Practice-Oriented Teaching of Malware Analysis: A Course Design and Implementation Study

Jing Wang[1], Zhi Lin[2,*]

[1]School of Mathematics & Computer, Wuhan Polytechnic University, Wuhan, China
[2]School of Electronic Information Engineering, Wuhan Donghu College, Wuhan, 430212, China
*Corresponding author's e-mail: linzhi@wdh.edu.cn

Abstract. To address the gap between theoretical instruction and practical skills in cybersecurity education, this paper presents a practice-oriented teaching framework for a malware analysis course. The framework follows a progressive pipeline—"environment setup → basic analysis → in-depth reverse engineering → comprehensive practice (hands-on combat)"—and employs four representative malware categories: registry persistence, path hijacking, self-modifying code, and encryption/packing. The curriculum integrates IDA Pro static reverse engineering, dynamic debugging, and ANTLR-based program analysis techniques, establishing a complete technical chain from control-flow graph construction and data-flow tracing to symbolic-execution pre-analysis. The paper elaborates on the principles, critical operational steps, and technical nuances of combining static and dynamic analysis in each case study. Teaching practice demonstrates that this approach effectively enhances students' malware analysis capabilities and adversarial thinking in cyber defense.

Keywords: Malware Analysis, Practice-oriented Teaching, Reverse Engineering, Control-flow Analysis, Symbolic Execution.

1 Introduction

As cyberattack methods continue to evolve, malware has become a primary vehicle threatening critical information infrastructure. According to statistics from the Qi'anxin Threat Intelligence Center, the number of new Windows malware samples in 2025 increased by 23.7% year-over-year, with persistence techniques present in over 60% of samples—registry Run keys, scheduled tasks, service installations, and IFEO hijacking being widely adopted across various families. Cultivating practical malware analysis talent is a core mission of cybersecurity education. However, traditional instruction tends to emphasize theoretical lectures, leaving students with insufficient hands-on experience in analyzing real-world samples and developing analytical tools [1]. Therefore, building a practice-oriented malware analysis teaching system holds significant practical value.

© The Author(s) 2026
M. B. Badzis et al. (eds.), *Proceedings of the 2026 2nd International Conference on Education Research and Training Technologies (ERTT 2026)*, Advances in Social Science, Education and Humanities Research 1042,
https://doi.org/10.2991/978-2-38476-628-4_11

2 Course Objectives and Overall Design

2.1 Teaching Objectives

This course aims to enable students to master the fundamental principles and methods of both static and dynamic malware analysis; gain proficiency in using IDA Pro for reverse engineering; understand program control-flow and data-flow analysis techniques and acquire the ability to extract code features using ANTLR; independently complete analysis of advanced evasive samples such as packed and self-modifying executables; and cultivate practical threat intelligence extraction and attack attribution skills.

2.2 Course Architecture

The course follows a four-stage pedagogical model: "principle lecture → case demonstration → hands-on practice → comprehensive assessment" [2]. The principle-lecture phase covers essential fundamentals including PE file format, function calling conventions, exception handling mechanisms, anti-debugging techniques, and packing principles. The case-demonstration phase involves the instructor walking through typical samples to illustrate the complete analysis workflow and key technical points. The hands-on practice phase requires students to independently analyze similar samples and write reports. The comprehensive assessment phase presents unknown samples to evaluate students' overall competence.

3 Laboratory Environment Setup

The laboratory environment adopts an isolated "host + virtual machine" architecture. The host machine runs Windows 10 and is equipped with IDA Pro 8.4 Free (including built-in debugger), the ANTLR 4.0 toolchain, and Python 3.9. The virtual machine runs Windows 7 with Python installed, serving as the execution environment for samples and configured with snapshot mechanisms for rapid recovery. During analysis, the host handles static reverse engineering and code parsing, while the virtual machine executes samples dynamically. The two communicate through a Host-Only network to ensure a secure and controllable analysis process.

4 Representative Case Studies

4.1 Case 1: Static Detection of Registry Persistence Malware

Registry persistence is a classic technique by which malware maintains its foothold—modifying startup entries such as Run, RunOnce, and Services to auto-execute upon system boot [5]. The goal of static analysis is to identify persistence behavior without executing the program. This case introduces ANTLR for assembly-code parsing and

feature extraction, allowing students to grasp program analysis fundamentals alongside IDA Pro operation [3].

The experiment begins by loading the target sample in IDA Pro and inspecting the import table to locate registry-related APIs such as RegOpenKeyEx, RegSetValueEx, and RegCreateKeyEx. If these APIs are not directly present in the import table, the possibility of dynamic loading should be considered, and cross-references should be searched for relevant strings. IDA Pro's Graph View is then used to generate function-level control-flow graphs to identify key branch structures [7]. For example, in a sample modifying the Run key, the code typically checks the return value after RegOpenKeyEx; if successful, it branches into the RegSetValueEx path; otherwise, it may attempt an alternate key path. Under different conditional branches, register states change accordingly—EAX holds the return value, EIP points to the current instruction address, and ESP/EBP maintain the stack frame—all of which provide constraint conditions for subsequent symbolic execution analysis.

The next step involves ANTLR-based data-flow analysis. The assembly code exported from IDA is fed into the ANTLR parser to generate an abstract syntax tree (AST). A custom Listener or Visitor traverses the AST to extract critical instruction sequences and trace the propagation path of registry key names and values—from string constants or dynamically constructed buffers to the API parameter registers. By symbolically marking different branch paths, a mapping between program states and malicious behavior can be established. This step introduces students to the core idea of symbolic execution—symbolizing variables to explore multiple execution paths without actually running the program. Finally, based on the above analysis, static detection signatures for registry persistence are summarized, and YARA rules are written for rapid screening.

4.2 Case 2: Combined Static-Dynamic Analysis of IFEO Path Hijacking Malware

Image File Execution Options (IFEO) hijacking is a stealthy persistence technique on Windows systems. Attackers set a "Debugger" value under the IFEO registry key for a target process, redirecting execution of legitimate programs (e.g., calc.exe, notepad.exe) to a malicious file [5]. This technique is widely used in banking trojans such as Emotet and Qakbot.

The experiment begins by loading the sample in IDA Pro, identifying APIs like RegSetKeyValue and RegCreateKeyEx via the import table, and searching for the strings "Debugger" and "Image File Execution Options" to locate the relevant code region. The sample is also checked for anti-debugging code, which is bypassed if necessary during subsequent dynamic debugging [7]. The sample is then launched under the IDA debugger, and breakpoints are set at the system-call layer for RegOpenKeyEx and RegSetKeyValue—specifically, conditional breakpoints at ntdll!ZwSetValueKey are used to trigger when the key name contains "Debugger", allowing observation of parameters and recording of the target process name and malicious file path.

After dynamic analysis, the key addresses recorded at runtime are mapped back to the static disassembly view, cross-referencing the context to determine whether the sample writes the IFEO key only under certain conditions and whether it employs evasion mechanisms such as time delays or registry existence checks. During analysis, Python scripts leveraging the ctypes library can be used to programmatically monitor registry modifications, cross-validating the debugger observations. Finally, the list of target process names, malicious file paths, and file hashes are extracted to form a technical fingerprint of the family [5], culminating in a comprehensive analysis report.

4.3 Case 3: Dynamic Detection of Self-Modifying Code

Self-modifying code evades signature-based static detection and some dynamic analysis by rewriting its own instruction sequences at runtime [6]. Its typical technical characteristics include: code sections initially having read-write-execute permissions, or using VirtualProtect to change read-only permissions to writable and executable; instruction sequences that write data to code-section addresses; and control flow subsequently jumping to the modified code region to execute dynamically generated instructions. This technique is commonly found in metamorphic viruses and advanced persistent threat payloads.

The experiment begins by loading the sample in IDA Pro and inspecting section attributes via the section view. If a section is simultaneously writable and executable, or if the code section has unusual initial attributes, it is marked as suspicious. Calls to memory-permission modification APIs such as VirtualProtect and VirtualAlloc are searched for [7], and the disassembly is examined for instructions like call $+5 / pop reg—which are commonly used to calculate the current EIP address and often precede self-modifying code. The sample is then launched under the IDA debugger, and a hardware memory-write breakpoint is set on the starting address of the target code section. Single-stepping continues until the breakpoint triggers, recording the instruction address that triggered the write, the target address, and the data value.

After the breakpoint triggers, the key event chain is traced: first, observe the call to VirtualProtect changing code-section permissions from read-only to writable and executable; then, trace the mov instructions that write new bytes into the code section; finally, record the control-flow transfer (e.g., jmp, call, or ret) that jumps to the modified code region. At each control-flow transfer, the target EIP address is recorded to confirm it falls within the modified memory range, fully reconstructing the "write-modify-jump" execution chain [6]. The modified code captured dynamically is dumped from memory and compared with the original static disassembly to analyze the true functionality hidden by the self-modification. Since self-modifying samples may incorporate anti-debugging mechanisms, IDA's debugger hiding features should be employed during debugging.

4.4 Case 4: Analysis and Unpacking of Encrypted/Packed Code

Packing encrypts, compresses, or obfuscates the original instructions, making static disassembly unable to reveal the program logic directly [5]. The core of unpacking lies in the fact that the unpacking stub first decrypts the original code into memory at runtime and then transfers control to the original entry point (OEP). The goal of dynamic debugging is to dump and repair the in-memory original code precisely at the moment the OEP is reached.

The experiment begins with Detect It Easy to identify the packer type. If identification fails, section names and section entropy are manually examined—entropy values approaching 8.0 indicate highly random data, suggesting encryption or compression [7]. The sample is then loaded in IDA Pro in debug mode. The packer entry point typically exhibits structures like pushad/pushfd to save register context, followed by a call to the decryption function [5]; analyzing the entry code reveals the structural characteristics of the decryption loop. OEP localization can employ multiple methods: using IDA's Trace functionality to record execution trajectories and locating the original entry via stack-backtracking; setting breakpoints on APIs such as VirtualProtect and GetProcAddress, since the packer inevitably calls these during import table decryption; and watching for the popad/popfd sequence, after which a following jmp or ret typically points to the OEP [7].

After pausing execution at the OEP, IDA's Python scripting interface is used to write a memory-dump script that exports the critical section data from the current process memory space. The exported binary is then subjected to import table reconstruction by analyzing call instruction offsets and correlating them with resolved function addresses in memory to reverse-engineer the IAT. The repaired file is re-analyzed statically to verify successful unpacking—the decompiler should now display C/C++ pseudocode normally, allowing focused analysis of core malicious functionalities such as network communication protocols, C2 address extraction algorithms, and file encryption behavior [5].

5 Teaching Outcomes and Reflections

After two rounds of teaching practice involving 72 undergraduate cybersecurity majors, assessment data show that: 92% of students independently completed the static analysis in Case 1 and wrote YARA rules; 87% mastered combined static-dynamic debugging methods and completed full IFEO hijacking analysis in Case 2; 81% successfully traced self-modifying code in Case 3; and 76% completed the unpacking and post-unpacking analysis in Case 4. Several outstanding students developed detection tools that have been piloted in campus network threat monitoring. Student satisfaction with the course's practical orientation scored 4.5/5.0 [2].

Reflections on teaching identified the following areas for improvement: students' ability to analyze non-x86 architectures (e.g., ARM) needs strengthening, and an IoT malware module will be introduced in the future; insufficient training time for combined static-dynamic debugging skills warrants additional dedicated lab sessions; unpacking remains challenging for some students, and preliminary exercises with

basic packers such as UPX and ASPack should be added; and an inter-institutional sample-sharing mechanism should be established under legal and ethical frameworks to enrich the teaching material pool [4], along with incorporating security analysis content for domestic operating systems.

6 Conclusion

The practice-oriented malware analysis teaching framework, through its progressive "environment + fundamentals + depth + combat" design and its use of four representative scenarios—registry persistence, IFEO path hijacking, self-modifying code, and encryption/packing—integrating IDA Pro static reverse engineering, dynamic debugging, and ANTLR-based program analysis, effectively bridges the gap between theory and practice [1][3]. The four cases form a graded training system that progressively enhances students' reverse-engineering capabilities. Future work will continue to introduce cutting-edge technologies such as AI-assisted reverse engineering and automated symbolic execution to cultivate more cybersecurity professionals with practical combat readiness.

References

1. Xiao Xinguang, Wang Yaohua, Gao Xibao, et al. Laboratory Manual for Malware Analysis and Detection Techniques [M]. Beijing: Publishing House of Electronics Industry, 2025.
2. Wang Zhi, Jia Chunfu. Laboratory Environment Construction and Experiment Design for Malware and Its Defense Techniques Course [J] // Compute Education, 2009(18): 140-142.
3. Zhang Lei, Li Jie, Liu Liang. An Experimental Design for Cross-Language Malware Detection [J]. Experimental Science and Technology, 2025, 23(2): 112-118.
4. Sanders A, Walia G S, Allen A. Analysis of Software Vulnerabilities Introduced in Programming Submissions Across Curriculum [C] // Proceedings of IEEE Frontiers in Education Conference (FIE). Washington D.C.: IEEE, 2024: 1-9.
5. Sikorski M, Honig A. Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software [M]. San Francisco: No Starch Press, 2012.
6. Messahel W, Touili T. Analyzing a Concurrent Self-Modifying Program: Application to Malware Detection [C] // Proceedings of the 11th International Conference on Information Systems Security and Privacy (ICISSP). Porto: SCITEPRESS, 2025: 176-182.
7. Eagles M. The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler [M]. 2nd ed. San Francisco: No Starch Press, 2021.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial 4.0 International License (http://creativecommons.org/licenses/by-nc/4.0/), which permits any noncommercial use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

