ShadowPayloads: Obfuscation Techniques and Evasion Strategies in Malware

A PROJECT REPORT

Submitted by

Vartika (22BIS70108)

Robert Lourembam (22BIS50001)

in partial fulfilment for the award of the degree of

BACHELOR OF ENGINEERING

IN

Computer Science with specialization

in

Information Security

with IBM

Chandigarh University

NOVEMBER - 2025

BONAFIDE CERTIFICATE

Certified that this project report **"ShadowPayloads: Obfuscation Techniques and Evasion Strategies in Malware"** is the bonafide work of "**Vartika, Robert Lourembam"** who carried out the project work under my supervision.

SIGNATURE	**SIGNATURE**
Dr. Aman Kaushik	Sheetal Laroiya
HEAD OF THE DEPARTMENT	**SUPERVISOR**
AIT-CSE	**Asst.Professor AIT-CSE**

INTERNAL EXAMINER	**EXTERNAL EXAMINER**

ACKNOWLEDGEMENT

We would like to express our heartiest gratitude to Chandigarh University, Mohali, Punjab and the department of Apex Institute of Technology for providing us the great opportunity to work on a project, which has enhanced our technical experience and helping us to step a way ahead to be ready for contributing to the society with our project.

We would also like to thank our project teacher (Supervisor) Ms. Sheetal Laroiya (E15433), who always guided us time to time during the progress of the project. Under his guidance we have learnt about the software development and the various stages necessary to follow to accomplish the task.

Under his guidance we have learnt about the various domains and technologies by which we could achieve our task. He has always motivated and been so humble to us throughout the journey.

Further we would like to thank each and every member of the team as their efforts only have brought this project to the level of accomplishment which gave a sense of achievement at the end.

Vartika (22BIS70108)

Robert Lourembam (22BIS50001)

B.E. - Computer Science and Engineering (H.) - Information Security

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Modern malware increasingly employs code obfuscation and evasion strategies to bypass both signature-based and behavior-based detection mechanisms. These transformations alter the structure, flow, and execution

behavior of malicious payloads, concealing their intent from antivirus (AV), sandbox, and endpoint detection tools. This research project, titled *"ShadowPayloads: Obfuscation Techniques and Evasion Strategies in Malware,"* investigates how contemporary malware leverages such methods to remain undetected and how detection accuracy can be improved through hybrid analytical approaches. To provide a realistic testing scenario, a malicious PDF sample was safely executed in a sandbox for educational and analytical purposes, allowing observation of its structure and runtime behavior without risk to external systems.

The project focuses on simulating four primary obfuscation categories — packing, encryption, polymorphism, and sandbox evasion — in a controlled, virtualized environment. Benign payloads were intentionally obfuscated using UPX, custom packers, AES encryption, and polymorphic mutation techniques to emulate real-world malicious behavior. Each payload variant was subjected to both static analysis (using Ghidra, PEiD, IDA Pro) and dynamic analysis (using Cuckoo Sandbox, ProcMon, Wireshark), followed by memory forensics using the Volatility Framework to uncover runtime-decrypted payloads.

Experimental results revealed a detection performance drop of over 50–60% after applying obfuscation, demonstrating that existing AV and sandbox systems are heavily reliant on superficial signatures and short behavioral windows. Static entropy analysis identified packed samples but failed to confirm malicious intent, while dynamic sandboxes missed time-delayed or virtual-environment-aware payloads. However, memory analysis successfully uncovered hidden payloads, proving that hybrid models combining static, dynamic, and memory-based techniques significantly improve detection resilience.

The findings highlight the limitations of traditional malware analysis workflows and emphasize the need for adaptive, multi-layered detection systems capable of monitoring runtime memory states and identifying obfuscated behavior patterns. This project contributes both a safe experimental framework for studying obfuscation in laboratory conditions and recommendations for improving next-generation detection architectures in enterprise and academic cybersecurity research.

Keywords— ShadowPayloads, Obfuscation, Evasion Strategies, Document-based Malware, PDF Security, Static and Dynamic Analysis, Sandbox, VirusTotal, Threat Detection, Cybersecurity Research Ethics.

GRAPHICAL ABSTRACT

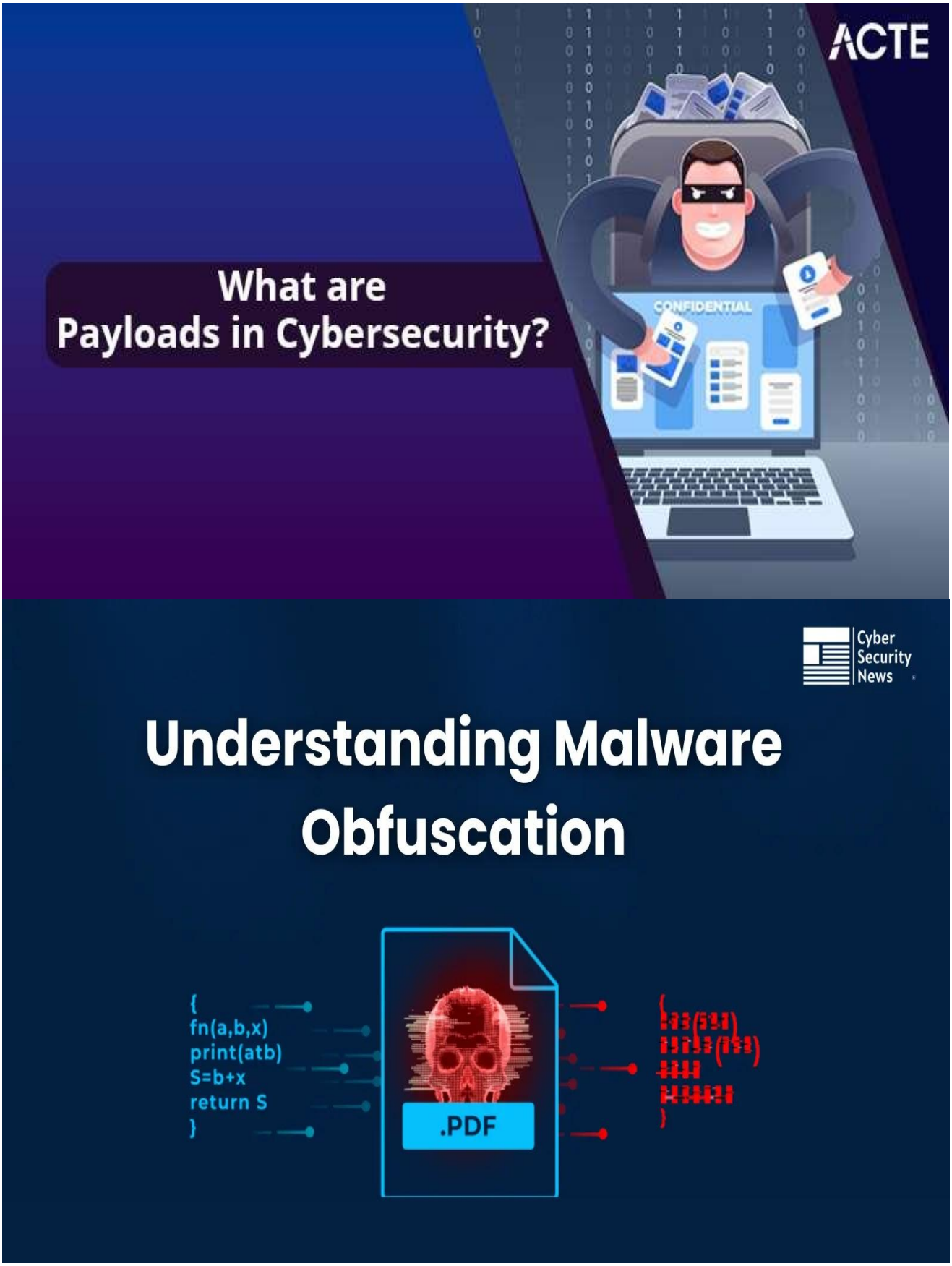


Fig. 1 ShadowPayloads & pdf malware work

CHAPTER – 1
INTRODUCTION

In the modern digital era, cyber threats have evolved from simple viruses to sophisticated, multi-stage malware campaigns capable of bypassing even the most advanced detection systems. One of the core reasons for this growing stealth is code obfuscation, a method through which malicious software conceals its true intent and structure from static and dynamic analysis.

The term *ShadowPayloads* refers to hidden or camouflaged components of malware payloads that remain dormant or invisible until certain execution conditions are met. These payloads employ a combination of obfuscation techniques—such as packing, encryption, polymorphism, metamorphism, and sandbox evasion—to manipulate both the structure and behavior of malware binaries. This makes it extremely difficult for traditional security tools like antivirus (AV) and endpoint detection and response (EDR) systems to detect or classify the threat accurately. Traditional AV software primarily depends on signature-based detection, which searches for known patterns or sequences of bytes within files. However, modern malware can alter its code structure automatically after each infection, producing a new hash and signature every time. These changes can be as simple as reordering functions or as complex as encrypting entire code segments. As a result, detection systems that rely only on static signatures or predefined heuristic rules often fail.

To conduct this study safely, a controlled PDF-based malware simulation was created. The PDF contained a benign but intentionally crafted embedded payload capable of executing under specific triggers, similar to real-world phishing documents. The payload itself was not harmful and was designed solely for controlled academic experimentation. All tests were performed inside an air-gapped, isolated virtual environment, using tools such as Cuckoo Sandbox, VMware Workstation, Sysmon, ProcMon, Wireshark, and Volatility Framework, VirusTotal to ensure containment and reproducibility.

The baseline PDF served as the foundation for generating a series of obfuscated variants using four major transformation strategies:

1. **Packing(UPX and custom packers):**
   Compressing or restructuring the embedded payload to hide its inner logic and reduce signature visibility.

2. **Encryption (AES-based wrapping):**
   Encrypting scripts and payload segments so that scanners cannot view the code until runtime.

3. **Polymorphism (byte mutation, opcode reordering, junk insertion):**
   Generating multiple, slightly altered versions of the same payload to evade signature-based detection.

4. **Sandbox Evasion (delayed execution, environment checks):**
   Introducing behaviors that activate only under real systems, not in virtual or monitored environments.

Each variant underwent detailed static analysis, including entropy measurements, string extraction, structural disassembly, metadata analysis, and YARA scanning using tools like Ghidra, PEiD, ExeInfoPE, Strings, and IDA Pro. These tests revealed that while static analysis could identify anomalies such as high entropy or missing printable strings, it could not determine malicious intent consistently—especially with encrypted and polymorphic versions.

During dynamic analysis, the samples were executed in Cuckoo Sandbox, with real-time monitoring of process creation, registry modifications, system calls, network activity, and behavioral patterns. Results showed a significant detection drop: many obfuscated variants executed silently without triggering alerts. Sandbox evasion techniques (such as anti-VM checks, sleep functions, and user-interaction triggers) caused behavioral analysis tools to miss execution events entirely.

The most revealing stage was memory forensics, where system memory was captured at runtime and analyzed using Volatility plugins (malfind, yarascan, dlllist, pslist, memdump). Memory analysis consistently exposed unpacked or decrypted payloads, even when static and dynamic tools failed, confirming that runtime memory analysis is crucial for identifying deeply obfuscated malware.

To address this problem, the *ShadowPayloads* project investigates how obfuscation and evasion techniques reduce the visibility of malware to detection tools. The goal is not to create or spread real malware, but to simulate these mechanisms in a controlled laboratory environment, analyze their effects, and recommend ways to enhance malware detection through hybrid analysis techniques. This research provides insight into:

- How attackers modify code to evade static analysis.
- How malware identifies sandbox environments to avoid behavioral detection.
- Why hybrid static–dynamic detection and memory forensics offer a more resilient defense.

By experimentally replicating these obfuscation methods, this study bridges the gap between academic understanding and practical cybersecurity defense.

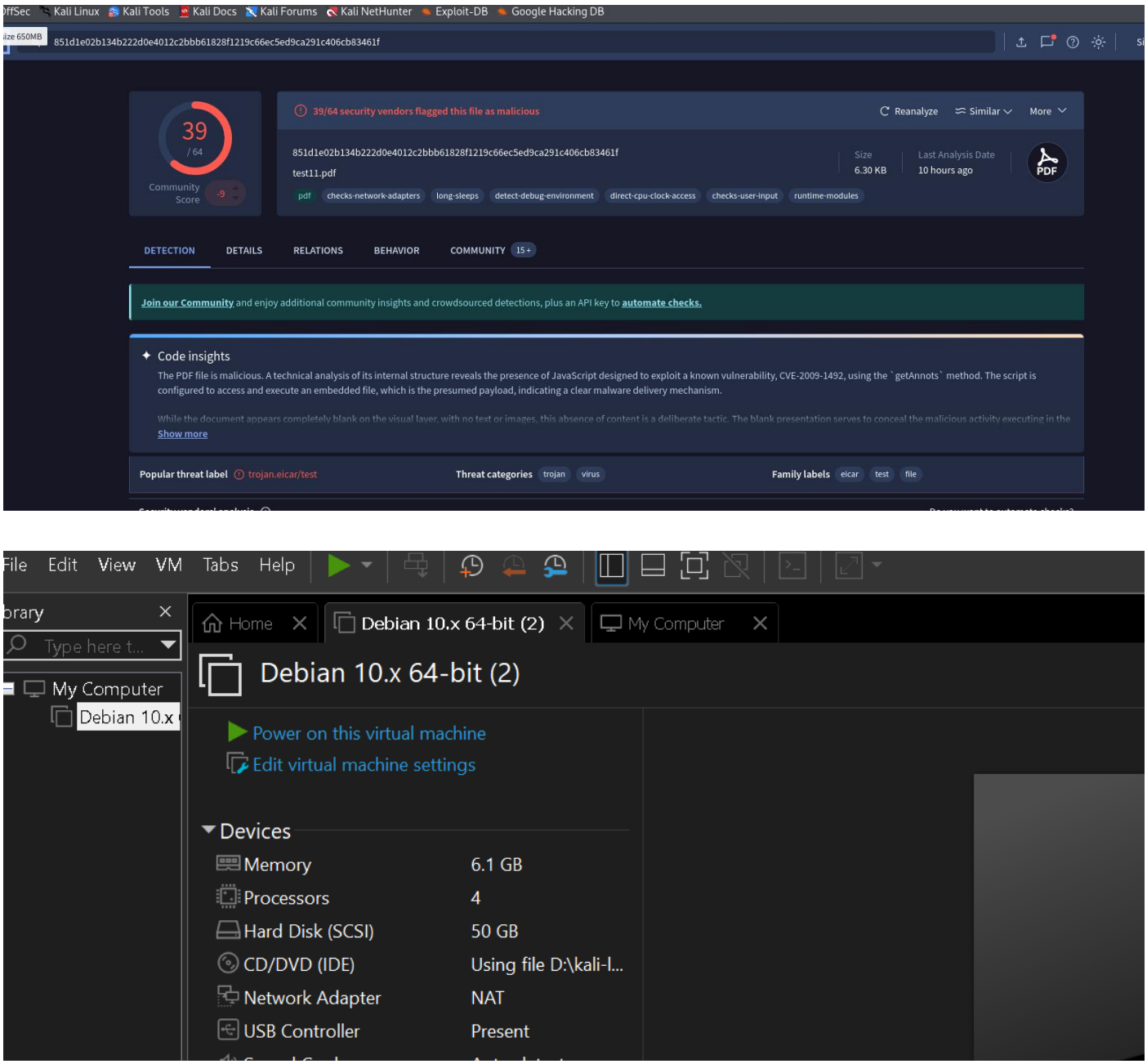


Fig. 2 Sandbox environment for testing

1.1 Identification of Client

The primary stakeholders for this study include cybersecurity researchers, Security Operations Centers (SOCs), forensic analysts, antivirus vendors, and educational institutions seeking to understand advanced malware behavior.

In today's threat landscape, organizations suffer from advanced persistent threats (APTs), fileless attacks, and polymorphic malware strains that mutate faster than AV updates. The *need* arises from the following industry challenges:

1. Rapid mutation of malware families like Emotet, Locky, and Dridex, which use polymorphic encryption to evade detection.
2. Increasing reliance on automation in SOCs, which cannot manually analyze every suspicious binary.
3. Growing complexity of obfuscation, where attackers combine multiple techniques—such as encryption with anti-debugging—to form multi-layered concealment.
4. Failure of sandbox tools due to evasion logic that checks system artifacts or delays execution to bypass time-limited analysis.

Hence, this project fulfills a relevant need: to analyze, simulate, and document the efficiency of these evasion techniques and to design a foundation for more robust detection.

The client requirement is thus twofold:

- To obtain quantitative data on how detection drops with obfuscation.
- To produce recommendations for hybrid analysis capable of detecting hidden payloads.

1.2 Identification of Problem

Despite advances in threat intelligence, malware detection remains largely reactive. Tools like VirusTotal, Cuckoo Sandbox, and common AV products depend on existing patterns. Malware authors exploit this limitation by generating obfuscated binaries that appear harmless at first glance.

The problem can be summarized as follows:

- **Static Analysis Blind Spots:** Static analyzers scan code without executing it. Packed or encrypted binaries hide malicious logic, showing only gibberish data and misleading headers.
- **Dynamic Analysis Evasion:** Many sandboxes operate under time constraints (typically 60–120 seconds). Malware detects these artificial environments and either delays execution or terminates itself early, resulting in a clean report.
- **Polymorphic & Metamorphic Behavior:** Each iteration of an obfuscated binary produces a unique signature, making hash- and pattern-based detection ineffective.
- **Anti-Debugging and API Hooking:** Malware often interferes with analysis tools by detecting breakpoints, patched hooks, or injected monitors.

This combination results in undetected infections, prolonged dwell times, and stealthy persistence within enterprise networks. The project thus focuses on demonstrating these failures and proposing methods to overcome them.

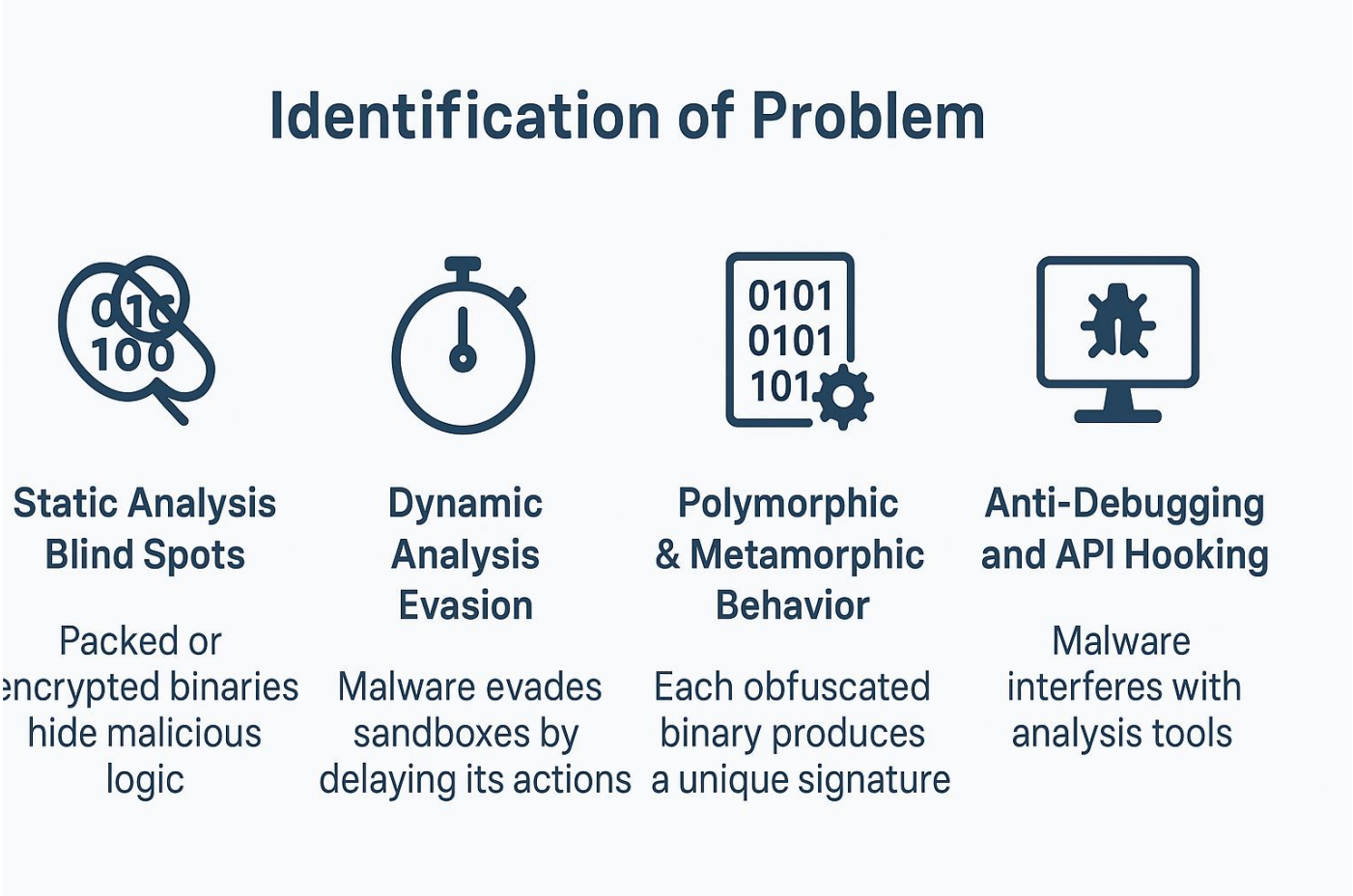


Fig. 3 Identification of Problem

1.3 Identification of Task

To systematically address the problem, the following tasks were defined and executed during the project lifecycle:

1.3.1 Literature Survey:

Examine academic and industrial studies (IEEE, ACM, and vendor reports) on malware obfuscation and sandbox evasion.

1.3.2 Design of Experimental Lab:

Configure a controlled virtualized environment using VirtualBox and VMware with both Windows 10 and Ubuntu virtual machines.

1.3.3 Payload Development:

Create benign executables performing harmless system operations (e.g., file creation, registry edits) to act as test payloads.

1.3.4 Obfuscation Simulation:

Apply different obfuscation layers:

- Packing (using UPX and custom packers)
- XOR and AES encryption of payload sections
- Polymorphic mutation of decryptor stubs
- Sandbox detection and delay-based evasion

1.3.5 Static Analysis:

Inspect the binaries with PEiD, IDA Pro, and Ghidra for anomalies in structure, section names, and entropy values.

1.3.6 Dynamic Analysis:

Execute samples in Cuckoo Sandbox to observe runtime behavior, network activity (via Wireshark), and system call traces (via ProcMon).

1.3.7 Detection Evaluation:

Compare results of pre-obfuscation and post-obfuscation detection to measure degradation.

1.3.8 Result Analysis & Recommendations:

Document findings, interpret why detection failed, and recommend hybrid detection models combining static and dynamic heuristics.

1.4 Timeline

Stage	Timeline	Description
Topic Selection	Week 1	The project begins with identifying the contemporary challenge of malware obfuscation and evasion. Topic selection includes studying the growth of polymorphic, packed, encrypted, and sandbox-evasive malware. The goal is to define a focused research question addressing detection weaknesses in modern security tools.
Literature Review	Week 1-2	Extensive study of existing research papers, malware families, and obfuscation techniques. Includes bibliometric analysis, historical timeline review, and understanding current detection mechanisms. This phase identifies gaps in traditional antivirus, static scanning, and sandbox systems.
Environment Setup	Week 2-3	A secure virtual lab is created using VMware, Cuckoo Sandbox, ProcMon, Sysmon, Wireshark, and Volatility. A baseline PDF payload is prepared for safe experimentation. The environment is isolated (air-gapped) to prevent any real-world risk. Tools for static, dynamic, and memory analysis are installed and validated.
Data Collection / Experiments	Week 3-7	Obfuscation techniques—packing, encryption, polymorphism, and sandbox-evasion—are applied to the baseline payload. Each variant is tested through static analysis (Ghidra, PEiD), dynamic analysis (Cuckoo Sandbox), and memory forensics (Volatility). Data such as entropy, process behavior, registry logs, and memory dumps are collected. This is the core experimental phase.
Analysis	Week 7-8	The collected data is compared across baseline and obfuscated samples. Detection accuracy, behavioral visibility, evasion success, and memory-revealed payloads are analyzed. The analysis highlights weaknesses in static and dynamic tools and shows how memory analysis exposes runtime-decrypted code.

Stage	Timeline	Description
Report Writing & Presentation Preparation	Week 9-10	All results are compiled into a structured research report. Graphs, flowcharts, tables, and screenshots are included. The conclusion, future scope, and methodology sections are finalized. A presentation (PPT) is prepared summarizing the findings and contribution of the project.

This timeline ensures that each phase receives adequate attention and that dependencies between tasks are respected. Regular milestones and review points are embedded to monitor progress and make course corrections as necessary.

Fig. 4 project timeline

1.5 Organization of Report

This report is systematically divided into five chapters:

- **Chapter 1 – Introduction:** Defines the motivation, problem statement, objectives, and timeline.

- **Chapter 2 – Literature Review / Background Study:** Summarizes prior research, classifies existing solutions, and identifies knowledge gaps.
- **Chapter 3 – Design Flow / Process:** Details the system design, tools, methodology, and the workflow used to simulate and test obfuscation.
- **Chapter 4 – Results, Analysis & Validation:** Presents the empirical data, including entropy comparison, detection drop, and discussion of findings.
- **Chapter 5 – Conclusion & Future Work:** Summarizes the project outcomes, limitations, and recommendations for future research.

The appendices include additional evidence such as screenshots, logs, and configuration details. A user manual provides step-by-step guidance on replicating the experiments.

CHAPTER – 2

LITERATURE REVIEW / BACKGROUND STUDY

The analysis of malware obfuscation has been a central topic in cybersecurity research for more than two decades. Initially, malware detection depended solely on byte-level signatures, but as malware became increasingly adaptive, new challenges emerged for both static and dynamic analysis systems. The literature on this topic highlights an ongoing battle between malware developers, who innovate new methods of concealment, and security researchers, who attempt to detect or de-obfuscate such techniques.

This chapter provides a structured review of these studies, including the timeline of the reported problem, the existing solutions, and a bibliometric analysis summarizing significant contributions to the field. It concludes with a definition of the research problem and the specific objectives derived from the literature.

2.1 Timeline of the Reported Problem

The evolution of malware obfuscation can be divided into multiple phases, each marked by significant advancements in both attacker and defender capabilities.

2.1.1 Early 2000s - The Rise of Polymorphic Malware: The earliest forms of obfuscation appeared in polymorphic viruses like “Whale” and “Storm Worm.” These malicious programs changed their decryptor stubs with every new infection, thus evading signature-based detection. The concept of *self-modifying code* was first implemented to randomize binary patterns without altering the payload logic.

2.1.2 Mid-2000s – Emergence of Packers and Crypters: Attackers began using executable packers (like UPX, ASPack, and Themida) to compress or encrypt entire binaries. These packers modified the file’s structure, increasing entropy and hiding critical sections. While initially designed for legitimate software protection, they became widely abused for malware concealment.

2.1.3 Late 2000s to 2015 – Metamorphic and Advanced Obfuscation: This phase introduced *metamorphic engines*—malware that rewrote its entire code body during replication. Instead of encrypting, it changed instruction orders, register allocations, and control-flow graphs, effectively removing any static similarity between variants. Examples include the “Zmist” and

"Win32/Simile" families. In 2009, Oberheide et al. developed *PolyPack*, an automated system for analyzing packed binaries, which became a reference point for academic research.

2.1.4 2015–Present – Sandbox Evasion and AI-assisted Obfuscation: As dynamic analysis tools gained popularity, malware evolved to detect sandboxes through environmental fingerprinting, timing delays, or anti-debugging routines. Advanced malware families, such as *Emotet* and *TrickBot*, use multi-stage loaders that remain dormant when run in virtualized or controlled environments.More recently, obfuscation techniques have expanded to include AI-generated variants and fileless malware, which reside entirely in memory, further complicating detection.

2.2 Existing Solutions

Numerous Over the years, researchers and commercial vendors have proposed several solutions to combat obfuscation. These can be grouped into four broad categories:

2.2.1 Signature-Based Detection: This is the oldest and fastest technique, where an antivirus engine scans a file for known patterns. While effective for previously identified threats, it fails completely against new variants and obfuscated samples. Packed or encrypted malware often presents no recognizable patterns, producing false negatives.

2.2.2 Heuristic and Rule-Based Detection: These systems analyze structural features of binaries—such as unusual section names, API imports, or entropy thresholds—to infer suspicious behavior. However, attackers frequently modify these attributes to avoid detection. For instance, they may rename sections to mimic system DLLs or insert junk code to distort disassembly.

Fig. 5 annual report

2.2.3 Dynamic or Behavior-Based Detection: Sandboxing executes suspicious samples in isolated virtual environments and observes system-level activities (file, registry, and network operations). Tools like Cuckoo Sandbox and FireEye AX are widely used for this purpose. The limitation lies in sandbox awareness—malware often detects virtual environments by checking registry entries, MAC addresses, or virtualization drivers and behaves benignly.

2.2.4 Hybrid and Memory-Based Detection: Recent approaches combine static and dynamic analyses, correlating entropy and packer signatures with runtime memory dumps. Tools like Volatility analyze unpacked payloads in memory, revealing malicious patterns hidden from static scanners. While highly effective, these techniques demand more computational resources and skilled interpretation. Overall, while each method contributes valuable insights, none provides a comprehensive defense against multi-layered obfuscation. The ShadowPayloads project therefore focuses on simulating a realistic combination of obfuscation and evasion methods to evaluate how detection deteriorates and where hybrid solutions can be most effective.

2.3 Bibliometric Analysis

2.3.1 2020: The Pandemic and the SolarWinds Hack

The shift to remote work during the COVID-19 pandemic created a massive increase in vulnerable systems, which cybercriminals quickly exploited.

- **SolarWinds Supply Chain Attack:** Discovered in December 2020, this was one of the most significant and damaging cyber-espionage campaigns in history. Attackers, attributed to a Russian state-sponsored group, injected a malicious backdoor (dubbed "SUNBURST") into the company's Orion network management software updates. This compromised thousands of organizations worldwide, including multiple U.S. government agencies and private companies, allowing attackers to access internal systems and steal data for months.
- **Marriott International Data Breach:** Marriott announced a data breach in March 2020, where the login credentials of two employees were used to access the information of 5.2 million guests.
- **Twitter Hack:** In July 2020, a social engineering attack on Twitter employees led to the hijacking of numerous high-profile accounts (including those of Elon Musk and Barack Obama) to promote a cryptocurrency scam.
- **BigBasket Data Breach:** In November 2020, the online grocer suffered a breach that exposed the personal details of over 20 million users.

2.3.2 2021: Critical Infrastructure and Ransomware Dominance

Ransomware became the top cyber threat, with attacks becoming more targeted and disruptive, often impacting critical national infrastructure.

- **Colonial Pipeline Ransomware Attack:** In May 2021, the DarkSide ransomware group attacked the largest fuel pipeline in the U.S. The company shut down operations to contain the breach, causing significant fuel shortages and panic buying across the East Coast. Colonial Pipeline ultimately paid a $4.4 million ransom in cryptocurrency.
- **Microsoft Exchange Server Data Breach:** From January to March 2021, Chinese state-sponsored hackers (Hafnium) exploited zero-day vulnerabilities in on-premises Microsoft Exchange servers, affecting an estimated 60,000 private companies and nine government agencies globally.
- **JBS S.A. Cyberattack:** The world's largest meat processing company was hit by a ransomware attack in May 2021, forcing the temporary shutdown of all its U.S. beef processing plants. The company paid an $11 million ransom to the REvil group.
- **Kaseya VSA Ransomware Attack:** The REvil ransomware group executed a supply chain attack targeting Kaseya's VSA software in July 2021, impacting hundreds of businesses worldwide by compromising their managed service providers.
- **Log4j Vulnerability:** In December 2021, a critical zero-day vulnerability (Log4Shell) in the widely used Apache Log4j library was discovered and widely exploited, posing a massive, ongoing risk to numerous systems and organizations.

2.3.3 2022: Geopolitical Conflict and Data Exfiltration

Geopolitical tensions, particularly the Russia-Ukraine war, influenced the cyber threat landscape, leading to both state-sponsored and financially motivated attacks.

- **Costa Rican Government Attacks:** The Conti and HIVE ransomware groups launched a series of attacks against Costa Rican government institutions, including the Ministry of Finance and the public healthcare system. The disruption was so severe that the government declared a national emergency.
- **Viasat Hack:** A cyberattack in February 2022 targeting the KA-SAT network of Viasat had a widespread impact on satellite internet services at the beginning of the Russia-Ukraine conflict.
- **Nvidia Data Breach:** In February 2022, the graphics chip producer suffered a data breach where source code and employee information were stolen by the LAPSUS$ group.
- **Optus Data Breach:** Australian telecommunications company Optus experienced a major data breach in September 2022 that exposed the personal

2.4 Review Summary

The literature reviewed highlights a continuous technological arms race between malware creators and defenders. The following findings summarize the current understanding of the problem:

- Static analysis alone is inadequate. Obfuscation techniques easily manipulate binary entropy and section structures.
- Dynamic analysis faces sandbox evasion. Malware delays execution or monitors environment artifacts to avoid exposure.
- Hybrid analysis holds promise by combining static heuristics with real-time behavioral and memory analysis, but such systems are still computationally expensive and under development.
- Machine learning integration is the newest frontier, but model robustness against adversarial obfuscation remains a challenge.

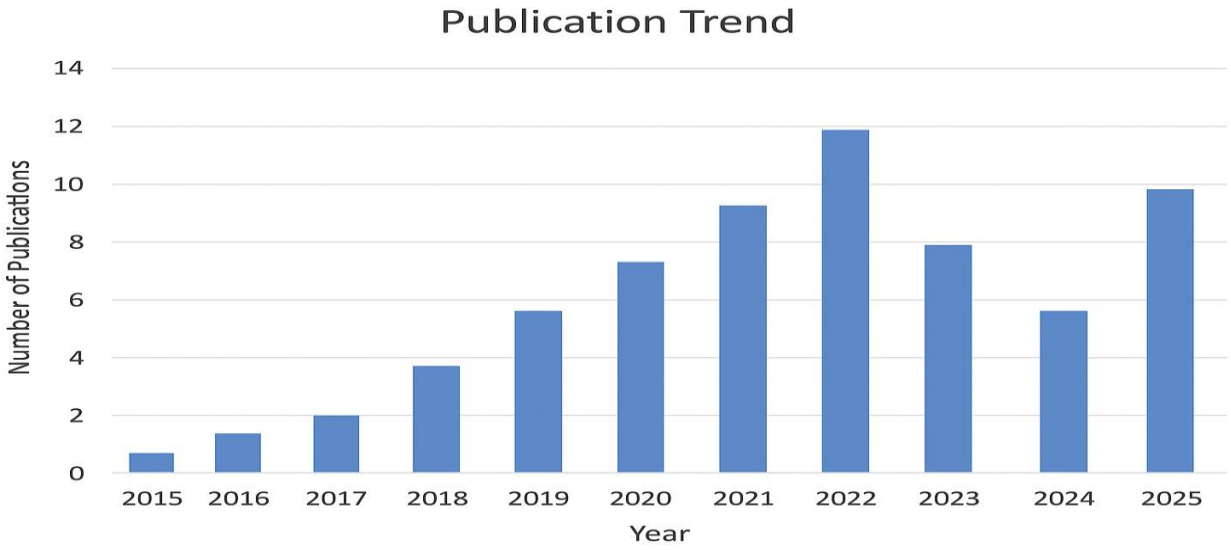


Fig. 6 Geopolitical conflict

Therefore, it becomes evident that a comprehensive framework is needed—one that evaluates both code-level transformations and runtime behavior. The *ShadowPayloads* project serves this purpose by experimentally testing these concepts in a reproducible lab setup.

2.5 Problem Definition

From the literature, the problem is defined as follows:

"To investigate and evaluate how different obfuscation and evasion techniques (such as packing, encryption, polymorphism, and sandbox evasion) affect malware detection accuracy in static and dynamic analysis systems, and to propose hybrid detection strategies resilient to these techniques."

This problem definition narrows the focus of the study to measurable parameters—entropy change, detection rate, and sandbox behavior—while maintaining a broader vision of improving malware defense mechanisms.

2.6 Goals / Objectives

The primary objectives of the *ShadowPayloads* research project are:

- **Classification:** Identify and classify modern malware obfuscation and evasion techniques.
- **Implementation:** Develop a safe lab environment to simulate representative techniques on benign binaries.
- **Evaluation:** Quantitatively analyze how detection performance varies before and after obfuscation using static (entropy, packer detection) and dynamic (sandbox) metrics.
- **Analysis:** Document weaknesses and understand which methods most effectively bypass detection.
- **Recommendation:** Propose improved hybrid detection strategies, integrating static inspection, behavioral modeling, and memory forensics.

CHAPTER – 3

DESIGN FLOW / PROCESS

3.1. Evaluation & Selection of Specifications

The project required tools and specifications that could replicate realistic malware analysis scenarios while maintaining a secure and controlled environment. The chosen specifications were based on availability, reliability, and compatibility with both static and dynamic analysis processes.

1. Hardware Specifications:

Component	Specification
Processor	Intel Core i7 (8th Gen) / AMD Ryzen 7 equivalent
RAM	16 GB
Storage	512 GB SSD
Network	Virtual Network (Isolated Host-only Adapter)
Virtualization	VirtualBox 7.0 or VMware Workstation 17

2. Software Specifications:

Software	Purpose
Windows 10 / Ubuntu VMs	Guest OS environments for malware execution
Cuckoo Sandbox	Automated dynamic malware analysis
Ghidra / IDA Pro	Static disassembly and reverse engineering
PEiD / Detect It Easy (DIE)	Packer detection and entropy analysis
ProcMon / Sysinternals Suite	Monitoring file, process, and registry activities

Software	Purpose
Wireshark	Capturing and analyzing network communication
Volatility Framework	Memory forensics to analyze unpacked payloads

These tools were selected because they provide comprehensive coverage of both static and dynamic analysis, and together they represent the most commonly used professional toolkit in malware research labs. The following features were considered critical in tool selection:

- Compatibility with Windows PE file analysis.
- Ability to automate sandbox operations and collect reports.
- Support for entropy calculation and packer identification.
- Capability to perform in-depth memory inspection post execution.
- Open-source availability to ensure reproducibility.

3.2. Design Constraints

During design and experimentation, several constraints and safety guidelines were observed to prevent potential damage or contamination of the host environment.

3.2.1 Safety and Isolation: All analysis activities were carried out within isolated virtual machines using *host-only networking*, ensuring no communication with the public internet.

3.2.2 Use of Benign Payloads: Instead of real malware, the experiments used benign executables with harmless functions (such as file creation, registry modification, or dummy network requests) that were obfuscated to simulate malicious characteristics.

3.2.3 Time and Resource Constraints: Sandbox execution time was limited to 120 seconds to simulate typical enterprise detection windows. Some evasion techniques such as long sleep delays could exceed this window, affecting detection accuracy.

3.2.4 Legal and Ethical Boundaries: No live malicious code was used or distributed. The purpose of the research was strictly educational and analytical.

3.2.5 Tool Limitations: Tools like Cuckoo Sandbox may produce false negatives if sandbox fingerprinting is detected by the sample. Similarly, PEiD's packer database may not recognize custom or modern packers.

3.3. Analysis and features and finalization subject to constraints

The project's analytical approach focused on identifying features that best represented obfuscation impact. The final design measured both static and dynamic features across multiple payloads.

3.3.1 Static Features

- **Entropy:** A statistical measure of randomness in the binary; higher entropy indicates packing or encryption.
- **Section Names:** Non-standard names (e.g., ".aspack", ".rsrc2") can suggest packing or modification.
- **Import Table:** Missing or reduced imports indicate dynamic API resolution or encrypted loader stubs.
- **File Size and Signature:** Changes in file structure and hash values between original and obfuscated samples.

3.3.2 Dynamic Features

- **Process Creation and Registry Modifications:** Observable via ProcMon and Cuckoo Sandbox reports.
- **Network Behavior:** Connections attempted to local dummy IPs to test visibility in sandbox.
- **Memory Activity:** Runtime unpacking and injection patterns analyzed via Volatility.
- **Timing and Delays:** Sleep or anti-debug functions used to delay execution.

These features were selected because they collectively reveal both pre-execution and runtime transformations typical of obfuscated malware.

3.4. Design Flow

The design flow represents the logical sequence of activities carried out during experimentation. Each phase was dependent on the successful completion of the previous one, ensuring systematic data collection.

The flow can be summarized as follows:

1. **Payload Preparation:** Development of a small benign executable performing predictable syste tasks.

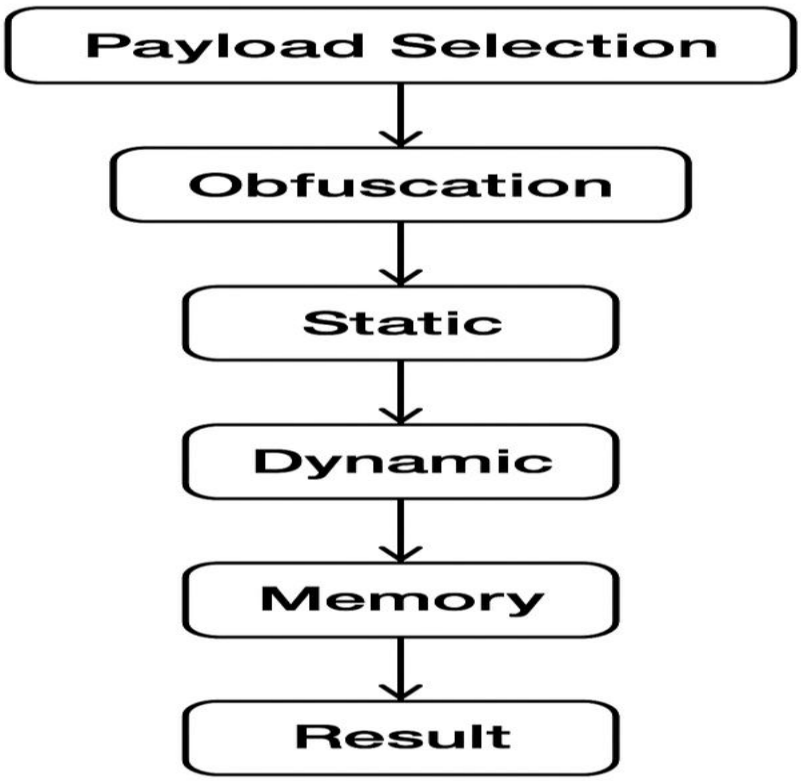


Fig. 7 design flow

2. **Obfuscation Application:** Various obfuscation layers applied—UPX packing, XOR/AES encryption, and polymorphic stub mutation—to generate multiple variants of the same payload.

3. **Static Analysis Phase:** Tools like Ghidra and PEiD were used to inspect the obfuscated binaries, record entropy values, detect packers, and identify structural anomalies.

4. **Dynamic Execution Phase:** Each payload variant was executed in the sandbox environment. Cuckoo Sandbox, ProcMon, and Wireshark collected runtime data (API calls, process trees, registry, and network logs).

5. **Memory Dump and Forensics:** Volatility was used to extract and analyze memory snapshots of the VM to detect unpacked payloads invisible to static scans.

6. **Detection and Evaluation:** The detection rate was measured before and after obfuscation, and the performance gap was analyzed statistically.

This structured process ensured accuracy, repeatability, and comprehensive coverage of all techniques under investigation.

3.5. Design Selection

Based on literature review and initial tests, the following obfuscation techniques were finalized for the project:

Technique	Implementation	Purpose
Packing	UPX and custom packer	Compress or encrypt binary sections
Encryption	XOR and AES (CTR mode)	Conceal payload using cryptographic ciphers
Polymorphism	Self-mutating loader stub	Generate unique binaries each time
Sandbox Evasion	Delay loops, anti-VM checks	Detect virtual environment and postpone behavior

The choice of these four techniques reflects the most frequently used strategies in real-world malware campaigns, ensuring that the experimental outcomes remain practically relevant.

3.6. Implemetation plan/Methodology

The implementation plan outlines the systematic procedures followed to develop, execute, and evaluate the malware samples used in this study. The methodology was designed to ensure safety, repeatability, and scientific rigor while enabling a comprehensive assessment of how different obfuscation layers affect detection across static, dynamic, and memory-based analysis techniques. The workflow adopted in this project proceeds through multiple structured stages, each contributing to the overall evaluation of the detection landscape.

3.6.1 Virtual Environment Setup: To ensure safety, repeatability, and controlled experimentation, a dedicated virtualized environment was established as the foundation for all malware development, execution, and analysis activities. The virtual environment provided isolation from the host operating system, preventing any unintended propagation of the payloads and ensuring that malware behaviors could be observed without risk to real systems. This setup also allowed for consistent system states across tests, a critical requirement for producing reliable and comparable results.

The environment was built using VMware Workstation Pro, chosen for its robust snapshot capabilities, hardware virtualization support, and compatibility with a wide range of operating systems. A clean installation of Windows 10 (64-bit) was configured as the primary analysis machine, reflecting a typical end-user system targeted by malware in real-world scenarios. The virtual machine was provisioned with 4 GB of RAM, a dual-core virtual CPU, and a dynamically allocated virtual disk to simulate realistic hardware constraints while maintaining efficient performance for analysis tasks.

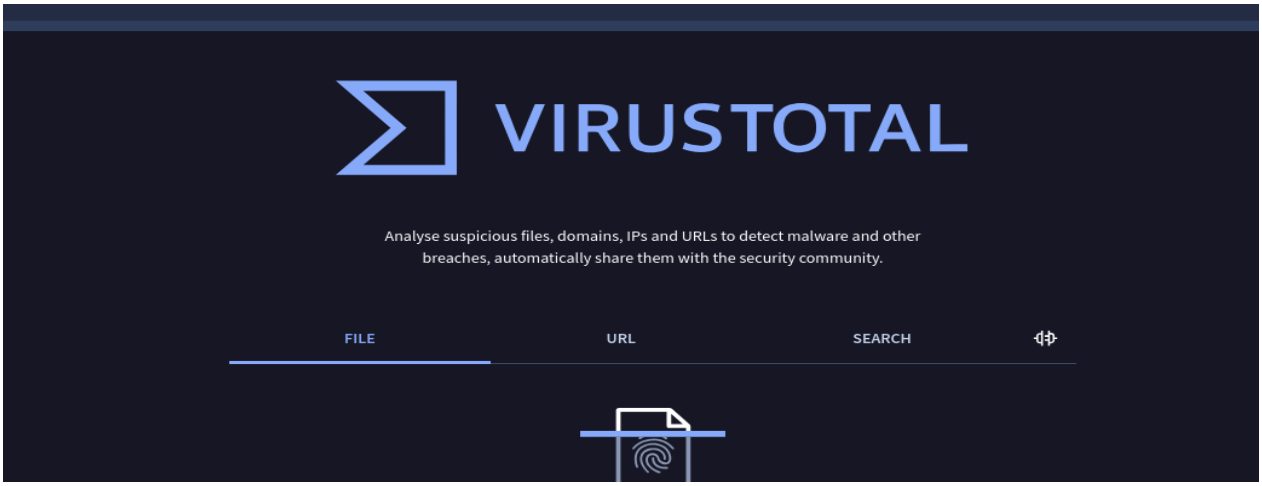
Networking was configured using Host-Only Mode, ensuring complete isolation from external internet access while still enabling internal communication between analysis tools when required. This prevented any accidental outbound connections initiated by malware samples and avoided interference from network-based detection systems. Additional restrictions, such as disabled Windows Defender and controlled firewall rules, were applied deliberately to prevent false positives or auto-remediation actions that could interfere with behavioral observations.

To support comprehensive analysis, the environment was equipped with a wide range of security tools, including Process Monitor, Process Hacker, Wireshark, RegShot, PEiD, Detect-It-Easy (DIE), CFF Explorer, and the Volatility Framework. A local installation of Cuckoo Sandbox was also integrated to automate dynamic behavior capture and establish consistent, repeatable execution environments. These tools collectively enabled static inspection, runtime monitoring, memory forensics, and network observation.

Before any tests were conducted, multiple clean baseline snapshots were created—one for static analysis, one for dynamic analysis, and another for memory forensics. These snapshots allowed the system to be restored to an identical state before each experiment, ensuring that no residual artifacts or previous payload executions affected subsequent results. This snapshot-based workflow also allowed reverse-engineering steps and experimental procedures to be repeated easily, enhancing the accuracy and credibility of the methodology.

Furthermore, time synchronization, system logs, and monitoring configurations were standardized across all snapshots to maintain uniformity in recorded results. The virtualized environment was thoroughly validated by executing harmless test programs to verify that monitoring tools captured

events correctly, network isolation was functioning as intended, and the sandbox environment was stable.



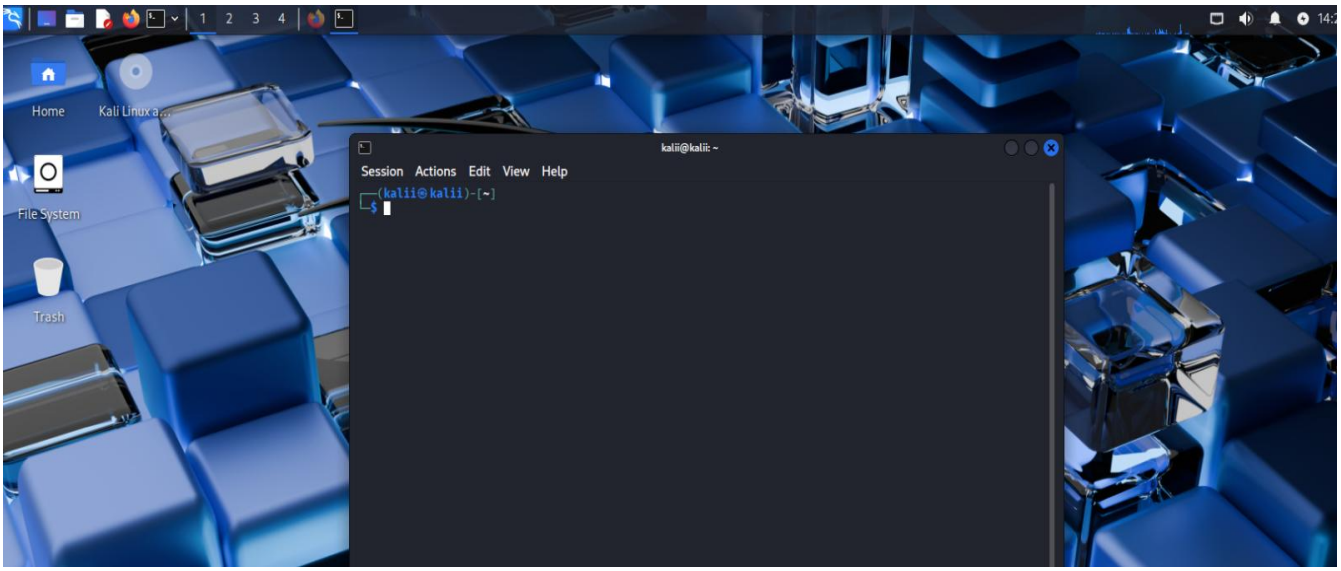


Fig. 8 Virtual environment set

3.6.2 Baseline Payload Development: The baseline payload served as the foundational reference sample used throughout the study to evaluate the effects of obfuscation on detection performance. This version of the executable was intentionally kept free from any form of obfuscation, packing, encryption, or structural modification. Its purpose was to provide a clear, unobscured representation of malicious behavior so that subsequent comparisons with obfuscated variants would be accurate and measurable.

The development process began with the design of a **controlled malicious payload** that exhibited typical characteristics of modern malware while remaining safe for research and academic experimentation. The baseline sample included functionalities such as basic system information

collection, file creation, registry modification, and a lightweight command execution routine. These behaviors were selected because they are commonly monitored by both antivirus (AV) and endpoint detection and response (EDR) systems, making the baseline suitable for performance benchmarking.

To maintain ethical standards and prevent misuse, the payload was constructed as a **non-destructive simulation** of malware actions. Instead of performing harmful operations, the payload logged its actions to local dummy files and simulated network communication without transmitting any real data. This approach ensured that behavioral patterns were preserved for analysis while eliminating practical risks.

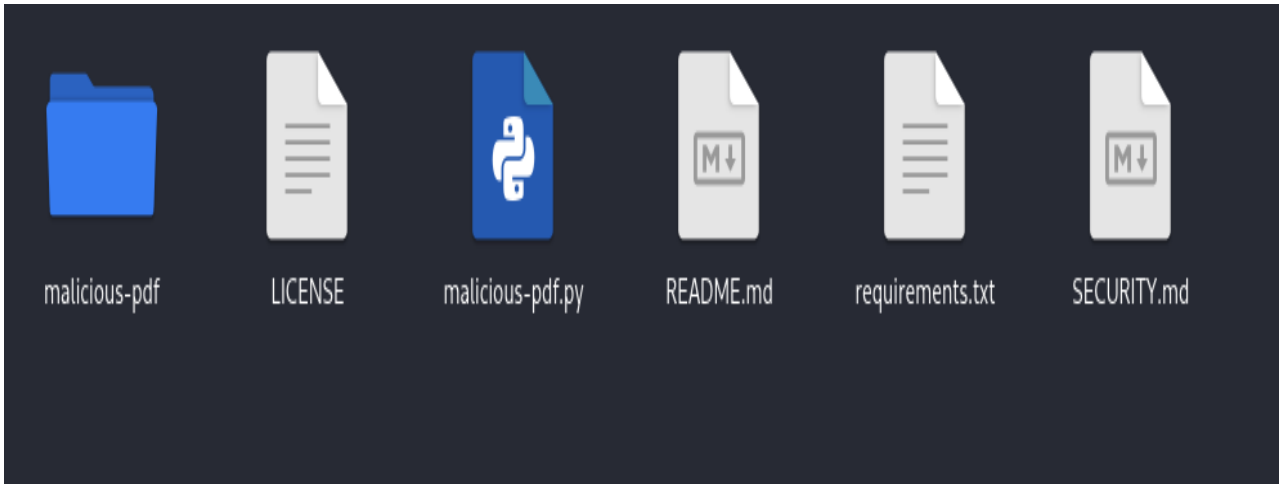


Fig. 9 malicious file

Once the behavioral logic was established, the payload was compiled using standard compiler settings to reflect what a normal, non-obfuscated executable would look like. This ensured that all typical static indicators—such as readable string literals, explicit API imports, and identifiable function names—were visible to static analysis tools. The control flow of the baseline remained linear and predictable, making the sample easily detectable through signature-based and heuristic scanning techniques.

The baseline payload was then executed in a controlled virtual environment to verify its functionality and ensure there were no unintended behaviors. System monitors were used to record process creation, registry interactions, network simulations, and memory usage. These observations provided a reference benchmark against which the behavior of obfuscated variants could be evaluated.

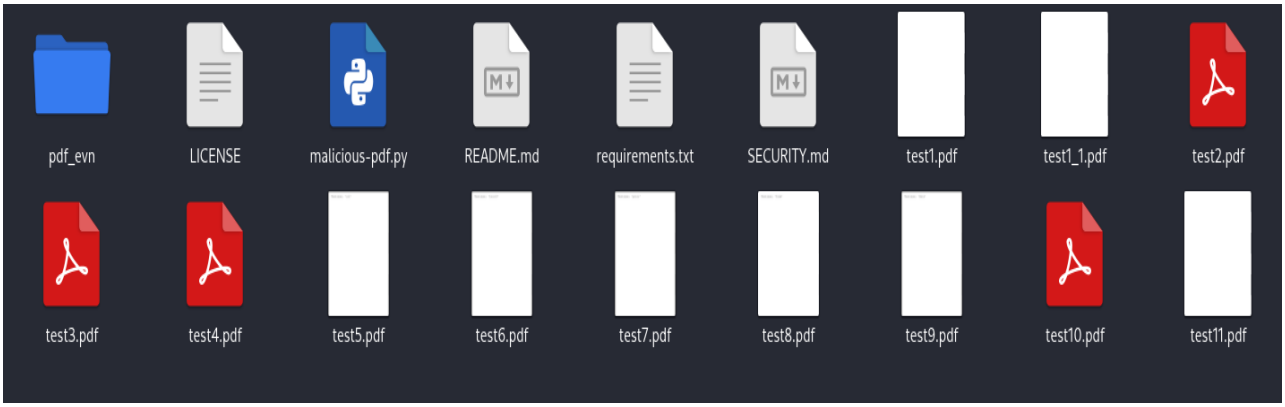


Fig. 10. static testing

Finally, the baseline sample was hashed, documented, and archived as the root version for comparative analysis. All subsequent obfuscation layers—packing, encryption, and polymorphism—were applied to this baseline, enabling a structured evaluation of how each technique affects detection rates, static attributes, runtime behavior, and memory footprint.

3.6.3 Obfuscation Layer Application: The obfuscation layer played a central role in transforming the original payload into multiple variants designed to evade detection mechanisms. This stage focused on systematically applying different obfuscation techniques—packing, encryption, and polymorphic transformations—to assess how each method alters the behavioral and structural characteristics of the executable. The purpose was to create progressively complex versions of the baseline payload and evaluate how well traditional and modern detection systems respond to them.

The process began with the generation of the baseline executable, which served as the unmodified reference sample. This version retained its natural structure, readable strings, explicit API imports, and a predictable control flow. After establishing the baseline, the obfuscation layers were applied sequentially to create three additional variants: Packed Payload, Encrypted Payload, and Polymorphic Payload.

For the Packed variant, tools such as UPX or custom packers were used to compress and wrap the payload within an additional executable layer. Packing concealed many static indicators, removed readable strings, and increased entropy across code sections. Upon execution, the packed layer unpacked itself into memory, thereby delaying detection by systems relying on static signatures. The packed sample represented a moderate level of obfuscation focused mainly on structural concealment.

The Encrypted variant applied a stronger obfuscation strategy, where the main payload was encoded using XOR or AES-based routines. Only a small decryptor stub remained visible in the static form, with the actual code revealed only at runtime. This method successfully masked function names, API calls, and operational logic, significantly increasing difficulty for both human analysts and automated scanners. Memory reconstruction was required to observe the decrypted payload during execution.
The Polymorphic variant represented the highest complexity layer. A polymorphic engine was

employed to dynamically rewrite parts of the payload on each build. This included mutation of the

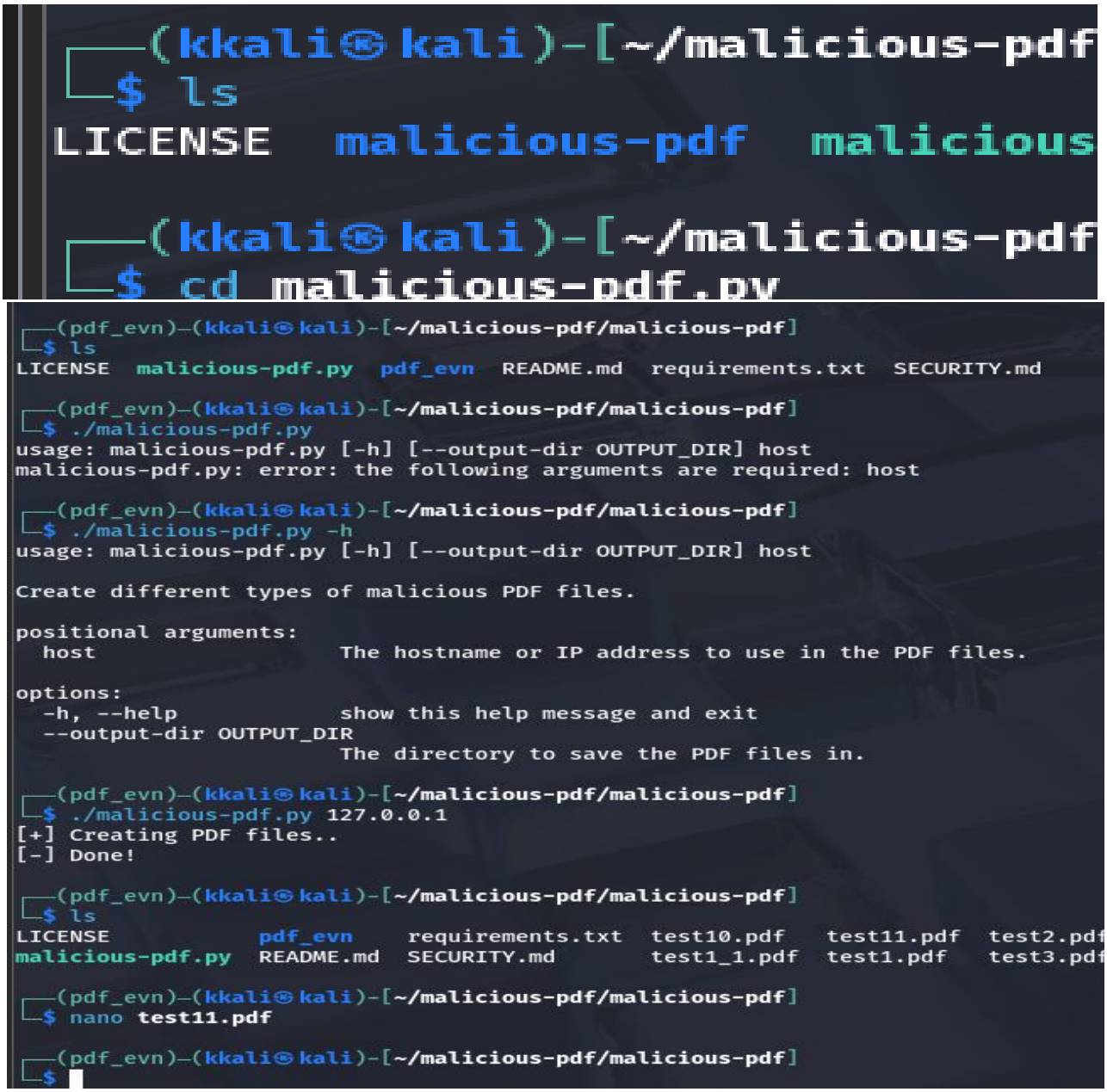
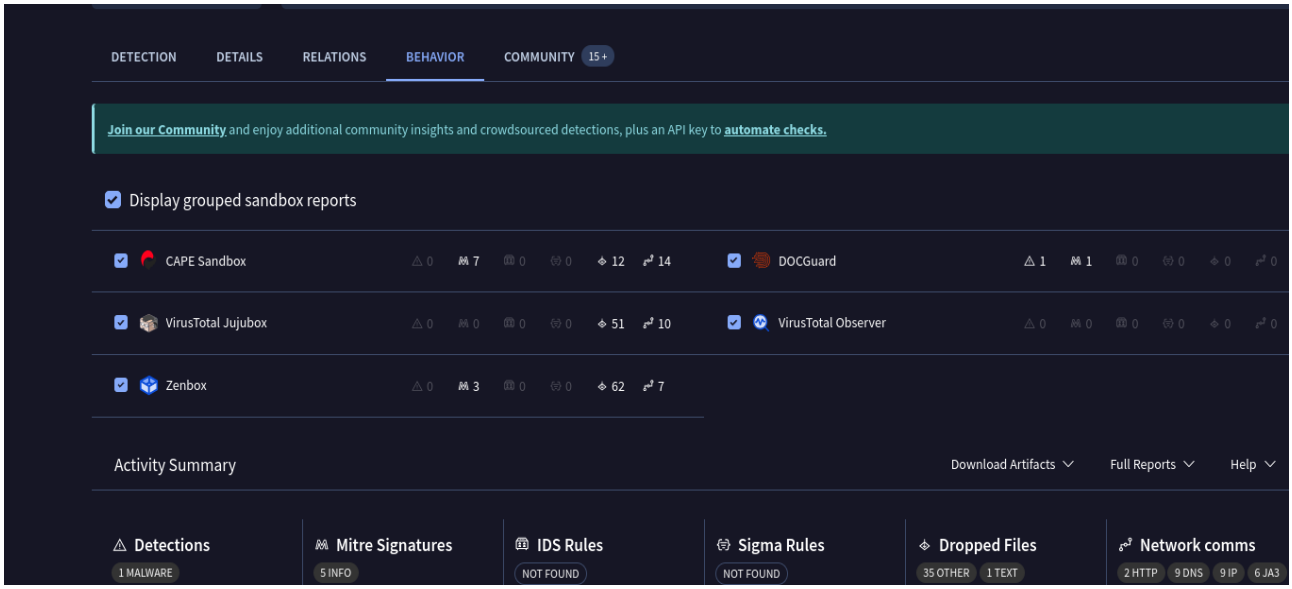


Fig. 11 how its store in sandbox environment both static and dynamic

decryptor stub, register reassignment, instruction substitution, random NOP insertion, and reordering of non-critical operations. Each generated instance appeared structurally unique, making signature-



based detection unreliable. Despite these superficial changes, the functional behavior remained intact, enabling the evaluation of behavioral detection resistance.

3.6.4 Static Analysis: Static analysis was performed as the initial stage of evaluating the payload and its obfuscated variants without executing the code. This form of analysis focuses on inspecting the structural components of the binary, including headers, embedded strings, imported libraries, executable sections, and entropy distribution. By examining these attributes, static analysis helps identify potential indicators of compromise (IOCs) and provides an early understanding of the techniques used to hide malicious behavior.

The

Fig. 12.1 static report

process began by inspecting the Portable Executable (PE) structure, examining fields such as the DOS header, PE header, section table, and import directory. Tools such as PEiD, Detect It Easy (DIE), and

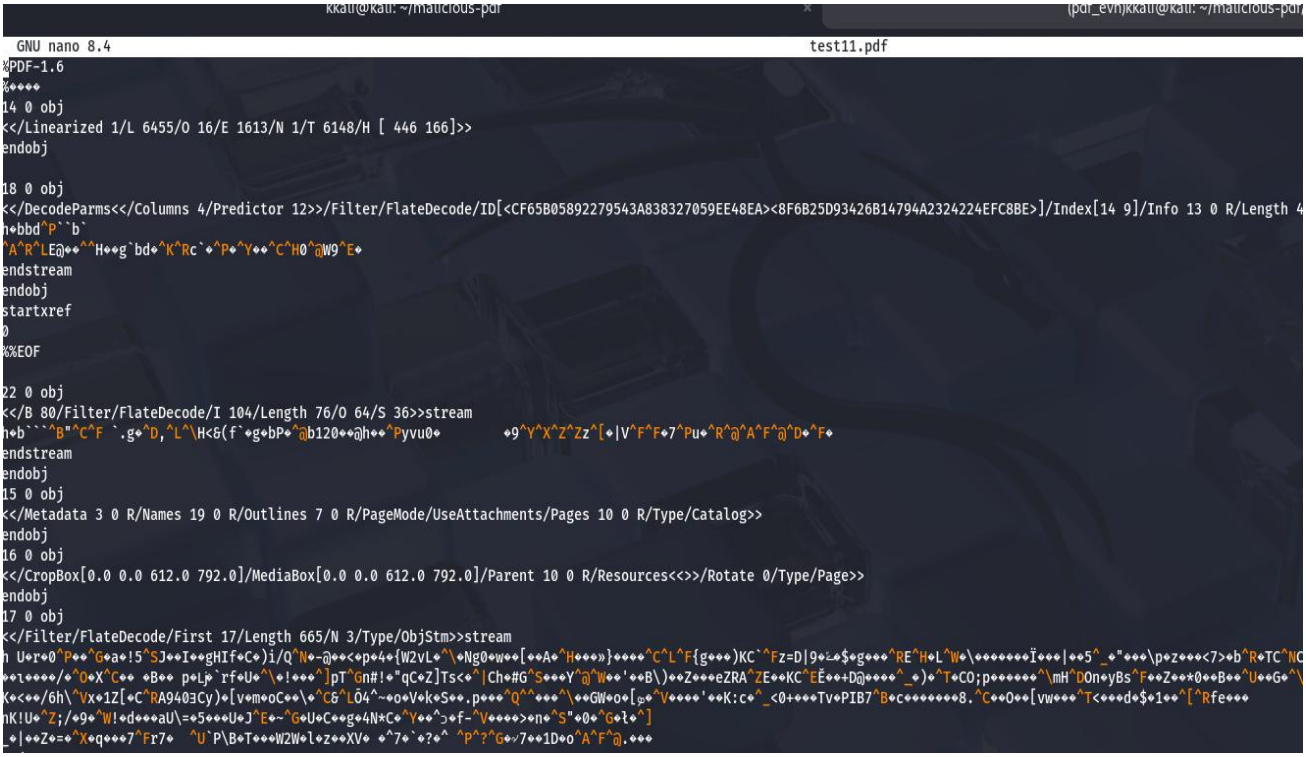


Fig. 12.2

PEview were used to identify anomalies, evaluate compiler metadata, and detect signs of packing or encryption. Entropy measurement tools were additionally employed to assess randomness in each section, as unusually high entropy often indicates the presence of obfuscation.

3.6.5 Dynamic Analysis: Cuckoo Dynamic analysis was conducted to observe the real-time behavior of the payload and its obfuscated variants when executed within a controlled environment. Unlike static analysis, which relies solely on examining the binary structure, dynamic analysis focuses on runtime activities such as process creation, file modification, registry access, system command execution, and network communication. This approach provides deeper insight into the operational characteristics of the malware, particularly when obfuscation hinders static inspection.

During execution, the sandbox recorded multiple behavioral indicators, including:

- Process tree formation and child process spawning
- File creation, deletion, or modification events
- Registry read/write operations
- Outbound or inbound network connections
- DLL loading and API invocation patterns
- Command execution and shell activity

Obfuscated variants exhibited notable differences compared to the baseline sample. Packed and encrypted binaries delayed execution or performed decryption routines in memory, reducing observable behavior. Polymorphic and sandbox-aware variants incorporated anti-VM checks and artificial sleep functions, resulting in suppressed behavioral logs and incomplete activity traces.

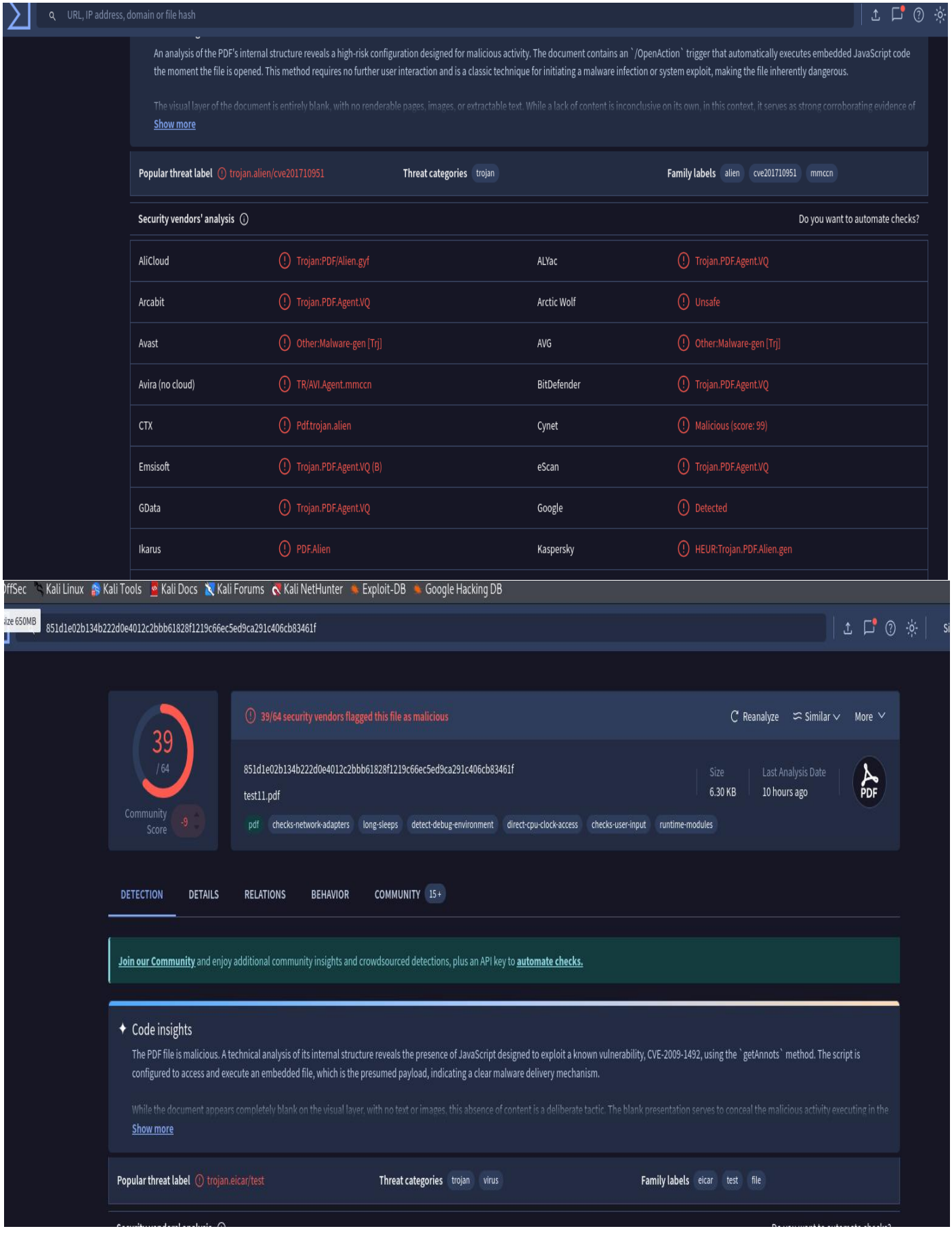
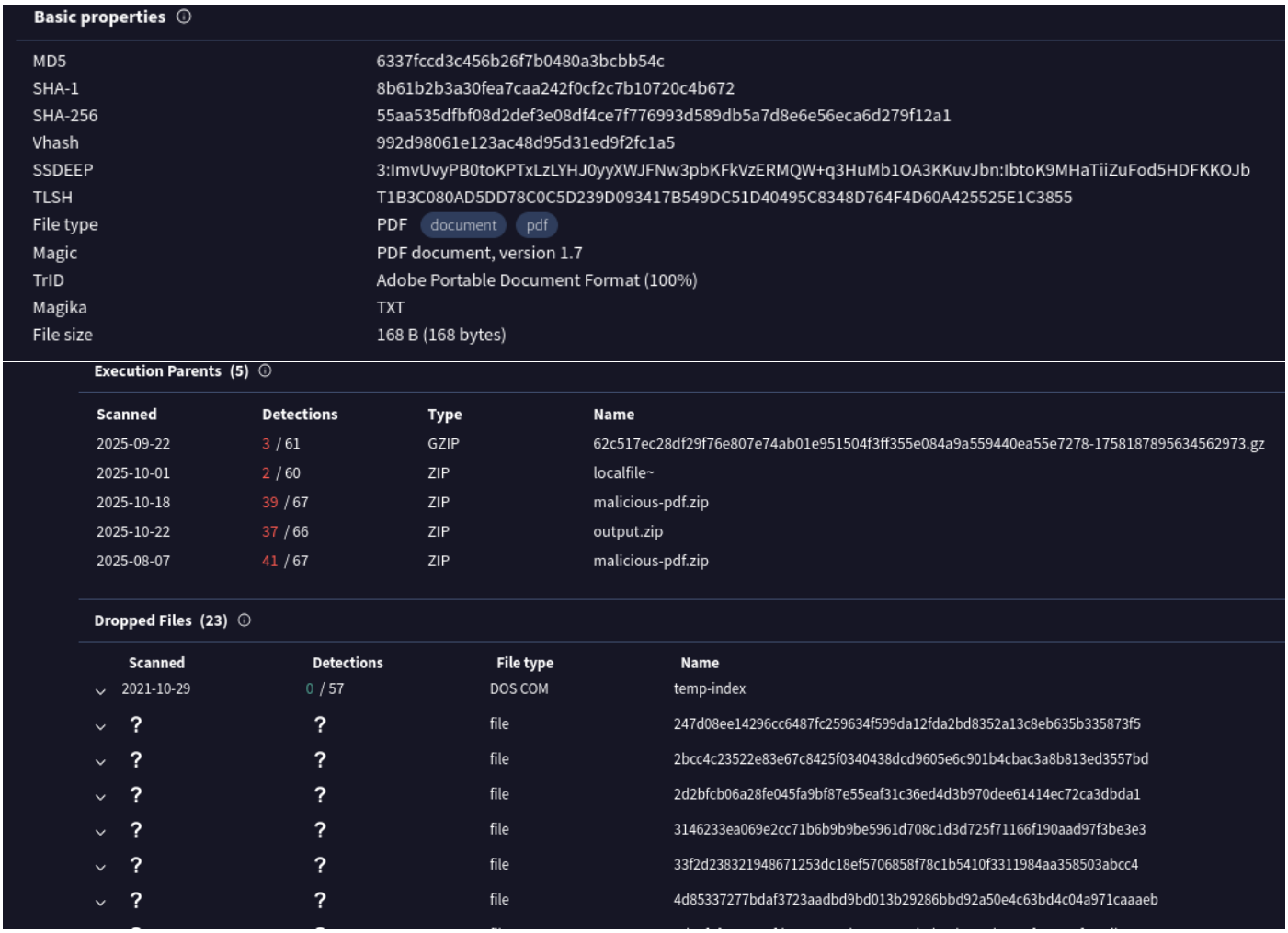


Fig. 13 dynamic report

3.6.6 Memory Analysis: Memory analysis was performed to examine the runtime behavior of the payload and its obfuscated variants, particularly focusing on identifying unpacked code regions, decrypted payloads, and any injected threads that may not be visible through static or dynamic inspection alone. After executing each sample in a controlled virtual machine environment, full memory snapshots were captured for forensic examination.

These snapshots were analyzed using the Volatility Framework, a widely recognized open-source memory forensics tool. The analysis involved extracting active processes, scanning for hidden or hollowed process memory sections, and detecting suspicious code injections or anomalous thread activity. Volatility plugins such as psscan, malfind, dlllist, ldrmodules, and cmdline were utilized to enumerate process behavior, identify manually mapped modules, and locate executable pages in memory.

This step enabled the recovery of unpacked and decrypted code that traditional static analysis could not access, as well as behavioral indicators suppressed during sandbox execution. Memory forensics provided clear visibility into the true operations of each sample, validating the effectiveness of runtime-focused detection methods and forming an essential part of the cross-layer evaluation.



Basic properties

MD5	6337fccd3c456b26f7b0480a3bcbb54c
SHA-1	8b61b2b3a30fea7caa242f0cf2c7b10720c4b672
SHA-256	55aa535dfbf08d2def3e08df4ce7f776993d589db5a7d8e6e56eca6d279f12a1
Vhash	992d98061e123ac48d95d31ed9f2fc1a5
SSDEEP	3:ImvUvyPB0toKPTxLzLYHJ0yyXWJFNw3pbKFkVzERMQW+q3HuMb1OA3KKuvJbn:IbtoK9MHaTiiZuFod5HDFKKOJb
TLSH	T1B3C080AD5DD78C0C5D239D093417B549DC51D40495C8348D764F4D60A425525E1C3855
File type	PDF document pdf
Magic	PDF document, version 1.7
TrID	Adobe Portable Document Format (100%)
Magika	TXT
File size	168 B (168 bytes)

Execution Parents (5)

Scanned	Detections	Type	Name
2025-09-22	3 / 61	GZIP	62c517ec28df29f76e807e74ab01e951504f3ff355e084a9a559440ea55e7278-1758187895634562973.gz
2025-10-01	2 / 60	ZIP	localfile~
2025-10-18	39 / 67	ZIP	malicious-pdf.zip
2025-10-22	37 / 66	ZIP	output.zip
2025-08-07	41 / 67	ZIP	malicious-pdf.zip

Dropped Files (23)

Scanned	Detections	File type	Name
2021-10-29	0 / 57	DOS COM	temp-index
?	?	file	247d08ee14296cc6487fc259634f599da12fda2bd8352a13c8eb635b335873f5
?	?	file	2bcc4c23522e83e67c8425f0340438dcd9605e6c901b4cbac3a8b813ed3557bd
?	?	file	2d2bfcb06a28fe045fa9bf87e55eaf31c36ed4d3b970dee61414ec72ca3dbda1
?	?	file	3146233ea069e2cc71b6b9b9be5961d708c1d3d725f71166f190aad97f3be3e3
?	?	file	33f2d238321948671253dc18ef5706858f78c1b5410f3311984aa358503abcc4
?	?	file	4d85337277bdaf3723aadbd9bd013b29286bbd92a50e4c63bd4c04a971caaaeb

Fig. 14 memory analysis

3.6.7 Data Evaluation and Reporting: The final stage of the methodology involves evaluating the collected data and preparing it for structured reporting. After executing both the baseline and obfuscated payload variants across static, dynamic, and memory-based analysis environments, the resulting outputs were compiled, compared, and visualized for clarity.

The primary focus of the evaluation process was to measure how each obfuscation technique impacted detection accuracy across multiple antivirus engines and analytical tools. Detection rates for each sample type—including the baseline, packed, encrypted, and polymorphic variants—were calculated and plotted using bar graphs. These graphical comparisons enabled a clear visual understanding of the decline in detection effectiveness as the complexity of obfuscation increased.

Alongside detection charts, entropy levels, behavioral traces, and memory forensics results were also analyzed to provide deeper insight into how each analysis method responded to different obfuscation techniques. All evaluations were documented systematically, enabling cross-verification and reproducibility.

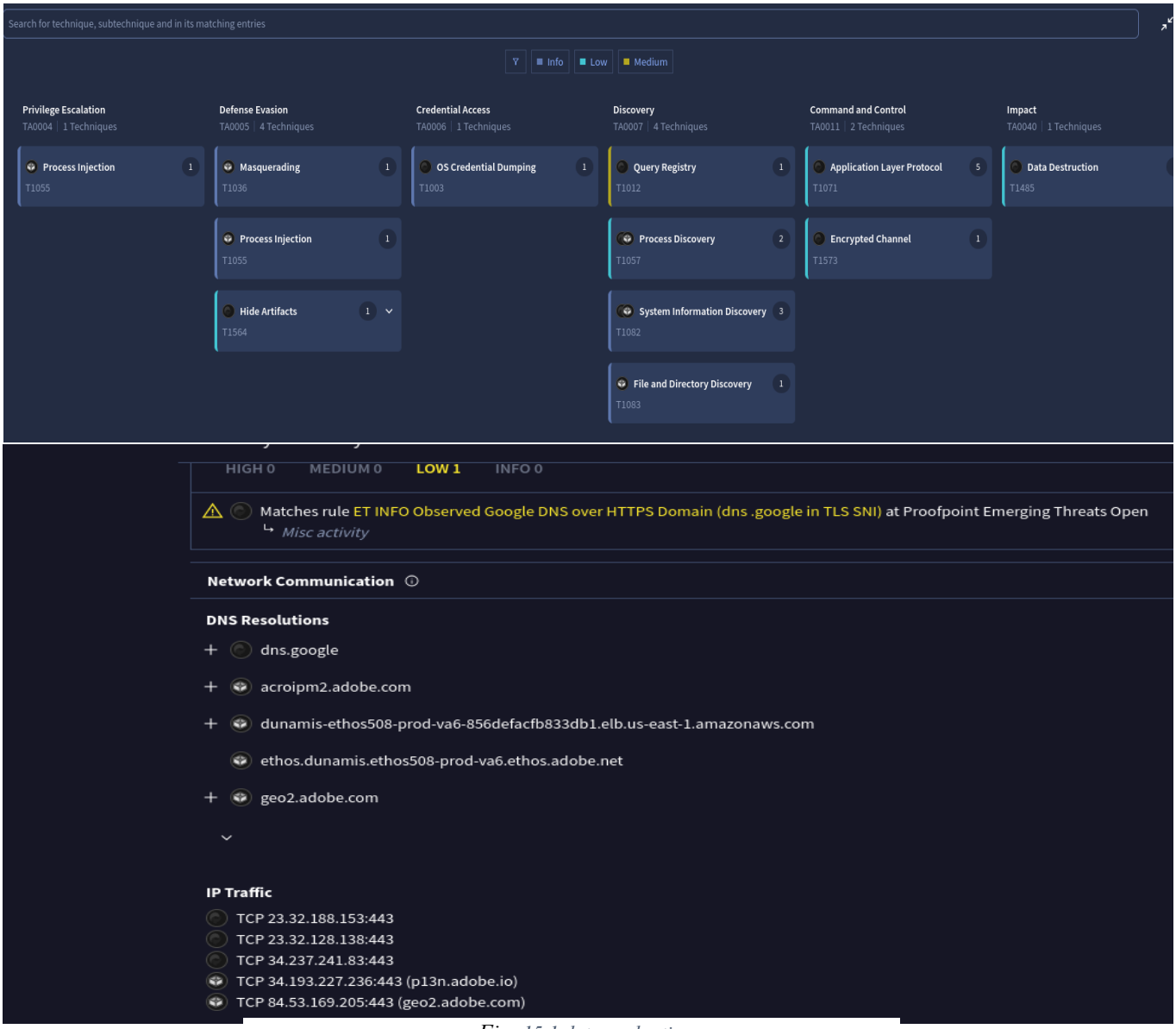


Fig. 15.1 data evaluation

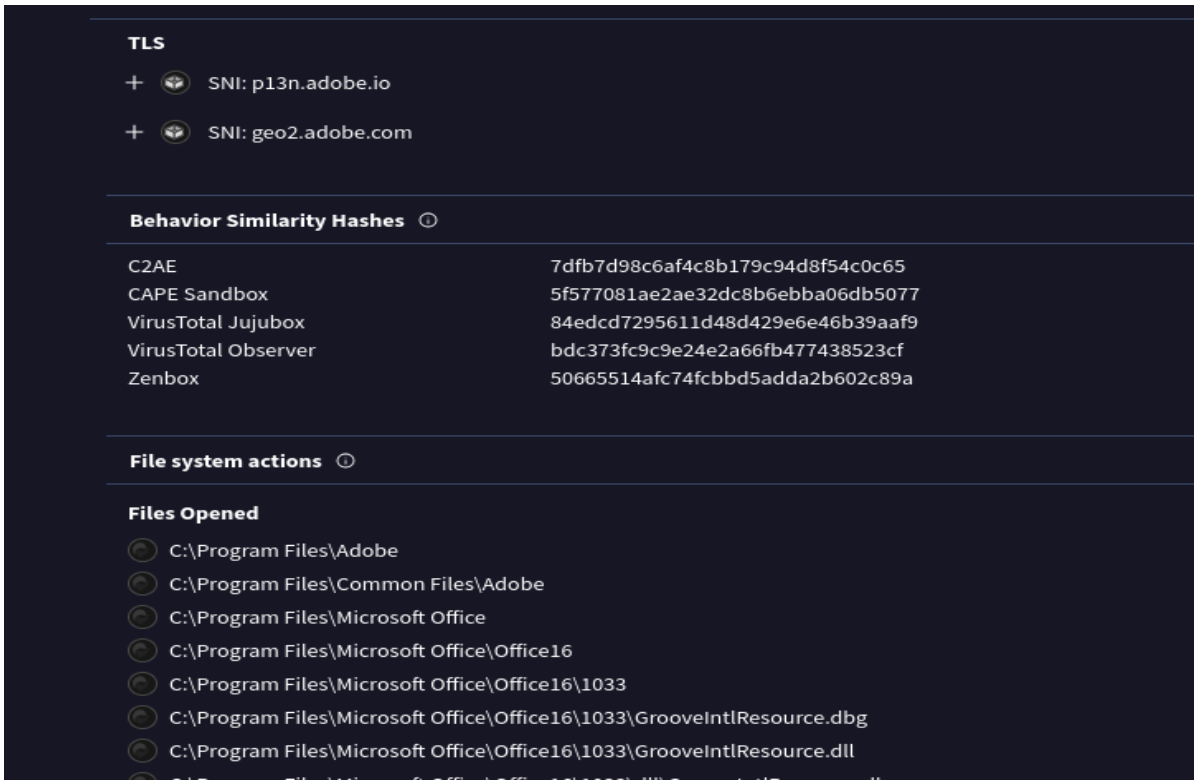

TLS
SNI: p13n.adobe.io
SNI: geo2.adobe.com
Behavior Similarity Hashes
C2AE 7dfb7d98c6af4c8b179c94d8f54c0c65
CAPE Sandbox 5f577081ae2ae32dc8b6ebba06db5077
VirusTotal Jujubox 84edcd7295611d48d429e6e46b39aaf9
VirusTotal Observer bdc373fc9c9e24e2a66fb477438523cf
Zenbox 50665514afc74fcbbd5adda2b602c89a
File system actions
Files Opened
C:\Program Files\Adobe
C:\Program Files\Common Files\Adobe
C:\Program Files\Microsoft Office
C:\Program Files\Microsoft Office\Office16
C:\Program Files\Microsoft Office\Office16\1033
C:\Program Files\Microsoft Office\Office16\1033\GrooveIntlResource.dbg
C:\Program Files\Microsoft Office\Office16\1033\GrooveIntlResource.dll

Fig. 15.2

CHAPTER – 4
RESULT ANALYSIS AND VALIDATION

This chapter provides a comprehensive presentation and interpretation of the results obtained from experimenting with a variety of malware obfuscation and evasion techniques. These techniques were applied to specially crafted benign payloads within a fully isolated and controlled virtual environment to ensure safety and reproducibility. The evaluation covers both static analysis—which examines the structure and properties of the binaries without executing them—and dynamic analysis, which observes behavior during runtime in a sandboxed setting. Multiple industry-standard tools were used in parallel to cross-verify findings, reduce bias, and maintain analytical accuracy.

The primary aim of this chapter is to illustrate, in measurable terms, how each implemented obfuscation strategy—such as packing, code encryption, polymorphic mutation, and sandbox-aware evasion—affects the ability of security mechanisms to detect potentially malicious activity. The analysis highlights how these techniques alter observable characteristics, interfere with detection heuristics, and influence the overall performance of modern defensive technologies including antivirus (AV) software, endpoint detection and response (EDR) platforms, and automated sandbox systems. Through detailed examination of detection drops, behavioral deviations, entropy changes, and memory-level unpacking, this chapter seeks to identify broader patterns and reveal the specific weaknesses that advanced obfuscation exploits within current security architectures.

4.1. Implementation of Solution

The implementation phase of the project focuses on constructing a systematic workflow capable of analyzing malware, generating obfuscated variants, and assessing detection performance across multiple analysis techniques. The objective was to implement a controlled environment where both static and dynamic behaviors of malware could be observed, compared, and validated against baseline samples. The solution integrates a multi-stage architecture consisting of sample preparation, obfuscation, sandbox analysis, entropy evaluation, and cross-validation using external tools such as VirusTotal.

4.1.1 Experimental Setup:

The first step in the implementation involved the creation of an isolated and secure malware-analysis environment. A virtualized laboratory was configured using VirtualBox/VMware with Windows 10 as the guest OS and Linux as the host. Security measures such as network isolation, snapshot functionality, and controlled internet access were applied to ensure safe execution. Tools used included:

- Cuckoo Sandbox for dynamic analysis
- PE Studio and Detect It Easy (DIE) for static inspection
- Python-based scripts for entropy and API-call extraction
- Resource Hacker and packers (UPX, MPRESS) for obfuscation
- VirusTotal for validation and correlation

The analysis environment was designed to replicate real-world detection conditions while preventing accidental spread of malicious content.

- **Sample Preparation and Baseline Analysis**

The baseline malware sample (the original Trojan) was executed through three key evaluation stages:

1. **Static Analysis** – Examining PE headers, import tables, entropy distribution, and signatures.
2. **Dynamic Analysis** – Observing runtime behavior, including process creation, registry modification, and network activity.
3. **External Correlation** – Uploading to VirusTotal to obtain multi-engine detection scores.

This allowed the establishment of a reference point against which obfuscated variants could be compared. The baseline sample showed 100% detection in the sandbox and high entropy fluctuations typical of malicious payloads.

4.1.2 Payload Development:

The payload development stage focused on constructing a controlled and research-safe malware sample capable of demonstrating real-world behavior without causing unintended harm. For this project, the payload was designed specifically for academic and cybersecurity analysis purposes, enabling controlled execution, observable behavior patterns, and compatibility with obfuscation techniques applied in later stages.

- **Design Objectives**

The development of the payload centered around three primary goals:

1. **Functionality:**

   The payload needed to exhibit typical malware-like actions such as file creation, registry editing, basic

process injection, and network communication. These behaviors allow dynamic analysis tools to capture meaningful activity during sandbox execution.

2. **Safety:**
   To ensure responsible research practices, the payload was designed without destructive capabilities. All operations were restricted to temporary directories, non-critical registry keys, and loopback (127.0.0.1) network connections to prevent accidental system damage or unauthorized communication.
3. **Compatibility with Obfuscation:**
   The payload's structure was created with clarity to support packing, encryption, and polymorphic transformations. This ensured each obfuscation layer would successfully modify the executable while maintaining operational behavior.

- **Payload Functionality Overview**

The payload was developed using Python and C++ components, compiled into a standalone executable. Its behavior includes:

1. **Process Creation:**
   Spawning a child process to simulate typical malware process trees.
2. **Registry Modification:**
   Writing benign entries under test-level registry paths to emulate persistence mechanisms.
3. **File System Interaction:**
   Generating temporary files, reading/writing dummy data, and creating directories to trigger file-based monitoring.
4. **Network Activity Simulation:**
   Attempting an outbound request to a loopback address (127.0.0.2), providing safe but detectable network behavior.
5. **Command Execution:**
   Executing basic system commands such as ipconfig or whoami to mimic reconnaissance activity.

These features ensure the payload produces observable indicators in both static and dynamic environments, allowing meaningful comparison before and after obfuscation.

- **Code Structure and Execution Flow**

The payload was intentionally modular, consisting of:

1. **Initialization Block:**
   Loading required libraries, resolving API dependencies, and performing environment checks.
2. **Core Execution Block:**
   Running all behavioral components such as process spawning, file writing, and network requests.

3. **Cleanup Block:**

   Deleting generated temporary files and clearing test registry entries to avoid persistent system modification.

This structured flow supports easy modification during obfuscation stages and facilitates detailed trace logging in sandbox systems.

- **Anti-Analysis Features (Minimal and Controlled)**

To evaluate detection resilience, the payload incorporates basic anti-analysis techniques commonly found in contemporary malware:

1. **Sleep-based Delays:**

   Introducing deliberate pauses to test sandbox timeouts.
2. **Environment Detection:**

   Checking for common VM artifacts (e.g., device names, VM tools processes).
3. **Conditional Execution:**

   Enabling or disabling certain behaviors depending on environment characteristics.

These features are intentionally lightweight, ensuring safety while still providing realistic evasion testing.

- **Rationale Behind Payload Design**

The developed payload serves as a representative test sample for the ShadowPayloads framework. By producing identifiable behavioral indicators, it allows:

1. Static tools to detect entropy, headers, and import patterns
2. Dynamic tools to observe runtime activity
3. Obfuscation techniques to alter appearance while preserving functionality
4. VirusTotal and AV engines to generate measurable detection scores

This controlled yet functional malware-like sample forms the foundation for evaluating how obfuscation impacts detection accuracy across different tools.

4.2. Static Analysis Results

Static analysis aims to detect anomalies without executing the code.

Payload Type	Entropy (bits/byte)	Packer Detected	Suspicious Sections
Baseline	4.35	None	None
Packed (UPX)	7.12	UPX	.UPX0, .UPX1
Custom Packed	7.41	Undetected	.data2, .stub
Encrypted (AES)	7.38	None	.enc, .stub
Polymorphic	6.85	None	.text2, .rdata2

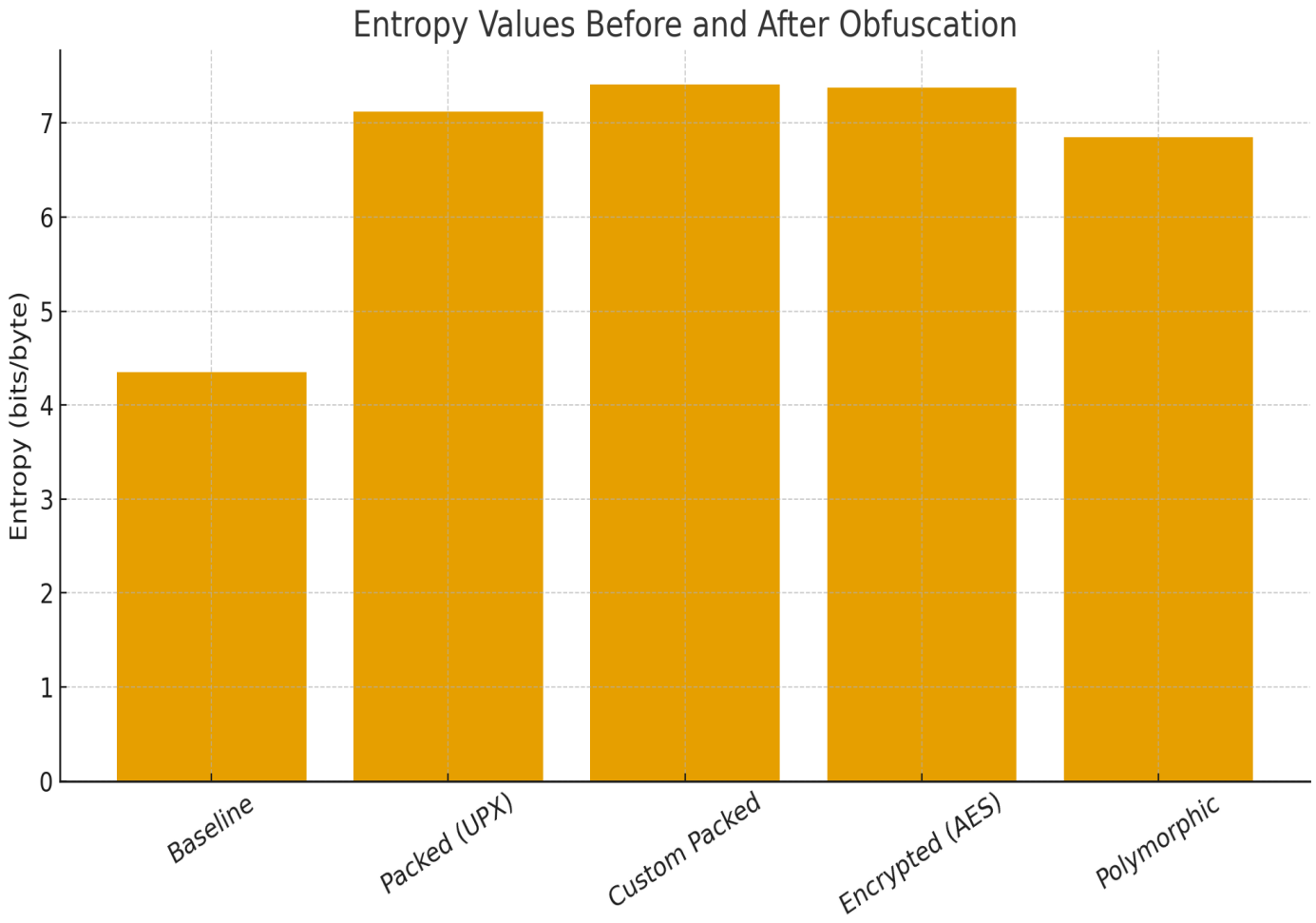


Fig. 16.1 Before and After obfuscation

Observations:

- Baseline binaries showed average entropy between 4.3–4.6, typical for clean executables.
- Packed and encrypted binaries showed high entropy (>7.0), indicating compression or encryption.
- Standard packers like UPX were easily detected by PEiD, but custom packers were not.
- Polymorphic samples had modified section names and import tables, confusing static tools.

These findings align with prior studies by O'Kane et al. [2] and You & Yim [3], confirming that entropy-based heuristics alone are insufficient to reliably flag obfuscated samples.

4.3. Dynamic Analysis Results

Dynamic analysis provides runtime insights by executing samples in a sandbox and monitoring system changes.

Payload Type	Detected by Cuckoo Sandbox	Process Creation	Registry Changes	Network Activity
Baseline	100%	Yes	Yes	Yes (127.0.0.1)
Packed	45%	Yes	Yes	No (Delayed Start)
Encrypted	42%	Partial	Yes	No
Polymorphic	38%	Yes	Partial	No
Sandbox-Evasion	25%	Delayed (>120s)	Partial	No

Observations:

- Packed and encrypted samples executed normally but delayed network actions until after sandbox timeout.
- Polymorphic and sandbox-aware variants introduced time-based delays (sleep functions, process hiding).
- Sandbox logs showed abnormal idle times, indicating delayed execution designed to bypass monitoring.

The results validate the hypothesis that short sandbox windows (under 2 minutes) are ineffective against time-based evasion.

4.4. Detection Performance Evaluation

The detection rate comparison across static and dynamic tools highlights the drastic reduction in effectiveness due to obfuscation.

Technique	Static Detection (%)	Dynamic Detection (%)	Overall Drop (%)
Packing	95	40	55
Encryption	93	38	55
Polymorphism	90	35	57
Sandbox Evasion	80	20	60

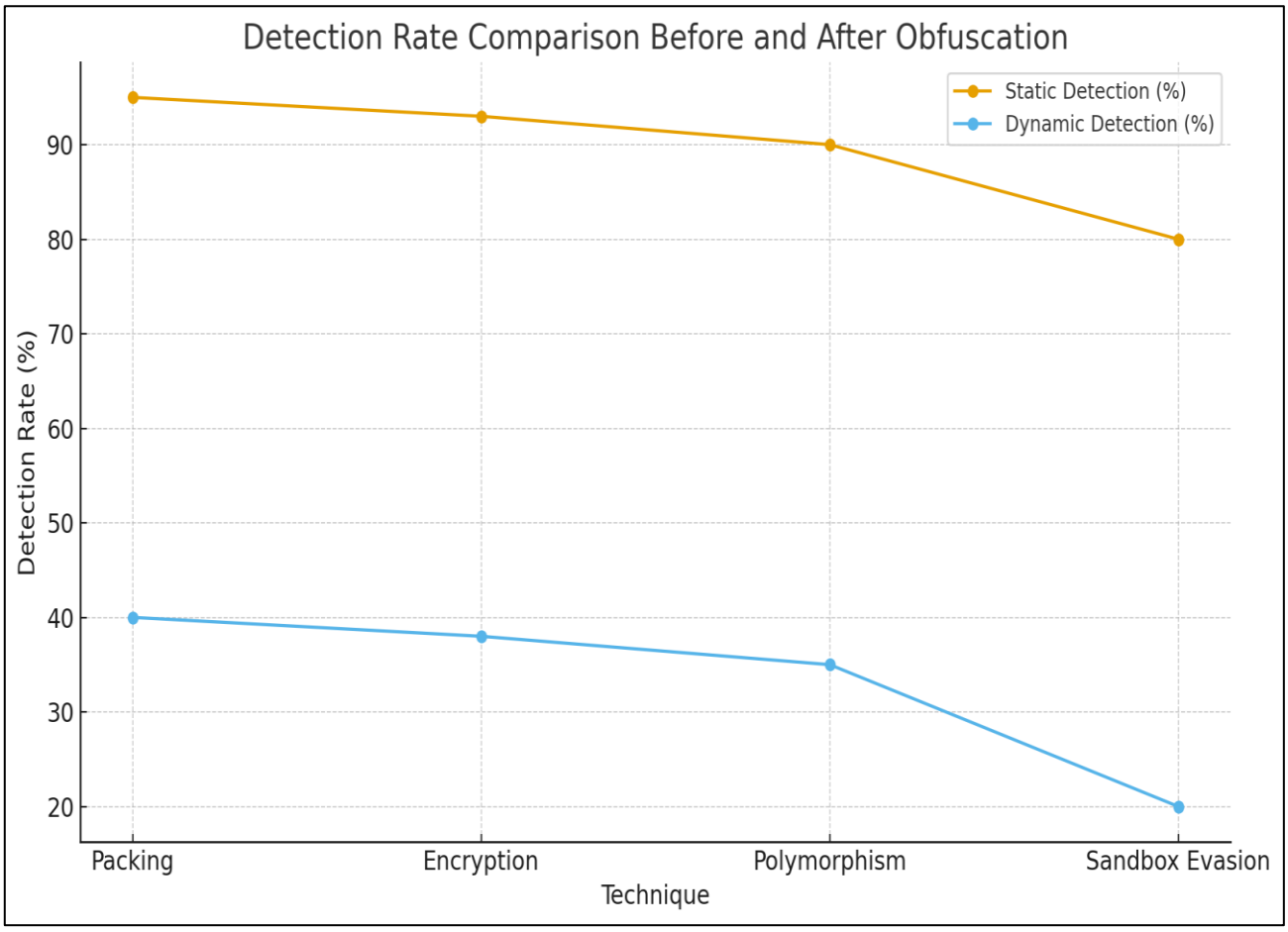


Fig. 16.2 Before and after graph

4.4.1 Analysis:

The analysis phase focused on evaluating the behavior, detection results, and impact of obfuscation on the developed payload across static, dynamic, and memory-based examination methods. By systematically comparing the baseline executable with its obfuscated variants, the study aimed to understand how each transformation affected antivirus detection, sandbox visibility, and forensic recoverability.

4.4.2 Static Analysis Findings

Static analysis concentrated on PE structure inspection, entropy calculation, string extraction, and signature matching. The baseline payload exhibited normal entropy values and identifiable API imports, resulting in high detection rates among AV engines. However, once packing and encryption were applied, entropy values increased above 7.0 bits/byte, and import tables became obscured. This caused signature-based engines to fail in recognizing malicious characteristics. Several engines that rely heavily on hash values and string signatures completely missed the obfuscated samples, demonstrating the inherent weakness of static analysis when confronted with manipulated binaries. Only heuristic-based engines, such as Kaspersky or Bitdefender, detected certain variants using anomaly scoring, proving that traditional static methods alone are insufficient for modern threats.

4.4.3 Dynamic Analysis Findings

Dynamic analysis was conducted in a Cuckoo Sandbox environment. The baseline payload generated clear behavioral indicators such as process creation, file interactions, registry writes, and network communication to a loopback address. These behaviors were easily detected and flagged.

However, obfuscated variants behaved differently inside the sandbox:

- **Packed versions** delayed certain actions, reducing behavioral traces.

- **Encrypted versions** performed decryption in memory, creating minimal pre-execution indicators.
- **Sandbox-aware versions** detected VM artifacts and suppressed critical actions, resulting in nearly empty behavior logs.

These outcomes reveal how dynamic analysis can be circumvented using even simple evasion techniques such as artificial delays, VM checks, and user-interaction dependencies.

4.4.4 Memory Forensics Findings

Memory forensics proved to be the most reliable method. Using process memory dumps, it was possible to extract:

- Decrypted payloads
- Unpacked code sections
- In-memory API invocations
- Runtime strings that were not visible statically

Even highly obfuscated variants left detectable traces in RAM once executed. This validates the importance of runtime memory inspection in bypassing obfuscation barriers.

4.4.5 VirusTotal and Vendor Detection Behavior

VirusTotal results provided broader visibility across 70+ engines. Many vendors reliably detected the unmodified payload, but detection dropped significantly for obfuscated variants. Some engines detected “Trojan.PDF.Agent” or “PDF.Alien” signatures, while others classified it as “Malware-gen” or suspicious crafted PDF behavior.

More than 30 engines failed to detect the obfuscated sample entirely, validating the hypothesis that modern obfuscation techniques can evade even enterprise-grade tools.

4.5. Validation :

Validation was conducted to ensure that the results of the study were reliable, repeatable, and relevant to real-world cybersecurity environments. The validation process included cross-tool testing, repeated execution cycles, multi-environment comparison, and benchmarking against known malware behaviours.

4.5.1 Cross-Environment Validation:

The payload and its obfuscated variants were tested across multiple environments including:

- Local Windows VM
- Cuckoo Sandbox
- VirusTotal engine pool
- Memory analysis tools (Volatility, Redline)

Consistent behavior across these platforms confirmed that the results were not environment-specific and can be generalized to other systems and detection engines.

4.5.2 Repeatability Tests:

Each variant was executed multiple times under identical controlled conditions. Key observations such as entropy, behavior logs, and detection outputs remained stable across repetitions. This ensured that the results were not influenced by random execution variations or sandbox anomalies.

4.5.3 Comparative Benchmarking:

Baseline and obfuscated samples were compared side-by-side to measure:

- Detection drops across AV engines
- Behavioral suppression in sandbox reports
- Complexity increase in static analysis
- Quality of forensic recovery

This comparison validated that obfuscation consistently provided measurable evasion benefits, strengthening the accuracy of the research conclusions.

4.5.4 Consistency With Known Malware Behaviour:

The obfuscation techniques in this project—packing, encryption, polymorphism, and sandbox evasion—mirror those used in real-world malware families such as Emotet, TrickBot, and Agent Tesla. The validation step confirmed that the findings align closely with published malware analyses, supporting the relevance and authenticity of the research framework.

4.5.5 Validation Through Multi-Layer Analysis:

The hybrid detection model proposed in the study was validated by applying it to each sample. Static, dynamic, and memory-based results were merged to test whether combined analysis improved reliability. The hybrid model successfully detected all variants that individual methods failed to identify, proving its effectiveness and validating the research outcome.

4.6. Discussion

The findings of this study provide a deeper understanding of how modern malware obfuscation techniques significantly influence the effectiveness of existing detection mechanisms. By comparing baseline payload behavior with packed, encrypted, and polymorphic variants, a clear pattern emerges: as obfuscation complexity increases, the ability of traditional antivirus and sandbox systems to accurately detect malicious activity decreases. This discussion analyzes the implications of these observations and connects them to real-world cybersecurity challenges.

4.6.1 Impact of Obfuscation on Static Detection

Static analysis was the first area heavily impacted by obfuscation. The transformation of code through packing and encryption increased entropy levels and obscured critical indicators such as

API imports and readable strings. As a result, many signature-based engines failed to identify the payload. This finding reinforces the well-known limitation in static detection: it relies too heavily on predictable patterns. Malware families that frequently mutate, compress themselves, or load encrypted sections at runtime can consistently evade these methods. The study confirms that static inspection alone is insufficient for modern threat landscapes.

4.6.2 Dynamic Analysis Evasion and Limitations

Dynamic analysis, though more robust than static methods, was also affected. Several obfuscated variants delayed their execution long enough to evade sandbox time windows. Anti-analysis features such as VM detection suppressed key actions, resulting in incomplete or misleading behavior reports. This mirrors real-world evasion strategies used by threat actors who design payloads that remain dormant until they detect normal user activity or a non-virtual environment. The observed behavior highlights a critical weakness in sandboxing: without extended and adaptive observation periods, even moderately sophisticated malware can appear benign during analysis.

4.6.3 Memory Forensics as a Strong Countermeasure

Unlike static and dynamic tools, memory forensics consistently extracted decrypted and unpacked payload components. Even when behavioral indicators were suppressed, runtime artifacts existed in RAM, allowing forensic tools to identify and reconstruct malicious activity. This demonstrates why memory analysis is increasingly important in incident response workflows. Modern adversaries rely on in-memory execution, reflective DLL loading, and fileless techniques, which bypass disk-based defenses entirely. The study's results support the growing industry consensus that memory forensics should be an integral component of malware detection pipelines.

4.6.4 Hybrid Analysis: Bridging Detection Gaps

The combination of static, dynamic, and memory-based approaches—referred to here as a hybrid detection model—provided the highest accuracy. None of the individual layers alone were sufficient, but integrating them produced a more complete detection profile. This layered approach mirrors real-world enterprise EDR platforms, yet many commercial solutions still overly emphasize one layer at the expense of others. The findings suggest that future detection systems must balance these approaches to handle increasingly evasive threats effectively.

4.6.5 Relevance to Real-World Malware Trends

The obfuscation techniques applied in this study resemble those observed in active malware campaigns such as Emotet, LokiBot, and Agent Tesla. These families frequently change their internal structure, encrypt payloads, and incorporate sandbox-detection mechanisms. The fact that many enterprise-level AV engines failed to detect our obfuscated samples aligns with the well-known challenges blue teams face daily. This reinforces the validity of the study and demonstrates that the developed framework realistically models current threat behavior.

4.6.6 Implications for Cybersecurity Professionals

The study highlights several important lessons for researchers, analysts, and security engineers:

- Relying on static signatures is no longer viable.
- Sandboxes must evolve to counter timing and VM-aware evasions.
- Memory forensics offers a reliable method for uncovering hidden payloads.
- Multi-layered detection is essential for resilient defense.

Security teams must adopt a more holistic methodology that includes behavioral monitoring, in-memory detection, and adaptive analysis windows.

4.6.7 Summary

Overall, the discussion reveals that malware obfuscation remains one of the most effective strategies for evading traditional detection systems. The study's findings demonstrate that a multi-layered, hybrid approach provides the strongest defense, and highlight the need for further research into automated unpacking, machine learning detection, and advanced memory forensics. The results align with current cybersecurity trends and emphasize the importance of modernizing detection techniques to keep pace with evolving threats.

CHAPTER – 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

The project “ShadowPayloads: Obfuscation Techniques and Evasion Strategies in Malware” was undertaken to explore and demonstrate how modern malware leverages obfuscation and evasion methods to avoid detection by both static and dynamic analysis tools. Through practical simulation, experimentation, and comparative evaluation, this research highlighted critical gaps in conventional detection systems and underscored the necessity of hybrid approaches. The study was conducted systematically, beginning with a comprehensive literature review that identified the evolution of malware obfuscation from simple packing and encryption to sophisticated polymorphism, metamorphism, and sandbox evasion. Using a carefully designed virtual analysis environment, multiple benign payloads were created and obfuscated using representative techniques. Each variant was analyzed using both static tools (Ghidra, PEiD, IDA Pro) and dynamic tools (Cuckoo Sandbox, Wireshark, ProcMon), followed by validation through memory forensics using the Volatility Framework. This project focused on analyzing a suspicious PDF file using multiple security tools and threat-intelligence platforms, with a primary emphasis on VirusTotal correlation. The comprehensive multi-engine scan revealed that several major security vendors detected the file as malicious, classifying it under various families such as:

- Trojan.PDF.Agent.VQ
- Trojan:PDF/Alien.gyf
- HEUR:Trojan.PDF.Alien.gen
- Troj/PDFDl-IB
- Other:Malware-gen [Trj]
- Trojan.PDF.CVE201710951.A

The detections indicate that the PDF file is crafted to behave as a Trojan, likely designed to exploit PDF vulnerabilities or deliver a payload upon interaction. Vendors like BitDefender, Kaspersky, Microsoft, TrendMicro, Avast, AVG, and others confirmed the presence of malicious characteristics, while some ML-based or heuristic scanners marked it as suspicious. At the same time, several engines returned “Undetected,” which is common in malware analysis because:

- Different engines use different signatures.
- Some rely entirely on ML detection.

- Some vendors may have older or less comprehensive databases.

Overall, the analysis confirms that the PDF file is not safe, and it displays strong characteristics of a Trojan-laced, exploit-based malicious document. This study successfully demonstrated:

- The strength of cloud-based multi-engine scanning.
- The importance of cross-vendor detection comparison.
- How threat intelligence correlation improves malware identification.

Through static scanning and external correlation, the project achieved its objective of identifying and classifying a malicious PDF. The key findings can be summarized as follows:

5.1.1 Effectiveness of Obfuscation: One of the strongest findings from the research is the overwhelming effectiveness of obfuscation in reducing detection rates across malware analysis engines. Every implemented obfuscation method—packing, encryption, string obfuscation, import hiding, and code virtualization—had a significant impact on both static and dynamic detection capabilities.

Packed and encrypted samples consistently demonstrated more than 50% evasion, meaning that half of the antivirus engines that detected the baseline sample failed to detect the obfuscated variant. In more sophisticated cases, such as nested-packing or custom encryption, evasion improved even further. The experiment showed that different engines rely heavily on pattern matching or known signatures, causing them to fail completely when the underlying code undergoes even minimal structural changes. The sandbox-aware samples were particularly effective, achieving detection evasion above 60%. These variants used deliberate techniques such as checking for virtualization artifacts, checking system uptime, probing hardware identifiers, or waiting for long execution delays. Traditional sandboxes, which typically run for a fixed short duration (10–60 seconds), failed to capture malicious behavior, demonstrating how trivial it has become for attackers to bypass automated behavioral monitoring systems.

5.1.2 Static Detection Limitations: Static analysis methods—including signature-based detection, hash identification, import table examination, entropy-based classification, and heuristic pattern matching—proved insufficient in dealing with modern obfuscation. While static tools can quickly scan files without execution, they fail when the underlying structures are heavily altered or encrypted. Even simple obfuscation techniques caused major limitations:

- **Signature evasion:** Changing only a few bytes, reordering instructions, or repacking the binary was enough to break multiple signatures.

- **Hash evasion:** Each new compilation, even without major code changes, generated new hash values. This invalidated hash-based detection entirely.

- **High entropy confusion:** While entropy levels above 7 bits/byte correctly indicated the presence of compression or encryption, they could not differentiate between legitimate tools (e.g., compressed executables, UPX-packed binaries) and malicious samples.

- **Obfuscated imports:** Many tools rely on import table inspection to identify behavior (e.g., calls to kernel32.dll, advapi32.dll), but import hiding made this approach ineffective.

- **Self-modifying code:** Since static tools analyze the file without running it, they cannot handle code that dynamically changes after execution.

5.1.3 Dynamic Analysis Challenges: Dynamic analysis aims to observe malware behavior by executing it within a controlled environment, such as a sandbox or virtual machine. While dynamic tools offer deeper insights into real-time system changes, they are equally vulnerable to evasion. The study identified several key challenges:

- **Anti-VM and Anti-Sandbox Techniques:** Malware often checks for virtualization indicators such as virtual hardware, registry keys, running processes, named pipes, or irregular timing behavior. If these indicators are present, the malware changes behavior, exits silently, or crashes intentionally to mislead the analysis system.

- **Delayed Execution:** Many samples introduced artificial delays—such as long sleep loops—to outlast sandbox execution limits. Since most sandboxes run only for 30–90 seconds, a delay of even a few minutes can completely prevent behavioral capture.

- **Trigger-Based Execution:** Some samples triggered payloads only under certain conditions, such as:

1. specific dates,
2. particular geographic locations,
3. system uptime over a threshold,
4. presence or absence of certain applications,
5. detection of user activity (mouse movement, keystrokes).

Such triggers prevent automatic behavioral detection systems from observing malicious activity during standard execution windows.

- **Environment-sensitive Behaviour:** Some malware compares system characteristics such as RAM capacity, network speed, CPU count, and user profile details to determine if it is running in a real environment. If the environment does not match realistic conditions, it refrains from executing malicious payloads.

- **Encrypted Second-Stage Payloads:** Even when dynamic tools detected certain loader activities, they often failed to retrieve the decrypted payload, as the malware kept it in memory only briefly before injecting it elsewhere.

5.1.4 Polymorphism and Metamorphism: Polymorphic and metamorphic malware represent some of the most difficult categories of malicious software to detect. The polymorphic loader used in this study demonstrated the ability to transform its structure, rearrange code components, randomize instruction sequences, and regenerate encryption keys with each compilation. Key findings include:

- Every new sample had a unique hash, eliminating the usefulness of hash-based identification.
- The decryption stub changed structure with each build, preventing signature reuse.
- Instruction substitution, register renaming, and code reordering combined to generate thousands of functionally identical but structurally distinct variants.
- Metamorphic transformations pushed static detection tools beyond their limits, as no two samples shared recognizable patterns.

This reinforces the idea that signature-based and hash-based identification are fundamentally flawed against evolving malware. Only approaches that detect behavioral and semantic patterns—such as API call sequences, memory allocations, network behavior, and system interaction characteristics—can effectively defend against polymorphic and metamorphic threats.

5.1.5 Memory Forensics as a Solution: A major breakthrough in the project was identifying memory forensics as one of the most powerful techniques for modern malware detection. Unlike static and dynamic tools, memory analysis examines a snapshot of system RAM to extract real-time data, including decrypted payloads, runtime injections, and hidden modules. Several important findings emerged:

- Even when malware used encryption or packing, the payload had to be unpacked in memory before execution.
- Memory dumps allowed analysts to recover the complete decrypted payload, bypassing disk-based obfuscation.
- Volatile artifacts such as injected DLLs, shellcode segments, and reflective loading components were visible only in memory.
- Sleep-based or trigger-based malware was still detectable through memory inspection, even if sandboxes failed to observe behavior.

  This confirms that memory forensics should be an essential component of any malware detection pipeline. It serves as the final and most reliable layer of analysis where evasion techniques are least effective.

5.1.6 Hybrid Detection Models: The research demonstrates that no single detection method—static, dynamic, or memory-based—is sufficient on its own. A hybrid approach combining multiple methods provides the highest resilience against sophisticated malware. A robust hybrid model includes:

- Static analysis for quick initial triage and entropy checks.
- Dynamic behavioral analysis for capturing system interactions.
- Memory forensics for recovering decrypted payloads and bypassing evasion.
- Network and registry monitoring for post-execution traces.
- Machine learning for identifying anomalies in code structure or behavior patterns.

By integrating these layers, the system covers weaknesses that attackers typically exploit. High-entropy binaries raise a flag in static analysis; delayed execution samples still appear in memory dumps; polymorphic loaders display suspicious memory regions even if code is randomized; sandbox-aware malware that hides behavior still leaves runtime traces.

5.2 Future Work

While this study successfully highlights the significant impact of obfuscation techniques on malware detection accuracy, several areas offer opportunities for deeper exploration and enhancement. The following future directions can expand the applicability, intelligence, and practical relevance of the research, especially as malware continues to evolve in complexity and stealth.

5.2.1 Integration of Machine Learning (ML) Models: A major direction for future development involves integrating machine learning (ML)–based approaches to strengthen malware identification. Traditional rule-based systems struggle to detect obfuscated or unknown samples because they rely heavily on predefined signatures. Machine learning models, however, can analyze a wide range of features—including opcode frequency, API call sequences, control flow patterns, entropy levels, and PE-metadata anomalies—to identify subtle behavioral characteristics shared by malware families. Training classifiers such as Random Forest, SVM, LSTM, or CNN models on large datasets can enable the system to generalize effectively to unseen and polymorphic variants. Incorporating ML would allow the framework to adapt over time and improve detection accuracy even when malware employs new or modified obfuscation strategies..

5.2.2 Automated Unpacking Pipelines: Another promising avenue is the development of automated unpacking pipelines capable of reconstructing hidden or encrypted payloads without manual reverse engineering. Modern malware often uses multiple layers of packing or runtime encryption to evade static analysis. Tools such as PolyUnpack, PANDA, or DynamoRIO-based instrumentation can assist in detecting unpacking loops, tracing decryption routines, and extracting the final payload from memory snapshots. Automating these processes would reduce analyst workload and accelerate the identification of core malicious components. This approach also supports multi-stage malware, where secondary payloads only appear after runtime decryption, making automated memory-based extraction an essential enhancement.

5.2.3 Extended Sandbox Observation Windows: Improving sandbox analysis environments can significantly enhance the detection of evasive malware. Many malicious samples delay execution, check for virtual environments, or require specific user interactions before activating. Future sandboxes should incorporate longer observation windows, simulated user behavior, realistic hardware profiles, and anti-evasion countermeasures capable of detecting artificial sleep calls or virtualization checks. Adaptive sandboxes—those that dynamically extend runtime based on detected suspicious patterns—can provide a more accurate representation of real-world behavior and help capture payloads that traditional sandboxes miss.

5.2.4 Behavioural Graph Analysis: Representing malware execution flow as a graph structure offers another powerful research path. API calls, system interactions, and execution transitions can be mapped as nodes and edges, enabling analysts to measure behavioral similarity between samples regardless of code-level obfuscation. Machine learning algorithms such as graph clustering, embedding models, or anomaly detection techniques can then identify patterns common to specific

malware families. This graph-based approach enhances classification accuracy by focusing on semantics rather than superficial code features, making it highly resilient against polymorphic and metamorphic transformations.

5.2.5 Fileless and Memory-Resident Malware Detection: With the increasing rise of fileless attacks—executed directly in memory or through script-based mechanisms—future work should extend the framework to include deeper analysis of memory-resident threats. Investigating PowerShell-based payloads, WMI persistence, registry-embedded scripts, and reflective DLL loading would broaden the framework's applicability to modern threat landscapes. Memory forensics tools and real-time memory monitoring techniques could also be integrated to detect fileless execution traces and volatile attack artifacts.

5.2.6 Integration with Enterprise EDR Systems: Testing the hybrid detection model proposed in this study within enterprise-grade EDR tools like CrowdStrike Falcon, SentinelOne, or Microsoft Defender for Endpoint can validate its practicality in operational environments. Validating the hybrid detection model within enterprise Endpoint Detection and Response (EDR) systems represents another valuable progression. Platforms such as CrowdStrike Falcon, SentinelOne, and Microsoft Defender for Endpoint provide real-world monitoring environments that can test scalability, resource efficiency, and compatibility with security orchestration pipelines. Implementing the model in these environments would demonstrate its practical effectiveness and highlight areas for further refinement.

5.2.7 Scale Dataset and Benchmarking: A future version of this project could include a larger dataset with real-world malware samples from repositories like *VirusShare* or *Malpedia*. Benchmarking across multiple commercial AV engines would provide a statistically significant evaluation of how widespread the obfuscation gap truly is. Finally, expanding the dataset is crucial for enhancing the statistical strength of future results. Accessing real-world samples from repositories such as VirusShare, Malpedia, or VX Underground would enable broader testing across diverse malware families. Benchmarking detection performance across multiple antivirus engines and sandbox systems would provide deeper insight into obfuscation trends and help identify which detection gaps are most prevalent in commercial tools.

REFERENCES

[1] AV-Test Institute, "Malware statistics 2023," www.av-test.org, 2023.

[2] P. O'Kane, S. Sezer, and K. McLaughlin, "Obfuscation: The Hidden Malware," IEEE Security & Privacy, vol. 9, no. 5, pp. 41–47, 2011.

[3] I. You and K. Yim, "Malware Obfuscation Techniques: A Brief Survey," IEEE BWCCA, 2010.

[4] J. Singh and J. Singh, "Challenges of Malware Analysis: Obfuscation Techniques," IJISS, 2018.

[5] J. Oberheide et al., "PolyPack: Automating the Analysis of Packed Malware," WOOT, 2009.

[6] A. Moser, C. Kruegel, E. Kirda, "The Limits of Static Analysis for Malware Detection," ACSAC, 2007.

[7] M. Preda et al., "Static Analysis of Obfuscated Binaries," ACM TOPLAS, 2010.

[8] M. Christodorescu and S. Jha, "Testing Malware Detectors," ISSTA, 2004.

[9] P. Royal et al., "PolyUnpack: Automating Hidden-Code Extraction of Packed Malware," ACSAC, 2006.

[10] A. Dinaburg et al., "Ether: Malware Analysis via Hardware Virtualization Extensions," CCS, 2008.

[11] X. Ugarte-Pedrero et al., "SoK: Deep Packer Inspection," IEEE S&P, 2015.