2, B1, B3, C1
Codisasm	A2, B1, B3, C1
Binunpack	A1, A2, B3, C1
Malflux	A2, B1, B2, C1
Roamer	Follows the practices

Table 3.4: This table presents an assessment of the conducted experiments in the reviewed publications based on Rossow's Prudent Practices for Malware Experiments. This assessment focuses on the practices for the design of a data set. As shown, nearly all publications violate at least two of these practices.

and the environment have been omitted. Table 3.4 lists which prudent practices are violated in each contribution. With the exception of Roamer, all reviewed contributions violate at least one prudent practice, up to six. The experiments for Roamer have been constructed with these practices in mind. Overall, the assessment reveals shortcomings in the data sets in all reviewed contributions. This has also been observed by Alkhateeb et al. [56] in their study on packing mitigation techniques in 2023, who criticize the use of outdated malware samples and a narrow range of packers. Similarly, Ugarte-Pedrero et al. [2] have pointed out that the lack of high-quality data set containing packed malware hinders a thorough evaluation.

As a result, most prior publications on generic malware unpacking have lacked a rigorous evaluation due to the absence of a high-quality data set and a clear set of requirements. Malpedia, a diverse malware data set by Plohmann et al. [10], addresses many of these shortcomings by adhering to Rossow's [8] Prudent Practices. It strives to include representative samples of every version of every malware family. In addition, Malpedia also provides an unpacked representative for most samples, along with the YARA rules [57, 58] that have been generated using unpacked or dumped representatives. This makes it a substantial advancement in malware data set design: the diversity of included families implies a corresponding diversity in actors, used tools, and, therefore, packing methods. This means a set of requirements can be derived from Malpedia and in turn be used to assess the previous unpacking methodologies. Such a requirement set en-

ables both a theoretical assessment of unpacking methodologies without access to the appropriate samples and the creation of more representative data sets.

One publication that attempts to measure the complexity and capabilities of packers is the study by Ugarte-Pedrero et al. [2] discussed in Chapter 2.3.1. Although valuable, the study does not provide a comprehensive list of the capabilities of real-world malware. Instead, it focuses on a systematization of packer complexity while omitting aspects such as usage of API functions, the life time of code, and unpacking spanning multiple processes. Therefore, the information presented by Ugarte-Pedrero et al. is incomplete and insufficient for evaluating the genericity of a generic malware unpacker.

The next section outlines how this dissertation aims to advance the state of the art by creating such a set of requirements.

3.3 Blueprint for the Construction of a Generic Malware Unpacker

To address the shortcomings identified in this chapter, this dissertation proposes a blueprint for constructing a generic malware unpacker tailored to any given set of malware. In addition, the blueprint produces a set of requirements that can be used to assess existing generic malware unpackers and to support the development of new ones.

Initially, studies are needed to measure the unpacking behavior of malware. These insights allow for categorizing behaviors, leading to the creation of generic handling strategies for each category. A generic malware unpacker must accommodate each category's specific behavior from the study's data set. Thus, an unpacker that successfully tackles all unpacking techniques can handle every malware in the data set.

However, to clearly describe the findings in this dissertation, a precise definition of *unpacking* is required. As previously established, the related work lacks a consistent definition of the tools' purpose and the malware's behavior. This dissertation, therefore, proposes the following definition for *unpacking*:

Definition for *Unpacking*:

Let P be a program that reveals a set of instructions H during its execution. Unpacking describes the process of making H from a given P statically observable.

This definition generalizes unpacking to encompass all forms of self-modifying code, providing a more accurate reflection of the state of the art, as packer and malware are indistinguishable. This definition can also be used to name the behavior of the malware. Therefore, it will serve as the working definition of *unpacking* throughout this thesis.

For the remainder of this thesis, the blueprint is applied to Microsoft Windows malware running natively on x86 and x86_64 processors. AVAtlas has found that 900 million unique samples have been identified for Windows, 900,000 for MacOS, 4.7 million for

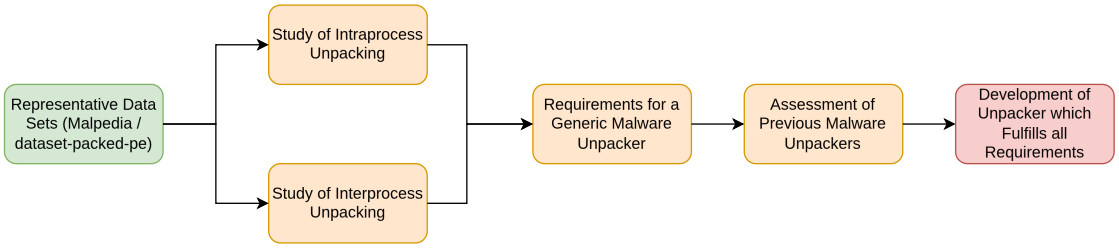

Figure 3.2: Flow for the remainder of this thesis. The two data sets Malpedia and dataset-packed-pe are used to conduct two studies on their unpacking behavior. The results of these studies inform a set of requirements for a generic malware unpacker. These requirements are then used to assess previous malware unpackers. The insights gained from this assessment guide the development of a generic malware unpacker that fulfills all requirements.

Linux, and 35 million for Android up to October 2024 [1]. This highlights the prevalence of Windows malware, thus justifying the focus of this thesis on this malware ecosystem. In addition, the malware unpackers reviewed in this chapter were also developed for this platform, so the requirements can be applied to these unpackers.

As mentioned above, the first step to execute the blueprint is selecting a set of packers from which to derive the requirements for a generic malware unpacker. Given the focus on Windows malware and the goal of maximizing applicability for the resulting unpacker, the selected data set should be diverse in families and ideally include a representative of each version of every family. This ensures diversity in actors and their tools, including packers. Additionally, the inclusion of unpacked malware representatives as encountered in the wild is highly desirable, as these can serve as ground truth for evaluating the resulting malware unpacker. For this purpose, Malpedia by Plohmann et al. [10], described above, was chosen. It is organized in a GIT repository and the commit used in this dissertation is *e55fbb6b8*. Malpedia follows Rossow's Prudent Practices [8, 10] and limits the risks of overly clinical and constructed data sets, as suggested by Muralidharan et al. [16]. However, exploring open-source or commercially available off-the-shelf packers applied to benign software is also valuable, as it highlights the contrast between packing used for malicious purposes versus those used in benign ways. Therefore, the data set *dataset-packed-pe* [11], which consists of benign samples packed with off-the-shelf packers alongside their unpacked binaries, is used to complement real-world malware. This data set is termed synthetic.

The initial decision to construct a generic unpacker is to determine how the unpacking is to be recognized. Past malware unpackers that assume a strict separation between packer and malware, typically define the end of unpacking as the moment when the program executes the original entry point (OEP) of the original binary. As previously established, this assumption does not reflect reality.

Alternative heuristics have also proven insufficient. For example, identifying unpacking steps based on the resolution of API functions [59, 53] is also unreliable, as there are several techniques to obfuscate resolution and even use of APIs. This heuristic, therefore, rests on the assumption that malware is not using these types of obfuscation.

The hump-and-dump heuristic assumes that unpacking has occurred when a large number of write operations are followed by a jump. [48] Unfortunately, this heuristic still implies that a lengthy loop is necessary to write the unpacked code. However, malware is also known to use functions such as RTLDecompressBuffer or LoadLibrary during unpacking. Consequently, it would be impossible for Hump-and-Dump to spot the unpacking steps that employ these methods.

The heuristic used in Roamer is also inadequate, as it assumes that malware does not utilize code caves or the stack to write its unpacked data.

Since all these heuristics rely on assumptions about the functionality and properties of malware, the write-then-execute heuristic, also known as WxE, has been chosen. Figure 3.1 displays how WxE helps distinguish between code, data, and unpacked code. WxE identifies an unpacking step by observing the malware's memory writes followed by the execution of a written byte. This heuristic is deliberately broad and makes no assumptions about the malware's functionality since it is based on the idea that unpacked code must be written to get executed. As shown above, WxE is also the most popular heuristic in prior research.

As a reminder, write-then-execute refers to the act of writing data to memory followed by the execution of that written data. This behavior can be observed inside of a single process, as described in Chapter 2.3.2.1. Therefore, the first study focuses on exploring write-then-execute events that are confined to a single process. Given that this dissertation focuses on Windows malware, it is also necessary to account for WxE unpacking behavior that crosses process boundaries, which is generally described as host-based code injection attacks [60]. To study this behavior, the second study focuses on measuring the multi-process write-then-executes. The combination of both studies yields a comprehensive picture of the unpacking behavior, allowing the categorization of the unpacking behavior and the derivation of requirements for a generic malware unpacker. These requirements are then used to evaluate the unpackers described above to identify the most promising approaches. Finally, these approaches are extended to create a new generic malware unpacker that fulfills all the requirements. This workflow is illustrated in Figure 3.2.

3.4 Conclusion

This chapter reviewed 16 publications on creating generic malware unpacking spanning from 2005 to 2019. The analysis of these works was structured along the three axes: the terminology used for unpacking, the heuristic employed to detect unpacking behavior, and the data sets used for evaluation. Examining the terminology for unpacking revealed inconsistencies in the academic landscape. Therefore, a new definition for unpacking is proposed: one that reflects the current state of the art and can be used to name the malware's behavior. The review of the used heuristics shows that the write-then-execute heuristic is the most widely used. Lastly, the most significant problem in all the reviewed work concerns the appropriate data sets, as shown by the review of the evaluations. The findings of the data sets are also reflected in other related work [2, 56, 8].

To remedy this state of affairs, a blueprint is proposed for creating a generic malware unpacker. To execute this blueprint, Windows malware has been chosen as the main focus. Since the blueprint calls for a set of packers, two representative data sets have been chosen that will be studied.

The subsequent chapters, Chapter 4, and Chapter 5, discuss studies on malware unpacking behavior. These studies provide the groundwork to equip future researchers with the knowledge to design, implement, and evaluate their generic malware unpackers.

Main Takeaways of this chapter:

A comprehensive review of the academic landscape reveals that it is not yet adequately equipped to create generic malware unpackers. In particular, the field fails to provide both an understanding of how malware packing works and a means to evaluate a generic unpacker. Therefore, this dissertation proposes a new definition of malware unpacking and a blueprint to create this understanding. The Windows ecosystem has been chosen to execute this blueprint for the remainder of this thesis.

4 Intra-Process Unpacking Behavior

The content of this chapter and some of its figures are mostly a retelling of prior research conducted by the author of this dissertation. It has been restructured from an independent article to fit the flow of this dissertation. The author of this dissertation has published their findings in the following paper:
Jenke, Thorsten, Elmar Padilla, Lilli Bruckschen (2024). "Towards Generic Malware Unpacking: A Comprehensive Study on the Unpacking Behavior of Malicious Run-Time Packers". In: Fritsch, L., Hassan, I., Paintsil, E. (eds) Secure IT Systems. NordSec 2023. Lecture Notes in Computer Science, vol 14324. Springer, Cham. [5]

4.1 Introduction

This chapter explores the unpacking behavior of malware within a single process called intra-process unpacking by measuring it under five different aspects. These aspects have been chosen to support the creation of requirements for a generic malware unpacker. As established in Chapter 3.3, write-then-execute is the chosen unpacking heuristic with its two defining properties, the writing and subsequent execution of code, and therefore heavily influences the aspects.

To write unpacked code, malware must access the memory where unpacked code is written and possess the means to write the code. Therefore, the first aspect is the techniques and functions that are used to obtain the memory to write the unpacked code, and the second aspect is the techniques and functions that are used to write the unpacked code. The next aspect is the number of unpackings that are performed, which means the number of unpacking layers. When multiple layers of packing are used, it might mean that the malware overwrites the intermediate stages to hide certain behavior from the analyst. These stages may yield information on the tool chain used, which may leak information on the threat actor group. Therefore, the next aspect is whether malware overwrites unpacked code during its unpacking process. The last aspect examined is the distinction between malware and packer. As shown in Chapter 3, the consensus in the related work presumes a clear distinction between packers and malware, with the primary goal of malware unpacking being the reconstruction of the original malware as it existed prior to the application of a packer. Therefore, this chapter explores whether malware and packer are more intertwined than is generally assumed. One missing aspect from the write-then-execute heuristic is the execution of the unpacked code. This aspect is further discussed in the discussion in Chapter 4.5.

To make this behavior observable, this work proposes a novel unpacking model that divides the execution of a program into unpacking layers. This model is implemented using the write-then-execute unpacking heuristic into a packer measurement system, which takes a program as input and returns the unpacking behavior using the unpacking model. The resulting measurement system is subsequently implemented for a study using Malpedia and dataset-packed-pe as data sets, as described in Chapter 3.3.

So, in total, the following research questions (RQ) are explored:

- *RQ1.1*: What are the methods/techniques used by malware to obtain the memory to write unpacked code?
- *RQ1.2*: What are the methods/techniques used by malware to write code?
- *RQ1.3*: Is the unpacked code overwritten during the execution of the malware?
- *RQ1.4*: How many layers of packing are used?
- *RQ1.5*: Is it possible to differentiate between malware and packer code?

This chapter is structured as follows: first, the unpacking model is described. This is followed by the implementation of the packer measurement system and the subsequent study. The chapter ends with a summary and conclusion.

4.2 Unpacking Model

To answer the research questions, this section establishes the necessary theoretical foundations to model the unpacking behavior. The research questions concern the logistics around the writing of code. In malware unpacking, the writing of code is conducted by the malware or, more precise, the unpacking layers of the malware. Therefore, the concept of unpacking layers is introduced that divides the entire malware behavior into unpacking layers. This is the first aspect examined in this section. These unpacking layers interact by providing, overwriting, or sharing each other's code. Thus, how layers interact with each other's code is examined to establish the relationships between the layers and used to answer the research questions. This is the second aspect explored in this section.

4.2.1 Unpacking Layer

As introduced above, the purpose of the proposed unpacking model is to divide the full behavior of the malware into unpacking layers, making the unpacking behavior observable. The first unpacking layer begins when the input program is executed, and each unpacking introduces a new unpacking layer. An unpacking is determined by the chosen unpacking heuristic.

As can be seen in Figure 4.1, an unpacking layer L is denoted as:

$$L = (I, E, W, A) \tag{4.1}$$

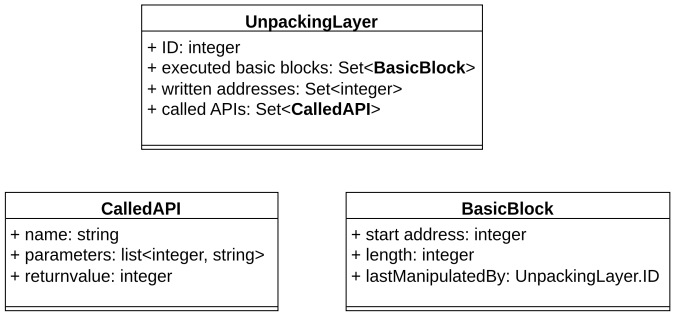

Figure 4.1: Representation of an unpacking layer, called APIs, and basic blocks.

where $I \in \mathbb{N}$ represents the identifier of the unpacking layer, E represents a set of executed basic blocks, W represents a set of written addresses w where $w \in \mathbb{N}$, and A represents a set of called API functions. A basic block b is a sequence of code that does not involve any control manipulation and is defined as

$$b = (s, l, I) \tag{4.2}$$

with $s, l, I \in \mathbb{N}$ and s being the start address and l the length of the basic block. I refers to the identifier of the unpacking layer that was the last to manipulate at least one byte of this basic block.

Let $executedAddresses(E)$ be a function that returns a set of executed bytes for a given set of basic blocks E. This means that for each b in E, add the addresses b_s until b_{s+l} to a singular set of addresses.

Every API call $a \in A$ being a tuple with

$$a = (name, P, r) \tag{4.3}$$

with $name$ being the name of the called function, $P \subset \mathbb{N}$ being the list of the in- and out-parameters with $r \in \mathbb{N}$ being the return value.

If during execution an unpacking is detected, a new layer L_n is introduced, and each subsequent basic block, write operation, and API call will be assigned to this layer.

Therefore, a single basic block can belong to multiple layers, which differs from the code waves model, as seen in [2, 44], where jumps back to previous layers are possible. In contrast, this unpacking model is sequential and once a new layer is introduced, the execution does not jump back to any earlier layer.

4.2.2 Interpretation of Unpacking Layers

A subsequent interpretation step extracts high-level semantics from the model described in the previous section. This step is explained in this section.

Because a single malware sample may employ multiple packing methodologies, different behaviors can be measured. The semantic representation system must therefore be

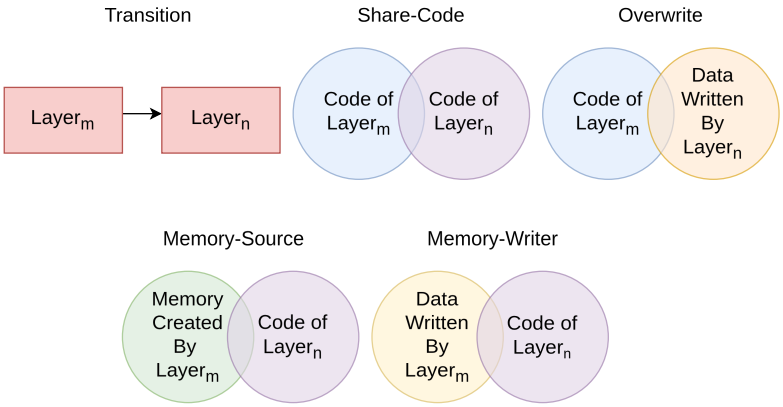

Figure 4.2: There are five different tags that describe the relationship between two unpacking layers. This figure shows the different possible layer tags with $Layer_m$ being created before $Layer_n$.

flexible enough to represent this diversity. A strict grouping system that places each sample exclusively into one group is too strict and may hinder accurate analysis of varied behaviors. Instead, a tagging system is chosen to describe the relationships between unpacking layers, ensuring that diverse behaviors of different unpacking methods are adequately captured.

As explained above, the writing of code is done by the malware itself. In particular, the different unpacking layers write or overwrite the code of the other layers. Therefore, the high-level semantic insights on the unpacking behavior needed to answer the research questions is how these unpacking layers interact with the code of the other layers. Accordingly, a tag is a relationship between two unpacking layers. With a tag R being

$$R = (L_m, L_n, T) \tag{4.4}$$

with L_m and L_n being unpacking layers and T being the identifier for a tag. Five different tags are needed for the study.

- **Transition Tag:** Organize all layers into a list $L_1, L_2, ..., L_k$. Assign the *Transition* tag to every two consecutive layers (L_n, L_{n+1}).
- **Share-Code Tag:** For each pair (L_m, L_n) where $n > m$, compute the intersection of their executed basic blocks $B_n \cap B_m$. If $B_m \cap B_n \neq \emptyset$, assign the *Share-Code* tag to (L_m, L_n). Therefore, this tag indicates that the two layers share code that has not been altered, indicating a functional dependency between the two layers.
- **Overwrite Tag:** For each pair (L_m, L_n) where $n > m$, compare the write operations W_n of the later layer with the executed addresses $executedBytes(E_m)$ of the earlier layer. If $W_n \cap executedBytes(E_m) \neq \emptyset$, assign the *Overwrite* tag to (L_m, L_n). This indicates that one or more basic blocks executed in L_m were partially or entirely overwritten by write operations in L_n. Multiple layers can overwrite the same earlier layer. However, the initial layer is not included in this, as its code can be reproduced from the image file and does not require unpacking.

Therefore, this tag is used to determine whether all layers of the code are accessible at the end of execution.

- **Memory-Source Tag:** If a layer L_n executes a basic block whose bytes originated or were generated by the memory of a previous unpacking layer L_m, assign the *Memory-Source* tag to (L_m, L_n). However, this only indicates that L_m gained the handle on a memory section that is used by L_m or a later layer to write the code of L_n. For this, the API calls used for memory allocation and retrieval need to be analyzed, such as the set $MemGen_m$ is a set of memory addresses that are allocated or retrieved by $Layer_m$ and $MemGen_m \cap executedBytes(E_m) \neq \emptyset$ to assign this tag to (L_m, L_n).
- **Memory-Writer Tag:** For each pair (L_m, L_n) where $n > m$, compare the write operations of W_m of L_m with the executed basic blocks $executedBytes(E_n)$ of L_n. If $W_m \cap executedBytes(E_n) \neq \emptyset$, assign the *Memory-Writer* tag to (L_m, L_n). This indicates that the code executed in L_n was written by L_m. *Memory-Writer* shows that L_n wrote parts of the code belonging to L_m.

4.3 Packer Measurement System

This section presents the technical implementation of the unpacking model presented in Section 4.2.

As established in Chapter 3.3, the chosen unpacking heuristic is write-then-execute. So, the unpacking model described above must be applied to this heuristic: each time code that has been written to memory is subsequently executed, a new unpacking layer is created.

Each written byte, executed basic block, and API call are added to the current layer L_n. Whenever a basic block is executed, the system checks if a written address $w \in W$ exists, such that $s \leq w < s + l$, with s and l being the start and length, respectively, of the executed block. If such a written address w exists, a new layer L_{n+1} is introduced and each written byte, the executed basic block and any API call are added to this new layer.

Next, a framework is chosen to implement the component that records the information needed to measure the unpacking layers and then apply tags.

4.3.1 Framework

The write-then-execute unpacking heuristic necessitates recording both written bytes and executed basic blocks, which, in turn, requires recording low-level details of the execution of the program. Attaching a debugger to the malicious process could record the executed and written bytes. However, debuggers are prone to detection and are easily bypassed by malware [20]. Therefore, it was decided to avoid this risk, and a more covert approach was pursued.

An option considered was virtual machine introspection such as Drakvuf [61] based on LibVMI [62]. Unfortunately, experiments using LibVMI with both KVM and XEN to record each executed and written byte proved too slow for such fine-grained analysis.

Alternatively, QEMU-based [18] emulation introspection frameworks offer a viable solution. QEMU is an open-source machine emulator capable of full-system emulation using the Tiny Code Generator (TCG) [63]. TCG translates the code of the guest system, in the granularity of basic blocks, into platform-independent code (Tiny Code) and is then compiled into the architecture of the host.

The open-source nature and documentation of QEMU have led to the development of several emulation introspection frameworks [64, 9, 65, 66, 67, 68]. These frameworks fork QEMU and modify its source code with calls to make execution and interaction with the memory observable. Among these, TEMU has not been maintained for over a decade and was therefore discarded. Instead, DECAF [66], DECAF++ [67], and PANDA [9] have been evaluated in previous experiments by setting them up and developing proof-of-concept plugins to gather the information needed. DECAF and DECAF++ proved unreliable and crashed in almost all attempts to run the plugin. Only PANDA has worked reliably, offers a stable Python API and remains actively maintained [69]. Therefore, PANDA is selected to implement the packer measurement system.

4.3.2 Implementation of Recording Plugin

This section discusses the implementation of the packer measurement system using the PANDA framework chosen in Section 4.3.1. The overall workflow achieved in this section is denoted in Figure 4.3. The recording of the execution trace is implemented through the PyPANDA [65] plugin interface.

Since this dissertation focuses on malware for Microsoft Windows, as explained in Chapter 3.3, the packer measurement system is only compatible with this operating system. To meet the requirements laid out in Section 4.2.1, three events must be recorded: executed basic blocks, written memory bytes, and called APIs. PyPANDA offers, among other functions, callbacks that are called whenever a memory byte is written or a basic block is executed. So, functions have been added to these callbacks to record the unpacking behavior. Each function produces a log line that contains the current PID, thread identifier, callback type identifier, and current program counter. Specifically:

- Basic-Block Callback: log line includes information about the base address and size of the basic block.
- Memory-Writer Callback: log line includes the address, the size of the written data, and the actual data.

Since PANDA has a view on the full system, these callbacks would normally be called for every process and kernel, resulting in a significant amount of data. Therefore, the recording is restricted to malicious processes that run on the system. Upon virtual-machine startup, all callbacks are deactivated except for the callback that gets called whenever the value in the CR3-register changes. The CR3-register contains the ASID number and is, therefore, an indicator for the currently running process. When a malware

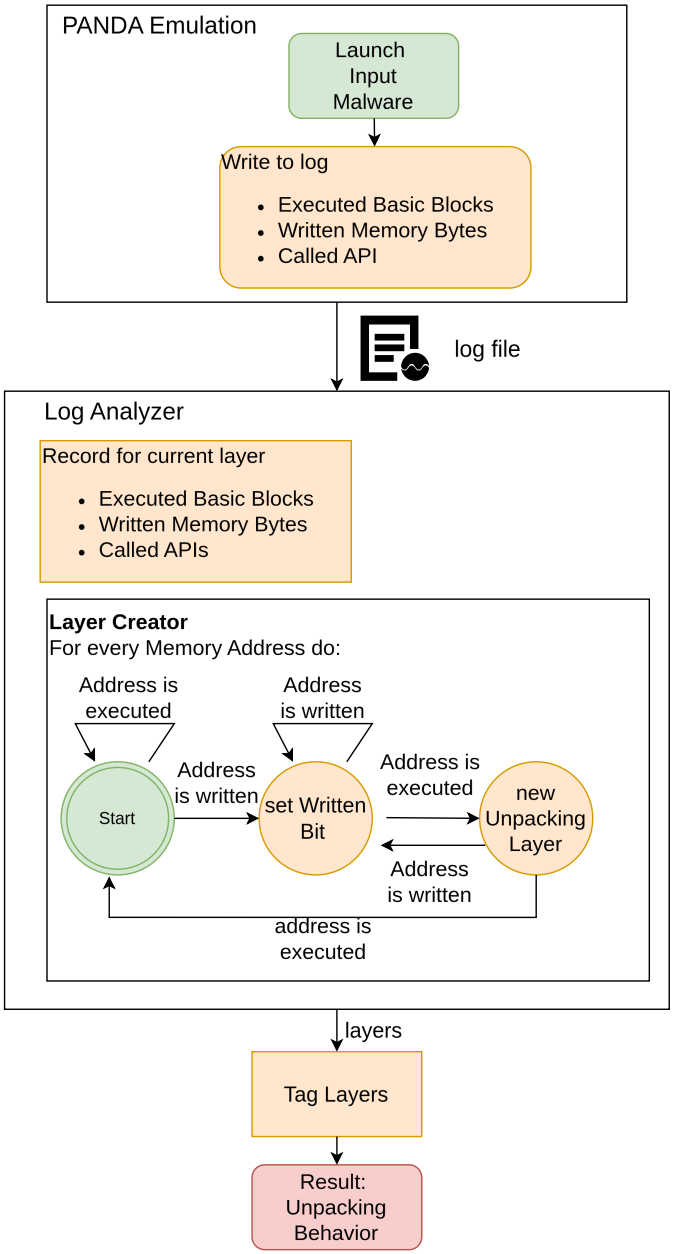

Figure 4.3: This figure illustrates the workflow of the packer measurement system. First, the malware sample is run in the PANDA emulation, and the executed basic blocks, written addresses and API calls are recorded into a log file. The resulting log file is analyzed. The executed, written addresses and called APIs are recorded for the current layer. The automaton in the **Layer Creator** detects when a new unpacking layer is introduced. The resulting layers are tagged and are the final output.

sample is first executed, the initial malware process has a predetermined name, and the system waits until the CR3 callback is called for the switch into the process with this predetermined name. At that moment, the PID of the malware process is added to a

watch list of processes, and all recording callbacks are activated. Whenever the ASID changes to a PID not on the watch list, the callbacks are deactivated, and when the execution returns to a watched PID, callbacks are reactivated. If the initial malware process performs a code propagation as explained in Chapter 5, the PID of the new malicious process is added to the watch list.

Furthermore, the model described in Section 4.2 also requires the recording of API calls. So, a generic API hooking framework is built into the packer measurement plugin for PANDA. To compile a list of API function addresses using APIScout [70], ASLR is disabled in the guest system, ensuring that the libraries and, therefore, the function offsets are always at the same address. Each of these addresses is hooked using the PANDA plugin hooks2 [69]. Additionally, the function names and their signatures, i.e. name, input, and output parameters, are added to a lookup table. Hook functions receive the signature of their respective functions.

Whenever a hook is triggered, it reads the input parameters according to the process architecture and creates a new hook on the return address of the API function. The input parameters are printed alongside the function name, current PID, and TID. Whenever the hook on the return address is triggered, the output parameters of that function are read and printed with the function name, current PID, and TID. The return hook deactivates and removes itself before returning.

The result of this plugin is a log file that contains every written memory byte, every executed basic block, and every API call done by the malware sample.

4.3.3 Generating the Unpacking Layers

The generation of unpacking layers is analogous to the write-then-execute methodology, described in Renovo [41]. The log file described in Section 4.3.2 is processed sequentially, line by line. Initially, a new empty unpacking layer is created. Each log line contributes to the current layer.

The following events in the log can be encountered:

- If a **memory write** is encountered, the written bytes are added to the set of written bytes W.
- If a **code execution or code translation** is encountered, the executed basic block is added to the set of executed blocks E.
- If an **API function** is encountered, the following lines are skipped until the return of that function. This optimization avoids attributing memory writes and executed/translated basic blocks by library functions, as they should not be attributed to the malware. Once the return of the function is encountered, the function is added to the set of functions called A.
  - If an **API function writes memory**, the affected bytes are derived from the parameter and return values and added to the written bytes W.

– If an **API function manipulates a file**, this file is added to a list of manipulated files, so that when a function of the LoadLibrary family is called, the set of manipulated files can be checked to see if the LoadLibrary function is used for unpacking.

A new layer L_{n+1} is created when a byte is executed that has already been added to the written bytes of the current layer L_n. At this point, the current layer has been completed, and every subsequent event is attributed to the new layer.

The write-then-execute heuristic is very lenient and does not make assumptions about the capabilities and functionalities of the malware; consequently, it is prone to false positives. Certain libraries execute generated code, which triggers the write-then-execute heuristic and generates a new layer. This unintentional behavior should not be included in the output.

Now, the tags can be applied to the output, as described in Section 4.2.2.

4.4 Study

To answer the research questions outlined in Section 4.1, a study was conducted to measure the unpacking behavior of two diverse sets of malware and packed executables using the packer measurement system described in Section 4.3. The resulting tags, as described in Section 4.2.2, are interpreted to gain a deeper understanding of packer behavior and answer the research questions. As explained in Chapter 3.3, the chosen data sets are Malpedia and dataset-packed-pe.

Malpedia Initially, 3,123 32-bit portable executables (PE) were obtained from Malpedia. Of these, 632 (15.91%) samples are written for the .NET framework by Microsoft [71] and, therefore, excluded from the data set, since the packer measurement framework is not compatible with the .NET framework. An additional 218 (5.49%) samples could not be analyzed, and 43 (1.08%) other samples have not been fully analyzed; these were also removed. Among the remaining samples, unpacking behavior was detected in 1,301 (41.66%) samples. These samples constitute the Malpedia data set for the study.

dataset-packed-pe The second data set, dataset-packed-pe, contained 2,558 samples. Of these, 119 (4.06%) are .NET and, therefore, were removed. Another 254 (8.67%) samples were deemed broken, and 16 (0.55%) samples could not be fully analyzed. Among the remaining samples, unpacking behavior was detected in 2,001 (78.23%) samples, which are further analyzed in the study.

4.4.1 Setup

A ten-minute timeout was chosen for the sandboxing to mitigate the impact of PyPANDA on the performance of the emulation. The chosen operating system is Windows 7 (32-bit) with 2 GB of RAM allocated to each virtual machine. The operating system

has Visual C++ redistributables installed. Although Windows 7 is an older platform and its user base has largely migrated to newer or other operating systems, it remains compatible with Malpedia malware. In a previous experiment, it was found that a higher number of malware is compatible with Windows 7 compared to Windows 10. In addition, PANDA is more compatible with Windows 7 32-bit than subsequent versions of Windows.

4.4.2 Results and Interpretation

This section presents and interprets the results of the study to answer the research questions posed in Section 1.1. The results for each tag in Section 4.2.2 are presented on their own and are interpreted to answer the corresponding research question.

4.4.2.1 Memory-Source Tag

This subsection presents and interprets the results for the memory-source tag, answering:

RQ1.1: What are the methods/techniques used by malware to obtain the memory to write unpacked code?

Used Memory Source	Malpedia Samples [%]	Packed-PE Samples [%]
image	67.03	84.91
VirtualAlloc	64.34	22.79
NtMapViewOfSection	26.29	19.84
RtlAllocateHeap	10.99	0.40
codecave	7.92	5.55
NtAllocateVirtualMemory	6.46	0
VirtualAllocEx	6.00	0
stack	5.00	0
LoadLibraryExW	4.07	0
LocalAlloc	3.61	0
LoadLibraryExA	2.84	0
GlobalAlloc	2.69	0.10
HeapCreate	2.15	0.25
LoadLibraryA	1.46	0
MapViewOfFile	0.61	0
malloc	0.54	0
LoadLibraryW	0.38	0

Table 4.1: This table shows the memory sources used by samples from the data sets to gain access to a memory region.

The results are presented in Table 4.1, which categorizes memory sources either by purpose or allocation method and reveals the types of memory malware uses to write its unpacked code.

- **Image** refers to the image sections that are allocated when the PE file is loaded into memory. In the Malpedia data set, 67.03% samples leveraged their own image sections for unpacking, while 84.91% samples of the dataset-packed-pe did so. For both data sets, it is the most widely used memory location.
- **VirtualAlloc** [26] takes the size and optionally the address to allocate a new memory section in the caller process. For Malpedia, 64.34% of the samples used VirtualAlloc, narrowly trailing the image section. In contrast, only 22.79% of dataset-packed-pe samples use VirtualAlloc, showing a clear bias towards real-world malware and indicating that off-the-shelf packers rely far less on this memory source.
- **NtMapViewOfSection** [72] is used to map the same piece of memory to other processes or within the own process. Its prevalence is similar between the data sets: 26.29% of Malpedia samples and 19.84% of dataset-packed-pe samples use this function.
- **RtlAllocateHeap** [73] allocates new memory on a given heap of a given size and is the fourth most prevalent memory source. 10.99% of the Malpedia samples used it, but only 0.4% of the samples in dataset-packed-pe.
- Other heap allocations showed a similar disparity, suggesting that the heap is almost irrelevant for off-the-shelf packers:
  - **GlobalAlloc**: 0.1% (dataset-packed-pe) vs 2.69% (Malpedia), can be used to allocate memory on the heap [74].
  - **HeapCreate**: 0.25% (dataset-packed-pe) vs 2.15% (Malpedia), creates a new heap [75].
  - **LocalAlloc** [76]: 3.61% (Malpedia; none in dataset-packed-pe)
  - **malloc** [77]: 0.54% (Malpedia; none in dataset-packed-pe)
- **Codecaves** [78] were used in 7.92% of the samples in Malpedia and in 5.55% of the samples in the dataset-packed-pe. Here, codecaves refer to writing code into the memory regions of a module and executing it from there.
- **NtAllocateVirtualMemory** [79] and VirtualAllocEx [80] were used in 6.64% and in 6% of samples in Malpedia and none in dataset-packed-pe.
- **The stack** is used by 5% of the samples in Malpedia to write and execute code. By default, code that has been pushed to the stack cannot be executed, since the stack section is marked as not executable. However, this can be circumvented.
- **LoadLibrary** [81] is used by some malware in Malpedia. This malware writes code as a library on the hard drive and loads it via one of the LoadLibrary family of functions, thereby fulfilling the write-then-execute metric using the hard drive.

The packer measurement system detected 17 different memory-source techniques in the Malpedia data set, of which only seven are observable in dataset-packed-pe. This discrepancy further demonstrates that evaluations based solely on off-the-shelf packers provide a limited perspective on real-world malware capabilities.

Consequently, a generic malware unpacker must be compatible with all kinds of memory sections.

4.4.2.2 Memory-Writer Tag

This segment presents the observed memory writing techniques and interprets the findings to answer the research question:

RQ1.2: What are the methods/techniques used by malware to write code?

Used Write Function/Technique	Malpedia Samples [%]	Packed-PE Samples [%]
direct	91.85	100.00
memcpy	17.22	7.45
LoadLibraryExW	4.07	0
RtlDecompressBuffer	3.77	0
memmove	3.00	0
LoadLibraryExA	2.84	0
WriteProcessMemory	2.61	0.10
LoadLibraryA	1.46	0
RtlMoveMemory	0.77	0.50
LoadLibraryW	0.38	0

Table 4.2: This table summarizes the methods and strategies used by samples in the data sets to write data that is subsequently executed. The term *direct* denotes immediate access to memory without using an API function.

The observed memory writing techniques are presented in Table 4.2. The *used write function/technique* column specifies the method the samples used to write code for the unpacking procedure.

- **Direct access to memory** occurs in 91.85% of the Malpedia samples and in every sample of the dataset-packed-pe. This means that code is written directly to the memory location without the use of an API function.
- **Memory copy functions** are also used in both data sets.
  - **memcpy** [29] is a function dedicated to copying memory from one buffer to another. It is used in 17.22% and 7.45% of the samples in Malpedia and dataset-packed-pe. Some compilers inline the memcpy function, which turns a usage of the memcpy function into an immediate access to memory.
  - **memmove** [82] is used by 3% of the samples in Malpedia, also moves memory from one buffer to another.
  - **RtlMoveMemory** [83] is used by 0.77% of samples in Malpedia and 0.5% of samples in dataset-packed-pe and also moves memory from one buffer to another.
- **WriteProcessMemory** [30] is used in both data sets with 2.61% and 0.1%. Because this API function does not perform writing operations in user mode, an unpacker that only observes user-space behavior is not able to detect these memory writes.

- **RtlDecompressBuffer** [84] decompresses a given buffer and is a packing function included in the Windows API. It is used in 3.77% of the samples in Malpedia but was not observed in dataset-packed-pe.
- **LoadLibrary** [81] usage is also reflected in the memory writing techniques, analogously to the memory locations.

Although memory writing techniques are less diverse than the observed memory-source techniques, important requirements for a generic unpacker can be deduced. Specifically, a generic unpacker must be compatible with memory access in user mode and memory access done through the API.

4.4.2.3 Code-Overwrite Tag

In this section, the phenomenon of code being overwritten during the unpacking phase is explored. The interpretation of these results determines whether a generic malware unpacker must track overwritten code during the unpacking process and answers the research question:

RQ1.3: Is unpacked code overwritten during the execution of the malware?

Code Overrides	Malpedia Samples [%]	Packed-PE Samples [%]
Yes	21.60	28.79
No	78.40	71.21

Table 4.3: This table indicates how many samples in each data set exhibited overwritten code.

Table 4.3 presents the results of the analysis for the *Code-Overwrite* tag. In Malpedia, more than 20% of the samples overwrite code during unpacking, while in dataset-packed-pe nearly 30% of the samples do so, which cannot simply be extracted from the unrun binary.

This is a significant portion of the binaries. A generic unpacker that fails to return every layer to the user risks inadvertently hiding crucial information about the tool chain of the actors and malware. Therefore, a generic malware unpacker must return every piece of executed code to the user.

4.4.2.4 Transition Tag

This section examines the number of unpacking layers observed per sample. The resulting layer counts address the research question:

RQ1.4: How many layers of packing are used?

The number of packing layers observed by the packer measurement system is summarized in Table 4.4. A majority of samples in both data sets use more than two layers: 36.97% of Malpedia samples and 44.88% of dataset-packed-pe samples. Moreover, approximately 25% of all samples in both data sets exceed three layers. The maximum

Number of Unpacking Stages	Malpedia Samples [%]	Packed-PE Samples [%]
2	36.97	44.88
3	24.14	25.54
4	18.14	19.74
5	9.30	0.40
6	5.30	0.30
7	1.08	0.30
8	0.77	0.25
9	0.46	5.80
10	0.61	0.15
11	0.15	0.15
12	0.08	0.10
13	0.15	0
≥15	2.84	2.40

Table 4.4: Number of unpacking layers per sample.

observed layer counts are 953 in the Malpedia data set and 487 in dataset-packed-pe. These results invalidate the assumption that malware has a single original entry point. Therefore, a generic malware unpacker must be compatible with any number of unpacking layers.

4.4.2.5 Share-Code Tag

Using the *Share-Code* tag explained in Section 4.2.2, the distinction between malware and packer is discussed to answer the research question:

RQ1.5: Is it possible to differentiate between malware and packer code?

The traditional unpacking model posits a definitive hand-off point between the packer and the malware, implying that their respective layers are functionally independent. Functional independence is defined by the absence of shared code between two layers. If this unpacking model holds true, every sample must exhibit such an observable hand-off point. This means that at least two disjoint sets of unpacking layers (one for the packer, one for the malware) are totally functionally independent, because both the packer and the malware might have multiple layers. Figure 4.4 illustrates an example for this scenario.

At least two Functionally Independent Programs	Malpedia Samples [%]	Packed-PE Samples [%]
No	80.40	90.85
Yes	19.60	9.15

Table 4.5: This table shows the number of samples that contain at least two functionally independent programs.

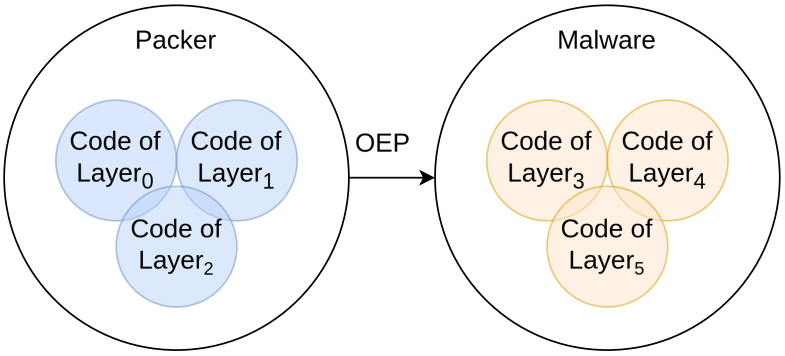

Figure 4.4: This figure illustrates two distinct programs, *Packer* and *Malware*, that are divided by a single hand-off point, the original entry point (OEP). Each program consists of three sequential layers, and within each program those layers share basic blocks. Crucially, there is no overlap between any layer of the Packer and any layer of the Malware, demonstrating complete functional independence.

The number of samples that contain at least two separate programs is displayed in Table 4.5. In 19.60 % of the Malpedia samples, the packer and malware functionalities are significantly intertwined; this proportion falls to 9.15 % in dataset-packed-pe samples.

These findings challenge the traditional assumption of packer and malware as two different entities, revealing instead a continuum between the terms packer and malware. Therefore, a generic unpacker must recognize that a strict distinction between the two is not possible.

4.5 Limitations

There are two major limitations to this study:

The implementation is limited to 32-bit executables. As modern malware might be increasingly compiled for 64-bit, more modern unpacking techniques are not measured by this approach.

This framework does not include a means to determine that all unpacking stages have been observed. The ten-minute sandbox timeout attempts to mitigate that fact by allowing sufficient runtime for most packers to complete. But there is no ground truth to verify the correctness. Also, there are packers that potentially generate an unlimited amount of unpacking layers, such as the packers that pack and unpack code on demand.

An additional aspect of the write-then-execute heuristic is the execution part, which remains unexamined in this study. For native binaries, the code is strictly executed in the CPU. However, code execution can be conducted in various ways on other platforms, such as .NET, for example just-in-time compiled or executed in a virtual machine. Therefore, for platforms with more variable code execution, the research question on how the written code is executed should be added.

In future work, a definitive end of the sandboxing could be implemented by scanning the memory with the Yara rules provided by Malpedia or searching for the unpacked representatives for the samples in dataset-packed-pe.

4.6 Conclusion

This chapter has presented a comprehensive study of intra-process malware unpacking. A theoretical model of unpacking behavior was introduced, segmenting unpacking into distinct layers and defining five inter-layer relationship tags. This model was implemented into a packer measurement system using the dynamic analysis system PANDA.

Using the packer measurement system, a study was conducted to observe the unpacking behavior of malware in the wild. Two data sets were analyzed: a diverse selection of real-world malware from Malpedia and a *clinical* data set containing benign binaries packed with off-the-shelf packers. Consistent with other work [16, 8] suggesting that clinical data sets are not suitable for studies of malware behavior, the comparison demonstrates that clinical data sets alone do not capture the entire range of malware behavior.

In the study, it was found that malware uses a plethora of different ways to write data to differing types of memory. It also regularly overwrites the code executed during the unpacking process and generally employs more than two unpacking layers. Additionally, a distinction between malware and packer is not always possible, as they are present on a spectrum.

Main Takeaways of this chapter:

This chapter provides a theoretical model to describe intra-process unpacking behavior. A study using a PANDA-based unpacking measurement tool reveals that real-world malware is much more diverse than off-the-shelf packers and benign software. The study has shown that malware uses the API and direct memory addressing to write its code to any place in its process. It also uses multiple unpacking layers and deletes unpacked code as part of the unpacking process, while also blurring the line between what can be considered packer and malware.

5 Multi-Process Unpacking Behavior

This chapter, including several of its figures, primarily recounts the previous research conducted by the author of this dissertation. The evaluation has been truncated to focus on the findings relevant to this dissertation. The findings have been published in the following paper: *Jenke, Thorsten, Simon Liessem, Elmar Padilla, and Lilli Bruckschen. "A Measurement Study on Interprocess Code Propagation of Malicious Software." International Conference on Digital Forensics and Cyber Crime. Cham: Springer Nature Switzerland, 2023.* [6]

5.1 Introduction

Chapter 4 examined the write-then-execute behavior of malware confined to a single process. However, some operating systems, most notably Microsoft Windows, permit write-then-execute behavior across multiple processes, rendering an intra-process analysis insufficient for a comprehensive evaluation of Microsoft Windows malware. This chapter therefore addresses the multi-process aspect of write-then-execute behavior with a study.

First, the concept of *code propagations* is introduced to define the multi-process behavior regarded as unpacking under the write-then-execute heuristic. A theoretical model of code propagation is then presented and implemented in a code-propagation measurement tool. Finally, this tool is employed in a study to explore the multi-process write-then-execute behavior of malware.

The research questions for this study are:

- *RQ1.6*: What methodologies are used by malware to achieve multi-process write-then-executes?
- *RQ1.7*: What functions are used to implement multi-process write-then-executes?

5.2 Code Propagation

The capabilities of multi-process write-then-execute behaviors are examined by first establishing which behaviors qualify. Prior research by Barabosch et al. [85, 60, 86] offers a comprehensive systematization of process injections techniques but omits the need for injected code to be executed. To bridge this gap, a definition of code propagation is introduced that explicitly includes the execution of written code, thereby conforming to

the write-then-execute heuristic. Following this, a theoretical model of code propagation is presented as a methodological basis to guide the implementation of the code propagation measurement tool. This model encompasses both the propagation topology and their implementation.

5.2.1 Definition

In general, a code propagation is defined as an instance of write-then-execute behavior that occurs outside the memory space of the currently observed process.

In more detail, a code propagation conducted by a process $\mathcal{M}$ fulfills the following criteria:

1. The code is written and executed outside of $\mathcal{M}$'s currently running process.
2. $\mathcal{M}$ uses previously written binaries to spawn new processes or injects code into newly created or running processes.
3. The code is executed immediately after writing.
4. Interaction from the user or other processes is not required.

Criterion 1 requires that the code is executed outside of the currently infected process, thereby ensuring the multi-process property of code propagation.

Criterion 2 specifies the way code propagations are carried out, mandating either through the spawn of a malicious process or the injection into another process.

Criterion 3 distinguishes code propagations from persistence techniques [87], in which code is placed for delayed execution to harden the malware against disinfection.

Criterion 4 further separates code propagations from persistence techniques and ensures that code propagations are performed autonomously by the malware.

5.2.2 Representation

In this section, the mathematical representation for code propagations is presented. As mentioned earlier, this dissertation focuses on Windows Malware and, therefore, uses Windows-specific nomenclatures, though the model applies to any operating system that offers multi-process write-than-execute.

Code propagations are the interaction between different processes, and only an infected process can create another code propagation, and therefore spread the infection. One process is also capable of infecting more than one process. This means that there is an infection chain from the original malware process to the last infection, making a graph-based model a natural choice. Therefore, code propagations can be described as a weighted, directed graph:

$$G_{mov} = (V, E) \tag{5.1}$$

with each vertex $v \in V$ representing an infected process and each edge $e \in E$ representing the code propagation between them. There are no reflective edges in E because a code propagation can occur only between two different processes. Therefore, the tuple

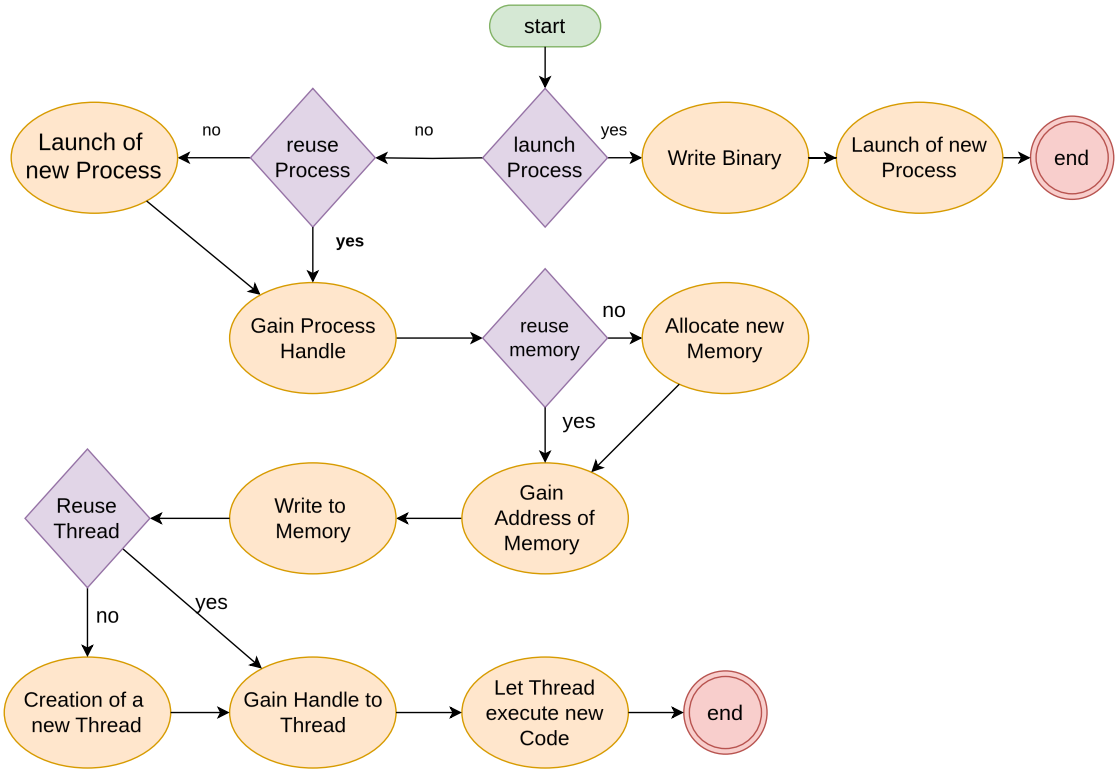

Figure 5.1: Decision tree to implement code propagations: the graph is segregated into two major paths depending on whether a malicious process is launched. The first path (right) describes the spawn of a malicious child process, while the other (left) describes the family of process injections [6].

(G_{mov}, f) denotes the code propagations for a given input binary, and f being the weight function called code propagation implementation, which is explained in the next subsection. In this dissertation, G_{mov} is called the code propagation topology.

5.2.3 Code Propagation Implementation

This section further elaborates on the weight function $f : E \to X$ of the code propagation graph, where $x \in X$ represents the implementation of the code propagation.

Figure 5.1, inspired by the work of Barabosch et al. [60], illustrates two primary paths of how code propagations can be carried out: spawning a malicious child process or injecting code into another process. If the decision lands on spawning a malicious process, a malicious binary is needed to launch a new process: either written to the hard drive by the malware or employing its original binary that spawned the initial process. The binary is then used to spawn a new process.

The other path denotes the various ways through which malware injects its code into other processes. Successful injections depend on three resources: a victim process, a target memory, and a thread. For each resource, the malware decides whether to reuse existing ones or allocate new ones.

- First, the malware must choose a victim **process**. Malware may either appropriate existing processes or spawn a new one from benign binaries for injection.
- The second resource is the **memory** to write malicious code. Once a process is chosen, its already allocated memory regions can be overwritten and, therefore, reused.
- The third resource needed is the **thread** to execute the malicious code written inside the victim process. Similarly to memory, each process already has running threads, but new threads can also be spawned. For example, malware can launch a new process in suspended mode, to manipulate the main thread of the new process after the malicious code has been written.

Measurements of code propagations, and malware behavior in general, inevitably capture only a snapshot of the current landscape. Because future implementation cannot be predicted, the chosen representation is deliberately open to ensure general applicability and a possible future-proof way of describing code propagations. For the representation, an eight-bit vector X encodes each decision step from Figure 5.1: a bit set to 1 denotes that the corresponding action occurred, while a bit set to 0 denotes that it did not.

Bit 1 : Was a new process spawned?

Bit 2 : Was a handle on a process obtained?

Bit 3 : Was a new thread spawned?

Bit 4 : Was a handle on a thread obtained?

Bit 5 : Was new memory allocated?

Bit 6 : Was a memory region reused?

Bit 7 : Was the data written?

Bit 8 : Was the thread resumed?

This representation also accommodates mixed strategies, in which resources can be reused and new ones generated. In addition, it also captures incomplete code propagations. The high diversity of malware projects and malicious actors [10], together with differences in the actors' motivation and capabilities, yield code propagations with superfluous steps or even missing steps, so some code propagations may be incomplete.

5.3 Measuring Code Propagations

The aim of this study is to characterize the multi-process unpacking behavior of malware by measuring the code propagations of real-world malware. To this end, the concepts of the code propagation topologies of the previous section need to be implemented into a code propagation measurement system, as outlined in Figure 5.2. Initially, API calls are gathered into logs. These produced logs are interpreted, and the code propagations are identified. Both the API call recorder and the interpretation step are detailed in the upcoming sections. Since code propagations represent the multi-process unpacking behavior of malware, a code-propagation measurement system provides the means of studying how malware relocates and executes code across process boundaries.

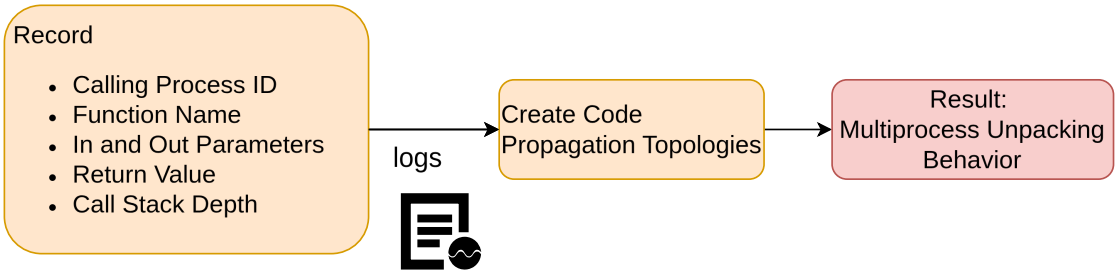

Figure 5.2: The flow of the code-propagation measurement system: API calls are traced and written into logs, which are interpreted to identify code propagations and thus reveal the samples' multi-process unpacking behavior. The interpretation step is displayed in Figure 5.3.

5.3.1 Recording API calls

As noted previously, the code-propagation measurement system focuses on Windows malware. Since code propagations are defined by interactions between processes, which require interaction with the kernel, the API calls for conducting code propagations need to be gathered. Addressing this issue through static analysis is highly time-consuming and impractical due to the numerous potential levels of obfuscation, especially malware packing. The alternative is a dynamic analysis system that records API calls, which puts the code propagation measurement system into the field of dynamic analysis systems.

Therefore, to build the code propagation topologies as described in Chapter 5.2, the code propagation measurement system must gather information on the API calls performed by the malware. This information is as follows:

- function name
- in and out parameters passed to the function
- return value
- depth of the call stack
- Calling Process (process ID)

The name of the function is needed to identify the API call performed. The in and out parameters are essential for attributing the API call to the target process. For example, if the malware first creates a handle on a process, the target's PID is an in parameter to that function, whereas launching a new process before injecting its code, the launched process' PID is in the API call's out parameters. The return value may contain information generated by the API call or the status code indicating the call's success or failure. The depth of the call stack indicates whether the API call is performed by the Windows API or by malware directly. Lastly, the identifier of the performing process should be recorded to identify the origin process.

The analysis plugin presented in the previous chapter would have been a valid candidate but it generates unnecessary performance overhead due to the fine-grained analysis. Also, a solution based on Virtual Machine introspection such as LibVMI would have involved the development of an API-hooking framework. However, Microsoft has released

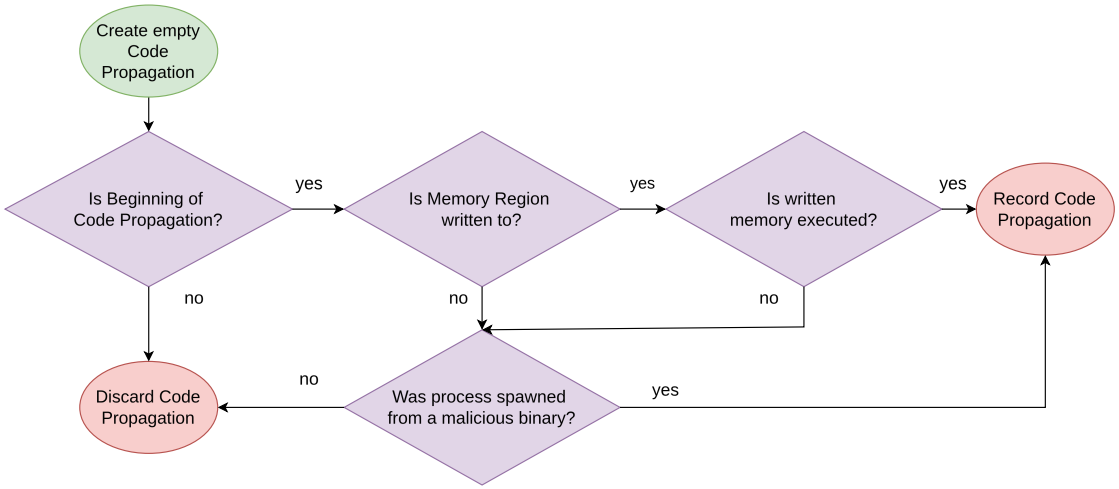

Figure 5.3: Decision tree to interpret the interactions between two processes to identify code propagations.

an actively maintained and thoroughly documented framework called Detours [88]. Detours allows hooking API calls in Windows by offering the tool chain to build libraries that are loaded into processes. Experiments have shown that Detours offers a stable solution with minimal programming overhead. As a result, Detours was chosen as the framework to capture and record API calls. It achieves this by overwriting the first few instructions of the target function with a jump to a function belonging to the Detours hooking library. Therefore, arbitrary code can be inserted into any API call.

For this study, a hooking library was created that is very similar to Microsoft's *Traceapi* [89] example plugin. Each time a hooked function is called, the library creates a log line containing the function's name, in parameters, and depth of the call stack. Then the original function is called to preserve the original functionality of the program. Afterwards a second log line captures the return value, the depth of the call stack, and the out parameters. This approach ensures that all the information required to construct the code propagation have been logged. The library is injected into the processes via the *AppInit_DLL* [90] registry key. Each process writes the log lines into its own log file.

5.3.2 Identifying Code Propagations

In this section, the log files from the API call recorder are processed line by line to identify the code propagations and create the code propagation topology for the input binary. All API-call entries are gathered, matched with their exit lines, and put into a list. Each API call's origin is then examined: If the API call was invoked by an API function instead of directly by the malware, it is recorded in a list as internal API calls. Each of these API calls represents an edge in a directed graph, with the origin vertex being the process that conducts the API call and the end vertex being the process that is interacted with.

A preliminary directed graph is defined as $G_{prelim} = (V_{prelim}, E_{prelim})$ with every captured process in V_{prelim} and all API calls between them in E_{prelim}. An empty set of vertices V and an empty set of code propagations E are also created. The flow of interpretation is shown in Figure 5.3. Each pair of two vertices $v_1, v_2 \in V_{prelim}$ is checked if there is at least one API call $e \in E_{prelim}$ between them. If no edge exists, the next pair of vertices is examined. Otherwise, it is checked whether the API call denotes the start of a code propagation according to Figure 5.1.

On Microsoft Windows, these functions are:

- OpenProcess & NtOpenProcess & CreateProcessW & CreateProcessA & WinExec
- CreateProcessInternalA & CreateProcessInternalW & NtCreateUserProcess
- ShellExecuteExW & ShellExecuteExA & ShellExecuteA & ShellExecuteW

If these functions are not encountered, then the next pair of vertices is examined. If these functions are encountered, the set of edges between the examined vertices could contain a code propagation and is therefore further analyzed, and a weight for the edge between v_1 and v_2 is created with all bits set to zero. If a process is spawned, the first bit is set to 1, and if a process is opened, the second bit is set to 1.

Next, it is checked whether the memory is being manipulated between the two processes. The functions checked for this interaction in Microsoft Windows are:

- VirtualAllocEx & VirtualAllocExNuma & NtAllocateVirtualMemory & NtMapViewOfSection
- WriteProcessMemory & NtWriteVirtualMemory & VirtualQueryEx

The functions *WriteProcessMemory*, *NtWriteVirtualMemory*, and *NtMapViewOfSection* signify that the data have been written to the target process and are used to set the seventh bit. It is not always possible to detect the reuse of a memory section through the usage of the API. So, if memory is being written inside another process that has not been previously allocated using *VirtualAllocEx*, *VirtualAllocExNuma*, or *NtAllocateVirtualMemory*, the sixth bit is set to 1. However, if they were used for allocating, the fifth bit is set to 1.

The last resource to check are the threads. The functions to manipulate and allocate threads on Microsoft Windows are:

- OpenThread & SetThreadContext & NtSetContextThread & NtOpenThread
- ResumeThread & NtResumeThread & CreateRemoteThread & QueueUserAPC

In case a thread is created or opened to execute the written code, the third or fourth bits are set to 1, respectively. Manipulation of a thread that is allocated with the spawn of a new process is denoted by the fourth bit being set to 1. Next, it is checked whether a thread is started to execute the written code, which is denoted by the eighth bit. Now that all bits have been set, it can be checked whether the written code has been executed inside the victim process.

If the functions between the two processes represent a functioning code injection, the edge between v_1 and v_2 is added to E, the resulting weight is added to the weight

```
479024d: 2072: +CreateProcessW(<NULL>,C:\Windows\SysWOW64\upnpcont.exe,0,0,0,0,0,<NULL>,18ff04,18ff50)
47b824f: 2072: -CreateProcessW(,,,,,,,,,) -> 1 (proc:2168/f0, thrd:2172/ec
47b828e: 2072: +OpenProcess(1f0fff,0,pid=2168)
47b8339: 2072: -OpenProcess(,,) -> f8
47b838c: 2072: +VirtualAllocEx(f8,0,9a4a,1000,40)
47b83c7: 2072: -VirtualAllocEx(,,,,) -> 100000
47b841a: 2072: +WriteProcessMemory(f8,@100000..109a49,27bea0,0)
47b8515: 2072: -WriteProcessMemory(,,,,) -> 1
47b8573: 2072: +CreateRemoteThread(f8,0,0,100000,0,0,18ff6c)
47b87d5: 2072: -CreateRemoteThread(,,,,,,) -> f4 (2176
488e0fa: 2072: +ExitProcess(0)
```

Figure 5.4: Truncated snippet of a log file containing a code propagation [6].

function f, and v_1 and v_2 are added to V. Otherwise, the binary used to spawn the process is examined. If the binary has previously been written by a malicious process, the code propagation is recorded. Otherwise, the code propagation is discarded because no malicious code has been executed. This process is repeated with the next pair of vertices contained in V_{prelim}.

The resulting sets V, E, and the weight function f fully describe the code propagation of the input malware. This program has been implemented in a Python program that takes the logs of the malware sandboxing step as input.

Figure 5.4 presents an example snippet of the previously mentioned log file, heavily truncated to improve readability. The snippet shows that the malware creates a new process using CreateProcessW, which takes, among other fields, a string containing the path to a binary as input parameter and a structure called LPPROCESS_INFORMATION as an output parameter [91]. The string denotes the binary that shall be used to spawn the new process, and the struct contains the process ID, thread ID of the main thread, a handle to the process, and the thread when the call returns. In this example, the ID of the created process is 2168.

Using the ID of the process, the malware uses OpenProcess to open the spawned process [92], which is unnecessary because CreateProcessW already returns the handle $f8$. This handle is used as input for VirtuaAllocEx to allocate a memory section of 1,000 bytes, and the base address of this section is returned [80]. Next, the malware calls WriteProcessMemory using the process handle and the base address of the memory to write its code [30]. The written code is executed using CreateRemoteThread, which takes the handle of the process and the address of the written code to create a thread [93]. Lastly, the malware process is terminated using ExitProcess [94].

5.4 Code Propagation Study

This section presents the study of code propagation to explore the multi-process unpacking behavior of malware. However, it is heavily truncated compared to the published article [6], with emphasis placed on the behaviors required to derive the requirements for a generic malware unpacker. First, the setup of the study is described. Then, the implementations of code propagations are analyzed to answer the research questions posed in Chapter 5.1.

5.4.1 Setup

The setup is analogous to the one presented in Chapter 4.4.1, employing four instances of VirtualBox virtual machines with 4 GB of RAM. Guest operating systems ran Windows 7 64-bit SP1 with Visual C++ redistributables installed and have been set up as described in [95].

As mentioned in Chapter 3.3, two data sets, Malpedia and dataset-packed-pe, were chosen, but since none of the dataset-packed-pe samples exhibited multi-process unpacking behavior in the experiments, this data set was excluded from this study. The remaining data set comprised all PE files including 64-bit executables and DLLs. In total, 4,889 samples have been sandboxed: 3,358 were executables and 1,531 DLLs. Of these, 1,456 samples, approximately 30%, performed code propagations. Only 86 of those samples were DLLs, indicating that code propagations are not frequently used in DLL malware. In fact, 41% of the executables performed code propagations, highlighting the importance of handling this behavior in malware analysis. The 1,456 resulting code propagation topologies are further analyzed in this study.

5.4.2 Results

The findings of the study are presented here. First, the measured implementations of code propagations are detailed. This answers the research question:

RQ1.6: What methodologies are used by malware to achieve multi-process write-then-executes?

This is followed by an assessment of which functions are used to realize them and of the distribution of function calls over the different samples to answer the research question:

RQ1.7: What functions are used to implement multi-process write-then-executes?

5.4.2.1 Code Propagation Implementations

This subsection discusses the methodologies of the code propagations. Our findings, depicted in Figure 5.1, show the type of propagation techniques by both their decimal and binary encoding. Spawning a new malicious process emerges as the most prevalent method, occupying both first place and also fourth place when the process is created in suspended mode and its main thread is explicitly resumed. The second most common technique, 217, is the injection into a new process, allocating new memory, and executing a reused thread. This technique is also commonly referred to as *process hollowing*. Injection into an already running process is in the third place, which is technique 86. This code propagation is the injection into a running process, the allocation of new memory, and the creation of a new thread to execute the written memory. Altogether, these findings confirm that the most common methods for multi-process write-then-execute are spawning new malicious processes from previously written binaries and injecting into new processes.

Code Propagation		Samples
Propagation	**Binary Code**	-
1	00000001	1018 (69.92%)
217	11011001	270 (18.54%)
86	01010110	101 (6.94%)
129	10000001	86 (5.91%)
225	11100001	85 (5.84%)
209	11010001	78 (5.36%)
85	01010101	43 (2.95%)
145	10010001	38 (2.61%)
82	01010010	34 (2.34%)
241	11110001	31 (2.13%)
249	11111001	28 (1.92%)
113	01110001	15 (1.03%)
210	11010010	14 (0.96%)
233	11101001	13 (0.89%)
81	01010001	12 (0.82%)
118	01110110	9 (0.62%)
117	01110101	7 (0.48%)
130	10000010	6 (0.41%)
50	00110010	6 (0.41%)
229	11100101	5 (0.34%)
114	01110010	5 (0.34%)
153	10011001	4 (0.27%)
214	11010110	3 (0.21%)
97	01100001	3 (0.21%)
230	11100110	3 (0.21%)
102	01100110	2 (0.14%)
246	11110110	2 (0.14%)
22	00010110	1 (0.07%)
242	11110010	1 (0.07%)

Table 5.1: Occurrence of code-propagation techniques in topology types. Each propagation is counted once per sample. The number in the first column is the decimal number representation of the 8 bit-vector. The most important code propagation techniques are the spawn of new processes, technique 217, (injection, newly created process, new memory allocation, executed by reused thread), and technique 86 (injection, running process, new memory allocated and executed by a spawned thread). The explanation for the binary codes can be found in Section 5.2.3.

This answers the research question *RQ1.6*:

- A generic malware unpacker needs to deal with code written to other processes.
- A generic malware unpacker needs to deal with code written to the hard drive.

5.4.2.2 Used API Functions

Function	Total Usage	Samples
CreateProcessW	2377	849 (58.31%)
WriteProcessMemory	48,671	574 (39.42%)
ResumeThread	801	555 (38.12%)
VirtualAllocEx	41387	510 (35.03%)
CreateProcessA	1602	459 (31.52%)
SetThreadContext	467	295 (20.26%)
VirtualProtectEx	1246	169 (11.61%)
OpenProcess	15513	162 (11.13%)
CreateRemoteThread	15347	155 (10.65%)
NtMapViewOfSection	256	151 (10.37%)
NtWriteVirtualMemory	731	133 (9.13%)
ShellExecuteA	123	96 (6.59%)
ShellExecuteExW	216	95 (6.52%)
ShellExecuteW	133	92 (6.32%)
NtResumeThread	98	69 (4.74%)
CreateProcessInternalW	63	56 (3.85%)
WinExec	65	52 (3.57%)
CreateProcessInternalA	22	22 (1.51%)
QueueUserAPC	10	10 (0.69%)
VirtualQueryEx	74	6 (0.41%)
ShellExecuteExA	5	5 (0.34%)
OpenThread	3	3 (0.21%)

Table 5.2: Functions, their number of usages, and the number of samples using the respective functions.

Now, the functions used to achieve the code propagations are discussed. Table 5.2 shows the results.

- **Process Creation and Manipulation**:
  - **CreateProcessW** [91] is the most used function, appearing in 58.31% of samples. This aligns with the analysis of the code propagation methodologies, in which spawning a new process is the most prevalent method. This is supported by the still frequent use of **CreateProcessA**. Other functions achieving similar results are the **ShellExecute** [96] family of functions, **WinExec** [97], and the low-level equivalents **CreateProcessInternalA** and **CreateProcessInternalW**.

  - **OpenProcess** [92] is used to create a handle on an already existing process. Handles are commonly used in functions to manipulate another process. It was observed in 11.13% of samples. Although this number may seem low, the CreateProcess family of functions also returns a handle to a process, making a call to OpenProcess obsolete.

- **Memory Writing:**
  - **WriteProcessMemory** [30] is the most used function with 48,671 total usages in 39.42% of samples. It is used to write data to be later executed and is therefore frequently used to implement code injections.
  - **NtWriteVirtualMemory** is used in 9.13% of samples and the low-level equivalent of WriteProcessMemory.
- **Thread Manipulation**:
  - **ResumeThread** [98] ranked third, with 38.12% of samples using it. However, this function is used not only in code injections, but also regularly used to spawn a malicious process. As mentioned above, it is possible to spawn a process in a suspended state and then use ResumeThread to run the main thread.
  - **NtResumeThread** is the low-level equivalent of ResumeThread and is used in 4.74% of samples.
  - **SetThreadContext** [99] can be used to manipulate an existing thread to execute the code of the code propagation. It was used in 20.26% samples.
  - **CreateRemoteThread** [93] was observed in 10.65% of samples and spawns a thread in another process, executing the injected code.
  - **OpenThread** [100] is used to manipulate an existing thread and was only used in three samples.
  - **QueueUserAPC** [101] is used to perform an APC injection and was used in ten samples.
- **Memory Manipulation:**
  - **VirtualAllocEx** [80] is the fourth most used function and is a necessary step in code propagations that do not reuse memory. It was observed in 35.03% of samples.
  - **VirtualProtectEx** [25] is closely related to this function as it is used to change permissions of a memory section. It was observed in 11.61% of samples.
  - **VirtualQueryEx** [102] was only used in six samples and retrieves information about a memory page.
  - **NtMapViewOfSection** [72] is used to transfer code to the memory of another process and was observed in 10.37% of samples. It enables the malware to write malicious code into a mapped section in its own memory space and then mapping that to the memory of another process to execute it. This concept is further explored in [103].

This analysis answers the research question:

RQ1.7: What functions are used to implement multi-process write-then-executes?

5.5 Limitations

One limitation of this implementation is the use of Detours or VirtualBox, both of which could be detected by malware. Upon detection, samples can hide their functionality and skip code propagations altogether.

Detours also hooks the APIs in the user space, whereas some malware may bypass these hooks by implementing the syscalls themselves [104].

Another limiting factor arises from evasion techniques, especially API hammering [21], where repeating the same API call many times overwhelms the system. In our experiments, API hammering caused the memory of the interpretation component to overflow, preventing the analysis of those samples.

In addition, no mechanism has been implemented that checks whether all code propagations have been performed. Therefore, there is no way to ensure that the behavior observed in the study is complete.

5.6 Conclusion

This chapter examined the malware's multi-process write-then-execute behavior, focusing on the method employed to realize this behavior and the functions to implement them. The results shall aid in understanding how multi-process write-then-execute behavior works and which functions need to be observed to spot it.

A mathematical model was proposed and implemented in a dynamic analysis system based on Microsoft Detours. Applying this system to a representative real-world data set containing Windows malware revealed that spawning a new process from a malicious binary and process hollowing are the dominant multi-process write-then-execute methods, while other process injections techniques occur far less frequently.

These findings imply that a generic malware unpacker must be able to deal with code, not only on the hard drive but also in the memory of other processes. In addition, the *CreateProcess* family of functions, *WriteProcessMemory*, *NtWriteVirtualMemory*, and *NtMapViewOfSection* emerged as strong indicators for code propagations.

The insights gained in this chapter inform the requirements for a generic malware unpacker. In the next chapter, the requirements are formulated, and an assessment is made to evaluate past generic malware unpackers using said requirements.

Main Takeaways of this chapter:

This chapter provides a theoretical model to display multi-process write-then-execute behavior. This model is used to conduct a study of real-world malware and reveals how malware spawns new processes and uses existing ones to spread its code to another process. Malware also uses the hard drive to write its code and spawns a process from it.

6 Past Generic Malware Unpackers

This chapter primarily recounts the previous research conducted by the author of this dissertation. It is restructured from an independent article into the flow of this dissertation. The author of this dissertation has published them in the following paper:
Jenke, Thorsten, Max Ufer, Manuel Blatt, Leander Kohler, Elmar Padilla, and Lilli Bruckschen. "Democratizing Generic Malware Unpacking." 2025 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). IEEE, 2025. [7]

6.1 Introduction

This chapter provides an in-depth analysis of the previous unpacking methodologies of Chapter 3 and their shortcomings based on the two studies discussed in Chapter 4 and Chapter 5. From these studies, the requirements for a generic malware unpacker are derived and stated. Finally, the previous unpacking methodologies are evaluated against these requirements.

This chapter answers the research questions:

- *RQ1*: What are the requirements for a malware unpacker that it must meet to be generic?
- *RQ2*: Do the previously proposed generic malware unpackers fulfill the requirements for a generic malware unpacker?

6.2 Requirements for a Generic Malware Unpacker

This section defines the requirements for a generic malware unpacker, as derived from the studies presented in Chapter 4 and Chapter 5.

The first set of requirements are derived from the types of memory writes that have been observed in the studies. The vast majority of malware uses direct access to memory to write the unpacked code. This leads to the first requirement:

R1:
A generic malware unpacker must be able to handle write operations carried out in user mode.

Malware has also been shown to use the API to write code into its own memory, to the hard drive or other processes. This leads to the second requirement:

R2:
A generic malware unpacker must be able to handle write operations through the API.

While some packers only use a limited number of different memory locations, the multi-process write-then-execute study has shown that malware uses any kind of memory location. This leads to the third requirement:

R3:
A generic malware unpacker must be compatible with all kinds of memory locations.

As mentioned previously, memory writes are also performed to implement code propagations, which means write operations to other processes or files on the hard drive. This leads to the fourth and fifth requirement:

R4:
A generic malware unpacker needs to cover injections into other processes.

R5:
A generic malware unpacker must be able to deal with new processes spawned from binaries on the hard drive.

Lastly, the topology of the unpacking layers is discussed. The intra-process write-then-execute study demonstrated that the majority of malware uses multiple packing layers. This leads to the sixth requirement:

R6:
A generic malware unpacker must be able to handle multiple unpacking layers.

In addition, these unpacking layers may also contain code that is deemed worthy of protection by malicious actors. They can be used to deduce knowledge about the tool chain used, which can yield information on the involved threat actor groups. This leads to the seventh requirement:

R7:
A generic malware unpacker must be able to unpack all unpacking layers.

Lastly, the intra-process write-then-execute study shows that malware and packer cannot be separated, rendering the search for an original entry point counterproductive in many cases. This leads to the eight requirement:

R8:
A generic malware unpacker does not rely on the dichotomy of malware and packers.

The behavior and their derived requirements are summarized in Table 6.1.

ID	Behavior	Requirement
Memory Writes		
R1	Direct write access to memory	Deal with memory writes in user mode
R2	API is used to write data	Deal with memory writes through API
Memory Locations		
R3	All kinds of memory locations are used	Deal with all types of memory
R4	Code is written to another process	Deal with code in other processes
R5	Code is written to the hard drive	Deal with code on hard drive
Topology		
R6	Multiple packing layers	Aware of multiple packing layers
R7	Unpacked code is deleted	Extract all unpacking layers
R8	Packer functionality in packed data	Handle mixed packer and malware code

Table 6.1: This table shows the exhibited behavior of malware and the requirements derived from them.

A generic malware unpacker must fulfill all these requirements. Since they are derived from the unpacking behavior observed in previous studies, failing any of them will prevent the unpacker from unpacking every sample analyzed in the studies.

6.3 Assessment

To answer research question *RQ2*, the related work introduced in Chapter 3 is evaluated against the stated requirements. For each contribution, it is stated whether they fulfill each requirement. The results are summarized in Table 6.2.

Malware Normalization

The unpacker presented in the *Malware Normalization* [4] paper fulfills requirement **R1**, although its compatibility with requirement **R2** cannot be inferred. It can locate malware in all memory locations within a process, thus fulfilling requirement **R3**, but is incompatible with code propagations and therefore fails to meet requirements **R4** and **R5**. Their unpacker is compatible with multiple layers to fulfill requirement **R6**. The authors claim that code being overwritten during unpacking is out-of-scope and,

Name	R1	R2	R3	R4	R5	R6	R7	R8
Normalization	✓	?	✓	✗	✗	✓	✗	✓
Polyunpack	N/A	N/A	✓	✗	✗	✗	✗	✗
Omniunpack	✓	✓	✓	✗	✗	✓	✗	✗
Renovo	✓	?	✓	✗	✗	✓	✓	✓
Etherunpack	✓	?	✓	✗	✗	✓	✓	✓
Eureka	N/A	N/A	✓	✗	✗	✓	✗	✗
Coogan	✓	✗	✓	✗	✗	✗	✗	✗
JUSTIN	✓	✗	✓	✗	✗	✓	✗	✗
Tracesurfer	✓	✗	✓	✗	✗	✓	✓	✓
Jeong	N/A	N/A	✗	✗	✗	✗	✗	✗
Malwise	✓	?	✓	✗	✗	✓	✗	✗
Codisasm	✓	✗	✓	✗	✓	✓	✓	✓
Binunpack	N/A	N/A	✓	✓	✓	✓	✗	✗
Malflux	✓	?	✓	✗	✓	✓	✗	✗
Roamer	N/A	N/A	✗	✓	✓	✓	✗	✓

Table 6.2: Assessment of previous malware unpacker based on the requirements. ✓ : unpacker fulfills requirement; ✗ : unpacker fails requirement; '?': definitive answer could not be derived from the publication; 'N/A': requirement is not applicable to that methodology. No unpacker meets all requirements.

therefore, requirement **R7** is violated. It does not differentiate between malware and packer, to satisfy requirement **R8**.

Polyunpack

Polyunpack [36] is not based on the tracing of executed bytes or written memory, therefore requirements **R1** and **R2** are not applicable. Because Polyunpack is agnostic to the memory in which the unpacked code is written to, requirement **R3** is met. However, it is completely unaware of multi-process unpacking and multiple unpacking layers, and it assumes a clear distinction between malware and packer code. The requirements **R4**, **R5**, **R7**, and **R8** are not met. Regarding multiple unpacking layers, the authors say that their algorithm should be applied multiple times, but since it is not part of the core algorithm, they violate **R6**.

Omniunpack

Omniunpack [47] treats all types of memory and memory writes uniformly, thereby satisfying requirements **R1**, **R2**, and **R3**. However, their program is not compatible with multi-process unpacking, so requirements **R4** and **R5** remain unfulfilled. The program can handle multiple unpacking layers, meeting requirement **R6**, but the inclusion of antivirus software leads to the extraction of only the layer on which the antivirus hits, which violates requirement **R7**. Finally, Omniunpack assumes a dichotomy between malware and unpacker, failing to satisfy requirement **R8**.

Renovo & Etherunpack

Renovo[41] is compatible with write operations in user mode, satisfying requirement **R1**, but its documentation does not confirm whether it also traces write operations done in kernel mode, leaving requirement **R2** open. It does not discriminate between different types of memory, fulfilling requirement **R3**, yet multi-process unpacking is completely outside its scope, thereby violating requirements **R4** and **R5**. Renovo is compatible with multiple unpacking layers, unpacks all layers, and does not differentiate between malware and packer code, fulfilling the remaining requirements **R6**, **R7**, and **R8**.

Etherunpack [42], as a reimplementation of Renovo that reproduces its evaluation, violates the exact same requirements.

Eureka

Eureka [49] does not trace memory writes, so requirements **R1** and **R2** are not applicable. It does not discriminate between different types of memory, meeting requirement **R3**. Due to the threshold used for the code-to-data ratio, it can be argued that Eureka is compatible with multiple packing layers, meeting requirement **R6**. However, Eureka is incompatible with multi-process unpacking, does not unpack all unpacking layers, and assumes a difference between malware and packer code. Therefore, the requirements **R4**, **R5**, **R7**, and **R8** are not met.

JUSTIN

JUSTIN [3] traces memory writes in user mode and not kernel mode, fulfilling requirement **R1** and violating **R2**, and checks for all memory locations, satisfying requirement **R3**. It is incompatible with code propagation, which violates requirements **R4** and **R5**. The tool is also aware of multiple unpacking layers, but does not extract them, meeting requirement **R6**, but failing requirement **R7**. Lastly, because it uses an antivirus scanner to find the OEP, it assumes a separation between malware and packer and therefore does not satisfy requirement **R8**.

Coogan Methodology

Coogan et al. [14] trace the memory writes of the binary in their heuristic, satisfying requirement **R1**, but not of the used API, thereby failing requirement **R2**. They do not differentiate between different types of memory, fulfilling requirement **R3**, but multi-process unpacking and multiple unpacking layers are out of scope, violating requirements **R4**, **R5**, **R6**, and **R7**. Lastly, it assumes a clear distinction between malware and packer, thus failing requirement **R8**.

Tracesurfer

Tracesurfer [51] traces memory writes in user mode, fulfilling requirement **R1**, but not in kernel mode, violating **R2**. It is compatible with all memory locations, meeting requirement **R3**, but does not consider code propagations, failing requirements **R4** and **R5**. However, the tool handles and extracts all unpacking layers and does not distinguish between malware and packer, thus satisfying requirements **R6**, **R7**, and **R8**.

Jeong Methodology

Their methodology [52] does not track write operations or executions, requirements **R1** and **R2** are not applicable. The unpacker seems to be compatible with unpacking only into the image sections, while code injections and child processes are out-of-scope, thereby violating requirements **R3**, **R4**, and **R5**. Furthermore, they only support one unpacking layer and do not extract any other layers, failing requirements **R6** and **R7**. Finally, their model is heavily based on a dichotomy between malware and packer, so requirement **R8** is also not met.

Malwise

Malwise [43] traces the memory write in user mode satisfying requirement **R1**, but similarly to the tracer-based unpackers mentioned above, it cannot be inferred whether the unpacker also traces writes in kernel mode, leaving requirement **R2** ambiguous. It checks for every possible memory location, fulfilling requirement **R3**, yet does not handle code propagations, breaking requirements **R4** and **R5**. The tool is compatible with multiple unpacking layers, meeting requirement **R6**, but does not extract all unpacking layers, failing requirement **R7**, and considers malware and packer to be two different entities, thus not meeting requirement **R8**.

CoDiasm

CoDiasm [44] is compatible with memory writes in user mode, meeting requirement **R1**, but cannot track writes in kernel mode, violating requirement **R2**. Codisasm is compatible with all memory locations, fulfilling requirement **R3**. While it tracks the creation of new threads and processes, injections into new processes are not fully covered, so requirement **R4** is not met, but is fully compatible with requirement **R5**. The tool is compatible with multiple layers and also extracts all the unpacking layers, therefore fulfilling requirements **R6** and **R7**. Finally, their model acknowledges that packer and malware cannot always be differentiated, satisfying requirements **R8**.

Binunpack

Tracing of memory writes is not part of Binunpack [53], therefore requirements **R1** and **R2** are not applicable. It is compatible with all types of memory, detects multi-process

Requirements	Number of Unpackers
R1	15
R2	6
R3	13
R4	2
R5	4
R6	12
R7	4
R8	6

Table 6.3: This table shows how many unpackers fulfill each requirement. Entries marked "N/A" are counted as compatible with the requirements because they are not needed for their respective unpacker and, therefore, not a flaw. Entries flagged as "?", indicating uncertainty about compatibility is treated as unmet requirement.

unpacking, and is aware of multiple unpacking layers, fulfilling requirements **R3**, **R4**, **R5**, and **R6**. However, it only extracts the last layer and assumes a dichotomy between malware and packer, failing requirements **R7** and **R8**.

Malflux

Just as the other tracers, it cannot be inferred if Malflux [54] is able to follow write operations in the kernel but it does in user mode, therefore satisfying requirement **R1** but requirement **R2** is inapplicable. It is compatible with all memory locations, meeting requirement **R3**, yet does not account for code injections, breaking requirement **R4**, although it tracks newly created processes, fulfilling requirement **R5**. The tool can handle multiple unpacking layers, meeting requirement **R6**, but does not extract them, violating requirement **R7**, and does not consider malware and packer to be one entity, failing **R8**.

Roamer

Roamer [55] does not trace the memory writes, therefore requirements **R1** and **R2** are not applicable. It is not compatible with all types of memory, instead only with newly allocated sections, violating requirement **R3**. However, code written to other processes and processes launched from malicious binaries are covered by Roamer, fulfilling requirements **R4** and **R5**. Roamer is also compatible with multiple unpacking layers, meeting requirement **R6**, but only extracts the memory at the end of the unpacking, violating **R7**. Roamer does not differentiate between malware and packer, meeting requirement **R8**.

6.3.1 Analysis

Table 6.3 reports how many unpackers meet each requirement. The least fulfilled requirements are those related to code propagations. The widespread use of multi-process

Unpacker	Number of met Requirements
Binunpack	6
Codisasm	6
Roamer	6
Renovo/Etherunpack	5
Tracesurfer	5
Normalization	4
Malflux	4
Omniunpack	4
Eureka	4
JUSTIN	3
Polyunpack	3
Malwise	3
Jeong	2
Coogan	2

Table 6.4: This table shows the unpackers and the amount of met requirements. Binunpack, Codisasm, Renovo/Etherunpack, Tracesurfer, and Roamer are the best performing unpackers. "N/A" and "?" are handled as in Table 6.3.

execution by malware to hide its behavior and the failure of most tools to address it, represents a massive oversight in current research. This is followed in prevalence by the requirement that all layers need to be unpacked and that there is no distinction between malware and packer.

A summary of the evaluation can be found in Table 6.2, which shows that no existing unpacker satisfies all requirements. This may help explain why none of these academic unpackers has managed to provide a lasting solution. Notably, five tracer-based unpackers do not specify whether they are also tracing memory writes in kernel mode.

As can be seen in Table 6.4, the best performing unpackers are Binunpack, Roamer, and Codisasm with six fulfilled requirements. Renovo/Etherunpack and Tracesurfer follow with five met requirements. Three of the top five best performing unpackers are all based on a tracer. Etherunpack is not counted separately, since it is a reimplementation of Renovo. Binunpack and Roamer score highly because they fulfill the code-propagation-based requirements. Excluding those two requirements, the best performing unpackers are those that implement write-then-execute in its most naive way. In contrast, each time researchers added another heuristic to enhance write-then-execute in any way, they invariably introduced unwanted assumptions, which have been disproved in our studies, on the property of packers [47, 54, 43]. For example, Omniunpack relies on antivirus software to determine the end of unpacking, presuming a clearly defined termination point for unpacking or that the entire malware resides in memory. Malwise introduced entropy-based heuristics to check the unpacking status, assuming that the entropy in the memory must change during the unpacking procedure. Malflux similarly presumes a minimum number of write operations before the code is executed. These specific assumptions reduce genericity and are refuted by our studies.

Therefore, implementing an unmodified write-then-execute unpacker combined with a robust code propagation detector has promising potential to improve on the related work, as such a design would satisfy all eight requirements. The construction of such an unpacker is the subject of the following chapter. Alternatively, Binunpack or Roamer could have been extended to fulfill all requirements. However, Binunpack's rebuilt-then-execute heuristic assumes a strict separation between malware and packer. Similarly, Roamer's heuristic forbids that some sections contain malicious code, such as the stack of a process that has been injected because this is not a new section.

6.4 Conclusion

This chapter established a set of requirements for a generic malware unpacker, derived from the studies in Chapter 4 and Chapter 5. A survey of existing unpacking methodologies against these requirements revealed that none fulfill all requirements. Especially, the handling of multi-process unpacking is seldom covered, despite its frequent occurrence, as shown in Chapter 5. This deficiency likely explains why academic unpackers have seen limited practical adoption for real-world malware analysis.

The assessment also demonstrated that unpackers implementing the unmodified write-then-execute heuristic outperform those that augment it with additional assumptions. Therefore, the next chapter introduces a new generic malware unpacker that combines the unmodified write-then-execute heuristic with a robust code propagation detector.

Main Takeaways of this chapter:

This chapter summarizes the requirements for a generic malware unpacker as derived from the previous studies. These requirements are used to evaluate previous research on the construction of a generic malware unpacker. The assessment reveals that none of the previously proposed unpackers is able to meet all the requirements, indicating that they are not suitable for real-world application. Therefore, in the next chapter the strongest performing heuristic is selected and combined with a mechanism to detect code propagations.

7 GeMU: The QEMU-Based Generic Malware Unpacker

The content of this chapter is mostly a retelling of previous research paper by the author of this dissertation:
Jenke, Thorsten, Max Ufer, Manuel Blatt, Leander Kohler, Elmar Padilla, and Lilli Bruckschen. "Democratizing Generic Malware Unpacking." 2025 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). IEEE, 2025. [7]
The description of GeMU has been updated to its most recent version.

Figure 7.1: Logo of the generic malware unpacker GeMU

7.1 Introduction

The last chapter demonstrated that no existing unpacker in the literature satisfies all eight requirements, answering research question *RQ2*. To address this gap, this chapter presents GeMU (Generic Malware Unpacker), which combines a write-then-execute based unpacker with a code-propagation detector.

The first section discusses the methodology, explaining how the different aspects of the methodology satisfy all requirements of Table 6.1. Next, the QEMU-based implementation of this methodology is described. This implementation is used to conduct a study, asserting a generic malware unpacker's performance on three data sets: Malpedia [105], dataset-packed-pe [11], and MWB2024, a data set composed of malware samples that have been uploaded to a malware sharing site called Malware Bazaar [12] in 2024.

The results of this chapter answer the research question:

RQ3: Is it possible to develop a malware unpacker that can meet every requirement for a generic unpacker and works perfectly on the data set used to create the requirements?

7.2 Methodology

This section describes the methodology of the new generic malware unpacker GeMU. As established in Chapter 6, the chosen unpacking heuristic is write-then-execute combined with a way to handle code propagations.

The write-then-execute heuristic requires tracing both write operations and code executions. Whenever a memory address is accessed, the associated memory section is identified and marked as written. Tracing then resumes, and if code is executed in a previously marked-as-written section, that memory section is extracted from memory. Once extracted, the memory section is unmarked, and tracing resumes. Each extracted memory section constitutes the output. This process satisfies requirement **R1** by tracing the memory writes and code executions, and fulfills **R3** because no distinction between memory locations has been made. The continued tracing of the program and the immediate extraction from memory accounts for additional unpacking layers and prevents the code from overwriting unpacked code after execution, ensuring that requirements **R6** and **R7** are met.

However, requirements concerning code propagations must also be addressed. To ensure compliance with requirement **R2**, the unpacker must monitor all operating system functions capable of writing code and mark the written sections accordingly. Full compliance with requirements **R4** and **R5** further necessitates the observation of the functions that execute code in a target process or on the hard drive. If a write-then-execute occurs in a target process, this process must also be traced. Thus, whenever a traced process writes data either directly to memory or over the hard drive to another process, the associated memory sections must be marked as written and extracted when executed.

This methodology is completely agnostic to the overall functionality of the traced code. It refrains from inferring or approximating the purpose of code and, therefore, does not discriminate between different pieces of code. Consequently, it does not distinguish between malware and packer code and fulfills requirement **R8**.

7.3 Implementation

The previous section introduced the methodology of GeMU. This method can be implemented using tools for both static and dynamic analysis. As established, it is essential

to trace both write operations and code executions. While static analysis offers solutions such as symbolic execution or the methodology of Coogan et al. [14], symbolic execution is very performance intensive and provides capabilities beyond what is required for implementing an unpacker.

Among dynamic analysis techniques, debuggers, virtual machine introspection, and emulation are potential candidates. However, debuggers are easily detected by malware and therefore unsuitable. Virtual machine introspection, though less detectable, suffers from performance limitations due to frequent processor switches out of virtualization mode. Emulation, in contrast, offers a favorable balance: it allows fine-grained analysis comparable to symbolic execution, while maintaining the performance of dynamic analysis approaches suitable for large-scale analysis. Therefore, emulation was chosen for the GeMU's implementation.

In Chapter 4 PANDA [9] has been used as an emulation-based dynamic analysis system. However, PANDA is based on an outdated QEMU [18] version. Since then, the QEMU code base underwent improvements, including speed-ups. Therefore, GeMU has been implemented using QEMU version 8, which was the most recent version available at the time of development.

The development also focuses on native binaries for Microsoft Windows. Given that most historic and a significant portion of contemporary malware is released for Microsoft Windows [10, 1], this choice ensures compatibility with the largest malware ecosystem. Specifically, GeMU is developed for Windows 7 and Windows 10, supporting historical and contemporary malware to enable longitudinal studies and future-proofing the implementation.

GeMU is implemented as a library that is imported by QEMU at an early stage of execution. The interactions between GeMU and QEMU are denoted in Figure 7.2. GeMU's internal state consists of a set of observed PIDs, memory maps of each observed process, and the relation from handles to PIDs. To enable tracing, QEMU's source code has been altered to invoke GeMU's three hooking functions for three different execution stages: writing function, translation function, and syscall/sysret function. Each of these functions share a prologue that checks whether the current process should be observed. This is achieved by extracting the current PID from memory and checking its presence in the list of observed PIDs. The initial malware process is recognized by its name and added to the observed PIDs.

7.3.1 Writing Function

The writing function sets the writing bit on memory pages, signaling that such pages should be extracted from memory if the code contained therein is executed. In QEMU, memory writes can occur via two paths: the *fast path* and the *slow path*. The fast path is conducted inside the TCG via TinyCode instructions, which is very fast. The slow path, however, is invoked when a memory write targets pages that are not currently cached or contain translated basic blocks. The slow path is not part of the TinyCode.

GeMU's *writing function* is invoked whenever QEMU takes the slow path for a memory write. This ensures that the correct page is loaded and that the cache of translated basic

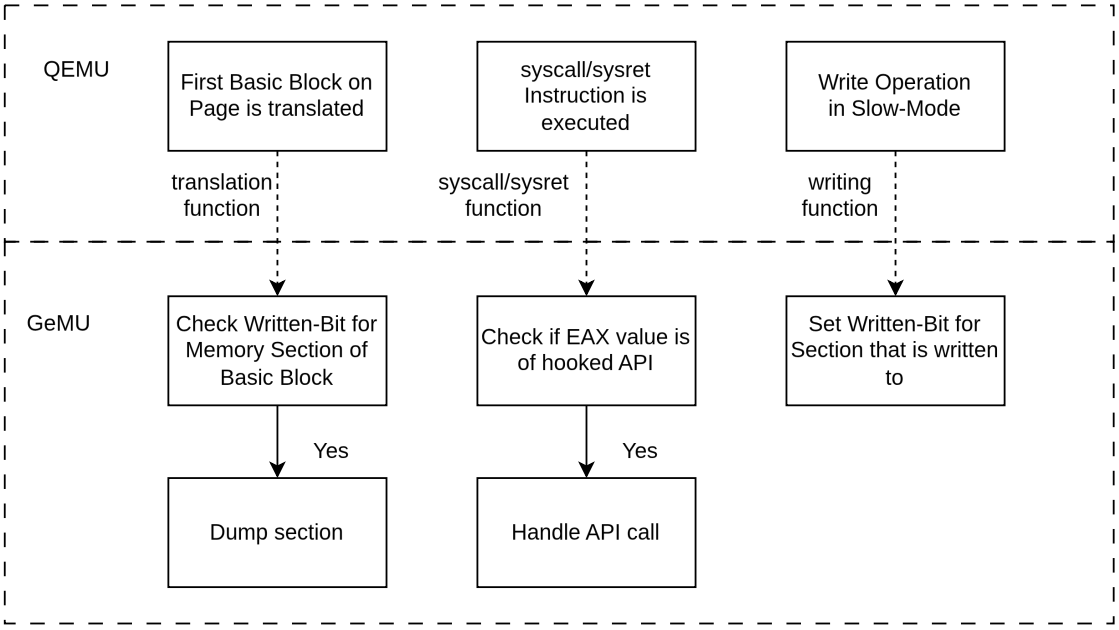

Figure 7.2: This figure illustrates the interactions of QEMU with GeMU. The *translation function* handles extracting the written sections. The *syscall/sysret function* handles the API calls, and the *writing function* marks the memory sections when they are written to.

blocks on that memory page is flushed. This flushing is essential on Intel architecture, where code and data share memory and code can be altered during execution. Flushing the translated basic blocks accounts for the self-modifying properties of code, as cached blocks can no longer be trusted, once data on a page with cached blocks is manipulated.

These two aspects make it unnecessary for GeMU to force the slow path via manipulating the TinyCode. This approach leads to a significant speed-up of the emulation over fine-grained tracing that records every write operation.

7.3.2 Translation Function

Translation occurs whenever QEMU is going to execute code which has not been executed before. This code is transformed into TinyCode, which is then compiled to host code. Therefore, it was decided to define translations as the point of execution where an unpacking could be concluded. Each time the *translation function* is executed, the page of the currently translated basic block is checked for the written bit. However, calling GeMU for every translation is inefficient, as superfluous calls for basic blocks may occur on the same page without any writing operations between subsequent translations. To reduce redundant invocations, QEMU calls GeMU's translation function only on the first translation of any given page.

The unpacking process starts with updating the memory map currently saved by GeMU to ensure that all memory sections are of the correct size. GeMU then identifies the section to be dumped and retrieves all directly adjacent sections, and also the

adjacent sections to these, and so on. The result is a continuous piece of virtual memory which is extracted from memory. This ensures that the resulting dump potentially contains data associated with the unpacked code, including header information, the import (address) table, and auxiliary data such as strings. At the end, all extracted memory sections are unmarked.

7.3.3 Syscall/Sysret Function

The *syscall/sysret function* handles the observation of API calls. It is invoked whenever QEMU executes a syscall or sysret instruction, which on x86_64 processors are used to perform API calls to the kernel, thus allowing to observe API calls. The call to GeMU is inserted into TinyCode during the translation of the basic blocks containing a syscall or sysret instruction. This method is more efficient than checking every basic block whether its address is the start of an API call, as it avoids invoking GeMU at the start of every basic block execution. To identify API calls, the function inspects the value in EAX/RAX, which contains a system call number. This number is compared against a list of all relevant call numbers that must be observed. These call numbers are extracted from the DLLs that handle syscalls.

In case an API call is detected, the in and out parameters are read. Also, the addresses for the out parameters are read so that their contents can be retrieved, if the sysret function is eventually executed.

The observed functions are:

- NtCreateUserProcess
- NtWriteFile
- NtWriteVirtualMemory
- NtMapViewOfSection
- NtOpenProcess
- NtTerminateProcess

The **NtOpenProcess** function is observed to capture the translation from PIDs to handles. This is useful, as most of the functions that manipulate a running process do not take PID as input and instead require a process handle. This is generally achieved by calling the **NtOpenProcess** function. Therefore, whenever this function is called, the mapping of PID to handle is saved to GeMU's internal state.

The **NtWriteFile** is observed to capture write operations on the hard drive. Every time a file is written to the hard drive, it is also extracted. Similarly, the **NtWriteVirtualMemory** function is observed to monitor write operations using the API, including write operations to a target process. The written memory is also extracted and provided as an output. In addition, the PID of the process that is written is added to the set of observed PIDs.

NtMapViewOfSection is used by malware to write to memory inside its own process and then map that memory section to another process to launch the code, as described in

Chapter 5. To reflect this behavior in GeMU, each mapped memory section has a shared written state, which is shared between the different processes to which this section is mapped.

NtCreateUserProcess and **NtTerminateProcess** are directly opposed. Every process that is created by a traced process is added to the list of observed PIDs in GeMU's state. This is an approximation of malware that uses child processes to perform write-then-execute behavior. In addition, the returned handles for the created processes are added to the internal state of GeMU. Conversely, **NtTerminateProcess** removes the PID of the terminated process from the observed PID set. If the set of PIDs is empty after removal, the analysis is terminated prematurely.

7.3.4 Management

The management of GeMU is performed via a Python script, which launches GeMU, the malware, and processes the output. It takes a binary as input and encapsulates it into an ISO file, which is loaded into GeMU's CD drive so that its content can be copied to the guest.

Inside the guest, the Python script uses QEMU's "sendkey" commands to simulate keyboard input into a command line launched with administrator permission. The binary is copied to the desktop and renamed to a predetermined filename, allowing the initial malware process to reliably recognize the malware based on its name. Subsequently, the binary is executed via the same command line. During execution, the script sleeps for a predetermined amount of time. Upon expiration of this timeout, the script shuts down GeMU.

7.3.5 Limitations

As mentioned earlier, GeMU is only compatible with Windows Malware and native binaries. This is a significant limitation, as GeMU is incompatible with programs using managed code, especially malware written in .NET, given its increasing prevalence in recent years. This shortcoming is shown later using a data set comprising samples of Malware Bazaar.

Furthermore, the current approach is not hardened against evasion techniques. In the upcoming study, evasion techniques detecting QEMU ask for user interaction or use stalling code regions [21], which had a detrimental effect on the correctness of the output. A more in-depth look is given in the next section.

7.4 Study

This section describes the study to evaluate GeMU's real-world applicability by assessing both the output correctness and the time required for the samples to unpack. First, the experimental setup is discussed, followed by an overview of the data sets and the methodology used to determine the correctness of the output. The results are presented by first reporting the correctness of the output, followed by the unpacking speed.

7.4.1 Setup

For this study, the samples were run on a Windows 10 64-bit version 1803 machine and on a Windows 7 SP1 64-bit machine, each with the Microsoft Visual C++ Redistributables installed. The guest machines, in which the malware runs, are not connected to the Internet. Four identical instances of these guest machines are used and running on the same host with 32 GB of RAM and an AMD EPYC 7352. The timeout for GeMU is ten minutes.

7.4.2 Data Sets

As stated in Chapter 3.3, the first two data sets are Malpedia [105] and dataset-packed-pe [11]. To further assess the real-world effectiveness of GeMU, an additional data set comprising malware samples submitted to Malware Bazaar in 2024 [12] was analyzed.

7.4.2.1 Malpedia

detected packer	**number of samples**
unknown	1648
UPX	138
PECompact	11
Themida	9
VMProtect	8
PELock	6
Confuser	6
ASPack	6
Smart Assembly	5
ASProtect	4
MPRESS	6
ASProtect	3
Yoda's Protector	2
.NET Reactor	2
(Win)Upack	1
Crypto Obfuscator	1
NsPacK	1
ConfuserEx	1
Enigma	1
tElock	1

Table 7.1: This table presents the outermost layer of packers in Malpedia, as identified by detect-it-easy. For 1,648 samples, no packer could be identified.

Malpedia provides both packed malware samples and their corresponding unpacked or dumped representatives. These representatives are used to generate Yara rules that

detected packer	number of samples
Exe32Pack	119
MEW	107
RLPack	106
MPRESS	105
Packman	105
BeRo	103
Yoda's Protector	102
NsPacK	97
PECompact	97
ASPack	94
UPX	91
(Win)Upack	89
FSG	73
Petite	70
Yoda's Crypter	35
unknown	34
tElock	29

Table 7.2: The packers on the most outer layer for the dataset-packed-pe samples that did not crash during the unpacking by Detect-It-Easy. Curiously enough, for 34 samples no packer could be identified.

prevents false positives with the representatives of other families. In this study, these rules are used to determine the correctness of the results. If at least one dump in the results of a run matches with the Yara rules, then this run is correct.

From the Malpedia data set, all executables are selected for the experiments. Samples that use .NET, lack a YARA rule, or have an unpacked representative on which their Yara rule does not match are removed. This filtering yielded 1,860 remaining samples in the data set. However, a further 35 (1.92%) samples that crashed during analysis were also removed, resulting in a final set of 1,825 samples. The packers on the outer layer of these samples were identified with the Detect-It-Easy packer identification tool [106], selected for its prominence on Github. The results can be found in Table 7.1. For 1,648 (90.3%) samples no packer could be identified, indicating that most packers encountered in the wild are largely unknown. This underlines the necessity of a malware unpacker that does not require prior knowledge of the packer used. The second largest group is UPX with 138 (8,56%) samples.

7.4.2.2 dataset-packed-pe

As mentioned above, the dataset-packed-pe data set comprises benign samples that are packed with off-the-shelf packers. It does not include rules to automatically assess the correctness of unpacking. To address this, signatures were generated from the unpacked programs: each rule has been created from the first 50 bytes at the base of the code as

regular expressions, in which the accessed addresses are replaced by wildcards to address ASLR. If the correct signature matches GeMU's output, the unpacking is considered successful. Although this method may falsely classify partial unpackings as fully unpacked, it is very robust to code reordering by the packer.

From this data set, all samples are compatible with the implementation. However, since this implementation does not support any user interaction, samples requiring a click to launch trial-version packers were excluded, as were any samples that did not provide its unpacked form. Therefore, 1,456 samples remained for the experiments. However, a further 138 (9.48%) samples crashed and were also removed. The packers of the remaining samples can be found in Table 7.2 as they have been detected by Detect-It-Easy [106]. For 34 samples, the tool was unable to identify the packer. According to the data sets labels, these unidentified samples comprise 4 samples packed by BeRo, 14 samples packed by FSG, 13 samples packed by UPX, 2 samples by (Win)Upack, and 1 sample by Packman.

7.4.2.3 Malware Bazaar 2024

To determine GeMU's effectiveness in real-world scenarios, it is tested using a third data set that was not curated to be representative and instead reflects the samples that malware analysts encounter in their daily work. The purpose of this evaluation is to determine whether GeMU is compatible with the malware that is actively of interest to malware analysts.

To this end, malware samples uploaded to Malware Bazaar [12] in 2024 were selected. Malware Bazaar is a platform for sharing malware samples with the wider security community. A key advantage is that the samples are analyzed by the platform and uploaded to other third-party analysis platforms to provide more context information to the samples.

Among these services is the malware unpacking service unpac.me [107]. This makes it possible to reduce the amount of samples uploaded to Malware Bazaar for constructing a filtered data set based on two criteria:

- successful unpacking by unpac.me
- tagging created dumps with Malpedia's Yara rules by unpac.me.

This filtered set, hereafter referred to as MWB2024, can be used to verify the correctness of GeMU-produced dumps. In summary, the samples in this data set are samples uploaded to Malware Bazaar in 2024 that have been successfully unpacked by unpac.me and were tagged with Malpedia's Yara rules.

To build MWB2024, the complete list of samples uploaded to Malware Bazaar was downloaded. From this list, Windows executables were selected. Malware Bazaar offers the analysis reports of these samples as download, so for each of the Windows executables uploaded in 2024, the corresponding analysis reports was downloaded, including the results of unpac.me. Only those samples that both exhibited unpacking behavior and were tagged with Malpedia's Yara rules were retained.

After downloading these samples, samples that were identified as .NET and where Malpedia's Yara rule matched on the packed sample have been removed. This resulted in the removal of 6,609 .NET samples, leaving 4,763 samples in the data set for analysis. 1,946 samples crashed during their execution. While this is a significant portion of the samples, 1,761 of these samples belong to the *stealc* family and an additional 134 samples to the *formbook* family. The remaining 51 samples span eight different families. The full list of families of crashed samples can be found in Table 7.3. A closer analysis of the 1,946 samples that crashed during execution was performed using Detect-It-Easy [106] and the results are displayed in Table 7.4. The largest group of samples is packed with Themida, a commercial packer that employs virtual machine emulation, which is out of scope for GeMU, and various evasion techniques to protect its payload.

Table 7.5 lists the identified packers in the samples that did not crash. Notably, Detect-It-Easy failed to detect the packer for 2560 (90.88%) samples. This further emphasizes the need for a generic malware unpacker that does not rely on prior knowledge of the packer type.

After removing the samples that crashed, 2,817 samples remained in the MWB2024 data set.

Family	**Count**
stealc	1761
formbook	134
lumma	34
neshta	8
amadey	2
lockbit	2
smokeloader	2
ave_maria	1
juicy_potato	1
vidar	1

Table 7.3: Distribution of samples across families for the crashed samples in MWB2024. The majority of samples belongs to the *stealc* family.

detected packer	**number of samples**
Themida	1734
unknown	202
UPX	10

Table 7.4: Identified packer by Detect-It-Easy for the MWB2024 samples that crashed during execution. 89.1% of these samples use Themida [108].

detected packer	number of samples
unknown	2560
UPX	145
Themida	96
MPRESS	6
VMProtect	3
ASPack	3
PECompact	1
MoleBox	1

Table 7.5: Identified packers of the samples that did not crash in MWB2024 were not detected by Detect-It-Easy. For 90,88% of samples, no packer could be identified.

7.4.3 Correctness

This section presents the correctness of the runs. To this end, each of the data sets is analyzed by themselves.

	Correct	Incorrect	Total
Malpedia	1,715 (93.97%)	110 (6.03%)	1,825
dataset-packed-pe	1,336 (91.76%)	120 (8.24%)	1,456
MWB2024	2,503 (88.85%)	314 (11.15%)	2,817

Table 7.6: This table depicts the correctness of the study divided across the three data sets. 93.97% of the samples in Malpedia, 91.76% in dataset-packed-pe, and 88.85% in MWB2024 were unpacked successfully.

7.4.3.1 Malpedia

First up is the analysis of the results of Malpedia. The correctness of the results is shown in Table 7.6. GeMU successfully unpacks 93.97% of the samples, demonstrating both the practical applicability of the methodology and implementation and its robustness against malware in the wild.

However, samples that have not been unpacked correctly may point to oversights in the derived requirements. Therefore, these 110 samples have been manually reversed to determine why the unpacking was unsuccessful. 98 of these samples are packed with an unknown packer, 6 are packed with UPX, 3 with VMProtect, 2 with ASPack, and 1 with PECompact.

For unsuccessful runs on Malpedia:

- 34 samples were successful after a manual rerun.
- 15 samples were correctly unpacked, but the Yara rule did not match because it is too narrow. The rules automatically generated in Malpedia demand that the dump is below a certain size. After removing this limitation from the rules, the Yara rules matched.

- 9 samples are incompatible with details of the implementation, such as the number of dumps that can be generated.
- 4 samples were correctly unpacked when allowed a longer timeout.
- 2 samples crashed during execution
- 46 samples did not work due to evasion techniques, which are as follows:
  - 34 detect QEMU
  - 5 use stalling code regions
  - 3 require user interaction
  - 3 reboot the system
  - 1 checks for network connection

The results indicate that evasion techniques have the most detrimental effect on GeMU. Importantly, none of the evasion techniques directly targets the GeMU unpacking routine. Consequently, these failures reflect implementation limitations rather than methodological shortcomings.

7.4.3.2 dataset-packed-pe

Next up is the analysis of the results of dataset-packed-pe run. The results are shown in Table 7.6. 91.76% of samples have been correctly unpacked, while 8.24% of the samples failed.

Manual analysis of the 120 incorrect samples revealed that 6 samples were correctly unpacked but could not be identified by the rules. The remaining 114 samples crashed due to missing DLLs or the binary being malformed.

87 of all 120 incorrect samples belong to the Exe32Pack packer, and 10 belong to tElock. The remaining 23 samples are spread over 12 other packers, each contributing between one to four samples.

The results demonstrate that GeMU works on most samples packed with off-the-shelf packers.

7.4.3.3 MWB2024

In this section, the results of the MWB2024 sandboxing run are analyzed. The raw results can be seen in Table 7.6. For MWB2024, 2,503 samples (88.85%) of samples have been correctly unpacked. These samples span 76 families, confirming a broad compatibility consistent with the Malpedia findings. There are 798 *Formbook* samples, 580 *Smokeloader* samples, and 338 *Stealc* samples as the three biggest families in this group. The top 15 families are listed in Table 7.7.

However, the failure rate of samples for MWB2024 is 11.15%, higher than in the other data sets. Failures occur in 25 different families, led by 159 *Remcos* samples, 70 *Formbook* samples, and 25 *Lockbit* samples. Detect-It-Easy identified 275 samples using

Family	Count
formbook	798
smokeloader	580
stealc	338
unidentified_045	123
stop	107
gcleaner	59
lumma	49
remcos	49
tofsee	49
flawedammyy	47
amadey	45
emotet	39
qtbot	25
sality	19
dbatloader	12

Table 7.7: Distribution of samples across the top 15 families that created a correct result in the MWB2024 data set.

an unknown packer, 38 using UPX, and one using PECompact. Further analysis of the 314 failed runs revealed that 198 samples are packed with an AutoIT-based packer called CypherIT using evasion techniques to thwart the analysis in sandboxes [109].

Of the remaining 116 samples, it was found:

- 27 samples were already unpacked but have not been tagged as such.
- 24 samples were successful after a manual rerun.
- 13 samples crashed during manual execution.
- 4 samples matched a benign file that the malware sometimes uses for injections.
- 2 samples needed a longer timeout.
- 2 samples are out of scope because they are based on Visual Basic.
- 1 sample had the same problem with the size restriction of the Yara rule as explained above.
- The remaining 43 samples used the following evasion techniques:
  - 35 detect QEMU
  - 4 require user interaction
  - 1 checks for internet connection
  - 1 sample requires a command line parameter
  - 1 uses virtualization
  - 1 sample uses stalling code

Family	Count
remcos	159
formbook	70
lockbit	25
stealc	20
phobos	8
karius	4
azorult	3
lumma	3
amadey	2
hawkeye_keylogger	2
nymaim	2
pony	2
socks5_systemz	2
blacksuit	1
cybergate	1
darkpulsar	1
dbatloader	1
gandcrab	1
oski	1
ozone	1
sinowal	1
smokeloader	1
stop	1
unidentified_045	1
webmonitor	1

Table 7.8: Distribution of samples across families that were incorrect in MWB2024. The biggest family is *Remcos*.

The results of MWB2024 are very similar to those of Malpedia. The most detrimental effect on the failure rate in this data set can be traced back to the samples packed with the AutoIT packer. Removing these samples from the data set yields a failure rate of 4.1%, aligning more closely to the failure rates in the other two data sets.

However, construction of MWB2024 has shown that a significant amount of contemporary malware uses the .NET framework indicating a need for an unpacker capable of handling these types of samples.

7.4.4 Speed

Unpacking speed denotes the time elapsed between the launch of the malware and the appearance of the first unpacked dump. Therefore, the unpacking speed is only measured for correct runs. Table 7.9 presents these durations in seconds.

For Malpedia, the fastest unpacking occurred in under one second. The average time is about 33 seconds and the median close to 10 seconds, with the slowest being 10 minutes. By contrast, the dataset-packed-pe samples unpacked significantly faster and much more uniform: the lower quartile, median, average, and upper quartile fell below 4 seconds. The maximum is also much lower, with only 118.27 seconds.

The samples in MWB2024 exhibited the greatest variability. The median unpacking time was about 15 seconds, the average exceeded 50 seconds, and the upper quartile reached 86 seconds: almost three times slower than Malpedia and 40 times higher than dataset-packed-pe. These observations confirm that packing with off-the-shelf packers is much more uniform than what is encountered in real-world malware. This further highlights the difference between benign samples packed with the off-the-shelf packers and real-world malware, which has already been observed in the studies of the previous chapters.

	Malpedia	dataset-packed-pe	MWB2024
Min	0.21 seconds	0.85 seconds	0.55 seconds
25%	3.73 seconds	2.01 seconds	3.53 seconds
median	9.68 seconds	2.45 seconds	15.31 seconds
Average	33.23 seconds	3.27 seconds	51.33 seconds
75%	32.03 seconds	2.84 seconds	86.08 seconds
max	594.89 seconds	118.27 seconds	444.74 seconds

Table 7.9: This table presents the unpacking speed in seconds, defined as the elapsed time from the moment the initial process is first traced until the first correct dump is observed. The unpacking speed averages 33.23 seconds for the Malpedia samples, 3.27 seconds for the dataset-packed-pe samples, and 51.33 seconds for MWB2024 samples.

7.5 Discussion

GeMU successfully unpacks the vast majority of samples across all three data sets. Its high success rate in Malpedia demonstrates compatibility with most historical and contemporary families. Results from dataset-packed-pe confirm that GeMU handles the majority of off-the-shelf packers.

The third data set, MWB2024, highlights that GeMU also performs effectively on real-world malware samples. MWB2024 exhibits the highest diversity in families among successfully unpacked samples compared with those that failed. However, constructing the MWB2024 data set required excluding a significant amount of .NET-based samples. Adding support for .NET based samples would improve GeMU's real-world applicability.

Unpacking speed is another concern. With an average unpacking speed of thirty to fifty seconds, some use cases may find this duration excessive. Testing alternative implementations, for example using virtual machines or instrumentation, could reduce unpacking time.

Another concern stems from using Malpedia and dataset-packed-pe for both deriving requirements and evaluating the tool. This practice risks overfitting, with requirements tailored too specific to these two data sets that they may fail the transfer to other data sets. However, as especially Malpedia is still very actively maintained, with new samples added constantly, the data sets provide a near complete picture of the current malicious unpacking landscape. Omitting them from the evaluation would introduce significant blind spots.

To guard against overfitting, the MWB2024 data set served as an independent validation corpus. Its results closely align with those from Malpedia and dataset-packed-pe, indicating a low overfitting effect and that GeMU is highly compatible with real-world malware. Nevertheless, ongoing testing with new samples will help ensure and verify GeMU's applicability with novel unpacking techniques.

In addition, concerns also arise in assessing correctness. In the study, the output was verified using Yara rules and regular expressions, which match only the specific unpacking layer containing the correct data. When unpacking unfolds across multiple

layers, it cannot be assessed whether the entire body of desired data is unpacked or just the part on which the rule coincidentally matches. Requirement **R7**, which demands extraction of every layer, cannot be verified with these aforementioned rules alone. To address this issue, a data set should be constructed that includes all possible unpacking methodologies and provides the ground truth to assess the correctness of every unpacking layer.

7.6 Summary

This chapter aims to construct a generic malware unpacker that meets all established requirements and to demonstrate its applicability in real-world scenarios. To this end, the write-the-execute heuristic has been combined with a component to detect code propagations to build a new unpacker called GeMU.

First, the methodology of GeMU was presented, which can be adopted in different runtime systems. It was highlighted how the different aspects of the methodology satisfy the requirements for a generic malware unpacker, confirming that these requirements are built into the methodology.

In the next section, the implementation of this methodology was presented.

Finally, a study was conducted to demonstrate the correctness of the approach. Three data sets have been selected to assess its applicability in real-world scenarios. GeMU achieved unpacking coverage above 90% for Malpedia and dataset-packed-pe, and 88.85% for MWB2024. Unpacking failed primarily due to missing hardening against evasion techniques and incompatibility with implementation details.

Main Takeaways of this chapter:

This chapter introduces GeMU, a new generic unpacking methodology along with its proof-of-concept implementation. GeMU meets all requirements derived in the previous chapter and successfully unpacks almost all samples from the chosen data sets. The few remaining samples that failed stem from evasion techniques.

8 Conclusion

This chapter concludes this dissertation by first evaluating its cumulative impact, then addressing its limitations, including a brief discussion of the applicability to other platforms, and finally outlining an outlook for possible future work.

8.1 Overall Impact

This dissertation makes two principal contributions to the field of generic malware unpacking:

1. The theoretical model for building generic malware unpackers for any kind of platform is established.
2. An in-depth analysis of generic malware unpacking is provided by applying this theoretical base to native Windows malware, the most prevalent group of malware.

Therefore, this dissertation delivers both foundational theoretical advancements in generic malware unpacking and actionable practical solutions.

The models for displaying malware unpacking within a single process and across multiple processes expose the unpacking behavior of any kind of software. When applied to a representative data set containing different malware, these models reveal a set of different unpacking behaviors, which, in turn, informs a set of requirements for a generic malware unpacker that is compatible with all samples included in the data set.

The practical contribution of this dissertation begins with a critical review of previous generic malware unpackers, identifying ambiguities in the definition of *unpacking* and their shortcomings. To address these, a new definition of *unpacking* is provided and requirements for a generic malware unpacker are generated. These requirements are created by conducting two studies using two representative data sets of real-world malware and off-the-shelf packers from the ecosystem of native Windows malware. The studies are practical applications of the unpacking models. One study covers the unpacking behavior within one single process and the other study covers the unpacking behavior across multiple processes. Using the results of these studies, the requirements for a generic malware unpacker for native Windows malware are derived and then used to evaluate existing generic malware unpackers.

Although the evaluation revealed that no existing unpacker covered all requirements, the most promising approaches were extended to fulfill the requirements, leading to the creation of GeMU, the Generic Malware Unpacker. The implementation and subsequent evaluation of GeMU show that most of the samples are correctly unpacked, while the few unpacking failures are attributed to evasion techniques unrelated to the unpacking. This

confirms that GeMU is capable of unpacking all the samples from which the requirements have been derived.

Although the practical results described in this dissertation can only be a snapshot of the current malware landscape, and the arms race between malware analysts and malware developers dictates that new unpacking techniques will be found. However, the theoretical blueprint presented here provides a durable foundation to adjust the requirements for a generic malware unpacker to the ever-changing state of the art. Thus, the theoretical contributions of this dissertation will most likely never become obsolete.

8.2 Impact of Research Questions

This section discusses the impact of this thesis on the field of executable packing, malware analysis, and reverse engineering along the different research questions. Note that the research questions mostly refer to the practical contributions of this dissertation, while the theoretical contributions stem from the real-world need for a generic malware unpacker to deal with the overwhelming amount of Windows malware.

Since Research Questions 1.1 to 1.7 are required to answer Research Question 1, their individual impacts are also stated here.

Research Question 1:
What are the requirements for a malware unpacker that it must meet to be generic?

Before Dissertation: The detailed requirements for a generic malware unpacker were unknown. Researchers relied on preconceived notions generated from their own personal experience in malware analysis to build unpackers or evaluate others. Furthermore, researchers used data sets of widely varying quality for their evaluation that do not reflect the full spectrum of unpacking behavior encountered in the wild. As a result, many unpackers fail to unpack a significant portion of the malware encountered in the real world. This shortcoming has perpetuated the so-called packer problem even two decades after the first generic malware unpacker has been proposed.

After Dissertation: With the standardization of these requirements, researchers can use them to evaluate their own generic malware unpackers against these standardized requirements instead of relying on a practical evaluation.

Research Question 1.1:
What are the methods/techniques used by malware to obtain memory to write unpacked code?

Before Dissertation: The memory used by malware to write its unpacked code was largely unknown. In previous unpacking methodologies, some unpackers assumed that only the image sections were used [52, 46].

After Dissertation: A detailed overview is provided, describing the types of memory that malware uses and how spread these types are. This knowledge will aid future researchers in detecting unpacked malware in the memory.

Research Question 1.2:
What are the methods/techniques used by malware to write code?

Before Dissertation: The methods and techniques used by malware were largely unknown. Consequently, researchers had to rely on preconceived notions on how to trace the execution and write operations of malware.

After Dissertation: Due to the results presented in this dissertation, researchers can now rely on a detailed overview of the different options malware uses to write its code. This will aid future research in automatic unpacking to build analysis systems for each of the different write operations.

Research Question 1.3:
Is unpacked code overwritten during the execution of malware?

Before Dissertation: It was considered sufficient for generic malware unpackers to only extract the unpacking layer containing the malware. Any extraction of all unpacking stages occurred incidentally rather than by deliberate design.

After Dissertation: The results demonstrate that a significant portion of malware overwrites unpacked code during execution. Such runtime modification can obscure information from analysts. Therefore, this dissertation highlights the importance of extracting the intermediate unpacking stages to provide a complete picture of the executed code.

Research Question 1.4:
How many layers of packing are used?

Before Dissertation: Ugarte-Pedrero et al. [2] have analyzed the amount of unpacking layers used in malware and found that malware utilizes multiple unpacking layers.

After Dissertation: These findings have been reproduced, reinforcing the need for a generic malware unpacker to deal with multiple unpacking stages.

Research Question 1.5:
Is it possible to differentiate between malware and packer code?

Before Dissertation: The malware and packer dichotomy was a strong assumption that permeated the field of generic malware unpacking. Although Ugarte-Pedrero et al. [2] mentioned in their packer types that the distinction between malware and packer might not always be clear, they still searched for a distinction between malware and packer. This is reflected in a significant number of existing generic malware unpackers sharing this assumption and therefore lacking in generality.

After Dissertation: The dichotomy of malware and packer is no longer the state of the art. The results in this dissertation demonstrate that, for a significant number of samples, determining the difference between malware and packer is impossible. Therefore, a generic malware unpacker must operate without assuming a clear distinction between malware and packer.

Research Question 1.6:
What methodologies are used by malware to achieve multi-process write-then-executes?

Before Dissertation: The previous research by Barabosch et al. [110] focused on process injections but did not address child-process creation or the notion of multi-process write-then-executes. Therefore, no empirically grounded framework existed to build an unpacker.

After Dissertation: This dissertation provides the knowledge needed to build a generic malware unpacker capable of detecting multi-process write-then-executes. Also, the findings of this dissertation have shown that code propagations can be observed in a significant portion of malware. These insights can also be leveraged in other dynamic analysis approaches that rely on following malware through multiple processes.

Research Question 1.7:
What functions are used to implement multi-process write-then-executes?

Before Dissertation: Ugarte-Pedrero et al. [2] and Barabosch et al. [110] have provided overviews of the functions used in Windows Malware to spread its code to other processes.

After Dissertation: The findings of Barabosch et al. are improved, since they did not monitor the execution of malicious binaries using process creation. Compared to Ugarte-Pedrero et al., the findings in this dissertation provide a slightly different picture, as many more process creations were conducted in this dissertation's studies. Therefore, researchers looking to create a generic malware unpacker for Microsoft Windows can apply the findings of this dissertation to gain a strong understanding of the functions for multi-process write-then-executes.

Research Question 2:
Do the previously proposed generic malware unpacker meet the requirements of a generic malware unpacker?

Before Dissertation: The effectiveness of previously proposed unpackers in real-world scenarios was unknown. Withholding their source code prevented an extended third-party evaluation and obfuscated their applicability.

After Dissertation: None of the previously proposed unpackers are capable of thwarting all possible packer behavior, since they do not fulfill all requirements for a generic malware unpacker. This fact explains why none of these unpackers solved the packer problem.

Research Question 3:
Is it possible to develop a malware unpacker that can meet all the requirements of a generic unpacker and works perfectly on the data set used to create the requirements?

Before Dissertation: It was unknown whether it was possible to create a generic malware unpacker capable of meeting all requirements and therefore handle a wide range of packer behaviors.

After Dissertation: It is possible to construct such an unpacker. The presented methodology for a generic malware unpacker can be adopted in various dynamic analysis scenarios. As a result, this dissertation ultimately democratizes the creation of generic malware unpackers, empowering analysts who are unable or unwilling to use commercial platforms such as UnpacMe [107].

8.3 Limitations

The primary limitation that plagues the implementations and evaluations in this thesis lies in the ignorance towards evasion techniques. Malware authors commonly use evasion techniques to thwart analysis, thereby reducing the reliability of the results of this thesis, as has been shown in Chapter 7.4.

Another limiting factor stems from the exclusive focus on Windows Malware for x86 and x64. All implementations and studies conducted in this dissertation target this platform, leaving managed code, especially .NET, unaddressed despite their widespread use in contemporary malware. This limitation extends to packers that convert the payload into managed code. Furthermore, the blueprint presented in this dissertation is based on the write-then-execute heuristic. While this heuristic is very broad and aligns with the Intel's Von Neumann Architecture, which mixes code and data in memory and therefore allows self-modifying code, it is not applicable on platforms which enforce a strict separation of code and data. Unpackers targeting such platforms must instead adopt new heuristics capable of detecting unpackings under these constraints. Nonetheless, the model to measure unpacking behavior within a single process is agnostic towards the unpacking heuristic.

8.4 Future Work

Future research can advance both the theoretical framework for creating a generic malware unpacker and the practical application of the proposed blueprint.

Other platforms: As of now, the current blueprint has only been applied to Windows malware using the write-then-execute heuristic to spot unpacking. Applying this methodology to other malware ecosystems would test its genericity.

One highly relevant platform is Android, given its huge user base and the rapidly growing number of malware for Android. Therefore, applying the blueprint to this platform would yield a significant benefit to malware analysts.

Managed code also remains unaddressed. Especially .NET, which is commonly encountered as seen in Chapter 7.4.2.3 in the MWB2024 data set, is a blind spot for the implementations presented in this dissertation, yet the current implementation of the write-then-execute heuristic is inherently incompatible with managed code like .NET. Extending the unpacker compatibility to malware written using managed code would greatly improve the results in this dissertation.

Extending the data sets: The practical results of this dissertation could be improved by incorporating additional data sets beyond Malpedia [10] and dataset-packed-pe [11]. Expanding the data set may expose more unpacking techniques that have not been covered yet. Another avenue is the creation of a data set which provides a high diversity in packing techniques and the means to confirm that the unpacking has worked. This has already been identified as future work in [2].

Evasion techniques in Windows Malware: As seen in Chapter 7, GeMU's effectiveness can be improved by addressing evasion techniques that are commonly used to thwart malware analysis. This aspect of malware has not been addressed in any study or in the implementation of GeMU. A dedicated study on evasion techniques used during the

unpacking phases is highly desirable, as it would greatly improve the results of this paper and help researchers decide which evasion techniques are detrimental to their unpackers.

8.5 AI-Tools Disclosure

The following tools have been used to fix language errors and improve wording:

- **Writefull For Overleaf**
- **Languagetool**
- **Overleaf's spell checker**

Bibliography

[1] AVTest, "Av-atlas." `https://portal.av-atlas.org/malware/statistics`. Accessed: 2025-05-15.

[2] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *2015 IEEE Symposium on Security and Privacy*, pp. 659–673, IEEE, 2015.

[3] F. Guo, P. Ferrie, and T.-C. Chiueh, "A study of the packer problem and its solutions," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 98–115, Springer, 2008.

[4] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith, "Malware normalization," tech. rep., University of Wisconsin-Madison Department of Computer Sciences, 2005.

[5] T. Jenke, E. Padilla, and L. Bruckschen, "Towards generic malware unpacking: A comprehensive study on the unpacking behavior of malicious run-time packers," in *Nordic Conference on Secure IT Systems*, pp. 245–262, Springer, 2023.

[6] T. Jenke, S. Liessem, E. Padilla, and L. Bruckschen, "A measurement study on interprocess code propagation of malicious software," in *International Conference on Digital Forensics and Cyber Crime*, pp. 264–282, Springer, 2023.

[7] T. Jenke, M. Ufer, M. Blatt, L. Kohler, E. Padilla, and L. Bruckschen, "Democratizing generic malware unpacking," in *2025 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pp. 30–38, IEEE, 2025.

[8] C. Rossow, C. J. Dietrich, C. Kreibich, C. Grier, V. Paxson, N. Pohlmann, H. Bos, and M. van Steen, " Prudent Practices for Designing Malware Experiments: Status Quo and Outlook ," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, 2012.

[9] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with panda," in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, pp. 1–11, 2015.

[10] D. Plohmann, M. Clauss, S. Enders, and E. Padilla, "Malpedia: a collaborative effort to inventorize the malware landscape," in *Proceedings of the Botconf*, 2017.

[11] "GitHub - packing-box/dataset-packed-pe: Dataset of packed PE samples — github.com." `https://github.com/packing-box/dataset-packed-pe`. Accessed: 2025-05-15.

[12] "Malware Bazaar," 2023. `https://bazaar.abuse.ch/`.

[13] E. Eilam, *Reversing: secrets of reverse engineering*. John Wiley & Sons, 2011.

[14] K. Coogan, S. Debray, T. Kaochar, and G. Townsend, "Automatic static unpacking of malware binaries," in *2009 16th Working Conference on Reverse Engineering*, pp. 167–176, IEEE, 2009.

[15] D. Plohmann, S. Eschweiler, and E. Gerhards-Padilla, "Patterns of a cooperative malware analysis workflow," in *2013 5th International Conference on Cyber Conflict (CYCON 2013)*, pp. 1–18, IEEE, 2013.

[16] T. Muralidharan, A. Cohen, N. Gerson, and N. Nissim, "File packing from the malware perspective: Techniques, analysis approaches, and directions for enhancements," *ACM Computing Surveys*, vol. 55, no. 5, pp. 1–45, 2022.

[17] Oracle, "Oracle virtualbox." `https://www.virtualbox.org/`. Accessed: 2025-05-15.

[18] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX annual technical conference, FREENIX Track*, vol. 41, p. 46, California, USA, 2005.

[19] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *2011 IEEE symposium on security and privacy*, pp. 297–312, IEEE, 2011.

[20] N. Galloro, M. Polino, M. Carminati, A. Continella, and S. Zanero, "A systematical and longitudinal study of evasive behaviors in windows malware," *Computers & Security*, vol. 113, p. 102550, 2022.

[21] C. Kolbitsch, E. Kirda, and C. Kruegel, "The power of procrastination: detection and mitigation of execution-stalling malicious code," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 285–296, 2011.

[22] various, "PE format," 2021. MSDN Article: `https://docs.microsoft.com/en-us/windows/win32/debug/pe-format` Accessed: 2025-05-15.

[23] M. Oberhumer, L. Molnár, and J. F. Reiser, "Upx: the ultimate packer for executables," 2004.

[24] various, "Virtual address spaces," 2021. MSDN Article: `https://learn.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/virtual-address-spaces` Accessed: 2025-05-15.

[25] Microsoft, "Virtualprotect function." `https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotect`. Accessed: 2025-05-15.

[26] Microsoft, "Virtualalloc function." `https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc`. Accessed: 2025-05-15.

[27] Microsoft, "Teb structure (winternl.h)." `https://learn.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-teb`. Accessed: 2025-05-15.

[28] Microsoft, "Peb structure (winternl.h)." `https://learn.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb`. Accessed: 2025-05-15.

[29] Microsoft, "memcpy, wmemcpy function." `https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/memcpy-wmemcpy?view=msvc-170`. Accessed: 2025-05-15.

[30] Microsoft, "Writeprocessmemory function." `https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory`. Accessed: 2025-05-15.

[31] Karl-Bridge-Microsoft, "PE Format - Win32 apps." https://docs.microsoft.com/en-us/windows/win32/debug/pe-format. Accessed: 2025-05-15.

[32] I. Guilfanov, "IDA Pro," May 1990. Company Website: `https://hex-rays.com/ida-pro/` Accessed: 2025-05-15.

[33] National Security Agency, "The Ghidra Software Reverse Engineering suite," 2019. Project Website: `https://ghidra-sre.org/` Accessed: 2025-05-15.

[34] B. Ninja, "Binary ninja. 2024," *URL: https://binary.ninja/*, vol. 24, 2024.

[35] N. M. Hai, M. Ogawa, and Q. T. Tho, "Packer identification based on metadata signature," in *Proceedings of the 7th Software Security, Protection, and Reverse Engineering/Software Security and Protection Workshop*, pp. 1–11, 2017.

[36] P. Royal, M. Halpin, and D. Dagon, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware, dec. 2006," *ACSAC, pp289-300*.

[37] J. Calvet, F. L. Lévesque, J. M. Fernandez, J. Marion, E. Traourouder, and F. Menet, "Waveatlas: surfing through the landscape of current malware packers," *Virus*, 2015.

[38] M. Krejdl, "Malware: It's all in the gift-wrapping," 2010. Website: `https://blog.avast.com/2010/12/20/malware-giftwrapping-services/` Accessed: 2025-05-15.

[39] F-Secure, "Packed:W32/MysticCompressor.gen!A," 2021. Website: `https://www.f-secure.com/v-descs/packed-w32-mysticcompressor-gen!a.shtml` Accessed: 2025-05-15.

[40] B. Zdrnja, "Opachki, from (and to) Russia with love," 2009. Website: `https://isc.sans.edu/diary/Opachki+from+and+to+Russia+with+love/7519` Accessed: 2025-05-15.

[41] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *Proceedings of the 2007 ACM workshop on Recurring malcode*, pp. 46–53, ACM, 2007.

[42] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM conference on Computer and communications security*, pp. 51–62, ACM, 2008.

[43] S. Cesare, Y. Xiang, and W. Zhou, "Malwise—an effective and efficient classification system for packed and polymorphic malware," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1193–1206, 2012.

[44] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "Codisasm: medium scale concatic disassembly of self-modifying binaries with overlapping instructions," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 745–756, ACM, 2015.

[45] S. D'ALESSIO and S. MARIANI, "Pindemonium: a dbi-based generic unpacker for windows executables," 2016.

[46] L. Bohne and T. Holz, "Pandora's bochs: Automated malware unpacking," *Master's thesis, RWTH Aachen University*, 2008.

[47] L. Martignoni, M. Christodorescu, and S. Jha, "Omniunpack: Fast, generic, and safe unpacking of malware," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pp. 431–441, IEEE, 2007.

[48] L. Sun, "Hump-and-dump: Efficient generic unpacking using an ordered address execution histogram," in *2nd International Computer Anti-Virus Researchers Organization (CARO) Workshop, 2008*, 2008.

[49] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A framework for enabling static malware analysis," in *European Symposium on Research in Computer Security*, pp. 481–500, Springer, 2008.

[50] R. Isawa, D. Inoue, and K. Nakao, "An original entry point detection method with candidate-sorting for more effective generic unpacking," *IEICE TRANSACTIONS on Information and Systems*, vol. 98, no. 4, pp. 883–893, 2015.

[51] W. Guizani, J.-Y. Marion, and D. Reynaud-Plantey, "Server-side dynamic code analysis," in *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*, pp. 55–62, IEEE, 2009.

[52] G. Jeong, E. Choo, J. Lee, M. Bat-Erdene, and H. Lee, "Generic unpacking using entropy analysis," in *2010 5th International Conference on Malicious and Unwanted Software*, pp. 98–105, IEEE, 2010.

[53] B. Cheng, J. Ming, J. Fu, G. Peng, T. Chen, X. Zhang, and J.-Y. Marion, "Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 395–411, 2018.

[54] C. Lim, K. Ramli, Y. S. Kotualubun, *et al.*, "Mal-flux: Rendering hidden code of packed binary executable," *Digital Investigation*, vol. 28, pp. 83–95, 2019.

[55] T. Jenke, D. Plohmann, and E. Padilla, "Roamer: The robust automated malware unpacker," in *14th International Conference on Malicious and Unwanted Software (MALWARE), Nantucket, MA, USA, 2019*, pp. 67–74, 2019.

[56] E. Alkhateeb, A. Ghorbani, and A. Habibi Lashkari, "A survey on run-time packers and mitigation techniques," *International Journal of Information Security*, pp. 1–27, 2023.

[57] V. M. Alvarez, "yara: The pattern matching swiss knife for malware researchers (and everyone else)." `http://virustotal.github.io/yara/`. Accessed: 2025-05-15.

[58] D. Plohmann, "Malpedia." `https://malpedia.caad.fkie.fraunhofer.de/stats/yara`. Accessed: 2025-05-15.

[59] B. Cheng and P. Li, "Bareunpack: Generic unpacking on the bare-metal operating system," *IEICE TRANSACTIONS on Information and Systems*, vol. 101, no. 12, pp. 3083–3091, 2018.

[60] T. Barabosch and E. Gerhards-Padilla, "Host-based code injection attacks: A popular technique used by malware," in *2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*, pp. 8–17, IEEE, 2014.

[61] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system," in *Proceedings of the 30th Annual Computer Security Applications Conference*, pp. 386–395, 2014.

[62] B. D. Payne, "Simplifying virtual machine introspection using libvmi.," tech. rep., Sandia National Laboratories (SNL), Albuquerque, NM, and Livermore, CA ..., 2012.

[63] "GitHub - airbus-seclab/qemu_blog: A series of posts about QEMU internals: — github.com." `https://github.com/airbus-seclab/qemu_blog`. Accessed 2025-05-15.

[64] H. Yin and D. Song, "Temu: Binary code analysis via whole-system layered annotative execution," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-3*, 2010.

[65] L. Craig, A. Fasano, T. Ballo, T. Leek, B. Dolan-Gavitt, and W. Robertson, "Pypanda: Taming the pandamonium of whole system dynamic analysis," in *NDSS Binary Analysis Research Workshop*, 2021.

[66] A. Henderson, L. K. Yan, X. Hu, A. Prakash, H. Yin, and S. McCamant, "Decaf: A platform-neutral whole-system dynamic binary analysis platform," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 164–184, 2016.

[67] A. Davanian, Z. Qi, Y. Qu, and H. Yin, "{DECAF++}: Elastic {Whole-System} dynamic taint analysis," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pp. 31–45, 2019.

[68] P. Dovgalyuk, N. Fursova, I. Vasiliev, and V. Makarov, "Qemu-based framework for non-intrusive virtual machine instrumentation and introspection," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 944–948, 2017.

[69] "GitHub - panda-re/panda: Platform for Architecture-Neutral Dynamic Analysis — github.com." `https://github.com/panda-re/panda`. Accessed 2025-05-15.

[70] D. Plohmann, S. Enders, and E. Padilla, "Apiscout: Robust windows api usage recovery for malware characterization and similarity analysis," *The Journal on Cybercrime & Digital Investigations*, vol. 4, 2018.

[71] J. Richter, *Applied Microsoft. NET framework programming*, vol. 1. Microsoft Press Redmond, 2002.

[72] Microsoft, "Zwmapviewofsection function." `https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-zwmapviewofsection`. Accessed: 2025-05-15.

[73] Microsoft, "Rtlallocateheap function." `https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-rtlallocateheap`. Accessed: 2025-05-15.

[74] Microsoft, "Globalalloc function." `https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-globalalloc`. Accessed: 2025-05-15.

[75] Microsoft, "Heapcreate function." `https://learn.microsoft.com/en-us/windows/win32/api/heapapi/nf-heapapi-heapcreate`. Accessed: 2025-05-15.

[76] Microsoft, "Localalloc function." `https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-localalloc`. Accessed: 2025-05-15.

[77] Microsoft, "malloc function." `https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/malloc?view=msvc-170`. Accessed: 2025-05-15.

[78] D. Benton, "The beginners guide to codecaves." `https://www.codeproject.com/Articles/20240/The-Beginners-Guide-to-Codecaves`. Accessed: 2025-05-15.

[79] Microsoft, "Ntallocatevirtualmemory function." https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-ntallocatevirtualmemory. Accessed: 2025-05-15.

[80] Microsoft, "Virtualallocex function." https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocex. Accessed: 2025-05-15.

[81] Microsoft, "Loadlibrarya function." https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya. Accessed: 2025-05-15.

[82] Microsoft, "memmove, wmemmove." https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/memmove-wmemmove?view=msvc-170. Accessed: 2025-05-15.

[83] Microsoft, "Rtlmovememory function." https://learn.microsoft.com/en-us/windows/win32/devnotes/rtlmovememory. Accessed: 2025-05-15.

[84] Microsoft, "Createprocess function." https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-rtldecompressbuffer. Accessed: 2025-05-15.

[85] T. Barabosch, S. Eschweiler, and E. Gerhards-Padilla, "Bee master: Detecting host-based code injection attacks," in *Proceedings of the 11th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), London, UK*, 2014.

[86] T. Barabosch, N. Bergmann, A. Dombeck, and E. Padilla, "Quincy: Detecting host-based code injection attacks in memory dumps," in *Proceedings of the 14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), Bonn, Germany*, 2017.

[87] Z. Gittins and M. Soltys, "Malware persistence mechanisms," *Procedia Computer Science*, vol. 176, pp. 88–97, 2020. Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 24th International Conference KES2020.

[88] Microsoft, "Microsoft detours." https://github.com/microsoft/Detours. Accessed: 2025-05-15.

[89] Microsoft, "Samples: Traceapi." https://documentation.help/Detours/Sam_Traceapi.htm. Accessed: 2025-05-15.

[90] Microsoft, "Appinit dlls." https://technet.microsoft.com/en-us/library/cc939696.aspx. Accessed: 2025-05-15.

[91] Microsoft, "Createprocessw function." https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessw. Accessed: 2025-05-15.

[92] Microsoft, "Openprocess function." https://learn.microsoft.com/de-de/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocess. Accessed: 2025-05-15.

[93] Microsoft, "Createremotethread function." https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createremotethread. Accessed: 2025-05-15.

[94] Microsoft, "Exitprocess function." https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-exitprocess. Accessed: 2025-05-15.

[95] D. Plohmann, "Knowledge Fragment: Hardening Win7 x64 on VirtualBox for Malware Analysis," 2017. Blog post for ByteAtlas: http://byte-atlas.blogspot.de/2017/02/hardening-vbox-win7x64.html Accessed: 2025-05-15.

[96] Microsoft, "Launching applications (shellexecute, shellexecuteex, shellexecuteinfo)." https://learn.microsoft.com/en-us/windows/win32/shell/launch. Accessed: 2025-05-15.

[97] Microsoft, "Winexec function." https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-winexec. Accessed: 2025-05-15.

[98] Microsoft, "Resumethread function." https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-resumethread. Accessed: 2025-05-15.

[99] Microsoft, "Setthreadcontext function." https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-setthreadcontext. Accessed: 2025-05-15.

[100] Microsoft, "Openthread function." https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openthread. Accessed: 2025-05-15.

[101] Microsoft, "Queueuserapc function." `https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-queueuserapc`. Accessed: 2025-05-15.

[102] Microsoft, "Virtualqueryex function." `https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualqueryex`. Accessed: 2025-05-15.

[103] M. Miller, "Using dual-mappings to evade automated unpackers," 2008.

[104] Microsoft, "Writeprocessmemory function." `https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory`. Accessed: 2025-05-15.

[105] Daniel Plohmann, Steffen Enders, "Malpedia." General Statistics: `https://malpedia.caad.fkie.fraunhofer.de/stats/general`.

[106] Horsicq, "Detect-It-Easy," 2014. GitHub Repository: `https://github.com/horsicq/Detect-It-Easy/` Accessed: 2025-05-15.

[107] S. Wilson and S. Frankoff, "UNPACME Project Overview," 2025. Website: `https://www.unpac.me/` Accessed: 2025-05-15.

[108] Oreans, "Themida overview." `https://www.oreans.com/Themida.php`, 2025. Accessed: 2025-05-15.

[109] raw data, "Exploring autoit fud crypter." `https://raw-data.gitlab.io/post/autoit_fud/`, 2019. Accessed: 2025-05-15.

[110] T. F. Barabosch, *Formalization and Detection of Host-Based Code Injection Attacks in the Context of Malware*. PhD thesis, Universitäts-und Landesbibliothek Bonn, 2018.