electronics

MDPI

Article

Pinicorn: Towards Automated Dynamic Analysis for Unpacking 32-Bit PE Malware

Gwangyeol Lee, Minho Kim, Jeong Hyun Yi and Haehyun Cho *

Graduate School of Software, Soongsil University, Seoul 06978, Republic of Korea; gwangyeal@gmail.com (G.L.); mhkim37@soongsil.ac.kr (M.K.); jhyi@ssu.ac.kr (J.H.Y.)

* Correspondence: haehyun@ssu.ac.kr; Tel.: +82-2-828-7360

Abstract: Original Entry Point (OEP) and API obfuscation techniques greatly hinder the analysis of malware. Contemporary packers, employing these sophisticated obfuscation strategies, continue to pose unresolved challenges, despite extensive research efforts. Recent studies, like API-Xray, have mainly concentrated on rebuilding obfuscated import tables in malware, but research into OEP obfuscation is still limited. As a solution, we present Pinicorn, an automated dynamic de-obfuscation system designed to tackle these complexities. Pinicorn bypasses packers' anti-analysis techniques and retrieves the original program from memory. It is specifically designed to detect and analyze trampoline codes within both OEP and the import table. Our evaluation shows that Pinicorn successfully deobfuscates programs hidden by three different packers, confirming its effectiveness through a comparative analysis with their original versions. Furthermore, we conducted experiments on malware obfuscated by Themida and VMProtect, analyzing the obfuscation techniques and successfully de-obfuscating them to validate the effectiveness of our approach.

Keywords: OEP obfuscation; API obfuscation; deobfuscation; unpacking; malware analysis

Citation: Lee, G.; Kim, M.; Yi, J.H.; Cho, H. Pinicorn: Towards Automated Dynamic Analysis for Unpacking 32-Bit PE Malware. *Electronics* **2024**, *13*, 2081. https://doi.org/10.3390/electronics13112081

Academic Editor: George Angelos Papadopoulos

Received: 30 March 2024
Revised: 20 May 2024
Accepted: 25 May 2024
Published: 27 May 2024

1. Introduction

In the field of computer security, the ongoing changes in challenges require us to come up with new solutions. Malware is a particularly pervasive threat, affecting a wide range of devices from servers to mobile phones [1]. To solve this, various malware analysis methods have been devised, yet attackers continue to innovate, developing sophisticated evasion techniques [2].

Attackers commonly deploy packers [2–4]. These tools go beyond simple compression; they integrate sophisticated anti-analysis and obfuscation techniques into binaries. Packed malware, with its original code encrypted, presents significant detection and analysis challenges, especially in debugger or sandbox environments [5–7].

Despite the proposal of generic unpacking methods, they are frequently undermined by obfuscation techniques, including those affecting the OEP and API. Packed malware typically has its original code encrypted and its entry point redirected to the packer's bootstrap code. This bootstrap code, upon execution, temporarily restores the original program in memory. However, OEP obfuscation can mislead unpackers, causing them to incorrectly detect the OEP or interact with trampoline code, which results in execution inconsistencies.

Moreover, whereas standard programs rely on the operating system's loader for runtime binding of API addresses to the Import Address Table (IAT), packed malware depends on its bootstrap code due to the packer often stripping away the import table. Recent obfuscators have made this situation more complex by inserting trampoline codes into the IAT, which hide the API calls. Unpackers attempting to identify and reconstruct the IAT may misinterpret these trampoline codes as API addresses, leading to unpacking failures [8].

Modern packers frequently use trampoline code to disguise both the OEP and IAT, thus complicating program analysis [9–12]. Execution of such packed programs incorporates trampoline code into the original data, changing its behavior.

The difficulty of OEP obfuscation depends on whether trampoline code is present or not. Sometimes, a branch to this code is embedded at the OEP, causing potential invalid memory accesses in unpacked programs. In some cases, trampoline code execution disrupts the original control flow, rendering accurate OEP detection a challenging task.

The complexity in de-obfuscating API obfuscation hinges on the sensitivity of trampoline code to arguments [8]. Argument-sensitive trampoline codes funnel all obfuscated APIs to a single point, with the specific API being determined by the provided argument [12]. Conversely, argument-insensitive trampoline codes associate every obfuscated API with a unique trampoline code, requiring the identification and tracking of each one [9–11]. This distinction is critical as previous API de-obfuscation approaches mainly focused on detecting and observing trampoline code, often neglecting the argument's role [8,13,14].

This paper explores methods for automatically uncovering the obfuscation techniques employed by modern packers, emphasizing trampoline codes in OEP and API obfuscation. We introduce Pinicorn, a dynamic approach aimed at recovering these techniques and unpacking packed programs.

Pinicorn initially utilized a Dynamic Binary Instrumentation (DBI)-based Obfuscated Program Execution Module to counteract the packer's anti-analysis techniques, reinstating the original program in memory for runtime execution. During this phase, Pinicorn monitors the process, identifying potential OEP candidates to effectively analyze the unpacked program. The resulting memory dump includes trampoline codes critical for obfuscating OEP and API, which are then analyzed using memory static analysis. Then, an Emulator-based Trampoline Code Execution Module examines all trampoline codes, uncovering key information that was hidden by obfuscation. This process helps restore the program's original structure and import table.

Contributions: This paper makes the following contributions:

- We provide an analysis of packers and their role in hindering reverse-engineering of programs.
- We propose an automated dynamic analysis approach for de-obfuscating and unpacking programs.
- We introduce an *Emulator-based Obfuscated Code Execution* method, demonstrating its efficacy in identifying and analyzing trampoline code in packed programs.

The remainder of this paper is organized as follows: Section 1 discusses the impact of packing, OEP, and API obfuscation techniques on binaries. It also reviews related work in unpacking and de-obfuscation approaches. Section 2 outlines the challenges associated with OEP and API obfuscation and details our proposed solutions. Section 3 presents our de-obfuscation results. Section 4 delves into potential challenges in analyzing obfuscated program and mitigation strategies. Finally, Section 5 concludes the study and suggests future directions. Packers play a pivotal role in malware obfuscation, often removing the entry point and import table during the packing process. While earlier research efforts have centered on detecting the OEP and rebuilding the import table of packed malware, packers have evolved, adopting advanced obfuscation techniques to counter these measures [5,7]. A notable study, API-Xray, has effectively tackled the challenge of reconstructing import tables in obfuscated malware contexts, revealing that 9.5% of resistant malware samples employed both OEP and API obfuscation [8].

1.1. The Impact of General Packers on Binaries

General packers, such as Themida, VMProtect, and ASProtect, are widely used to obfuscate malware binaries, making analysis and detection challenging. These packers employ various techniques to compress, encrypt, and manipulate the executable code, hindering static and dynamic analysis. Understanding the impact of these packers on binary structure and behavior is crucial for developing effective unpacking solutions.

Figure 1 illustrates the standard control flow of a Portable Executable (PE) file. In this process, the Windows Loader carries out essential tasks such as binding the API to the IAT and starting execution from the entry point. PE files utilize APIs from Dynamic

Link Libraries (DLLs), with the Windows Loader populating the IAT with these API addresses [15].

Once configured, the loader activates the program's entry point, commencing the main program functions. However, as depicted in Figure 2, packers modify the entry point to direct it to their bootstrap code. In execution, the bootstrap code unpacks the original program, manages library loading, and allocates resources before triggering the OEP to execute the *temporarily unpacked program*.

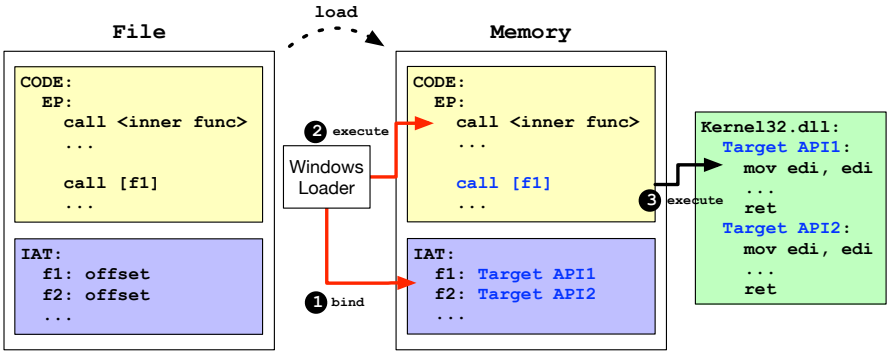


Figure 1. The control flow of normal program.

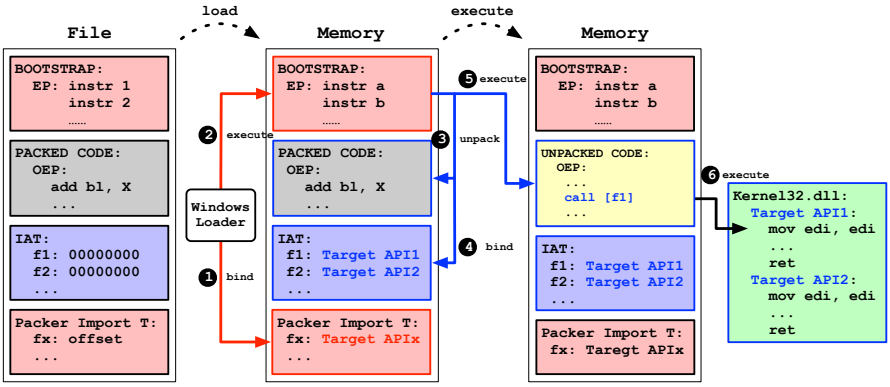


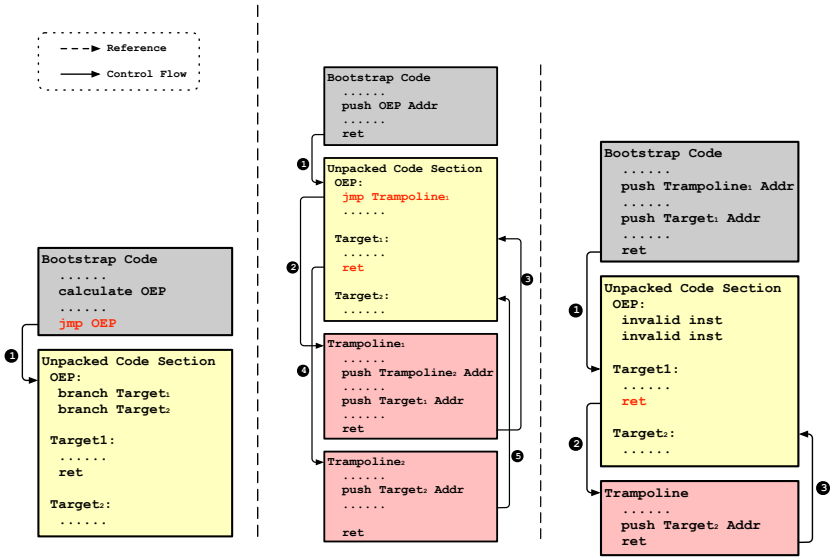
Figure 2. The control flow of packed program.

1.2. Anti-Analysis Techniques

Modern packers incorporate sophisticated anti-analysis techniques to evade detection by analysts and automated tools. These techniques include anti-debugging, anti-virtual machine, and anti-emulation strategies that complicate the analysis process. Previous studies have successfully bypassed such techniques in commercial protectors and unpacked malware protected by known packers [5,7,16,17]. However, challenges persist with unknown packers, and research is ongoing to overcome these obstacles from yet unidentified packers.

1.3. OEP Finding

The Original Entry Point (OEP) is a critical component in unpacking malware, as it represents the location where the unpacked code begins execution. Identifying the OEP in heavily obfuscated binaries is challenging. Traditional unpacking methods often employ a *Written-then-Execute* approach to locate the OEP by analyzing instructions restored at runtime [18–21]. However, commercial protectors like Themida [10], as shown in Figure 3b, implement OEP obfuscation techniques that challenge these conventional methods. These techniques may involve the use of trampoline code or not.



(**a**) ASProtect, Obsidium (**b**) Themida (**c**) VMProtect

Figure 3. The control flow of OEP obfuscation techniques.

1.4. Import Table Reconstruction

Reconstructing the import table is essential for understanding the external dependencies of a binary. Obfuscation techniques often modify or hide import table entries to thwart analysis. Our methodology leverages dynamic analysis to accurately rebuild the import table, revealing the true dependencies and enhancing the understanding of the malware's functionality. Some research has successfully deobfuscated by examining five argument-insensitive API obfuscation techniques [8,13,14]. Nevertheless, identifying obfuscated APIs becomes particularly challenging when dealing with argument-sensitive trampoline codes, which depend on API arguments.

1.5. OEP and API Obfuscation Techniques

Obfuscation techniques targeting the OEP and API calls are designed to mislead analysts and automated tools. These techniques include code virtualization, encryption, and polymorphism. We provide a comprehensive overview of these obfuscation strategies, highlighting their impact on the analysis process and the challenges they pose.

1.6. OEP Obfuscation: A Deep Insight

OEP obfuscation involves various methods to obscure the original entry point, making it difficult to locate and analyze. Techniques such as control flow manipulation and encryption are commonly used. Our deep dive into OEP obfuscation reveals the intricacies of these techniques and how our approach effectively counters them.

In approaches employing trampoline code, packers may remove branching around the OEP, as with Themida. Execution then involves either the bootstrap or trampoline code replicating the effects of these removed instructions, complicating OEP detection and risking program termination if untreated trampoline codes lead to invalid memory access at startup. Packers like VMProtect [9], illustrated in Figure 3c, might have bootstrap code directly execute targets without fully restoring the OEP, presenting additional challenges for its identification.

In contrast, techniques not involving trampoline code may further conceal the OEP or access it through multiple branches. Once reached, however, the execution mirrors that of non-obfuscated programs. Packers such as ASProtect [12] and Obsidium [11] adopt this method, indicating that conventional unpacking can successfully detect the OEP in these cases.

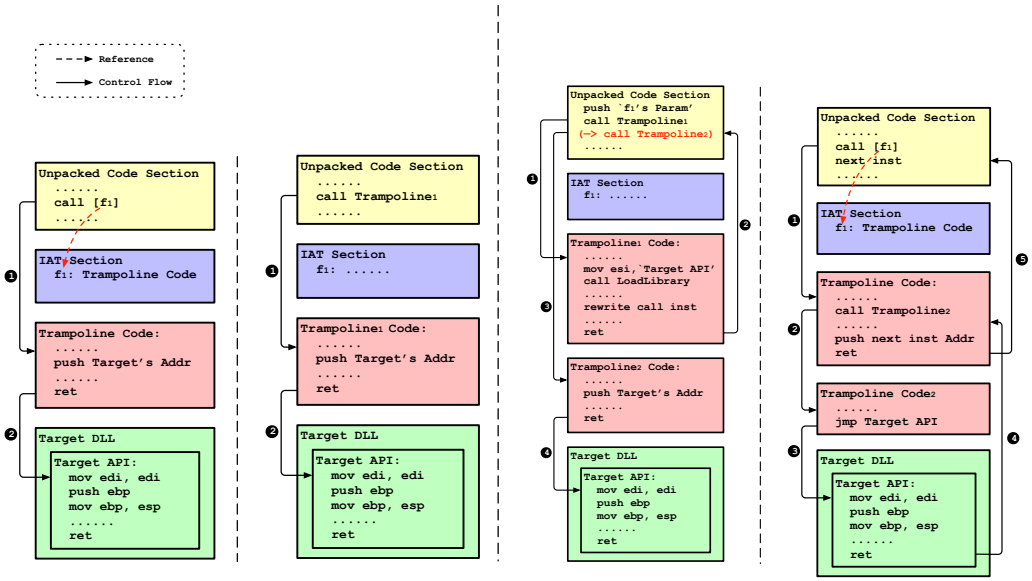
1.7. API Obfuscation: A Deep Insight

API obfuscation aims to hide or alter API calls to prevent detection and analysis. Techniques include API redirection and dynamic import resolution. We delve into these methods, explaining their mechanism and how our combined use of DBI and emulators successfully deobfuscate API calls, ensuring accurate analysis of the malware's behavior.

API obfuscation techniques, which alter API calls or IAT entries, are a prevalent strategy among packers. They reconstruct the import table exclusively during runtime, thus requiring dynamic analysis after the API has been bound. These techniques, including trampoline code, are designed to obfuscate control flow and complicate analysis by employing tactics like *Dead Code Inserting* and *Constant Propagation* [7]. Trampoline codes can be invoked in two primary ways:

1. **Indirect Calls:** Packers such as Themida use the IAT to bind trampoline codes at runtime, redirecting API calls to these codes (Figure 4a,d).
2. **Direct Calls:** Alternatively, packers like VMProtect and ASProtect modify API call instructions to point directly to trampoline codes, bypassing the IAT (Figure 4b,c).

These trampoline codes, integral to the execution logic, serve to conceal the control flow and call API addresses in a masked manner. Packers such as Themida, VMProtect, and Obsidium typically employ *argument-insensitive* trampoline code, where each obfuscated API is associated with its specific trampoline code. In contrast, packers like ASProtect use *argument-sensitive* trampoline codes, where the API address depends on the provided arguments, leading all obfuscated APIs to a single trampoline code. This variation presents a significant challenge for deobfuscation efforts [8].



(**a**) Themida (**b**) VMProtect (**c**) ASProtect (**d**) Obsidium

Figure 4. The control flow of API obfuscation techniques.

1.8. Summary of Related Works on Malware Unpacking and Analysis Techniques

Table 1 provides a comprehensive overview of the literature on malware unpacking and analysis techniques, summarizing the research design, methodologies, contributions, and results of related works. This table serves as a foundational reference for understanding the current state of the art and positioning our contributions within this context.

Table 1. Summary of related works on malware unpacking and analysis techniques.

Related Work	Research Design	Methodology	Contribution	Result
[21]	Comparative analysis	Uses emulation to extract hidden code.	Accurate hidden code extraction solution.	Successfully extracted original binaries from most executables.
[13]	Analytical study	Uses manual and automated analysis with Dyninst.	Detailed taxonomy of obfuscation techniques.	Enhanced analysis of obfuscated binaries.
[22]	Theoretical study	Uses taint propagation and code transformations.	Generalized deobfuscation technique.	Reconstructed logic of obfuscated programs.
[23]	Longitudinal study	Develops a framework and taxonomy for packer complexity.	Insights into evolution and sophistication of packers.	Revealed trends and increased complexity in packing.
[14]	Tool development	Uses DynamoRIO for static and dynamic analysis.	Accurate reconstruction of unpacked binaries.	Effective against self-modifying code and IAT destruction.
[5]	Empirical study	Uses static and dynamic analysis with Intel PIN.	First to deobfuscate Themida 2.4.5.0 version.	Reduced analysis time and enhanced deobfuscation techniques.
[24]	Experimental study	BinUnpack uses kernel-level DLL hijacking.	High-performance solution for multi-layered malware.	Completed unpacking within 0.5 s.
[6]	Comparative study	Utilizes packed IAT hooking on bare-metal OS.	Effective unpacking without simulated environments.	Outperformed existing approaches for environment-sensitive packers.
[16]	Evaluative study	Uses Pin tool to detect and bypass anti-debugging techniques.	Automatic detection and bypassing scheme.	100% success rate in tested protectors.
[7]	Hybrid analysis	Combines instruction emulation and direct execution.	Specialized tool for anti-reverse engineering techniques.	Unpacked all tested binaries and provided detailed logs.
[17]	Experimental study	Uses DBI tools to bypass anti-analysis techniques.	Demonstrates DBI tools' effectiveness.	Bypassed anti-analysis in various commercial protectors.
[8]	Experimental study	API-Xray uses hardware-assisted tracing and dynamic/static analysis.	Robust method for import table reconstruction.	Rebuilt import tables for 174,285 samples.
[25]	Experimental study	Uses HPCs and loop profiling with machine learning.	Novel hardware-assisted malware unpacking.	Successfully unpacked malware with selected hardware events.

2. Design

2.1. Overview

Our project's main goal is to implement Pinicorn, a system that automatically clears up confusions in Windows programs caused by obfuscation. We aim to do this in three ways: (1) finding *Trampoline Code*, (2) studying how trampoline code hides the OEP and API, and (3) fixing programs to effectively remove the obfuscation.

2.1.1. Scope

This research focuses on the obfuscation techniques employed by commercial protectors, particularly in the realms of OEP and API obfuscation. Prior studies have extensively explored API obfuscation but have often neglected the complexities of *argument-sensitive*

trampoline code, an area we aim to address. Additionally, not enough research has been done on OEP obfuscation, which is why we have decided to explore it. We have observed that both OEP and API obfuscation frequently involve the insertion of trampoline code addresses into binaries. Notably, tools like ASProtect extensively utilize *argument-sensitive* trampoline code, necessitating innovative deobfuscation strategies. Additionally, we have identified trampoline code insertions disrupting regular control flow in OEP obfuscation, particularly in protectors like Themida and VMProtect. Our study delves into these techniques, emphasizing argument-sensitive trampoline code, and aims to develop effective deobfuscation methods.

2.1.2. Challenges

In this project, we address the challenges involved in designing a system to deobfuscate OEP and API obfuscation. The project addresses two primary challenges in deobfuscating OEP and API obfuscation within obfuscated binaries:

1. **Counteracting Anti-analysis Techniques:** Commercial protectors often deploy evasion techniques such as Anti-Debug and Anti-VM, probing memory structures and system settings. The straightforward nature of these techniques enables obfuscated software, especially malware, to efficiently detect analyis tools. This critical aspect is often overlooked in previous research.
2. **Trampoline Code Analysis Problem:** Obfuscated binaries contain multiple trampoline codes in both original and bootstrap sections. Our goal is to list APIs and track how the Instruction Pointer branches from the OEP, requiring accurate execution and detailed analysis of each trampoline code.

2.1.3. Strategy

1. **Anti-analysis:** Packers use evasion techniques like debugger and virtual machine detection, complicating the analysis of packed malware. Our approach started with an in-depth study of well-known anti-analysis methods, summarized in Table 2. We aimed to adapt the execution context to evade detection by monitoring specific instructions and API calls.
    - Study of Anti-analysis Methods: We systematically reviewed various anti-analysis techniques used by popular packers. This included methods like API hooking and environmental checks.
    - Execution Context Adaptation: By tracking specific instructions and API calls, we developed strategies to adapt the execution environment dynamically, ensuring our analysis bypasses detection mechanisms.
2. **OEP obfuscation:** Packers disguise the OEP using complex bootstrap code patterns, posing unique challenges during unpacking. To address this, we studied commercial protectors to understand their methods, focusing on trampoline code that could lead to errors during unpacking. Our deobfuscation system is based on these findings, employing OEP detection algorithms, analyzing trampoline calls, and tracing the flow back to the original code.
    - Commercial Protector Analysis: We examined the techniques used by commercial protectors to obfuscate the OEP, with particular focus on their bootstrap code and trampoline patterns.
    - Deobfuscation Algorithm Development: Based on our analysis, we detected the OEP, identified trampoline calls, and traced them back to the original code flow.
3. **API obfuscation:** Packers frequently eliminate the import table from the original program, which makes the analysis of APIs more challenging. Our approach involves identifying the IAT through trampoline codes and rebuilding it. Modern packers, such as Themida and Obsidium, obfuscated APIs using trampoline code addresses in the IAT. Our method examines blocks containing trampoline codes and employs a basic block-level emulation approach to unveil obfuscated APIs.

- IAT Reconstruction: We designed a method to reconstruct the IAT by analyzing and deobfuscating trampoline codes.
- Basic Block Emulation: Utilizing basic block-level emulation, our approach reveals obfuscated API calls, enabling accurate identification of the APIs used by the program through the reconstruction of the IAT.

Table 2. A summary of anti-analysis techniques.

Packer	Version	Anti-Debug	Anti-VM
Themida	3.0.7	PEB (NtGlobalFlag), NtQueryInformationProcess, NtUserGetForegroundWindows	Registry Key and Value Checks, GeySystemFirmwareTable
VMProtect	3.0.9, 3.4.0	PEB (BeingDebugged), NtQueryInformationProcess, NtClose, Single-Step Exception	CPUID
ASProtect	2.7.8	PEB (BeingDebugged)	
Obsidium	1.6.7	PEB (BeingDebugged), NtQueryInformationProcess, NtClose	GeySystemFirmwareTable, IN

2.2. Proposed Scheme

Pinicorn, illustrated in Figure 5, is designed to tackle the aforementioned challenges. It comprises two principal modules: (1) a DBI-based program execution module and (2) an emulator-based trampoline code execution module. Pinicorn operates by executing the obfuscated program to capture the restored data, identifying inserted trampoline code, and running this code independently to uncover its obfuscated data. The end result is a reconstructed program with a deobfuscated OEP and import table.

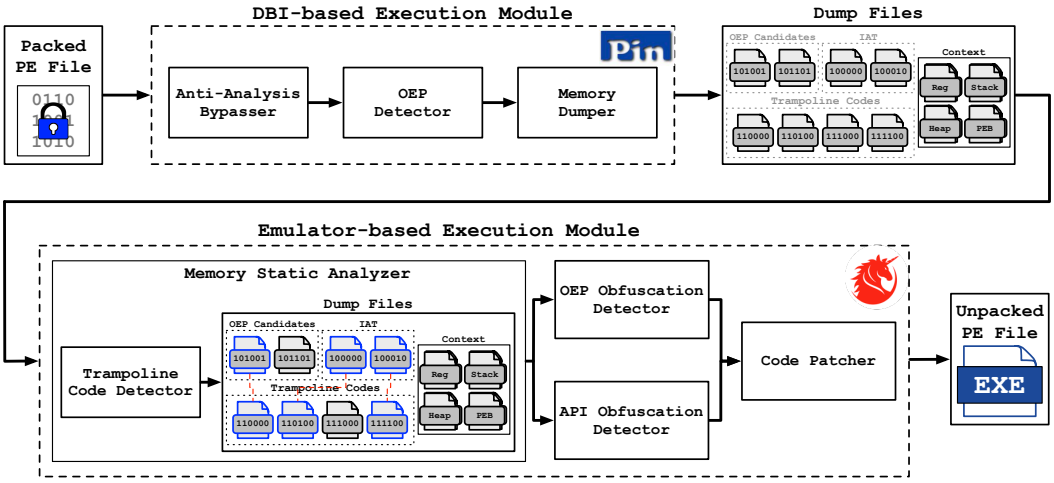


Figure 5. The overview of Pinicorn.

2.2.1. DBI-Based Obfuscated Program Execution

Why DBI? Dynamic program analysis tools such as debuggers, emulators, and DBIs vary in their objectives and applications. For Pinicorn, the goal is to execute obfuscated programs until they restore the original program, capturing this state. Despite various studied techniques, the Intel Pin, a DBI framework, is preferred for its resilience against anti-analysis techniques [16]. However, to avoid detection by some obfuscation tools, we developed a PinTool-based bypass module, enabling the analysis of programs by bypassing anti-analysis techniques.

Roles of DBI. Intel Pin dynamically instruments binary code using JIT (Just-In-Time) compilation [26]. Its plugin, PinTool, facilitates specific program analyses, allowing us

to inject code for targeted analysis. The primary role here is to execute the obfuscated program while avoiding anti-analysis and to capture a memory snapshot when the OEP is reached. This memory dump includes sections with trampoline code addresses and setup data for emulator-based analysis in subsequent stages.

2.2.2. Emulator-Based Trampoline Code Execution

Pre-Process. The DBI module captures process memory, including the original code and data (e.g., import table). The trampoline code's location depends on the specific OEP and API obfuscation techniques employed. For instance, in Themida's OEP obfuscation, the trampoline code is located at the OEP, as depicted in Figure 3. Similarly, for API obfuscation, Themida stores trampoline code addresses in the IAT, while VMProtect and ASProtect patch the API call to directly invoke the trampoline code, as shown in Figure 4. Pinicorn identifies these trampoline code locations to ensure they can be executed during emulation.
Why Emulator? The secondary aim of Pinicorn is to execute detected trampoline codes to decode obfuscated details. The emulator, using the memory dump as a reference, replicates the environment from the first module, allowing targeted execution of trampoline codes to decode obfuscated information.
Roles of Emulator. Unicorn [27], a CPU emulator built upon QEMU, facilitates direct binary code execution, making it ideal for reverse engineering tasks. Using Unicorn, Pinicorn focuses on specific program segments, detecting trampoline code locations and executing them via the emulator to pinpoint obfuscated data. This data is then used to reconstruct the deobfuscated binary.

3. Evaluation

In this section, we demonstrate the effectiveness of Pinicorn by deobfuscating and unpacking a program protected by three packers that use the OEP obfuscation and API obfuscation techniques.

3.1. Dataset

The Juliet Test Suite C/C++ [28] is a dataset of source code written in C/C++ languages, widely used to evaluate static, dynamic, and source code analysis tools. We selected this dataset because the Juliet Test Suite code is based on Common Weakness Enumeration (CWE) and includes code that causes security vulnerabilities. These vulnerabilities can potentially be exploited in malware [17]. Therefore, we assumed the obfuscated samples from this dataset as obfuscated malware for our experiments. We compiled this code to generate binaries and then applied OEP and API obfuscation techniques of three different packers. We then compared the original programs with the deobufscated ones to verify the effectiveness of the deobfuscation results.

Next, we used the VirusShare dataset, which includes a variety of samples such as malware, Trojans, and ransomware, and is widely used in malware analysis research [29]. From this dataset, we selected packed PE malware from 2021 to 2022 for our experiments. We verified the presence of packing using PyPackerDetect [30], and the results are presented in Tables 3 and 4. Most packers were primarily found in 32-bit malware samples, rather than in 64-bit ones. Notably, Obsidium was not detected in any samples. Consequently, our experiments focused on deobfuscating 32-bit malware samples packed with Themida, VMProtect, and ASProtect.

Table 3. The detection of 32-bit packed malware by year.

Packer	2021 [1]	2022
Themida	39	4376
VMProtect	432	8667
ASProtect	166	3328
Obsidium [2]	0	0

[1] The absolute quantity of the VirusShare 2021 dataset is small. [2] Failed to find Obsidium malware samples.

Table 4. The detection of 64-bit packed malware by year.

Packer	2021 [1]	2022
Themida	29	767
VMProtect	59	1386
ASProtect [2]	0	0
Obsidium [2]	0	0

[1] The absolute quantity of the VirusShare 2021 dataset is small. [2] Packed 64-bit malware was not found.

3.1.1. Experiment Setup

We implemented Pinicorn on Intel Pin (V3.20.98437) and Unicorn (V2.0.1). For analysis, we used a Windows 10 Virtual Machine (VM), Windows 10 Pro, 22H4, 32-bit for each dataset. The experiments ran on three host machines with Windows 10 (Intel i9-12900, 64 GB of RAM), utilizing a total of 15 VMs. Each VM was configured with 4 CPUs and 8 GB of RAM. Furthermore, we set a 30-min execution time limit for each malware during the *DBI-based Obfuscation Program Execution*.

3.1.2. Comparison Models

Pinicorn, leveraging both Pin and Unicorn, reverses OEP obfuscation and API obfuscation techniques that current tools cannot handle. Its performance is assessed by comparing Juliet Test Suite's original programs with their deobfuscated versions, and by contrasting its execution results and speed with those of PinDemonium and Unipacker, open-source unpackers based on Pin and Unicorn.

PinDemonium [18], a Pin-based tool for unpacking PE files, employs multiple OEP detection algorithms to pinpoint OEPs, uses Scylla [31] for memory dumps, and reconstructs import tables. However, as it is not fully automatic, users must manually choose the correct OEP from the candidates.

Unipacker [32], a Unicorn-based unpacking tool for PE files, automates the unpacking process through Python scripts without needing manual input. However, it is limited to certain packers like UPX [33], ASPack [34], and FSG and may struggle with unpacking more complex obfuscated programs.

3.2. Comparison of Deobfuscation Results

In this experiment, we analyzed 300 programs that had been obfuscated using techniques from three different packers: Themida, VMProtect, and ASProtect. These obfuscation techniques were applied to 100 original programs sourced from the Juliet Test Suite. The goal was to reverse the OEP and API obfuscation techniques employed by these packers, evaluating the deobfuscation efficiency of Pinicorn against PinDemonium and Unipacker, as detailed in Table 5.

Table 5. Deobfuscation results of Juliet Test Suite.

Packers	API Obfuscation			OEP Obfuscation		
	PinDemonium	Unipacker	Pinicorn	PinDemonium	Unipacker	Pinicorn
Themida	0/100	0/100	100/100	0/100	0/100	100/100
VMProtect	0/100	0/100	100/100	0/100	0/100	100/100
ASProtect	N/A [1]	0/100	100/100	N/A[1]	0/100	N/A [2]

[1] PinDemonium failed to unpack due to a blue screen error during operation. [2] ASProtect's OEP obfuscation technique does not utilize trampoline code.

3.2.1. Evaluation of PinDemonium

In this section, we evaluate PinDemonium's ability to handle the OEP and API obfuscation techniques of Themida, VMProtect, and ASProtect. We examine the effectiveness of its OEP detection algorithms and import table reconstructing methods. PinDemonium uses four heuristic algorithms to detect the OEP: Long Jump Heuristic, Jump Outer Section

Heuristic, Entropy Heuristic, and Yara Heuristic. Our experiments revealed mixed results: For Themida samples, PinDemonium accurately detected the OEP address. Howerver, a trampoline code branch instruction remained at the OEP, complicating further analysis. In the case of VMProtect samples, the algorithm incorrectly identified code within the bootstrap code section as the OEP. This misidentification indicates that the current heuristics are ineffective against VMProtect's obfuscation techniques. ASProtect samples presented additional challenges, as we could not test them due to unresolvable issues with both OEP and API obfuscation techniques.

PinDemonium relies on Scylla to dump the temporarily unpacked process into memory and modify the import table. This results varied across different packers: For Themida Samples, the *Fix Dump* operation generally failed, except for some APIs like *HeapFree* and *TlsGetValue*. This suggests incomplete reconstruction of the import table. VMProtect samples fared worse, with PinDemonium failing to record any APIs in the import table. This highlights a significant gap in its unpacking capability. Similar to the OEP detection, ASProtect samples were untestable for import table rebuilding due to the same underlying issues. The primary cause of these problems lies in the development environment of PinDemonium. It was developed using the Windows 7 operating system and Visual Studio 2010's compiler. When operated in environments beyond Windows 10, these outdated foundations led to compatibility issues and blue screen errors.

3.2.2. Evaluation of Unipacker

Unipacker also struggled to handle the obfuscation techniques of Themida, VMProtect, and ASProtect. Unipacker, based on the Unicorn CPU Emulator, does not heavily depend on the operating system environment. This flexibility allowed us to attempt unpacking ASProtect, unlike with PinDemonium. Despite this advantage, Unipacker failed to detect the OEP in all cases and could not properly fix the import table. The primary reason for these failures is that Unipacker is designed to handle packers that do not employ advanced obfuscation techniques. Consequently, it could not cope with the sophisticated methods used by Themida, VMProtect, and ASProtect.

3.2.3. Evaluation of Pinicorn

In this section, we evaluate Pinicorn, which uses Intel PinTool and the Unicorn Emulator, to determine its effectiveness against the obfuscation techniques of Themida, VMProtect, and ASProtect. First, we tested how well Pinicorn handles the OEP obfuscation techniques of the three packers. Themida and VMProtect employ OEP obfuscation by modifying the OEP instructions. They use trampoline code to redirect the execution flow, making it possible to identify the original execution flow by analyzing the final return address of the trampoline code. By inserting a branching instruction to the original execution flow, we can deobfuscate the program to run similarly to the original.

ASProtect's OEP obfuscation does not use trampoline code, and thus falls outside the scope of obfuscation techniques this paper addresses. Therefore, we applied PinDemonium's algorithm to attempt unpacking ASProtect. Although PinDemonium's algorithm failed to accurately detect the OEP address, manual analysis revealed that identifying the code executed immediately after the OEP serves as an effective entry point. Figure 6 presents the comparison of Control Flow Graphs (CFGs) for each program deobfuscated using our proposed system, Pinicorn, after applying OEP obfuscation techniques by Themida, VMProtect, and ASProtect. The CFGs for the original, Themida, VMProtect, and ASProtect versions reveal minor differences in the VMProtect version. However, these differences are limited to areas near the OEP instructions, with the rest of the CFGs showing close similarity.

Next, we evaluated how well Pinicorn handles the API obfuscation techniques of the three packers. Themida and VMProtect use argument-insensitive trampoline code for API obfuscation. In this scenario, Pinicorn detects all trampoline code and emulates it to identify the obfuscated API through the final return address of the trampoline code. ASProtect employs argument-sensitive trampoline code for API obfuscation. Pinicorn

detects all trampoline code and emulates from the basic block containing the trampoline call instruction. This process identifies different APIs that the same trampoline code branches to. Our experiments confirmed that Pinicorn effectively addresses the OEP and API obfuscation techniques of Themida, VMProtect, and ASProtect, though it encountered issues with some APIs, which will be further discussed in Section 4.5.2. The proposed system demonstrates robust capability in deobfuscating programs protected by these advanced techniques.

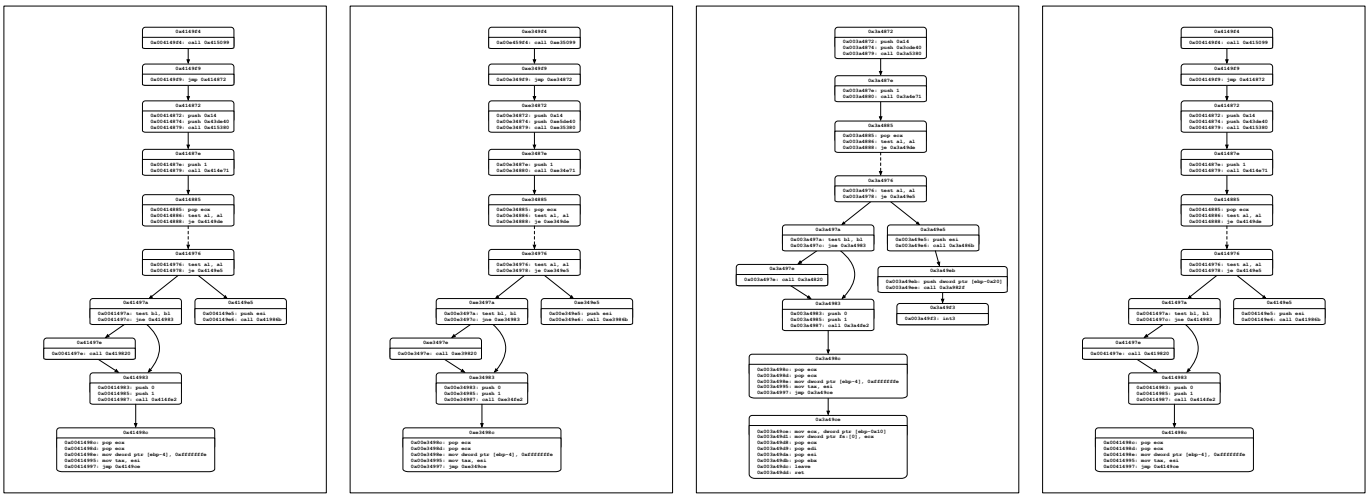


Figure 6. CFG extraction comparison: Original vs. Themida vs. VMProtect vs. ASProtect.

3.3. Malware Case Study

We conducted our experiments using real-world malware collected from VirusShare. The identification of OEP and API obfuscation techniques requires the malware to be temporarily unpacked in memory during runtime. Therefore, we used packing detection tools to identify malware samples packed with Themida and VMProtect from the diverse collection. Table 6 provides an overview of the presence of API and OEP obfuscation in the analyzed malware samples and the results of our deobfuscation efforts.

Table 6. Overview of obfuscation presence and deobfuscation results in VirusShare malware samples.

Packers	API Obfuscation	OEP Obfuscation
Themida	9/43	37/43
VMProtect	10/27	11/27

After detection, we randomly selected malware samples for case studies. We did not conduct experiments on ASProtect-packed malware samples. The reasons for this execution will be discussed in the limitation section.

3.3.1. Themida Malware Case Study

We used a total of 100 samples for the Themida malware case study. Among these, we successfully dumped 43 samples. Analysis of these 43 samples revealed that 9 samples used OEP obfuscation, 37 samples used API call obfuscation, and 8 samples employed both OEP and API obfuscation techniques.

The failure to dump the remaining 57 samples is attributed to the varying anti-analysis techniques used by different version of Themida. The version of Themida used in this study was 3.0.7.0, which differs from the 2.4.5.0 version analyzed in previous research. These differences highlight the evolution of anti-analysis techniques in newer versions. It is likely that even later versions have introduced further changes to these techniques.

3.3.2. VMProtect Malware Case Study

We used a total of 100 samples for the VMProtect malware case study and successfully dumped 27 of them. Analysis of these 27 samples revealed that 10 samples used OEP obfuscation, 11 samples used API obfuscation, and 2 samples employed both OEP and API obfuscation techniques.

Similar to Themida, the anti-analysis techniques used by VMProtect can vary across different versions. In this study, we used versions 3.0.9 and 3.4.0 of VMProtect. There is a possibility that earlier or later versions employ different anti-analysis techniques. Although we classified malware samples based on the type of packer, we did not categorize them by specific versions, which is a limitation of our study.

3.4. Performance Evaluation

We conducted experiments to measure how long it takes for Pinicorn to analyze the trampoline code used in obfuscation and extract data useful for deobfuscation. For comparison, we measured the execution time of programs with Themida's OEP and API obfuscation using Pinicorn, PinDemonium, and Unipacker. Figure 7 shows the average execution time of each unpacker running Themida samples 10 times. Notably, the execution time of Pinicorn in the DBI stage is only approximately 25 s longer than PinDemonium. This additional overhead occurs because Pinicorn (DBI stage) performs dynamic analysis of the packed program and dumps memory whenever an OEP candidate is identified.

Another key point is that the execution time of Pinicorn in the Emulator Stage, which involves analyzing numerous OEP candidates, identifying the correct OEP, and analyzing all trampoline code in the memory dump, is only approximately twice as long as the DBI stage. This indicates that although the Emulator Stage involves more extensive analysis, the additional time required is not excessively high. However, if the test program used more APIs, the execution time for the Emulator Stage could increase.

The fundamental reason Pinicorn takes longer to execute than other unpackers is that it identifies and analyzes all trampoline code used in obfuscation. This comprehensive analysis is essential for accurately deobfuscating the program.

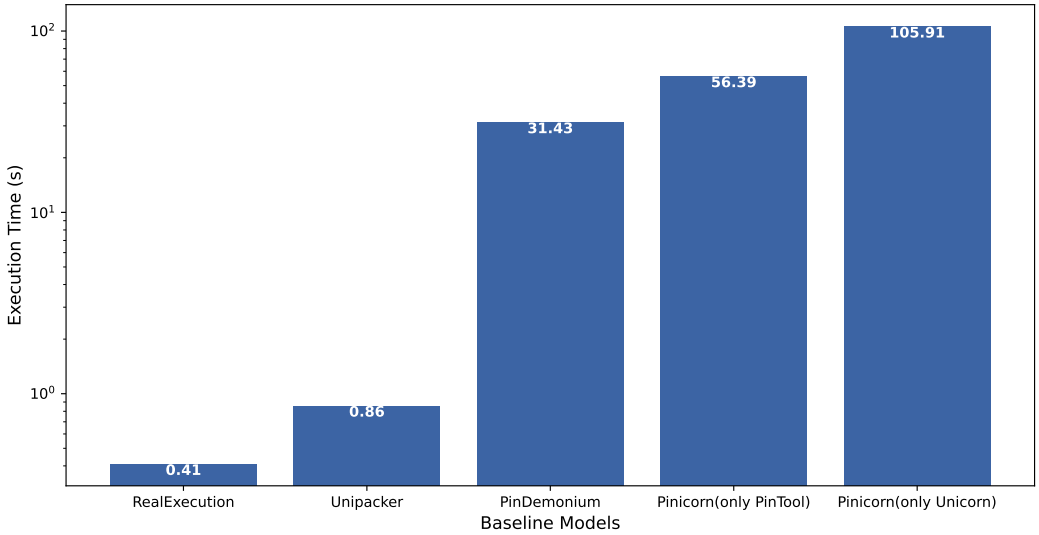


Figure 7. Elapsed time comparison for experiments.

4. Discussion

Analyzing malware is inherently challenging, and no method can claim to be perfect. Security researchers often select techniques that best suit their specific needs, while accepting inevitable trade-offs. Various approaches aimed at rendering malware into an *analyzable state* often face countermeasures implemented by sophisticated malware. This section discusses the role of DBI and emulators in malware analysis, how our proposed

system overcomes the limitations of existing systems, and its integration with existing knowledge, and it examines potential attacks against Pinicorn and its inherent limitations.

4.1. Role of DBI and Emulators in Malware Analysis

Dynamic binary instrumentation and emulation technologies have evolved significantly, becoming critical tools in malware analysis. DBI frameworks, such as Intel Pin and DynamoRIO, allows for dynamic analysis of binary executables by inserting instrumentation code at runtime. This capability enables detailed monitoring and analysis of program behavior without modifying the binary code.

Early DBI systems were limited by performance overhead and compatibility issues, which restricted their application in real-time malware analysis. However, advancements in DBI technology have addressed many of these limitations. Modern DBI frameworks offer improved performance, greater flexibility, and support for a wide range of architectures and operating systems. These enhancements have made DBI an invaluable tool for analyzing obfuscated and packed malware.

Emulators, such as QEMU and Unicorn, simulate the execution of binary code in a controlled environment, allowing analysts to observe malware behavior without the risk of infection. These tools have evolved to provide high-quality simulation of various CPU architectures and system environments, expanding their applicability in malware analysis.

4.2. Overcoming Limitations of Existing Systems

Traditional malware unpacking systems often struggle with several limitations, including the following:

- **Detection Evasion**: Advanced malware employs anti-analysis techniques to detect and evade previous analysis environments, rendering many tools ineffective.
- **Dynamic Analysis Limitations**: Dynamic analysis tools cannot execute or monitor all behaviors of a binary code. This limitation can prevent the analysis of certain obfuscated code, making them impractical for unpacking some malware. Specifically, dynamic analysis may miss code paths that are rarely executed or require specific triggers to activate. By not capturing all possible execution paths, these tools might fail to fully unpack or understand the malware's functionality, reducing their effectiveness in thorough malware analysis.

Our proposed solution leverages modern DBI and emulation technologies to overcome these limitations. By utilizing Intel Pin to dump the temporarily unpacked process in memory and performing static analysis on the memory dump to detect trampoline code used in obfuscation techniques, and employing Unicorn to analyze the trampoline code, we can achieve comprehensive monitoring of malware behavior. The integration of these approaches allows for the detection of anti-analysis techniques and the reconstruction of obfuscated code.

4.3. Integration with Existing Systems

Our approach builds on existing knowledge in malware analysis by integrating DBI and emulators into the broader framework of dynamic and static analysis techniques. Related works have demonstrated the effectiveness of DBI and emulators in various contexts, but few have explored their combined potential in tackling both OEP and API obfuscation simultaneously.

For instance, Park et al. [16] utilized Intel Pin to detect and bypass anti-debugging techniques, demonstrating the framework's robustness. Similarly, Choi et al. [7] combined instruction emulation with direct execution to handle anti-reverse engineering techniques, showcasing the power of emulation in malware analysis. Our work extends these studies by integrating DBI and emulation to provide a more comprehensive solution that addresses multiple layers of obfuscation. Furthermore, API-Xray [8] demonstrated the capability of reverse engineering by reconstructing the import tables of obfuscated malware. This study highlights the effectiveness of dynamic and static analysis techniques in overcoming

advanced obfuscation strategies, reinforcing the potential of our integrated approach to achieve similar successes in unpacking and analyzing complex malware.

4.4. Potential Attacks and Measures

4.4.1. DBI Detection Attacks

Intel Pin, as a DBI framework, enables the execution of packed programs that use anti-analysis techniques. Despite its resilience [7,16,17], some sophisticated methods can still detect Pin and similar DBI frameworks. Thus, despite protective measures, some packed malware may recognize Pinicorn's DBI module, triggering its bootstrap code and hindering memory dumping.

4.4.2. Anti-Debugging Attacks in Trampoline Code

Our study primarily investigated commercial packers known for regular updates, focusing on OEP and API obfuscation techniques. Although our reviewed techniques did not employ anti-debug APIs during trampoline code execution, other studies have reported such instances within API obfuscation's trampoline codes [8]. This implies the possibility that some sophisticated packed malware might detect emulation of their trampoline codes.

4.5. Limitation

4.5.1. Context Dump Problem

Most unpackers run the packed program until it naturally concludes [18,32]. In virtual or sandbox environments, this can result in incomplete unpacking due to time constraints. Pinicorn captures the moment when the bootstrap code temporarily restores the original program in memory, then performs a comprehensive memory dump. However, this approach encounters several challenges:

1. We utilized PinDemonium's OEP detection to recognize when the program's original state is restored. Pinicorn identifies trampoline code from several OEP candidates. Analyzing multiple candidates and performing memory dumps for each significantly impacts performance.
2. Operating in a virtual machine with a 30-min runtime limit, Pinicorn faced resource allocation challenges for malware dumping across 15 virtual machines. This occasionally resulted in restoring virtual machine snapshots before executing the packed malware's original code.
3. In experiments with malware, issues may arise from anti-analysis techniques in unknown versions of known packers, which cannot be completely bypassed. To improve this, there is a need to analyze as many versions of packers as possible and implement additional bypass modules.

4.5.2. Stolen Function

Some packers utilize the *Stolen Function* technique, copying entire Windows API instructions directly rather than redirecting through trampoline code [8]. This technique presents a challenge for Pinicorn in identifying these copies, as no redirection takes place. In our research, this issue was observed with only a few APIs, such as *GetCurrentThreadID*.

4.5.3. Unknown Argument-Sensitive Trampoline Code for API Obfuscation

Among the examined tools, only ASProtect employs *argument-sensitive* trampoline code in its API obfuscation. It switches from argument-sensitive to argument-insensitive call instructions at runtime, depending on preceding arguments. This complexity makes traditional deobfuscation methods, which primarily focus on trampoline code execution, less effective [8]. To overcome this, Pinicorn initiates emulation from the basic block that contains the call instruction (Figure 4c), covering necessary argument-related trampoline codes and facilitating effective tracking and precise identification of obfuscated API. However, malwares using unknown packers might employ other advanced argument-sensitive API obfuscation patterns, introducing further challenges.

4.5.4. Problems of Deobfuscation Approaches

Despite efforts to unpack malware for analysis, completely reversing certain malware to its original state proves challenging, particularly in pinpointing crucial elements such as OEP and IAT, which vary with the obfuscation tool employed [5,17,21]. As a result, the reconstructed binaries might not functionally match their original counterparts, heightening the likelihood of deobfuscation failures.

4.5.5. Deobfuscation Challenges with Obfuscated Malware by ASProtect

We conducted deobfuscation experiments on malware, focusing on samples obfuscated with Themida, VMProtect, and ASProtect, sourced from the VirusShare dataset. Unfortunately, we were unable to deobfuscate any of the samples obfuscated with ASProtect. The main challenge was the difficulty in accurately identifying the ASProtect version used in the malware. Pinicorn analyzed the obfuscation techniques applied in ASProtect version 2.78, but like many obfuscation tools, older version of ASProtect could potentially use completely different obfuscation patterns.

4.5.6. Obsidium Malware Sample Were Not Found

We used malware samples from VirusShare, specifically targeting PE samples, to identify packing with *PyPackerDetect*. While this tool successfully detected malware packed by Themida, VMProtect, and ASProtect, we did not find any samples packed with Obsidium. Consequently, we did not conduct experiments on Juliet Test Suite samples protected by Obsidium.

4.5.7. Application to Other Operating Systems

Our study primarily focuses on Windows-based malware and obfuscation techniques. However, it is important to consider the applicability of our proposed methods to other operating systems such as Linux and macOS. Each operating system has unique characteristics and environments that can result in differences in the application process. Firstly, we used the Intel Pin DBI framework. Intel Pin operates only in Intel CPU-based environments and does not work on other architectures such as ARM. However, alternative frameworks like DynamoRIO are available for these environments. Secondly, we analyzed the anti-analysis techniques used in Windows operating system and implemented modules to bypass them. However, the anti-analysis techniques used in Linux and macOS can vary. For example, Linux malware might leverage *ptrace*-based anti-debugging techniques, requiring different bypass strategies compared to those used for Windows-based packers. Thirdly, we focused on analyzing the technique used by protectors that apply packing and obfuscation to PE binaries in the Windows operating systems. Packers targeting ELF and Mach-O binaries may employ different forms of obfuscation techniques. Addressing these would require additional analysis. In conclusion, while our current approach demonstrates potential for Windows-based malware, applying and extending it to other operating systems will require further work. Such efforts are expected to enhance the ability to unpack and analyze malware across diverse operating system environments.

4.6. Out-of-Scope

4.6.1. Limited Scope of Examined Commercial Protectors

Our study focuses on four prevalent packers, Themida, VMProtect, ASProtect, and Obsidium [11], selected for their consistent updates and implementation of OEP and API obfuscation techniques. While these packers offer unique obfuscation strategies, addressing unknown packers and their techniques remains a formidable challenge. Ongoing efforts are needed to analyze and deobfuscate a wider range of packers to effectively counteract unidentified malwares.

4.6.2. Macro-Based Source Code Obfuscation

The packers in our study employ various obfuscation techniques, including source code level obfuscation prior to binary compilation, a method known as *macro-based obfus-*

cation [10]. This includes techniques like *code virtualization*, *string obfuscation*, and *resource obfuscation*. However, our research primarily focuses on the trampoline code-based obfuscation of OEP and API, excluding macro-based source-level obfuscation from our purview.

5. Conclusions

Recent research on deobfuscation has primarily targeted API obfuscation techniques that complicate the reconstruction of the import table. However, these efforts have not successfully reconstructed import tables obfuscated using trampoline codes that are sensitive to the API's arguments. Moreover, the lack of research on OEP obfuscation techniques, combined with their application in malware, has led to failures in unpacking such obfuscated malware. To address these gaps, this paper analyzes the OEP and API obfuscation techniques employed by various packers and introduces an effective deobfuscation approach.

Our proposed system leverages dynamic binary instrumentation tools and emulators to identify information obfuscated by trampoline codes sensitive to API arguments. It also discerns the presence or absence of trampoline code calls among OEP candidates and identifies deleted OEP instructions. This approach enables the deobfuscation and unpacking of malware obfuscated through trampoline codes. However, as unknown packers may employ different techniques for OEP and API obfuscation, and given the potential for obfuscating a wide range of information beyond OEP and API, challenges in malware analysis are likely to continue.

Limitations and Future Works

Firstly, Pinicorn is designed to primarily unpack malware that has been protected using prevalent packers such as Themida, VMProtect, ASProtect, and Obsidium. Should Pinicorn encounters malware packed with different tools, it may not successfully unpack it. Consequently, it is essential to identify the packing tool used on the malware to effectively utilize Pinicorn. Additionally, Pinicorn targets deobfuscation techniques primarily focused on OEP obfuscation and API obfuscation. Like other unpackers, Pinicorn may fail to unpack if it encounters obfuscation techniques beyond its designed capabilities. We note that unpacking unknown and obfuscated malware is a challenging problem, and the limitation is common in dynamic analysis-based approaches. We leave these limitations as future work.

Author Contributions: Conceptualization, M.K.; methodology, M.K.; software, G.L. and M.K.; validation, G.L.; formal analysis, G.L. and M.K.; investigation, G.L. and M.K.; resources, M.K.; data curation, M.K.; writing—original draft preparation, M.K.; writing—review and editing, M.K. and H.C.; supervision, J.H.Y. and H.C.; project administration, J.H.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government, Ministry of Science and ICT (MSIT) (No. 2017-0-00168, Automatic Deep Malware Analysis Technology for Cyber Threat Intelligence).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. AV-TEST—The Independent IT-Security Institute. Malware Statistics and Trends Report. 2024. Available online: https://www.av-test.org/en/statistics/malware (accessed on 1 April 2023).
2. FireEye. M-Trends 2020 Report. 2021. Available online: https://www.mandiant.com/sites/default/files/2021-09/mtrends-2020.pdf (accessed on 1 April 2023).
3. FireEye. M-Trends 2021 Report. 2022. Available online: https://services.google.com/fh/files/misc/rpt-mtrends-2021-en.pdf (accessed on 1 April 2023).

4. FireEye. M-Trends 2022 Report. 2023. Available online: https://services.google.com/fh/files/misc/m-trends-report-2022-en.pdf (accessed on 1 April 2023).
5. Suk, J.; Lee, J.; Jin, H.; Kim, I.; Lee, D. UnThemida: Commercial obfuscation technique analysis with a fully obfuscated program. *Softw. Pract. Exp.* **2018**, *48*, 2331–2349. [CrossRef]
6. Cheng, B.; Li, P. BareUnpack: Generic Unpacking on the Bare-Metal Operating System. *IEICE Trans. Inf. Syst.* **2018**, *101*, 3083–3091. [CrossRef]
7. Choi, S.; Chang, T.; Kim, C.; Park, Y. X64Unpack: Hybrid emulation unpacker for 64-bit windows environments and detailed analysis results on VMProtect 3.4. *IEEE Access* **2020**, *8*, 127939–127953. [CrossRef]
8. Cheng, B.; Ming, J.; Leal, E.; Zhang, H.; Fu, J.; Peng, G.; Marion, J. Obfuscation-Resilient Executable Payload Extraction From Packed Malware. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), Virtual event, 11–13 August 2021; pp. 3451–3468.
9. VMProtect Software. VMProtect Software Protection. 2003–2023. Available online: https://vmpsoft.com/ (accessed on 1 April 2023).
10. Oreans Technologies. Themida Overview—Oreans Technologies. 2004–2023. Available online: https://www.oreans.com/Themida.php (accessed on 1 April 2023).
11. Obsidium Software. Obsidium Software Protection System. 2023. Available online: https://www.obsidium.de/ (accessed on 1 April 2023).
12. StartForge. ASProtect. 2007–2023. Available online: http://www.aspack.com/asprotect32.html/ (accessed on 1 April 2023).
13. Roundy, K.A.; Miller, B.P. Binary-code obfuscations in prevalent packer tools. *ACM Comput. Surv. (CSUR)* **2013**, *46*, 1–32. [CrossRef]
14. Korczynski, D. Repeconstruct: Reconstructing binaries with self-modifying code and import address table destruction. In Proceedings of the 2016 11th International Conference on Malicious and Unwanted Software (MALWARE), Fajardo, PR, USA, 18–21 October 2016; pp. 1–8.
15. Microsoft. PE Format—Win32 Apps. 2023. Available online: https://learn.microsoft.com/en-us/windows/win32/debug/pe-format/ (accessed on 1 April 2023).
16. Park, J.; Jang, Y.H.; Hong, S.; Park, Y. Automatic detection and bypassing of anti-debugging techniques for microsoft windows environments. *Adv. Electr. Comput. Eng.* **2019**, *19*, 23–29. [CrossRef]
17. Lee, Y.; Suk, J.; Lee, D. Bypassing Anti-Analysis of Commercial Protector Methods Using DBI Tools. *IEEE Access* **2021**, *9*, 7655–7673. [CrossRef]
18. D'Alessio, S.; Mariani, S. *PinDemonium: A DBI-Based Generic Unpacker for Windows Executables*; Black Hat: Las Vegas, NV, USA, 2016 .
19. Kim, G.M.; Park, Y.S. Improved Original Entry Point Detection Method Based on PinDemonium. *KIPS Trans. Comput. Commun. Syst.* **2018**, *7*, 155–164.
20. Isawa, R.; Inoue, D.; Nakao, K. An original entry point detection method with candidate-sorting for more effective generic unpacking. *IEICE Trans. Inf. Syst.* **2015**, *98*, 883–893. [CrossRef]
21. Kang, M.; Poosankam, P.; Yin, H. Renovo: A hidden code extractor for packed executables. In Proceedings of the 2007 ACM Workshop on Recurring Malcode, Alexandria, VA, USA, 2 November 2007; pp. 46–53.
22. Yadegari, B.; Johannesmeyer, B.; Whitely, B.; Debray, S. A generic approach to automatic deobfuscation of executable code. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 674–691.
23. Ugarte-Pedrero, X.; Balzarotti, D.; Santos, I.; Bringas, P.G. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 659–673.
24. Cheng, B.; Ming, J.; Fu, J.; Peng, G.; Chen, T.; Zhang, X.; Marion, J. 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, Toronto, ON, Canada, 15–19 October 2018; pp. 395–411.
25. Cheng, B.; Leal, E.A.; Zhang, H.; Ming, J. On the feasibility of malware unpacking via hardware-assisted loop profiling. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, USA, 9–11 August 2023; pp. 7481–7498.
26. Levi, O. Pin—A Dynamic Binary Instrumentation Tool. 2007–2023. Available online: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html (accessed on 1 April 2023).
27. Quynh, N.; Vu, D. Unicorn—The Ultimate CPU Emulator. 2015–2023. Available online: https://www.unicorn-engine.org/ (accessed 1 April 2023).
28. National Institute of Standards and Technology. Juliet Test Suites. 2006–2023. Available online: https://samate.nist.gov/SARD/test-suites/ (accessed on 1 April 2023).
29. VirusShare. 2012–2022. Available online: https://virusshare.com/ (accessed on 1 April 2023).
30. PyPackerDetect. 2018–2023. Available online: https://github.com/cylance/PyPackerDetect (accessed on 1 April 2023).
31. Scylla—x64/x86 Imports Reconstruction. 2011–2015. Available online: https://github.com/NtQuery/Scylla (accessed on 1 April 2023).
32. Unipacker. 2019–2023. Available online: https://github.com/unipacker/unipacker (accessed on 1 April 2023).

33. The UPX Team. UPX—The Ultimate Packer for eXecutables. 1996–2023. Available online: https://upx.github.io/ (accessed on 1 April 2023).
34. StarForge. ASPACK Software. 2007–2023. Available online: http://www.aspack.com/aspack.html (accessed on 1 April 2023).