VADViT:Vision Transformer-Driven Memory Forensics for Malicious Process Detection and Explainable Threat Attribution

Yasin Dehfouli

A Thesis Submitted to the Faculty of Graduate Studies

In Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science

Graduate Program in Electrical Engineering and Computer Science

York University

Toronto, Ontario

May 2025

Abstract

Modern malware's increasing complexity limits traditional signature and heuristic-based detection, necessitating advanced memory forensic techniques. Machine learning offers potential but struggles with outdated feature sets, large memory data handling, and forensic explainability. To address these challenges, we propose VADViT, a vision-based transformer model that detects malicious processes by analyzing Virtual Address Descriptor (VAD) memory regions. VADViT converts these structures into Markov, entropy, and intensity-based images, classifying them using a Vision Transformer (ViT) with self-attention to enhance detection accuracy. We also introduce BCCC-MalMem-SnapLog-2025, a dataset logging process identifier (PID) for precise VAD extraction without dynamic analysis. Experimental results show 99% accuracy in binary classification and a 93% macro-average F1 score in multi-class detection. Additionally, attention-based sorting improves forensic analysis by ranking the most relevant malicious VAD regions, narrowing down the search space for forensic investigators.

Acknowledgement

I would like to express my sincere gratitude to my supervisor, Prof. Arash Habibi Lashkari, for his guidance and support throughout this journey. I am also deeply thankful to my friends and colleagues for their encouragement along the way. Above all, my deepest appreciation goes to my parents, whose patience, sacrifices, and unwavering love have been my greatest source of strength, bridging the distance between us and inspiring me every step of the way.

Contents

List of Tables

List of Figures

1 Introduction

Check Point Research reports that global cyberattacks surged by 30% in the second quarter of 2024, reaching an average of 1,999 attacks per organization per week. This represents the most significant increase in cyberattacks observed over the past two years [1]. In recent years, there has been an increasing number of studies focusing on utilizing information from volatile memory to detect malicious activities. This approach is particularly advantageous over other forensic techniques, as every incident leaves traces in memory, enabling more effective detection and analysis.

Malware detection techniques have been extensively explored through various methodologies, mainly categorized into signature-based, behavior-based, machine learning-driven, memory visualization, and hybrid approaches [2, 3]. Signature-based methods detect malware by matching predefined patterns extracted from known threats, making them highly effective for identifying previously cataloged malware [4, 5]. Research efforts have improved database efficiency and automated signature extraction to enhance detection speed and coverage [2, 6]. However, these methods struggle against rapidly evolving malware families. Behavior-based detection shifts focus to monitoring system calls, API sequences, and runtime anomalies, enabling the identification of malicious activity even in the absence of known signatures [7, 8]. Several studies have employed dynamic analysis tools and heuristic-based models to improve behavioral profiling and early threat identification [9, 10].

Beyond traditional techniques, machine learning-based approaches leverage extracted features from static and dynamic analyses to train models capable of distinguishing between benign and malicious files [11, 12, 13, 14]. Researchers have explored a variety of feature sets, including opcode sequences, API calls, and memory artifacts, to improve classifier robustness and adaptability [15, 16, 17, 18]. Additionally, memory visualization-based detection transforms raw memory content into image representations, allowing convolutional neural networks to analyze visual patterns indicative of malware presence [19, 20, 21, 22]. Studies have shown that deep learning models trained on these images can detect malware effectively without requiring handcrafted feature engineering [23, 24, 25, 26]. Finally, hybrid approaches integrate multiple detection methodologies, combining signature matching, behavioral heuristics, and machine learning models to

improve overall accuracy and resilience against obfuscation techniques [24]. These methods have been increasingly explored to leverage the strengths of different approaches while mitigating their individual weaknesses.

Despite these advancements, signature-based techniques fail against zero-day threats and obfuscation, relying on predefined patterns that require constant updates, making them reactive and ineffective against polymorphic malware [6]. Behavior-based detection adapts better but generates high false positives due to the challenge of differentiating benign from malicious actions and is susceptible to evasion tactics where malware mimics legitimate behavior, rendering it unreliable against sophisticated threats [9, 10]. Machine learning-based techniques mitigate some limitations but introduce new ones: feature extraction requires domain expertise, making models subjective to the selected features and prone to omitting critical malware characteristics, while visualization-based methods suffer from resizing gray-scale pictures, handling large memory dumps conversions to image, and vulnerability to adversarial perturbations that degrade accuracy [18, 27, 28].

Moreover, these ML-based methods often rely on outdated feature sets, struggle with generalization to novel malware, and lack explainability, limiting their forensic utility [29]. Hybrid techniques integrate multiple approaches to improve robustness but significantly increase computational complexity, making them inefficient for large-scale forensic investigations, while also introducing compatibility challenges and operational overhead that hinder real-world deployment [24, 26]. These persistent shortcomings across detection methodologies allow adversaries to continually exploit weaknesses, underscoring the need for more adaptive and scalable solutions. Another critical limitation of these approaches is their focus on binary classification, where the outcome merely indicates whether a memory dump is benign or malicious. While useful for security monitoring, this provides little practical value for forensic investigators, who must locate specific regions within memory containing injected code, malicious payloads, or execution traces for further analysis. Regarding the challenges in existing datasets, they primarily consist of system-wide snapshots taken at isolated points in time [30, 31], often failing to capture the temporal evolution of malware execution [32, 18]. Dynamic sandboxing offers an alternative by monitoring malware behavior over time, but it requires continuous execution tracking, which is impractical for large-scale memory analysis [33, 34].

To address these challenges, we propose the BCCC-MalMem-SnapLog-2025 dataset and the VADViT framework designed to improve forensic memory analysis and malware detection by using the Virtual Address Descriptor(VAD). BCCC-MalMem-SnapLog-2025 records process identifiers (PIDs), network activity, and system event logs, enabling investigators to track malware behavior over time. By logging the PID, it eliminates the need to attach a debugger, minimizing interference during execution. Tracking process identifiers across multiple dumps allows forensic analysts to observe malware evolution, isolating malicious activity from broader system attributes. Building upon this dataset, we introduce VADViT, a novel deep learning-based detection framework that utilizes VAD region visualization to detect malware and identify malicious activity within memory dumps. VADViT also pinpoints specific memory regions likely to contain malware, providing forensic investigators with actionable insights into potential threats.

VADViT is a deep learning-based framework designed for forensic memory analysis and malware detection by leveraging the Virtual Address Descriptor (VAD) structure of a process. Using the process identifier (PID), it extracts VAD regions from memory dumps, visualizing them as structured images. Three types of images are generated: Markov images to represent transition probabilities of memory bytes, entropy images to highlight randomness and encryption, and intensity images to capture security attributes like protection flags. These images are then processed by a Vision Transformer (ViT), which utilizes self-attention mechanisms to analyze the spatial and temporal relationships of the memory regions. By tracking the evolution of VAD regions across multiple memory snapshots, VADViT enables forensic investigators to observe malware execution over time without relying on handcrafted features. The attention mechanism of the ViT helps pinpoint the most suspicious regions, narrowing the search space for analysts and improving the efficiency of forensic investigations. The contribution of our work is as follows:

- **Temporal and Isolated Malware memory Dataset** – We introduce BCCC-MalMem-SnapLog-2025, a dataset that captures periodic 30-second memory snapshots, enabling forensic tracking of malware execution over time. By extracting process identifiers (PIDs) at runtime, it isolates malware processes from system-wide noise. Additionally, it logs network activity via TCPDump and system events (Sysmon, security, and application logs) at 3-second intervals, preserving key forensic data. This multi-perspective approach enhances

the detection of stealthy, time-delayed, and transient malware behaviors.

- **VAD-Based Memory Representation and Adaptive Region Prioritization** – We introduce a novel memory visualization approach that structures VAD regions to detect malicious processes. By tracking memory allocations, permissions, and behavioral modifications through VAD regions, VADViT identifies suspicious activity while preserving spatial relationships. It leverages a transformer-based attention mechanism to dynamically prioritize high-risk VAD regions, focusing on those with distinct structural properties and execution patterns. Our method generates Markov images (byte transition patterns), entropy images (randomness and encryption), and custom intensity images (protection attributes and VAD tags), ensuring minimal information loss during resizing.
- **Forensic-Guided and Explainable Memory Analysis** – VADViT enhances forensic investigations by moving beyond binary classification to pinpoint specific malware-affected memory regions. Using a transformer-based attention mechanism, it highlights the probable injected code, payloads, and execution traces, allowing investigators to focus on relevant areas instead of analyzing full-memory dumps. This approach improves detection accuracy while providing interpretable insights, making it more practical for real-world forensic analysis.

2 Literature Review

2.1 Previous Works in Memory Forensics

Memory forensics is a crucial aspect of digital investigations, focusing on volatile memory (RAM) to uncover evidence of malicious activities that may not leave traces on disk [35, 36]. It has evolved from simple string searches to advanced methods analyzing kernel data structures across operating systems [37]. Many researchers have focused on memory forensics and leveraged the rich information in the RAM to explore and develop methods to enhance incident response scenarios. This section discusses the previous work on memory forensic tools, techniques, and related studies. We use the categorization by Kim *et al.* [38] to classify memory forensic techniques into four categories, namely, string-based, signature-based, list traversal-based, and kernel object-based forensics, discussing their corresponding tools and studies.

2.1.1 String-based Forensics

Being the first approach, string-based forensics searches for specific text strings by converting binary data into readable characters to locate information like URLs and file names. This approach relies on keywords like command names or process IDs. Still, it often lacks context, making determining the relevance or origin of identified strings difficult, especially when padded with null bytes or unnecessary characters [38].

2.1.1.1 Strings-Based Forensic tools

The **Strings** tool, pre-installed on most Linux/UNIX systems, searches disk, directory, or file contents for readable text. The "Grep" command complements Strings by filtering and matching text, supporting extended and approximate searches, but adjustments may be needed if the file encoding isn't ASCII [39]. **Bulk Extractor** offers an unstructured analysis of memory dumps, efficiently extracting data like MAC/IP addresses, domain names, and network packets without parsing the filesystem. It includes a wordlist scanner similar to Strings, saving character arrays to output feature files [39]. In 2010, **Stevens** *et al.* [40] successfully recovered command line arguments and executed commands from memory using string-based searches, identifying and analyzing known command line structures stored in plaintext. Similarly, **Okolica** *et al.* [41] retrieved clipboard contents by scanning memory for unique signatures and specific data structures. String-based

tools were primarily used in the past due to limited alternatives and are now less common. Thus, we won't cover additional studies in this section.

2.1.2 Signature-Based Forensics

Signature-based memory forensics detects patterns in memory dumps using predefined rules. While more efficient and accurate than string-based methods, it cannot detect unknown threats without signatures and may struggle with large data sets [38, 42].

2.1.2.1 Signature-Based Forensic Tools

Three forensic tools use a signature-based approach, the most common being **YARA**. YARA identifies and classifies malware through textual or binary patterns, using flexible rules to scan memory dumps, files, or processes [43]. **Autopsy** is an open-source platform with a GUI, automating tasks like metadata extraction and email parsing [44]. **Belkasoft Evidence Center** automates the detection and analysis of digital evidence, identifying hidden or corrupted files and processing large datasets with a GUI [45].

2.1.2.2 Signature-Based studies

Schuster *et al.* [46] developed a signature-based method to extract active processes and threads from memory dumps using signatures like the EPROCESS block. This was later extended by **Schuster** *et al.* [47] to analyze Windows pool allocations, uncovering details about system operations and network connections. **Thantilage** *et al.* [48] created a tool to extract social media artifacts (with specific keywords and data structures) from volatile memory on Windows 10 systems, achieving 95% accuracy with less than 5% False Positive Rate (FPR). **Cohen** *et al.* [49] introduced "Context-Aware Scanning," using YARA and the Windows Page Frame Number (PFN) database to map memory pages to processes, reducing false positives in malware detection. Similarly, **Case** *et al.* [50] developed *HookTracer*, automating API hook analysis by emulating code execution and linking hooks to memory regions, minimizing false positives. **Bajpai** *et al.* [51] proposed a technique to recover cryptographic keys from memory during ransomware encryption, enabling real-time decryption without ransom payment. Moreover, **Sali** *et al.* [52] developed a toolkit integrating Volatility and YARA for the automated detection of malicious processes from RAM images. **Liebler** *et al.* [53] created MRSH-MEM, combining raw memory analysis and approximate matching to detect software loaded in memory by comparing code fragments without

file context reliance. The overview of the memory forensic studies utilizing the signature-based approach is shown in the table 1.

Table 1: Signature-Based Memory Forensic Studies

Authors	**Objective**	**Methodology**	**Proposed Tool Name**	**Results**
Schuster *et al.* (2006)	Extract active processes and threads	Signature-based scanning for EPROCESS block in memory dumps	N/A	Detailed extraction of active system processes and threads at capture time
Schuster *et al.* (2008)	Analyze Windows pool allocations	Scanning specific memory areas for process management and network connections	N/A	Detailed insights into system activities and resource allocations
Thantilage *et al.* (2019)	Extract social media artiForensic analysts can trace all running processes and their data by tracing these linksed keywords, metadata, and data structures	N/A	95% accuracy, ¡5% false positives, memory dumps analyzed in under 30 minutes (up to 16GB)	
Cohen *et al.* (2012)	Improve malware detection accuracy	Context-Aware Scanning using PFN database to map memory pages to processes	N/A	Significant reduction in false positives
Case *et al.* (2014)	Automate API hook analysis	Emulated code execution capturing hook execution paths, linked to memory regions	HookTracer	Minimized false positives, detailed analysis of malicious and benign hooks
Bajpai *et al.* (2018)	Counter ransomware	Extracted cryptographic keys during encryption from temporary key storage in memory	N/A	Recovery of symmetric and asymmetric keys, enabling successful data decryption across ransomware strains
Sali *et al.* (2021)	Detect malicious processes from RAM images	Toolkit with signature scanning and artifact extraction using Volatility and YARA	N/A	Automated detection and analysis of malicious processes
Liebler *et al.* (2022)	Detect software loaded in memory	Integrated raw memory analysis with approximate matching to identify code fragments	MRSH-MEM	Memory carving and identification of code segments without relying on file context, using approximate disassembly

2.1.3 List Traversal-Based Forensics

List traversal-based memory forensics generates a list of processes by traversing memory entries of process objects and using the associated data. In Windows, for example, processes are managed in the "_EPROCESS" structure, organized as a doubly linked list. By tracing these links, forensic analysts can identify all running processes and their data. This widely used technique effectively recovers OS data structures but is vulnerable to anti-forensic techniques like Direct Kernel Object Manipulation (DKOM) and relies heavily on specific OS data structures [38, 54].

2.1.3.1 List Traversal-Based tools

MemParser [54], analyzes Windows memory dumps by traversing the "_EPROCESS" structure to extract process information. It compiles a list of all running processes at the time of memory capture [54, 55]. Similarly, **Kntlist**, also from the DFRWS 2005 Forensics Challenge, uses the same method to identify active processes and threads, requiring regular updates to maintain accuracy [55, 56]. It employs the same methodology of traversing kernel memory structures to identify active processes and threads. It compiles a list of active processes and threads. The effectiveness of both tools depends on specific memory structures.

2.1.3.2 List Traversal-Based Studies

Unlike traditional methods that focus on isolated evidence, **Wang** *et al.* [57] introduced the Computer Forensic Analysis Model (**CERM**) for reconstructing evidence chains from volatile memory. It involves memory acquisition, memory image analysis, and a list traversal-based time relevance analysis, correlating temporal data across processes to build a robust evidence chain. **Sun** *et al.* [58] introduced REVIVER, a memory forensics tool that reverse engineers data structures from memory dumps without prior execution traces. Using a list traversal-based approach, it achieves 98.1% accuracy with minimal runtime overhead (1.8%). **Tran-Quoc** *et al.* [59] developed MemInspect, a memory forensics tool extracting OS-independent artifacts via paging structures in x86 and x86-64 CPUs. It translates virtual to physical addresses without OS-specific kernel information, reconstructing process-linked lists and detecting hidden processes.

2.1.4 Kernel Object-Based Forensics

This forensic method uses kernel object structures and attributes to extract and analyze data from memory dumps. It involves identifying the OS memory management system and reverse-

engineering kernel objects. It offers high accuracy and reliability by reanalyzing memory loading processes, making it more resistant to anti-forensic techniques like DKOM than list traversal-based methods. This robustness comes from focusing on kernel object structures instead of process lists[38, 60].

2.1.4.1 Kernel Object-Based Tools

Volatility analyzes RAM dumps by parsing OS data structures and extracting information like processes, kernel modules, and network connections. It offers various plugins for tasks such as process enumeration and registry extraction [61, 62]. **Rekall** builds on Volatility with improved performance and plugin versatility. It handles large memory dumps, though its discontinued development limits future use [63]. **Stadlinger** *et al.* [64] extended Rekall with plugins like ”heapdump” and ”heapsearch” for analyzing heap memory in Linux user space, identifying data like login credentials and SSH keys. **Redline** performs the forensic analysis by collecting artifacts without system shutdown, with timeline analysis capabilities [65]. **Memoryze** captures and analyzes system memory snapshots to detect threats, offering rootkit detection and post-mortem analysis [66].

2.1.4.2 Kernel Object-Based Studies

Cohen [67] addressed kernel variability by developing a profile indexing technique to identify the exact kernel version in memory dumps using pre-existing profiles. **Block** *et al.* [68] developed Rekall plugins, ”heapdump” and ”heapsearch,” for efficient Glibc heap data extraction. **Song** *et al.* [69] developed DeepMem, a Graph Neural Networks (GNN)-based tool for detecting kernel data structures, achieving over 99.5% precision and recall for detecting structures like ”_EPROCESS” and ”_ETHREAD”. Unlike list-traversal and signature-based methods, DeepMem is resilient to DKOM attacks. **Zhang** *et al.* [70] introduced MRB-PTE that tracks process memory in real-time with minimal overhead by using a reserved bit in the PTE of target memory pages. **Maggio** *et al.* [71] introduced Seance, a framework that adapts to changes in forensically important binaries, maintaining tool effectiveness. **Duke** *et al.* [72] compared memory forensic techniques across Apple M1 and Intel architectures to address tailored forensic approaches in different hardware environments. **Qi** *et al.* [73] enhanced binary-only memory forensics with a logic inference method for automatic kernel object identification. **Fernández-Alvarez** *et al.* [74] developed Modex and Intermodex on the Volatility 3 framework to enhance module extraction from Windows memory

dumps. Table 2 demonstrates a detailed overview of the studies.

Table 2: Kernel Object-Based Memory Forensic Studies

Authors	**Objective**	**Methodology**	**Proposed Tool Name**	**Results**
Cohen *et al.* (2015)	Address memory analysis challenges across Windows kernel versions	Profile indexing technique to identify the exact kernel version using pre-existing profiles	N/A	Solving kernel variability problem, effective across kernel versions
Song *et al.* (2018)	Detect kernel data structures in memory dumps	Graph Neural Networks (GNN) to construct memory graphs and classify segments using weighted voting	DeepMem	Over 99.5% precision and recall for detecting _EPROCESS and _ETHREAD
Zhang *et al.* (2018)	Live tracking of process memory	Using reserved bit in PTE to monitor process memory in real-time with minimal overhead	MRB-PTE	7.9% performance overhead when monitoring 10 memory pages
Maggio *et al.* (2021)	Detect and adapt to changes in binaries	Monitoring kernel objects and modifications, focusing on binary updates	Seance	Maintained effectiveness of forensic tools despite binary updates
Duke *et al.* (2021)	Compare forensic techniques across hardware architectures	Compared techniques on Apple M1 and Intel architectures using Volatility	N/A	Highlighted need for hardware-specific forensic approaches
Fernández-Alvarez *et al.* (2023)	Enhance module extraction from Windows memory dumps	Built plugins on Volatility 3 for intra/interdump module extraction	Modex, Intermodex	Enhanced detection of DLL hijacking, analysis of inconsistencies in module paths and sizes
Qi *et al.* (2022)	Enhance binary-only memory forensics	Method for generating profiles through logic inference to identify and analyze kernel objects	N/A automatically	Improved automated identification and analysis of kernel objects
Block *et al.* (2017)	Improve user space analysis with focus on heap data	Introduced plugins for Rekall framework, focusing on Glibc heap, with tools like "heapdump" and "heapsearch"	Heapdump, Heapsearch	Efficient extraction of heap-related data from user space

The taxonomy of general tools and techniques made by authors is shown in Figure 1 with Table 3 that categorizes tools by core functionality, with many, such as Volatility, handling multiple approaches, highlighting their versatility. Most tools support Windows, with about half supporting other OSes, indicating potential platform bias. There's a split between GUI tools, which are user-friendly, and terminal-based tools, which are more efficient but require expertise. Advanced

tools, particularly kernel object-based ones, have steep learning curves. Few tools handle memory acquisition directly, requiring external methods, while kernel object-based tools are gaining lots of attention due to their deeper OS inspection.

Researchers have conducted comparative studies on memory forensic tools. **Firoozjaei** *et al.* [75] compared Volatility, Autopsy, and Redline, focusing on malware analysis and resource usage. Volatility and Autopsy identified exited malware processes, while Redline struggled with injected processes. Volatility had the lowest resource consumption but lacked memory acquisition capabilities, requiring external tools. With its user-friendly interface, Autopsy was resource-intensive, and Redline, though it was easy to use, consumed more CPU. **Parekh** *et al.* [76] noted that Volatility supports multiple OSes but needs external tools for acquisition, while Autopsy excels at reporting but is resource-heavy. Memorize offers real-time analysis for Windows but is limited to that platform. Belkasoft is comprehensive but complex, handling evidence from hard drives and mobile devices. These studies lack technical details and omit many tools, highlighting the need for standardized evaluation frameworks due to the evolving nature of malware.

Table 3: Memory Forensics Tools Overview

Tool Name	Forensic Approach	Supported Platforms	Advantages	Disadvantages	Acquisition Capability	User Interface
Strings	String-based	Windows, Linux, macOS	Directly extracts ASCII and Unicode strings, effective for identifying plaintext data	Ineffective against encrypted or obfuscated data; lacks correlation with specific processes	No	Terminal
Bulk Extractor	Signature-based, Can perform String-Based Forensics	Windows, Linux, macOS	Rapid extraction of bulk data artifacts; supports parallel processing	Limited forensic context; may not associate artifacts accurately with originating processes	No	Terminal
YARA	Signature-based Can perform String-Based Forensics	Windows, Linux, macOS	Highly customizable with user-defined rules for pattern matching	Rule development requires deep expertise; performance may degrade with poorly written rules	No	Terminal
Autopsy	Signature-based, Can perform String-Based Forensics	Windows, Linux, macOS	Intuitive GUI with integrated analysis modules; includes robust reporting features	High resource consumption; requires familiarity with TSK for effective usage	No	GUI
MemParser	List Traversal-Based, Can perform String-Based and Signature-based Forensics	Windows	Specialized in parsing complex Windows memory structures; provides detailed output for advanced analysis	Steep learning curve; primarily limited to Windows	No	Terminal
Kntlist	List Traversal-Based, Can perform String-Based and Signature-based Forensics	Windows	Focused on kernel memory list traversal; deep inspection of Windows kernel objects	Limited documentation and support; requires strong understanding of Windows internals	No	Terminal
Volatility	Kernel Object-Based, Can perform others	Windows, Linux, macOS	Comprehensive toolset for memory analysis with a wide range of plugins	Requires significant technical expertise; plugin quality varies	Yes	Terminal
Rekall	Kernel Object-Based, Can perform others	Windows, Linux, macOS	Advanced memory analysis with live memory acquisition support; extendable with plugins	Documentation and support less extensive than Volatility; can be slow on large captures	Yes	Terminal
Redline	Kernel Object-Based, Can perform others	Windows	User-friendly interface for thorough forensic analysis, including artifact correlation	Limited to Windows; requires substantial system resources	Yes	GUI
Memoryze	Kernel Object-Based, Can perform others	Windows	Capable of memory acquisition and detailed analysis of processes, drivers, and services	Windows-only; challenging to use without specific knowledge of Mandiant's tools	Yes	GUI

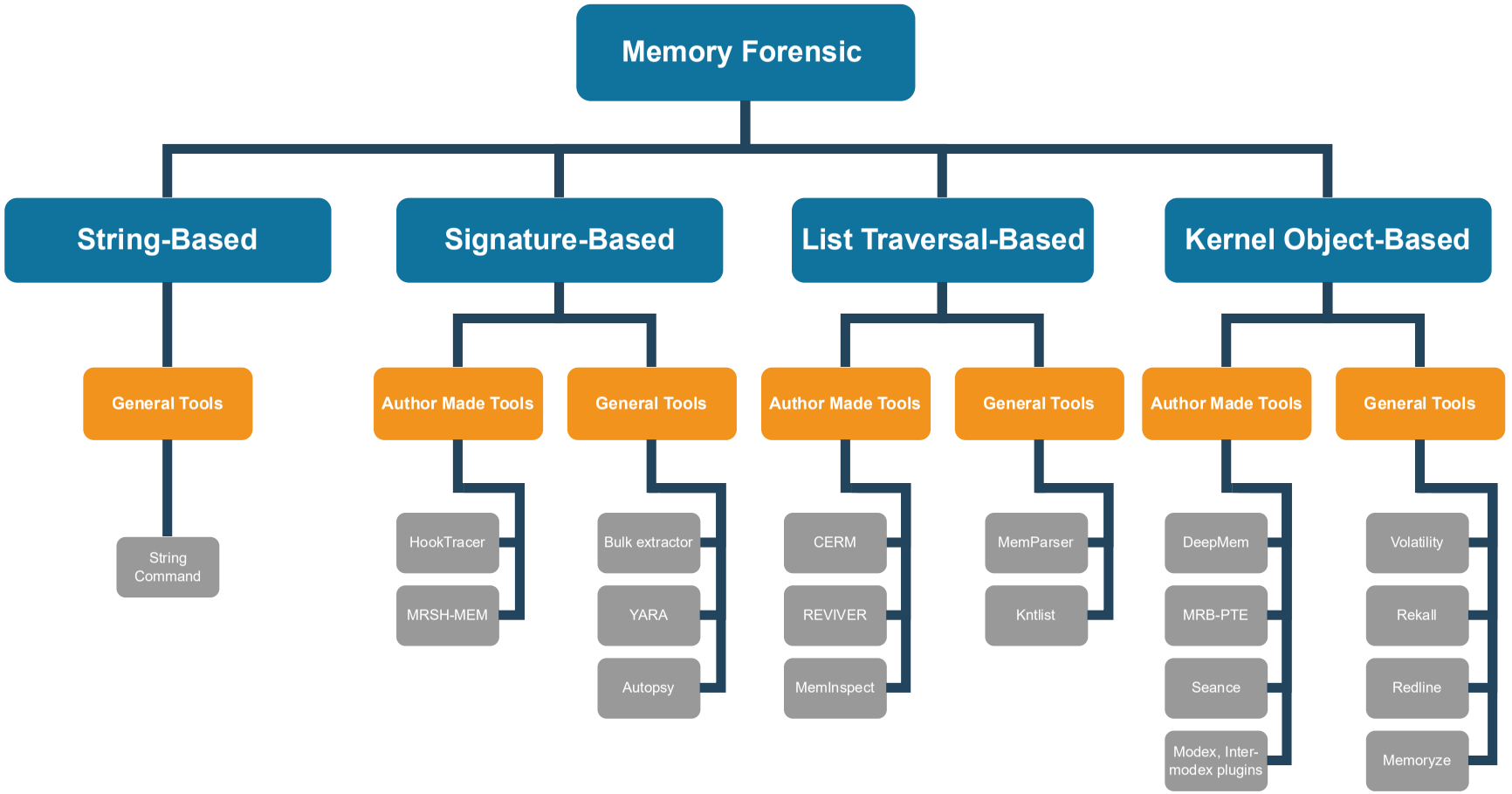


Figure 1: Memory Forensic Techniques and Tools Taxonomy

Regarding the challenges beyond tool comparison, **Ottmann** *et al.* [77] identified reliability issues with memory forensic tools, showing frequent inconsistencies due to system load and concurrency during memory dump acquisition. Their Volatility study revealed that many memory dumps were meaningless or inconsistent. Few tools can eliminate the need for expert intervention, and detecting process injection and termination remains challenging across all tools [75]. Another challenge is the reliance on kernel-specific profiles and the semantic gap, where low-level memory data cannot be easily mapped to higher-level structures. Custom OS kernels further complicate process identification without kernel-specific details [59].

2.2 Previous Works in Malware Detection

2.2.1 Non ML Approaches

Traditional malware detection uses signature-based detection and heuristic analysis to search memory for traces and anomalies, often involving manual expert analysis. While effective for common threats, these methods struggle with sophisticated malware due to memory dump complexity.

2.2.1.1 Signature-Based Approaches

Signature-based malware detection matches memory data patterns against a database of known malware signatures, similar to a fingerprint. It performs well in detecting known malware but cannot detect unknown variants [2]. **Michalas** *et al.* [6] developed **MemTri**, a signature-based memory forensics tool using Bayesian Networks and the Volatility framework. MemTri analyzes memory images, focusing on browser, messenger, and document artifacts, achieving 95.7% accuracy in normal mode and 80% in scan mode. Similarly, **Sanjay** *et al.* [4] proposed a methodology aimed explicitly at detecting fileless malware. Their workflow starts with malware delivery via phishing emails, leading to persistence through code injection into system processes and PowerShell script execution. They used sandboxing, script emulation, and YARA rules for pattern matching. With 500 malware and 500 legitimate software samples, their method achieved a 97.6% detection accuracy. **Botacin** *et al.* [5] introduced MINI-ME, a hardware-assisted tool that performs in-memory pattern matching for detecting fileless malware during write-to-read operations. It achieved 100% detection with less than 1% performance overhead.

2.2.1.2 Heuristic Approaches

Unlike signature-based detection, which matches data against a known database of malware signatures, heuristics are rules, approaches, and algorithms to detect anomalies indicative of malicious intent [3]. This approach is more effective in identifying new malware and addresses the main shortcomings of signature-based detection. **Case** *et al.* [78] proposed a heuristic-based method for detecting Objective-C malware on Mac OS X through memory forensics. Their approach used the "realized_class_hash" variable to enumerate classes, and they introduced "mac_observers" and "mac_swizzled" Volatility plugins to detect keystroke logging and method swizzling. The study analyzed 188 malware samples and 177 legitimate applications, achieving a 96.8% detection rate and 1.4% FPR. **Case** *et al.* [7] later developed Hooktracer, which detects keystroke loggers by emulating memory-captured code, generating API call traces. Tested on a Turla-infected VM using Volatility and the "hooktracer_messagehooks" plugin, Hooktracer identified logged keystroke data and network transmission without manual reverse engineering.

Yang *et al.* [8] proposed a Linux memory forensics-based model for network attack detection using a list traversal algorithm to scan memory dumps and identify anomalies in network processes. Integrated with system call tracing, it improved real-time detection speed and accuracy. Focusing on temporal analysis, DLL comparisons, and cross-system checks. **Pottaraikkal** *et al.* [9] developed a fileless malware detection method using 12 memory images captured at different timestamps. Tested on Windows VMs, their multiple memory image algorithm achieved 100% detection using "Malfind" and "Pslist" Volatility plugins. **Thakar** *et al.* [10] used baseline comparison with DumpIt and Volatility to detect malware by comparing clean and suspicious memory images, successfully identifying new processes and altered services.

2.2.2 ML Approaches

ML has been applied to malware detection in recent years, utilizing various classifiers and deep learning models. It outperforms traditional signature-based and heuristic methods in identifying patterns and anomalies. This section reviews recent research on machine learning-based malware detection.

2.2.2.1 Memory Visualization Approaches

Several studies have used computer vision to classify malware by converting it into images, achieving good detection results [25]. Many researchers have converted malware executables into bytes instead of memory dumps due to the large dump file size. However, visualizing memory data reveals differences between malicious and benign dumps and similarities within malware families [24]. This survey will focus on studies visualizing memory dump contents.

Bozkir *et al.* [23] used the Dumpware10 dataset to convert memory dumps into RGB images and extracted two descriptors: Histogram of Oriented Gradients (HOG) and Global Image Structure Tensor (GIST). Sequential minimal optimization (SMO) with RBF kernel achieved 96.39% accuracy. UMAP manifold learning improved detection accuracy for unknown malware. **Tran** *et al.* [26] introduced an OS-independent framework using MemGen to simulate virtual environments and generate memory dump datasets. After converting dumps to images using bin2png and extracting HOG and GIST features, Random Forest (RF) achieved 93.75% accuracy. In 2022, **Shah** *et al.* [21] improved image quality by applying CLAHE and wavelet compression, reducing image dimensions to 112x112 and achieving 97.01% accuracy with SVM. **Shah** *et al.* [22] later refined the approach using Non-Linear Means denoising and uniform image resizing (224x224 pixels), improving accuracy to 97.82%.

Earlier in the same year, **Naeem** *et al.* [19] applied t-SNE for dimensionality reduction on RGB images (224x224, 300x300), using Local Binary Patterns (LBP) and Gray Level Co-occurrence Matrix (GLCM) features with a Convolutional Neural Network (CNN), achieving 98% accuracy. A year later, **Naeem** *et al.* [20] expanded this work, achieving 99.1% accuracy for Windows, 94.3% for Android, and 99.8% for obfuscated malware using a deep ensemble model and explainable AI. **Liu** *et al.* [25] introduced MRm-DLDet, converting memory dumps into ultra-high-resolution images and cropping them into sub-images with a non-overlapping sliding window. The model achieved 98.34% accuracy using ResNet-18 for feature extraction. The overview table and taxonomy of existing memory visualization research in malware detection are demonstrated in table 4 and figure 2, respectively.

Table 4: Existing Memory Visualization Techniques in Malware Detection (Sorted by accuracy)

Authors (Year)	Dataset	Conversion approach	Image sizes	Preprocessing	Extracted Features	Model	Results (Accuracy)
Liu *et al.* (2023)	Liu *et al.*'s	RGB images with custom algorithm	High-resoloution RGB images cropped into sub-images of 224x224	Memory page deduplication, CLAHE	N/A	ResNet-18 with a GRU-attention mechanism	98.34%
Naeem *et al.* (2023)	Dumpware10, CIC-MalMem-2022, Android dataset	RGB and grayscale images with custom algorithm	224x224, 300x300	Explainable AI approach	LBP, GLCM	CNN+MLP	99.1% Windows, 94.3% Android, 99.8% Windows obfuscated
Naeem *et al.* (2022)	Dumpware10	RGB images using bin2png	224x224, 300x300	t-SNE for dimensionality reduction	LBP, GLCM	CNN	98%
Shah *et al.* (2022)	Dumpware10	RGB images using bin2png	224x224	Non-Linear Means denoising	GIST, HOG	SVM	97.82%
Shah *et al.* (2022)	Dumpware10	RGB images using bin2png	112x112, 56x56	CLAHE, wavelet transform compression	GIST, HOG	SVM	97.01%
Bozkir *et al.* (2021)	Dumpware10	RGB images using bin2png	224xfixed height, 224x224, 4096xfixed height	Lanczos interpolation	GIST, HOG	SMO (RBF)	96.39%
Tran *et al.* (2021)	Tran et al.'s	RGB images using bin2png	300x300	MemGen to simulate virtual environments	HOG, GIST	RF	93.75%

2.2.2.2 Byte-wise Feature Engineering Approaches

Exploring raw memory dump files goes beyond visualizing binary data, as these files hold extensive byte sequences representing all the information in memory. Therefore, researchers have used these byte sequences as feature vectors for machine learning classifiers.

Nissim *et al.* [79] introduced a malware detection method for VMs in private clouds by extracting 32-byte shingles from dumps and hashing them into MinHash signatures with 200 hash values each. The approach can compare dumps without storing them in memory achieving 100% True

Positive Rate (TPR) for ransomware and RATs. It also resulted in 1.8% FPR for ransomware and 0% for RATs, saving over 86% in time and storage. Metrics like accuracy and F1 score were not reported. **Kumara** *et al.* [80] proposed the A-IntExt system for detecting unknown malware at the hypervisor level, combining VM Introspection (VMI), Memory Forensics Analysis (MFA), and machine learning. The MFA part captured _EPROCESS patterns and used Volatility's "procdump" plugin to generate 4-byte N-grams. The features are refined using selection techniques like Information Gain, N-gram Frequency, and Chi-Square. The dataset included 3750 malware and 4500 benign files, and an RF achieved 99.55% detection accuracy in 10-fold cross-validation with 0.004 FPR and 6.3% performance overhead.

2.2.2.3 Memory Feature Engineering Approaches

Memory dumps contain significant information, including active and terminated processes, Dynamic Link Libraries (DLLs), running services, registry entries, and active network connections [81]. Consequently, extensive research has focused on extracting data from memory dump files and using this data as numerical or categorical features for machine learning algorithms. Although this approach requires domain knowledge, its efficiency has been validated by many studies. This section will focus on these papers categorized chronologically into malware detection and classification.

• Malware Detection

In one of the earliest studies, **Mosli** *et al.* [82] used the Term Frequency-Inverse Document Frequency (TF-IDF) on his dataset for feature selection on DLL, registry, and API function call features extracted using Volatility. They achieved 96% accuracy with an SGD model (hinge loss). **Sihwail** *et al.* [81] used six types of memory features (API calls, DLLs, etc.) with Information Gain (IG) and Correlation for feature selection, achieving 98.5% accuracy with SVM. Similarly, **Arfeen** *et al.* [14] focused on ransomware detection by extracting process-related memory features. Using Chi-square and Lasso for feature selection, they employed XGBoost, achieving 88.7% accuracy. In 2022, **Carrier** *et al.* [15] proposed a stacked ensemble method using RF, Naive Bayes (NB), and Decision Trees (DT), with Logistic Regression (LR) as the meta-learner, achieving 99.00% accuracy. Earlier in the same year, **Dener** *et al.* [16] achieved the highest accuracy of 99.97% using LR on 52 selected features. **Al-Qudah** *et al.* [13] applied Principal Component Analysis (PCA) to

reduce features and used One-Class SVM (OCSVM) for detection, achieving 99.4% accuracy. In 2023, **Khalid** *et al.* [27] employed RF for malware detection on normalized features, achieving 93.33% accuracy. Earlier in the same year, **Alani** *et al.* [11] used XGBoost with Recursive Feature Elimination (RFE) to reduce features to five, achieving 99.89% accuracy and identifying "services sharing processes" as the most significant feature using SHAP explanations. **Aljabri** *et al.* [12] achieved 97% accuracy in ransomware detection using RF with VolMemLyzer. Finally, in the latest study, **Maniriho** *et al.* [83] proposed a stacked ensemble method with deep autoencoders, achieving 98.82% accuracy for malware detection. The table 5 demonstrates the overview of existing memory feature engineering studies.

• **Malware Classification**

In 2021, **Lashkari** *et al.* [84] introduced VolMemLyzer and performed both binary and multi-class classification on their dataset. RF and DT achieved a 93% TPR and 6.6% FPR. In the same year, **Panker** *et al.* [85] focused on malware detection in Linux VM cloud environments. Their methodology leveraged hypervisor memory acquisition to avoid detection, and seven experiments were conducted using various classifiers. They achieved 99.93% TPR and 0.17% FPR for unknown malware detection, with 99.71% accuracy for fileless malware. In 2022, **Mezina** *et al.* [86] applied a dilated CNN featuring multiple convolutional blocks and a focal loss function to the CIC-MalMem-2022 dataset. They reached a 75.13% F1 score for multiclass classification and near-perfect binary classification accuracy with RF. **Sazzed** *et al.* [87] used Chi-square, ANOVA, and Mutual Information to reduce the feature set of the same dataset by 25% and 50%, with RF achieving a 0.75 F1 score on half of the features in a 5-class Trojan family classification.

Hasan *et al.* [17] implemented a memory dump analysis system using CIC-MalMem-2022. They performed ADASYN class balancing with an XGBoost classifier, achieving 94.27% F1 score and 99.99% accuracy in multi-class and binary classification, respectively. Most recently, **Hossain** *et al.* [18] proposed an ensemble-based system for obfuscated malware detection on the same dataset. Using Gradient Boosting and feature selection with Chi-square and Mutual Information, the model reached 100% accuracy for binary and multiclass classification with SMOTE and 99.96% without SMOTE. The overview of malware classification studies is shown in Table 6. Since all these studies also detect malware, the integrated taxonomy of memory feature engineering approaches is presented in Figure 2.

Table 5: Existing Memory Feature Engineering Techniques in Malware Detection (sorted by accuracy)

Name	Dataset	Tools utilized	Preprocessing	Feature Selection	Model	Results (Accuracy)
Dener *et al.* (2022)	CIC-MalMem-2022	N/A	Removal of duplicates, missing values handling, normalization	Selection of 52 features	Logistic Regression	99.97%
Alani *et al.* (2023)	CIC-MalMem-2022	N/A	RFE (5 features)	Recursive Feature Elimination (RFE)	XGBoost (with SHAP)	99.89%
Al-Qudah *et al.* (2023)	CIC-MalMem-2022	N/A	Removal of duplicates, handling of missing values, encoding, normalization	PCA reduction to 10 features	One class SVM	99.4%
Carrier *et al.* (2022)	CIC-MalMem-2022	VolMemLyzer-V2	SMOTE for dataset balancing	N/A	Stacked ensemble (NB, RF, DT with LR as meta-learner)	99.00%
Maniriho *et al.* (2024)	Maniriho *et al.*'s dataset	VolMemLyzer-V2	Removal of irrelevant values, standard scaling, label encoding	N/A	Stacked ensemble with autoencoder (SVM, SGD, BLR, CatBoost as the meta-learner)	98.82%
Sihwail *et al.* (2021)	Sihwail *et al.*'s 2021	Volatility	N/A	IG, Correlation	SVM	98.5%
Aljabri *et al.* (2024)	Aljabri *et al.*'s dataset	VolMemLyzer-V2	Handling NaN values, MinMaxScaler scaling	N/A	Random Forest	97%
Mosli *et al.* (2016)	Mosli *et al.*'s 2016	Volatility	N/A	TF-IDF	SGD with hinge loss	96%
Khalid *et al.* (2023)	Khalid *et al.*'s dataset	Volatility	StandardScaler normalization	N/A	Random Forest	93.33%
Arfeen *et al.* (2022)	Arfeen *et al.*'s dataset	Volatility	N/A	Chi-square, Lasso, correlation matrix	XGBoost	88.7%

2.2.2.4 Hybrid Approaches

Integrating volatile memory features with dynamic and static features, such as byte-wise or visual features, enriches the feature set and enhances malware detection.

Table 6: Existing Memory Feature Engineering Techniques in Malware Classification (sorted by F1 Score)

Name	Dataset	Tools utilized	Preprocessing	Feature Selection	Model	Results (F1 Score)
Hossain *et al.* (2024)	CIC-MalMem-2022	N/A	Handling missing data, label encoding, normalization, SMOTE for class balancing	Chi-square, Mutual Information (MI)	Gradient Boosting	1.0
Hasan *et al.* (2023)	CIC-MalMem-2022	N/A	Cleaning, standardization, label encoding, class balancing with ADASYN	N/A	XGBoost	0.9427
Sazzed *et al.* (2023)	CIC-MalMem-2022	N/A	N/A	Chi-square, ANOVA, MI	RF (5-class Trojan family)	0.75
Mezina *et al.* (2022)	CIC-MalMem-2022	N/A	Normalization, Handling missing values	N/A	Dilated CNN	0.7513
Lashkari *et al.* (2021)	Lashkari *et al.*'s dataset	VolMemLyzer	N/A	N/A	RF, DT	0.93
Panker *et al.* (2021)	Panker *et al.*'s dataset	Volatility	Class balancing	N/A	RF	Not provided

Table 7: Existing Hybrid Malware Detection Approaches (Ranked by Accuracy)

Name	Dataset	Tools utilized	Features	Model	Results (Accuracy)
Sihwail *et al.* (2019)	Sihwail *et al.*'s 2019 dataset	Cuckoo Sandbox, Volatility, WEKA	API function calls from IAT + behavior reports	SVM	98.5%
Dai *et al.* (2019)	Dai *et al.*'s dataset	Cuckoo Sandbox with a soft probe on the guest side	API call sequences + memory dump grayscale images + hardware performance counters	BiGRU, VGG19 CNN	96.9%
Sharafaldin *et al.* (2017)	Zeus botnet-infected VMs dataset	Volatility	DNS logs + memory dumps + DGA-generated domains	K-Means clustering	93.5%
Tomiyama *et al.* (2023)	Author-made dataset	Cuckoo Sandbox	API occurrence frequency + memory block size statistics	SVM	82.7%

Sihwail *et al.* [88] combined memory's IAT and behavior reports, yielding 6270 binary features. Using SVM with 10-fold cross-validation, they achieved 98.5% accuracy, surpassing standalone memory (94.8%) and dynamic analysis (97.4%). **Tomiyama** *et al.* [9] integrated the five most fre-

quent APIs and the mean and variance of memory block sizes allocated during execution from 100 dumps. Their SVM model achieved an F-measure of 84.2% in detecting malware. **Sharafaldin** *et al.* [89] developed BotViz for botnet detection. It collects DNS logs, detects domain generation algorithms, extracts Export Address Table (EAT), IAT, and inline memory hooks, and clusters hosts. Using Zeus botnet-infected VMs and benign domains, BotViz detected 93.5% of malicious domains with a 2% FPR. **Dai** *et al.* [24] proposed SMASH, a multi-feature ensemble learning method that integrates memory dump grayscale images, hardware performance counters (instructions retired, unhalted core cycles, and total performance status), API call sequences vectorized using Word2Vec. API sequences performed best, followed by memory dumps, achieving 98.1% F1-score and 96.9% accuracy. The overview of studies using a hybrid feature set has been illustrated in the table 7 and the overall taxonomy in figure 2.

The current research state of malware detection and classification has shifted from non-machine learning to machine learning-based approaches. The overall taxonomy of malware detection or classification with memory analysis in figure 2 demonstrates this transition, though traditional approaches still have their use cases. Signature-based and heuristic-based methods are now often combined with advanced techniques. Despite improvements in machine learning, challenges remain, such as handling large memory dump files, particularly in memory visualization. Large files pose problems for computer vision algorithms, requiring resizing, which leads to information loss. Chunking the images into smaller parts is resource-intensive, as shown in [25]. Hence, image descriptors like GIST and HOG have become more prevalent.

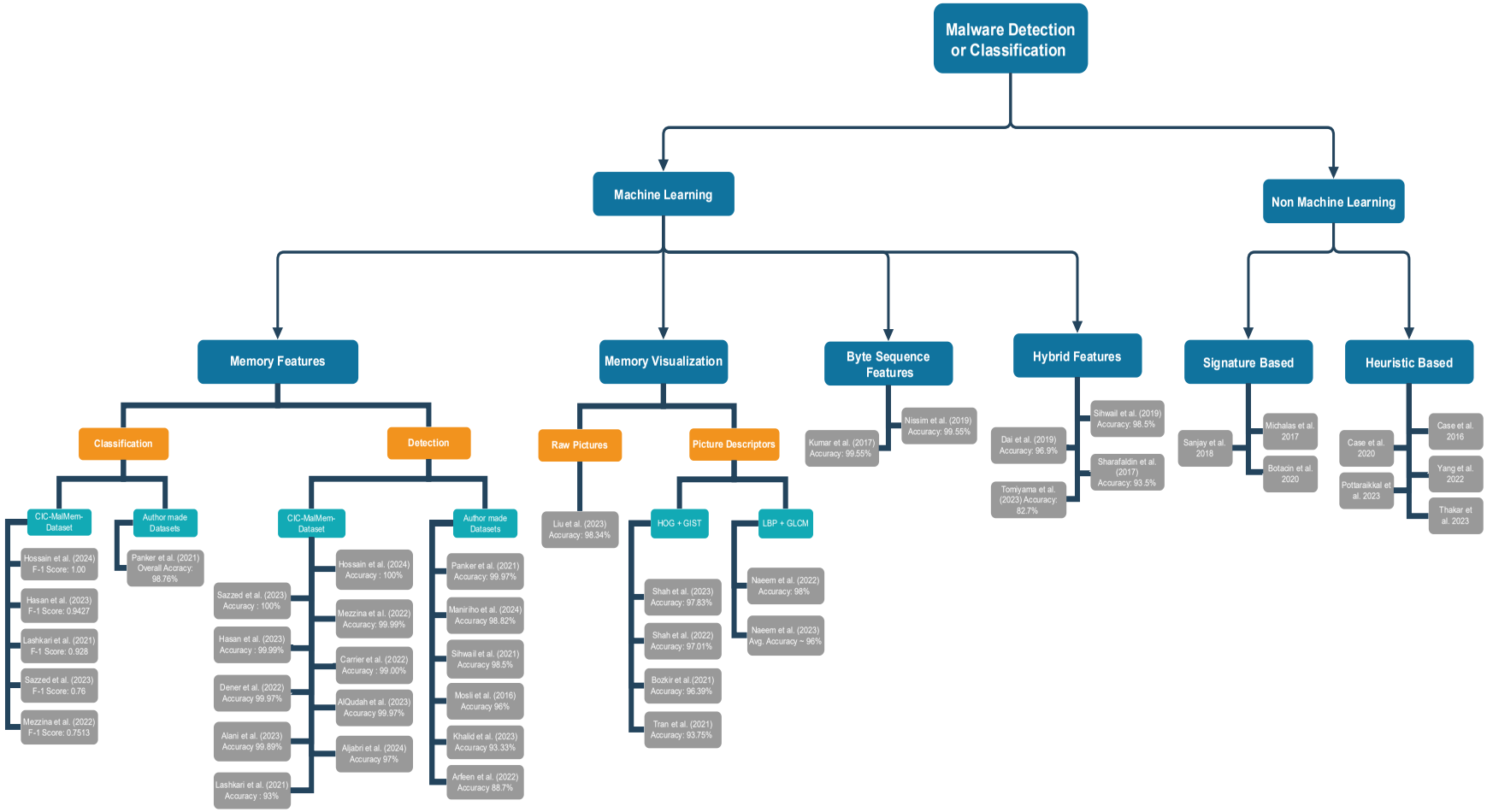


Figure 2: Taxonomy of Memory all approaches sorted by their Accuracy (If Applicable)

However, Machine learning approaches primarily focus on memory features, with tools like Volatility extensively utilized. These methods generally achieve higher accuracy but often require detailed feature extraction and selection, particularly when other datasets are used, necessitating significant domain knowledge. Byte-wise feature analysis offers another valuable method for malware detection in memory dump files, providing a relatively fast, detailed, low-level data view. However, it is susceptible to malware obfuscation techniques, which can hinder detection accuracy, and the large volume of binary sequence data presents processing challenges. Hybrid approaches offer improved accuracy but increase implementation complexity and resource costs.

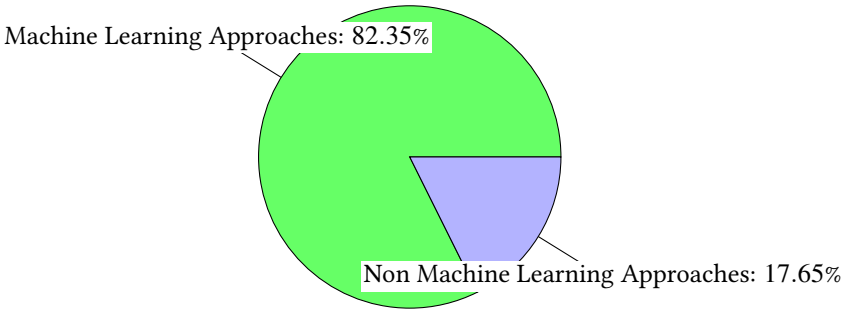


Figure 3: Distribution of malware detection or classification approaches in recent studies

The limited availability of datasets often forces researchers to create their own, and most of these datasets are heavily focused on Windows, making it harder for studies to apply across different platforms. Furthermore, some machine learning methods are resource-heavy, making real-time applications difficult without optimization.

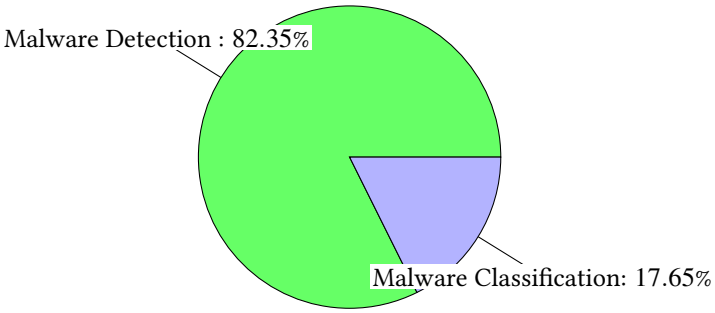


Figure 4: Distribution of multi-class and binary malware classification

Our statistical analysis, as demonstrated in the figures 3 and 4, reveals that approximately 85% of malware detection or classification studies employ machine learning techniques, with 82% focusing solely on detection and only 18% extending efforts to classify malware into families, highlighting a notable gap in classification.

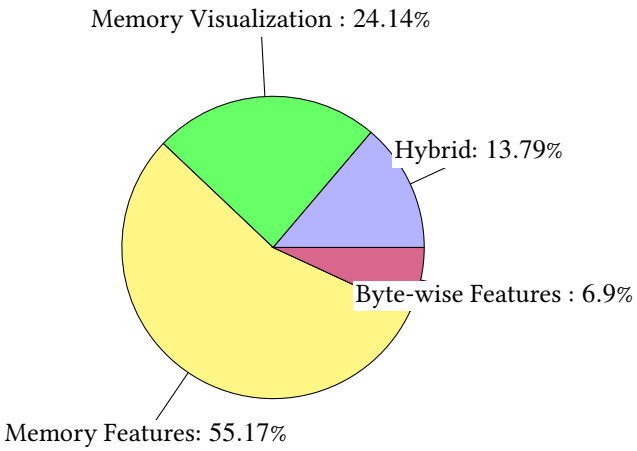


Figure 5: Distribution of Machine learning methods in malware detection and classification

Also, the trend from 2015 to 2024 shows a steady increase in the use of machine learning approaches, as illustrated in figure 6, with traditional methods becoming less common each year. The specific techniques in the ever-increasing machine learning approach are demonstrated in Figure 5. In conclusion, while machine learning has significantly improved malware detection and classification, challenges remain in handling large memory dump files, improving feature extraction accuracy, and developing datasets. The table 8 demonstrates an overall comparison of all approaches.

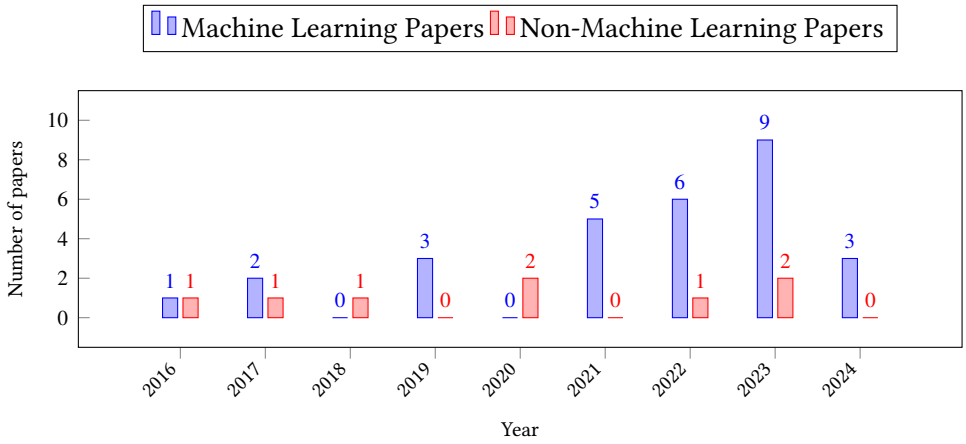


Figure 6: Number of malware detection or classification papers in each approach

Table 8: Overall Comparison of Approaches

Approach Name	Type	Advantages	Disadvantages	Primary Tools Used	Common Datasets	Resource Requirements
Signature-based Detection	Non-ML	Reliable for known malware, quick detection (Antivirus Software), low computational overhead	Ineffective against new or obfuscated malware, requires constant updates	YARA and custom tools	N/A	Low
Heuristic Analysis	Non-ML	Can detect new malware, adaptable (Antivirus Software)	High false positive rate, may miss subtle obfuscations, requires frequent tuning	Volatility	N/A	Medium
Memory Feature Engineering	ML	High accuracy, effective against diverse threats, scalable	Requires extensive feature extraction, domain knowledge needed	Volatility	CIC-MalMem-2022	Medium
Memory Visualization	ML	Effective in identifying patterns, supports large data sets, No domain knowledge needed	Requires resizing of memory dumps, potential information loss, high computational demand	Custom Tools (bin2png)	Dumpware 10	High
Byte-wise Feature Analysis	ML	Fast, detailed low-level view, robust against simple obfuscations	Affected by malware obfuscation, large data volume, requires efficient data handling	Custom Scripts	Custom Datasets	Medium
Forensic Analysis	Non-ML	Thorough, can uncover hidden malware, detailed forensic reports	Resource-intensive, time-consuming, requires deep system knowledge	Volatility	N/A	High
Hybrid Analysis	ML	Balances accuracy and performance, flexible, can handle a variety of threats	Complex to implement, high resource requirements, integration challenges	Depends	Custom Datasets	High

3 Background

Memory analysis techniques have evolved to address the challenges posed by advanced malware, including rootkits that manipulate kernel memory to evade detection [90, 91]. These techniques include memory-based feature extraction, which can identify the behavior and characteristics of malware by analyzing the memory layout and the processes running within it [92]. The effectiveness of these methods is underscored by their ability to detect processes that employ hooking techniques to masquerade as legitimate applications, thereby enhancing the accuracy of malware detection [92]. The richness of data obtained from memory analysis extends beyond mere identification of malicious software; it also encompasses insights into user activity, system vulnerabilities, and potential insider threats [93]. For instance, memory analysis can reveal traces of user actions and interactions with the system, which are invaluable for understanding the context of an incident and developing comprehensive incident response strategies [91]. Furthermore, integrating advanced techniques such as machine learning and deep learning into memory forensics has shown promise in improving detection rates and automating the analysis process [25].

3.1 Process Memory Internals

Process memory in modern operating systems is structured to enforce isolation, manage execution, and optimize performance across user-mode and kernel-mode operations. Each process operates within a private virtual address space, mapped through the Memory Management Unit (MMU) to physical memory, preventing direct interference between processes. Core memory structures include the stack (for function calls and local variables), heap (for dynamic memory allocations), and memory-mapped files (such as DLLs and shared libraries). Address Space Layout Randomization (ASLR) mitigates exploitation by randomizing memory locations, making attacks like return-oriented programming (ROP) more difficult. Analyzing these structures is essential in forensic investigations, as malware often manipulates heap allocations, injects code into process address spaces, or hooks system calls to maintain persistence and evade detection[94]. Figure 7 shows a high-level diagram of the typical contents of process memory.

Enumerating process memory involves analyzing page tables, Virtual Address Descriptors (VADs), and the working set list. Page tables store virtual-to-physical mappings, managed through multi-level structures such as Page Directories and Page Tables in x86 systems. The VAD tree, stored

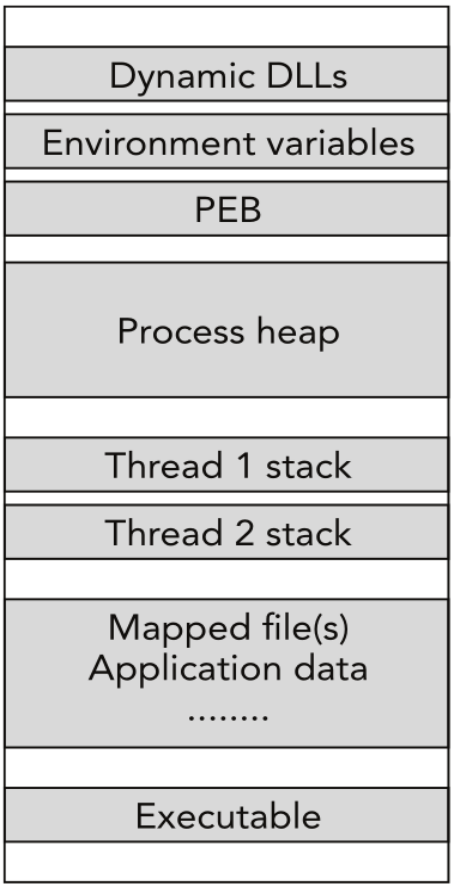


Figure 7: Process Memory Contents

in the _EPROCESS structure, tracks memory allocations and permissions, distinguishing between reserved, committed, and free memory regions. The working set list maintains pages in RAM, tracking their access frequency and residency status. These structures provide a forensic footprint of a process's execution state, revealing injected payloads, hidden threads, and memory anomalies indicative of compromise. Malware may attempt Direct Kernel Object Manipulation (DKOM) to unlink or modify these structures, concealing itself from standard enumeration methods[94]. Memory allocation follows a hierarchy from high-level API calls to low-level kernel routines. VirtualAlloc and HeapAlloc in Windows allocate memory through system calls like NtAllocateVirtualMemory, which ultimately interface with kernel-mode functions. Heap allocations are managed by the Windows Heap Manager, using control structures such as _HEAP and _HEAP_ENTRY, which track allocated blocks and metadata. Exploits such as heap spraying allocate predictable memory patterns to bypass ASLR, while shellcode injection often targets heap or stack memory for execution. Kernel structures like _MMVAD within the VAD tree dictate access permissions,

and tampering with these can alter execution privileges or enable stealthy persistence[94]. The overview of hierarchical memory allocation API calls in the Windows operating system is demonstrated in figure 8.

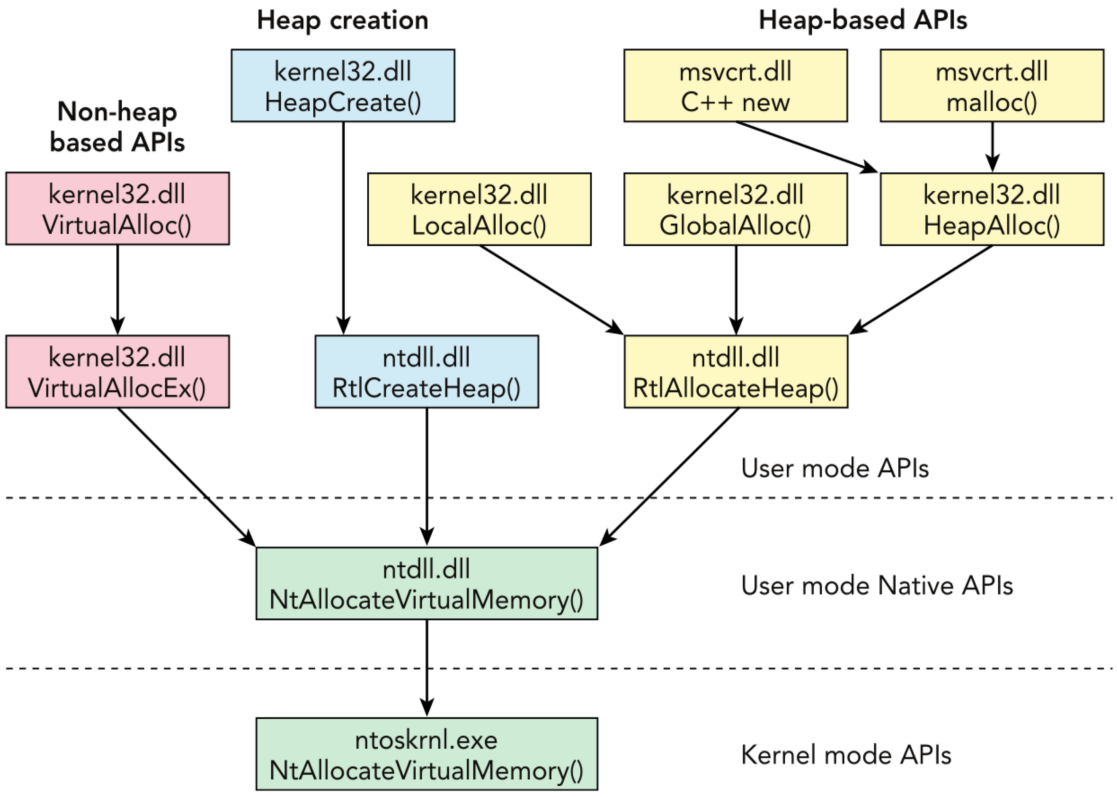


Figure 8: Process Memory Allocation API Calls

Forensic tools such as Volatility extract and analyze process memory through plugins like memmap, which lists memory regions, and vadinfo, which enumerates VAD structures. Dumping and inspecting memory segments allows investigators to identify non-resident pages swapped to disk, detect hidden injected code, and reconstruct malicious activity. Cross-referencing page tables with VAD structures helps uncover inconsistencies exploited by malware, such as rogue executable pages or phantom allocations. Advanced threats may leverage API unhooking or direct system call invocation to bypass user-mode monitoring, necessitating deeper kernel-level inspection [94].

3.2 Virtual Address Descriptors

Virtual Address Descriptors (VADs) are critical data structures the Windows operating system maintains to manage and track memory regions allocated to a process. Unlike page tables, which are handled at the hardware level by the CPU, VADs are purely OS-defined structures, storing metadata about allocated memory regions that the CPU does not inherently track. A VAD contains essential attributes, including the starting and ending virtual addresses of a memory range, initial protection settings (such as read, write, and execute permissions), and flags that indicate whether the memory region is private, shared, or associated with a mapped file. This metadata is particularly valuable in forensic investigations, as it allows analysts to associate virtual memory ranges with specific DLLs, executables, or dynamically allocated memory blocks[94].

Structurally, the VAD is implemented as a self-balancing binary tree, where each node represents a range of virtual memory assigned to a process. This organization ensures efficient searching, insertion, and deletion of memory regions. Nodes are positioned in the tree based on the memory range they describe: if a node represents a lower memory address than its parent, it is placed on the left; if it means a higher range, it is placed on the right. The tree structure enables rapid traversal and efficient memory management, unlike linear structures such as linked lists, which require sequential searching. The VadRoot member of the _EPROCESS structure serves as the entry point to the VAD tree. For forensic analysts, reconstructing this tree provides a comprehensive view of process memory layout, revealing allocated and deallocated regions that may contain remnants of malicious activity [94]. The design and implementation of VAD structures have evolved across different versions of Windows. In Windows XP and Windows Server 2003, the VadRoot pointer in _EPROCESS referenced an _MMVAD_SHORT, _MMVAD, or _MMVAD_LONG structure, depending on the complexity of the memory allocation. Starting with Windows Vista, Microsoft transitioned to the _MM_AVL_TABLE format, which provided a more generalized approach to AVL tree-based memory management. With the release of Windows 8.1 and later, the structure was further modified into an _RTL_AVL_TREE.

3.3 VAD Attributes

Virtual Address Descriptors (VADs) contain key attributes that define memory regions within a process. These attributes give forensic investigators critical information about memory allocation,

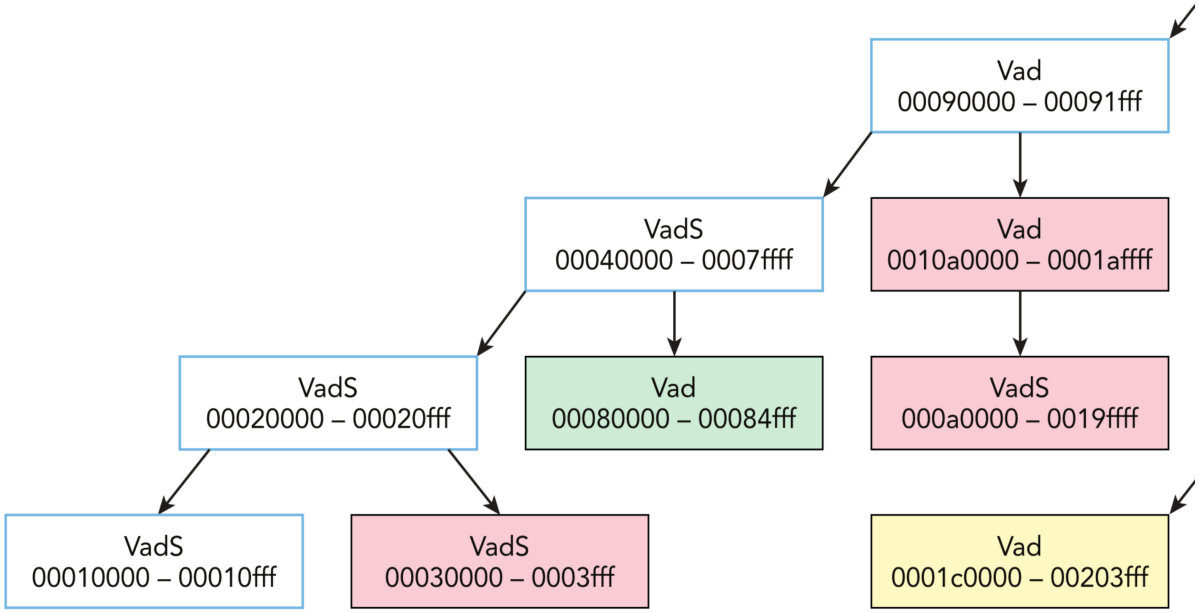


Figure 9: VAD Tree visualized as a self-balancing binary tree

permissions, and usage patterns.

3.3.1 VAD Tags

VAD tags are identifiers embedded within the _MMVAD structures that help categorize the type of memory allocation associated with a particular VAD node. These tags reside in the PoolTag member of the _POOL_HEADER structure, which exists in memory before the node. Volatility leverages these tags to determine whether a given node corresponds to an _MMVAD_SHORT, _MMVAD, or _MMVAD_LONG structure[94]. Table 9 maps the common VAD-related pool tags to their corresponding node types:

Malware often manipulates these tags to mask injected code or allocate memory regions that evade detection. Investigators should closely inspect VAD tags to identify anomalies such as unexpected

Table 9: A Mapping of VAD-Related Pool Tags to the Containing Structure Type

Tag	Node Type
Vad	_MMVAD_LONG
VadS	_MMVAD_SHORT
VadF	_MMVAD_SHORT

memory allocations without associated file mappings [94].

3.3.2 Commit Charge

Commit Charge is a field within the _MMVAD structure that specifies the number of committed pages in the associated memory region. This field is distinct from the MemCommit flag, indicating whether a memory range was committed at allocation. Attackers frequently pre-commit memory regions when injecting malicious payloads, making a high Commit Charge value a potential indicator of injected code [94]. Forensic analysts can leverage tools like Volatility's `vadinfo` plugin to inspect the Commit Charge of VAD nodes. By cross-referencing this value with expected application behavior, analysts can detect deviations that may indicate malicious activity [94].

3.3.3 Protection

The Protection field defines the access permissions applied to a memory range when it was first allocated. This value is loosely coupled with the memory protection constants passed to the virtual allocation APIs. The table 10 shows various types of protection memory regions can have.
A significant issue with the Protection field is that it reflects the initial settings rather than the current state of memory permissions. Attackers can allocate memory as non-executable and later modify it to execute code. Consequently, forensic analysts should supplement their VAD analysis with direct page table inspection to verify runtime permissions [94].
Despite its advantages, memory forensics is not without challenges. The trustworthiness of memory analysis can be compromised by malware that employs anti-forensic techniques to obscure its presence or manipulate memory data [90]. Additionally, the increasing complexity of malware and the sheer volume of data generated by modern systems necessitate the development of more

Table 10: Memory Protection Constants and Their Descriptions

Protection Constant	Description
PAGE_EXECUTE	Allows execution but prohibits writing.
PAGE_EXECUTE_READ	Allows execution and reading but prohibits writing.
PAGE_EXECUTE_READWRITE	Grants full access, making it a common setting for injected code.
PAGE_EXECUTE_WRITECOPY	Enables execute, read-only, or copy-on-write access to a mapped view of a file. It cannot be set by calling VirtualAlloc or VirtualAllocEx. DLLs almost always have this protection.
PAGE_NOACCESS	Blocks all access, sometimes used to conceal injected memory.
PAGE_READONLY	The memory can be read but not executed or written.
PAGE_READWRITE	The memory can be read or written but not executed.
PAGE_WRITECOPY	Enables read-only or copy-on-write access to a mapped view of a file. It cannot be set by calling VirtualAlloc or VirtualAllocEx.

sophisticated analysis tools and methodologies [95]. Recent studies have highlighted the need for frameworks that can efficiently handle the intricacies of memory analysis where traditional forensic methods may fall short [96, 97].

4 Dataset

The evolving complexity of malware and the increasing sophistication of detection methods underscore the need for data sets in malware memory forensics. The availability of reliable datasets is vital to validating and benchmarking new methodologies. Specifically, memory dump files are challenging to acquire due to their time-consuming process, which requires careful setup and execution to ensure data integrity. This section discusses existing memory analysis datasets across two categories, namely, Tabular Datasets and Raw File Datasets with their key characteristics.

4.1 Tabular Datasets

Most of the studies in the memory analysis are tabular datasets with their features extracted manually by the researchers using memory forensic tools. The most recent contributions in 2024 include works by **Malak Aljabri** *et al.* [12] and **Maniriho** *et al.* [83]. **Malak Aljabri** *et al.* released the most recent dataset to detect ransomware using memory dump features. It includes labels for 579 benign and 586 ransomware samples with 58 features. Sources include VirusShare, Bazar, Informer, and CNET. Data was acquired using the VolMemLyzer tool in a Cuckoo Sandbox. This dataset is not publicly available. On the other hand, **Maniriho** *et al.* introduced the MemMal-D2024 data set for malware detection using deep learning memory analysis. The dataset contains 29,298 benign and 28,870 malware samples with 60 features using deep autoencoders. The dataset, originating from Tristan *et al.* (2022), was collected using the Volatility framework on a Windows 10 system. It is publicly accessible.

In the year 2022, **Khalid** *et al.* [27] developed a fileless malware detection dataset containing 45 samples labeled as fileless malware or benign with 33 features extracted using Volatility. Data sources were diverse, including VirusShare and AnyRun, etc. The environment used VMware Workstation and the memory dumps were acquired using the ”vmss2core” tool from the VM snapshots, collected after executing the programs. accessibility is not specified. In the same year, **The CIC MalMem 2022 dataset**, developed by **Carrier** *et al.* [15], was gathered to test obfuscated malware detection methods. It contains 29,298 benign and 29,298 malware samples with 58 features from five categories extracted via VolMemLyzer-V2. Memory dumps were obtained using tools like Volatility. The dataset, collected using a virtual machine with 2 GB memory in debug mode, includes dumps created by executing software samples from VirusTotal for malware and

by running applications for benign samples. The dataset is publicly available [98]. Furthermore, **Lyles** *et al.* [99] introduced a dataset having 165 benign and 200 malware samples with features that were extracted using a custom Volatility plugin that included EPROCESS, ETHREAD, PEB, and VAD tree structures. The dataset was created by the authors and collected using the Volatility3 framework in a Windows 10 VM hosted on VirtualBox. Memory images were captured statically during the execution of applications. Accessibility is not specified.

In 2021, **Arfeen** *et al.* [14] developed a dataset for ransomware consisting of 900 memory dumps: 300 benign and 600 ransomware. Features were extracted using the Volatility and were transformed into CSV format for analysis. The environment used a VirtualBox VM with Windows 8. Memory snapshots were taken at 5-minute intervals. This dataset is publicly available on the Harvard Dataverse repository. Also, **Panker** *et al.* [85] developed a 2021 dataset comprising 10,900 benign and 10,900 malicious samples, covering various families with 171 features extracted using Volatility. Data sources include VirusTotal for malware and benign samples by the authors. The environment consisted of two virtual servers running DNS and HTTP on Linux Ubuntu 16.04, each with 4 GB RAM. Memory dumps were acquired by querying the hypervisor while the VM was temporarily frozen and collected periodically. Earlier in the same year, **Sihwail** *et al.* [81] and **Habibi Lashkari** *et al.* [84] introduced two datasets. **Sihwail** *et al.*'s dataset includes 966 benign and 2,502 malware samples with 8,898 features also extracted by volatility. Data sources include VirusTotal for malware and Windows OS files and utility applications for benign samples. The environment consisted of VirtualBox with a Cuckoo Sandbox for acquiring memory dumps. snapshots collected after 120 seconds of sample execution and the dataset is publicly accessible. **Habibi Lashkari** *et al.* introduced the VolMemLyzer dataset in 2021, designed for classifying malware. It has 1,127 benign and 773 malware samples with 36 features extracted by the VolMemLyzer tool. The dataset was created in a VirtualBox with 4 GB RAM and Memory dumps were acquired using the Volatility framework. The accessibility is not specified.

In older studies, **Sihwail** *et al.* [88] developed a dataset in 2019 that combines memory forensics with dynamic analysis. It comprises 400 benign and 1,200 malware samples covering over 80 families with 6,270 features extracted also by volatility. Sources include VirusTotal and Das Malwerk for malware, and Windows 7 for benign files. The environment used VMware Workstation and memory dumps were acquired using VMware's "createSnap" with each malicious sample executed for two minutes after VM rollback to the "Initial Stat" snapshot. Accessibility is not

specified. **Mosli** *et al.* [82] developed a dataset in 2016 including 100 benign and 400 malware samples with 59,270 combined features. Data sources include VirusShare, VX-Heaven, and the authors' collection. The environment consisted of a Windows 7 target system with a modified Cuckoo Sandbox to handle BSODs for memory acquisition. After running malware samples the system was revertined to a clean state using Fog. **Duan** *et al.* [100] introduced the Detective dataset in 2015, which consists of 3390 malware and 230 benign samples with features including loaded DLLs, n-tuples of DLL sequences, and HNB classification. Data sources include VirusTotal, Process Explorer, and Volatility. The environment consisted of a Windows 7 target system with Cuckoo Sandbox for acquiring memory dumps. Similar to Mosli, samples were collected after reverting the system to a clean state using Fog. Features were extracted from DLL lists, process information, clustering, and frequent item set mining techniques.
The overview of the tabular dataset with their key characteristics is shown in the table 11.

4.2 Memory Dump Raw File Datasets

The memory dump datasets consisting of the raw memory files without the manual extraction of features were also employed by researchers in various studies. **Liu** *et al.* [25] released a 2023 dataset to detect memory-resident malware that includes 1,010 benign and 1,050 malware samples from different families such as adware, ransomware, keylogger, downloader, and backdoor. Samples are memory dumps with features extracted as RGB images. The dataset originates from the authors' private collection and VirusShare, with memory images acquired using the Volatility tool on an Ubuntu host and Windows 7 guest OS. Feature extraction involves API call features from behavior reports. The next recent dataset was proposed by **Zhang** *et al.* [29]. They developed a dataset including 4,896 samples labeled as malware or benign, covering families like file-less malware, trojans, ransomware, keyloggers, and backdoors. Samples are memory dumps of the specified process of the executed PE files, dumped to a specified file using a custom process space dump algorithm. The dataset sources include VirusShare, MalShare, and Microsoft. The environment used VMware Workstation and memory dumps were acquired using a custom algorithm, collected every 10 minutes during sample execution.
In the year 2021, two datasets were proposed. The first dataset that is used by many computer vision-based studies is **Dumpware 10** which was developed by **Bozkir** *et al.* [23]. The dataset includes 608 benign and 3,686 malware samples, covering malware families such as adware (Ad-

poshel, Amonetize, BrowseFox, InstallCore.C, MultiPlug), worms (Allaple.A, AutoRun-PU), trojans (Dinwod!rfn, Vilsel), and viruses (VBA). Samples are memory dumps represented as RGB images, with features extracted using GIST and HOG descriptors in the Bozkir *et al.* study. Data sources include actual malware samples from an information security company and digitally signed Microsoft Windows applications. The environment was a Windows Sandbox with Windows 10 and 16 GB of allocated memory. Memory dumps were acquired using the Procdump tool after running PE files in the sandbox with a 5000 ms delay. This dataset is publicly accessible. The other dataset was developed by **Tran** *et al.* [26] for detecting malware using computer vision techniques. It includes 2,750 samples from eight malware types: Loki-Bot, Agent Tesla, FormBook, StormKitty, OskiStealer, RedlineStealer, and SnakeKeylogger. Samples are memory dumps represented as RGB images, with features extracted using HOG and GIST descriptors in their study. The dataset was generated using MemGen from MalwareBazar. The environment included an Ubuntu 20.04 host with VirtualBox running a Windows 10 guest OS with 4 GB RAM. Memory dumps were acquired using the ProcDump utility in the MemGen system and collected from benign and malicious machines.

In a more recent study in 2019, three datasets were proposed and gathered. **Nissim** *et al.* [79] developed a dataset for detecting malware in virtual machines using MinHash. The dataset includes 3,000 raw memory dumps, each approximately 1 GB, from virtual machines running Windows Server 2012R2. Shingles were extracted from these dumps with a window size of 32 bytes and an offset of 8 bytes, resulting in around 125,000,000 shingles per dump. The memory dumps were collected using VMware vSphere, with snapshots taken securely without the virtual machines' awareness. The malware sources are not specified, but the dataset, processed into MinHash signatures, is publicly accessible. **Sadek** *et al.* [101] introduced a dataset containing 1,530 memory snapshots of a compromised Windows 10 virtual machine, stored in Advanced Forensics Format (AFF4). The snapshots were acquired using the WinPmem tool after compromising the VM with obfuscated reverse shells. Each memory snapshot is accompanied by a list of running processes and a memory map for the malware. The dataset is publicly accessible. The summarization and overview of these datasets are shown in the table 12.

4.3 Dataset Comparison

The choice of the dataset can heavily influence the direction of malware detection research, sometimes narrowing the focus to specific threats or environments. Standardizing evaluation frame-

works to address these challenges and comparing and understanding datasets is essential. In memory dump datasets, these critical aspects include sample numbers, year of release, malware diversity, feature number, accessibility, and memory dump size. A scoring system is proposed to rank the available datasets by assigning points to the mentioned aspects to perform this comparison and ensure an appropriate dataset selection. The scoring system quantifies qualitative aspects and categorizes quantitative ones by assigning points based on predefined value ranges for each dataset characteristic. For example, datasets receive higher scores for having more samples, recent release dates, diverse malware types, numerous and relevant features, public accessibility, and balanced memory dump sizes. The exact scoring system is shown in the table 13.

Table 13: Memory Dump Dataset Scoring System

Comparison Aspect	**Criteria**	**Points**
Sample Numbers	Over 10000 samples	30 points
	3000 - 10000 samples	20 points
	1000 - 3000 samples	15 points
	500 - 1000 samples	10 points
	Under 500 samples	5 point
Malware Diversity	More than 5 families	25 points
	3 - 5 families	15 points
	Less than 3 families	8 points
Feature Number	Over 200 features	20 points
	50 - 200 features	12 points
	20 - 50 features	8 points
	Under 20 features	5 points
Year of Release	Last 3 years (2022 - 2024)	10 points
	4 - 6 years ago (2019 - 2021)	6 points
	Older	4 points
Memory Dump Size	More than 2 GB	10 points
	1 GB - 2 GB	6 points
	1 GB or less	3 points
Accessibility	Publicly accessible	5 points
	Restricted or private access	0 points
Maximum Points	(will be converted to scale of 0 - 10)	**100 points**

The datasets' scores are demonstrated in the table 14 ranked by highest to lowest score (answer to the D2 research question). For the raw memory dump datasets, the "feature number" score will not be counted, calculated out of 80, and then converted to the 0-10 scale.

A low score doesn't necessarily indicate that the dataset lacks value or usability; it may have been collected for specific research purposes. This suggests that there are more comprehensive datasets available that might be better suited for broader applications. The top 3 best-scoring tabular and

raw datasets have been visualized in figure 10 and 11, respectively.

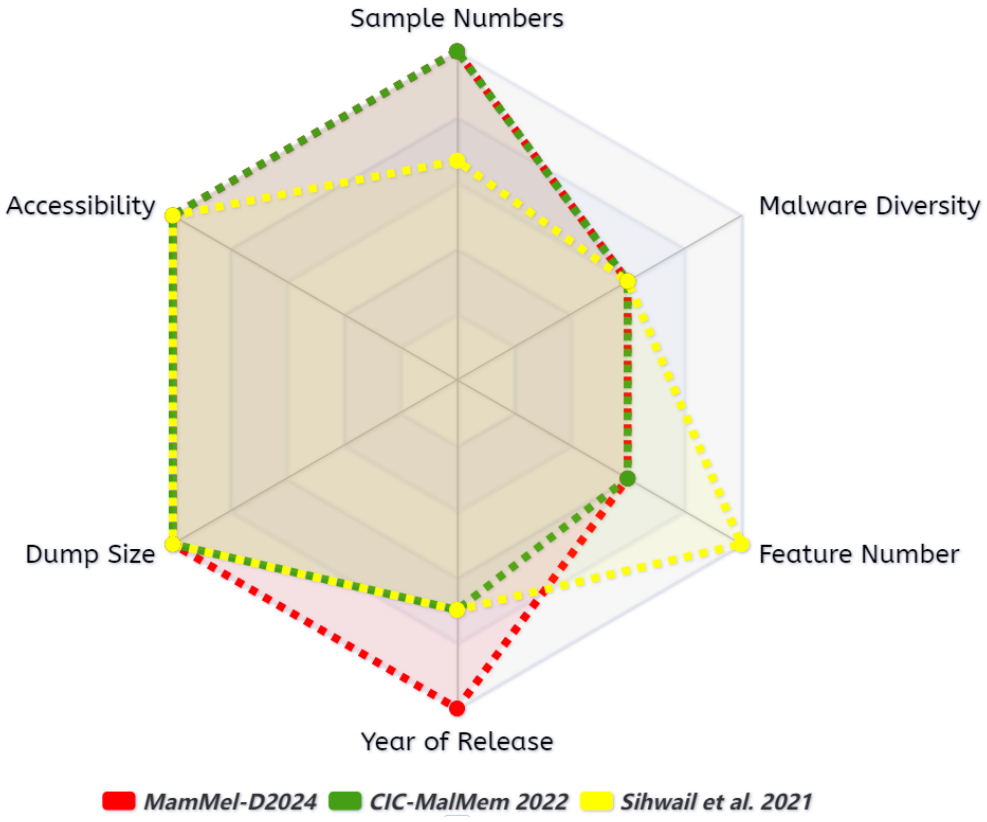


Figure 10: Top3 Tabular Datasets

4.4 BCCC-MalMem-SnapLog-2025 Dataset

Figure 12 illustrates the most comprehensive taxonomy of the currently available memory dump datasets.

Analyzing existing memory dump datasets in memory forensics reveals strengths and challenges. Datasets like MemMal-D2024 and CIC MalMem 2022 stand out due to their large sample sizes, recent data, and a more diverse selection of malware, which are clear advantages. However, the limited accessibility of many datasets holds back collaboration and innovation within the forensic community. There's also a strong focus on Windows systems, leaving a gap for datasets covering Linux or macOS. The variation in dataset sizes shows that smaller ones are often used for specific analyses, while larger datasets are typically used for training machine learning models.

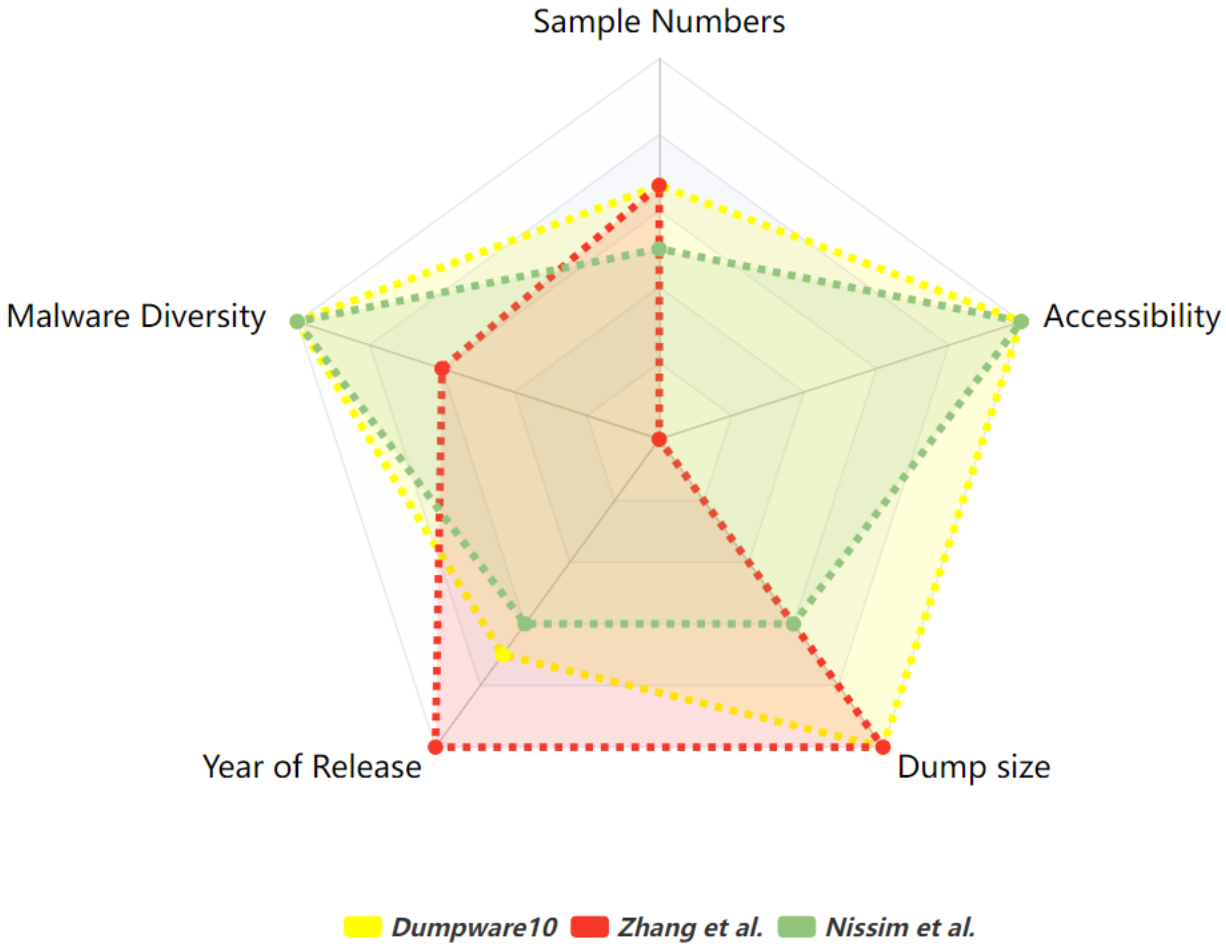


Figure 11: Top3 Raw Datasets

4.4.1 Dataset Generation Motivation

Datasets explored in the previous subsection capture memory dumps at specific instances and focus solely on system-wide attributes. These datasets may miss the particular characteristics of the malware process, as they do not isolate or identify the malware's presence effectively [30][31]. For example, while these instant dumps can provide a snapshot of the system state, they lack the granularity needed to analyze the behavior of individual malware processes, which is essential for understanding the full impact of a malware attack [19][102]. Furthermore, the temporal changes during malware execution may also be missed. To address these challenges, researchers have proposed various methodologies. Datasets that capture dumps at intervals may attempt to incorporate temporal features. Still, they also risk missing the presence of malware if the dumps do not coincide with the malware's execution [32][18]. This temporal aspect is critical as malware can exhibit different behaviors depending on the execution time and the system's state. Mem-

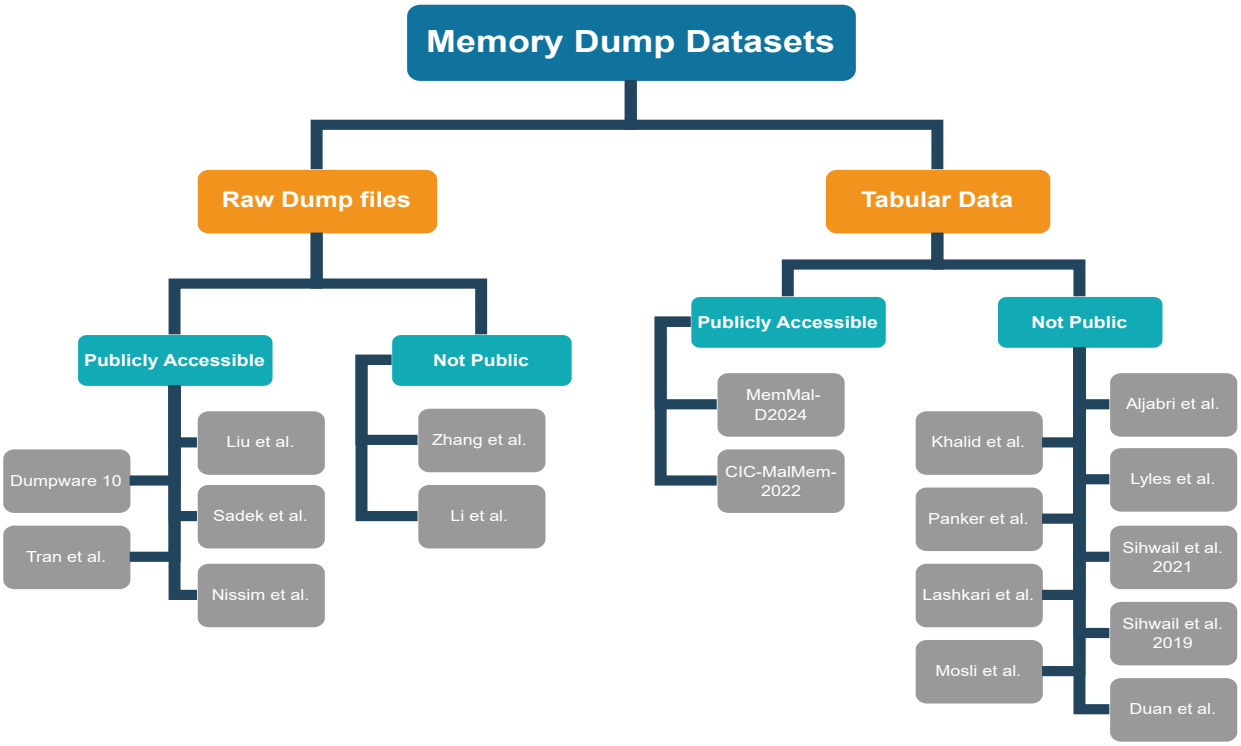


Figure 12: Taxonomy of Existing Memory Datasets

Scrimper is a technique that dumps the memory in intervals and compresses memory dumps by storing only the differences introduced by malware, which can help incorporate these temporal features and manage the large volume of data [103]. However, the presence and isolation of the malware process in these memory dumps remains a challenge.

On the other hand, dynamic sandboxing plays a crucial role in malware analysis. Still, it exhibits significant limitations that affect its utility in incident response scenarios, as it requires monitoring the malware execution process, which may not often be possible. Furthermore, dynamic sandboxing techniques often focus on capturing the memory of individual processes, which can lead to missing some system-wide effects of malware. [33][34]. This limitation means that if the malware is not actively monitored during its execution, critical information about its behavior and interactions with other system components may be lost, rendering the memory dumps less useful for comprehensive incident response [104][105].

Thus, to generate a memory dump dataset that addresses the aforementioned shortcomings of previous ones, we used an execution workflow to maximize the probability of malware presence in the memory dump and incorporate the temporal and system-wide features as much as possible to analyze malware behavior without monitoring the process directly.

4.4.2 Dataset Generation Method

The idea for generating our dataset is to identify the malware process identifier (PID) in the system and time the snapshots to ensure the malware process is active in memory. Additionally, we dump the memory at intervals to capture the temporal features that malware may exhibit across these dumps. The detailed methodology is illustrated in Figure 13.

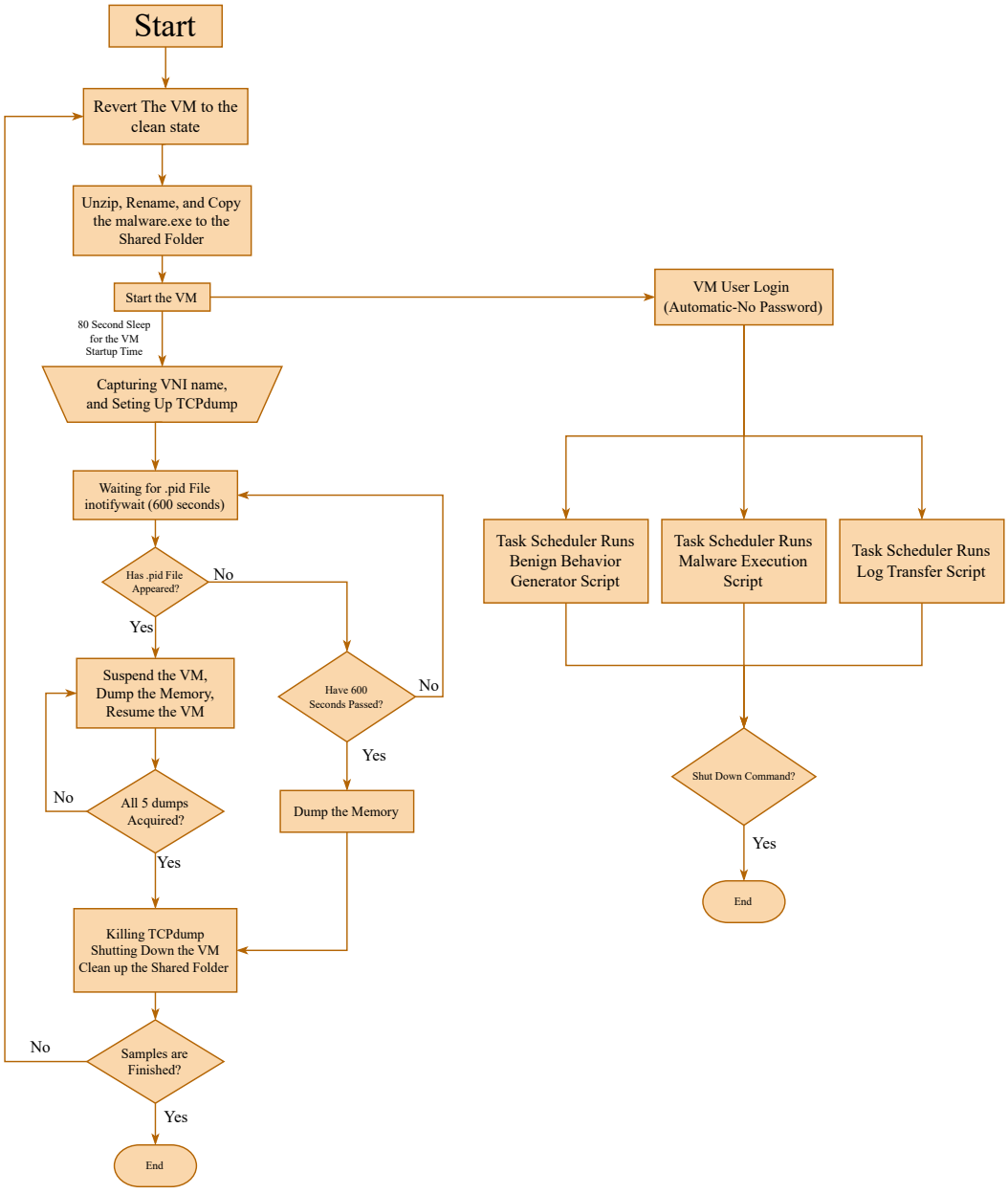


Figure 13: Detailed Execution Flow of the Automated Dataset Generation Script

To achieve this, a host system running Fedora 40 was used to set up a guest Virtual Machine

(VM) with Windows 11 Home Edition using QEMU. QEMU is a generic and open-source machine emulator and virtualizer that can be used for system emulation, providing a virtual model of an entire machine (CPU, memory, and emulated devices) to run a guest Operating System (OS). This framework allows the CPU to be fully emulated or to work with a hypervisor such as KVM or Xen to run directly on the host CPU [106].

Network connections (e.g., SSH or sockets) were not used as the communication protocol to minimize host interference with the guest OS. Instead, the built-in VirtIOfs shared folder was set up between the host and guest to transfer data. VirtIOfs is a shared file system that enables virtual machines to access a directory tree on the host, offering local file system semantics and performance [107]. The workflow begins with each malware sample being renamed to a standardized filename (malware.exe) on the host and placed in the shared folder, after which the VM is started. Upon startup of the Windows VM, a pre-configured task scheduler executes the malware.exe file immediately after the user logs in. This setup ensures automated and consistent execution across all samples, reducing variability.

A key requirement for this dataset was to capture the PID of each malware process accurately. Therefore, the Windows task scheduler runs a PowerShell script that initiates the malware and retrieves its PID. Considering that some malware samples may detect they are being executed in a virtual machine or may use delayed-start mechanisms, the script was designed with a try-catch block to verify the successful initiation of the malware process. This approach ensures that the PID is captured only upon successful malware execution. Additionally, the script incorporated a timeout of 600 seconds, allowing sufficient time for malware processes to initialize, even in cases where delays occurred due to system specifications or the nature of the malware.

Benign processes were simultaneously simulated within the VM. A benign behavior generator script by Shafi et al. [108] was also triggered at user logon to replicate activities such as checking emails, browsing, watching videos, downloading files, and interacting with office tools. Upon execution of the malware, the captured PID was communicated to the host by creating a "<PID>.pid" file in the shared folder, causing the host system to begin the memory dump process. The promptness of this alert was crucial to avoid missing malware processes that may be momentarily present in memory. This approach informs the host with the PID of the malware process as the name of the .pid file.

The Fedora host used the inotifywait mechanism to monitor the shared folder for the appearance

of the .pid file. Upon detection, the VM was suspended to prevent the malware process from terminating during memory dumping. The ”virsh memory-dump” command was then executed to capture the memory state of the Windows VM. Five memory dumps were captured at 30-second intervals to document the most comprehensive behavior of the malware without directly and dynamically monitoring the process. The dump files were named by the acquired PID, followed by the number of dumps in the intervals (e.g., 1234_1.vmem indicating the first dumped memory right after malware process identification).
Additionally, we measured the VM startup time across 20 iterations and averaged the results so the host system could approximately predict when to initiate the inotifywait command, using a 600-second timeout. It is worth noting that, despite these measures, specific scenarios where malware failed to execute within the allocated timeout were unavoidable. Factors such as malware-specific code behaviors and VM configurations were identified as potential causes. While such cases fall outside the scope of this dataset, the implemented workflow aimed to maximize the likelihood of capturing active malware processes.

4.4.3 Pcap and Log files

We also captured the network activity and gathered the VM logs to generate a comprehensive dataset that researchers from different scopes could use in their work. This dataset is the most extensive and complete memory dataset consisting of raw files that could be used for forensic analysis from various sources like interval dumps, network activity, and log files. This will give the researchers a multi-view behavior of malicious processes with their PID identified in the dumps.

4.4.3.1 Network Activity

Packet Capture Files (PCAP) are used to analyze VM traffic for security and performance assessment. These files record transmitted packets at the virtual network interface level, which allows the monitoring of internal and external communications, facilitating the detection of security threats and optimizing performance [109]. Tools such as Wireshark and TCPDump generate PCAP files, which are analyzed for threat detection and performance profiling [110]. Integrating packet capture data with machine learning techniques enhances network threat classification and response capabilities [111, 112]. Therefore, in our dataset generation workflow, we utilized the ”TCPdump” tool in the fedora host to capture the packets at the VM’s Virtual Network Inter-

face (VNI). The network source of the VM was set to **Virtual Network 'default':NAT**, resulting in the VM using the host internet connection.

The virtual network interface in VMs facilitates communication between VMs and the physical network and among VMs themselves. Implemented through a virtual network interface card (vNIC), it enables interaction with the host operating system's network stack [113][114]. Network Address Translation (NAT) is essential for IP address management and VM communication with external networks in virtualized environments. By modifying IP address information in packet headers, NAT enables multiple VMs to share a single public IP while maintaining unique private addresses within the local network. This approach conserves IPv4 addresses and simplifies network management in cloud and virtualization contexts [115]. NAT can function as a virtual network function (VNF) in virtualized setups, allowing dynamic network resource allocation and management without requiring physical hardware modifications [116].

When the malware is copied to the shared folder and the VM is started, the Fedora host script attempts to retrieve the name of the VNI, which is typically labeled as "vnet" followed by a numerical identifier (e.g., vnet0). This identifier changes with each execution of a sample and subsequent VM restart. If the previous network interface is not correctly cleaned up, the system assigns an incremented vnet number upon every VM restart. This behavior increases memory and resource overhead when running scripts on many malware samples. If not managed properly, it can cause crashes in the automated data-gathering script. The issue arises because the operating system has a limit on the number of VNet interfaces it can allocate for guest machines. After determining the vnet name, the script configures a TCPdump mechanism to monitor and capture network packets while the Windows VM runs, storing the collected data in a ".pcap" file. Regardless of whether the ".pid" file appears in the shared folder, this operation will continue for all malware samples.

4.4.3.2 Logs

Logs are essential for malware detection and analysis, providing structured data that enables chronological reconstruction of system events. This reconstruction aids in identifying malware behaviors, such as process creation, file modifications, and registry changes [117, 118].

•System Logs

System Logs track system-level events such as hardware failures, crashes, and resource utiliza-

tion, providing insights into the overall health and performance of the operating system. These logs are crucial for diagnosing issues and ensuring optimal performance. For instance, logging runtime information aids in analyzing system behaviors, which is vital for root cause analysis and troubleshooting [119]. The systematic collection of these logs can help identify patterns that indicate underlying problems, thus improving the system's reliability [120].

•Application Logs

These logs capture events related to application performance, errors, and usage patterns, offering detailed information about application behavior. This data is critical for debugging and performance tuning, leading to improved application performance and user experience. Integrating application logs with system logs provides a comprehensive view of system health, allowing for more effective monitoring and management of applications and the underlying operating system [121].

•Security Logs

Security logs are essential for monitoring system behavior, detecting anomalies, and investigating security incidents. They provide crucial details for forensic analysis and non-repudiation [122], though their volume poses analysis challenges. Effective logging captures relevant data without overloading systems, ensuring critical insights are not missed [123]. Integrating logs into SIEM systems enhances real-time threat detection and response [124].

•Sysmon Logs

Generated by the Sysinternals Sysmon tool, these logs provide detailed information about process creations, network connections, and changes to file creation times. These logs are handy for security monitoring and threat detection, revealing malicious activities that may not be captured by standard Windows event logs [125]. The granularity of Sysmon logs allows for advanced analysis techniques, including machine learning models that can identify lateral movement and other sophisticated attack patterns [126]. Leveraging Sysmon logs enhances threat-hunting capabilities, allowing for more proactive identification of potential security incidents [127].
Windows stores different types of logs in specific locations. The key log types and their respective storage paths are shown in the table 15. The Windows Task Scheduler executes a PowerShell script

to copy log files to the shared folder on the host machine. These log files continuously record system events and are processed in an infinite loop. The script iterates through predefined log files, checks for their existence, and copies them while preserving their original filenames with a ‘.evtx‘ extension. This copying occurs every three seconds. In the worst-case scenario, if the system shuts down just before the next iteration, the logs may be delayed by up to three seconds.

Table 15: Windows Logs and Their Storage Paths

Log Type	**Log File Path**
System Log	C:\Windows\System32\Winevt\Logs\System.evtx
Security Log	C:\Windows\System32\Winevt\Logs\Security.evtx
Application Log	C:\Windows\System32\Winevt\Logs\Application.evtx
Sysmon Log	C:\Windows\System32\Winevt\Logs\Microsoft-Windows-Sysmon%4Operational.evtx

Figure 14 illustrates the visualized dataset folder structure.

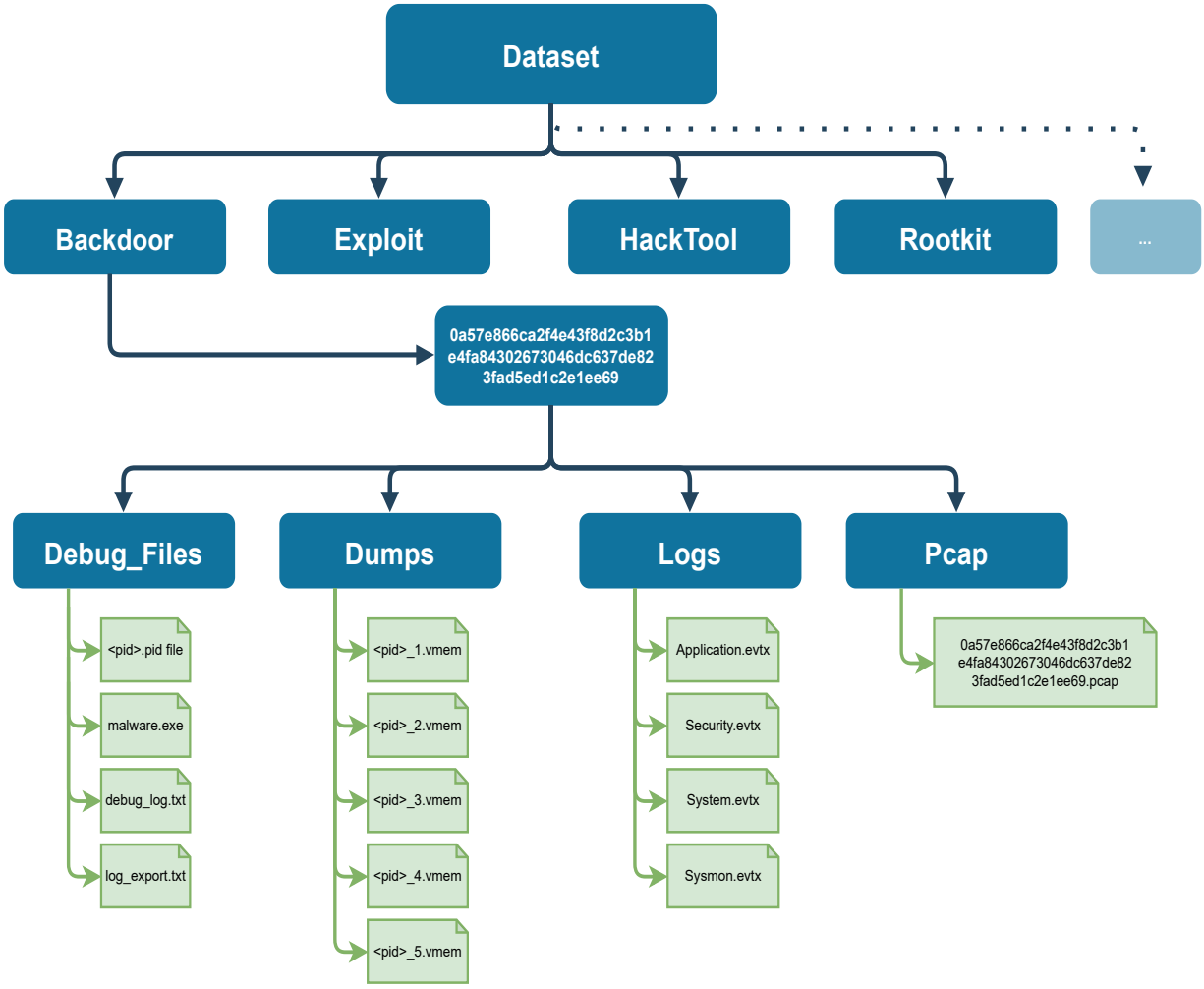


Figure 14: Dataset Folder Structure

Table 11: Tabular Memory Datasets

Name	Purpose	Sample Numbers	Malware Families	Feature and extractor	Feature Types	Sample (RAM) Size	Acquisition Method	Availability
Aljabri (2024)	Detecting ransomware using memory dump features	579 benign, 586 ransomware	Revil, Lockbit, BlackCat, Critroni, CryptLocker, CryptoWall, Locker, MATSNU, Reveton	58, VolMemLyzer with enhancements	Processes, DLLs, handles, modules, etc.	4 GB (Windows 7)	VolMemLyzer tool from Cuckoo Sandbox, during execution for 300s	N/A
MemMal-D (2024)	Detect malware using memory analysis with deep autoencoders and stacked ensemble learning	29,298 benign, 28,870 malware	Ransomware (Shade, Conti, Ako, Pysa), Trojans, Spyware	60, Volatility	Loaded DLLs, injected code, handles, processes, API hooks, and timestamps (Month, Year)	16 GB (Windows 10)	Volatility framework, samples collected from repositories	Public
CIC MalMem (2022)	Test obfuscated malware detection methods through memory	29,298 benign, 29,298 malware	Spyware, Ransomware, Trojan Horse (various families)	57, VolMemLyzer	Traces of malware in memory	8 GB (Windows 10 VM)	Memory dumps created by executing software samples, normal behavior collected by running applications	Public
Khalid (2022)	Machine-learning-based fileless malware detection	45 samples total	Fileless malware including Powershell, WMI, Macros, VB script	33, Volatility	List of processes, DLLs, network connections, registry hives	2 GB (Windows 7)	vmss2core tool from VM snapshots, post-execution snapshots	N/A
Lyles (2022)	Analyze memory images for process characterization and malware detection using machine learning	165 benign, 200 malware	DLL injection, PE injection, process hollowing, shellcode injection	24, Custom Volatility plugin	EPROCESS, ETHREAD, PEB, VAD tree structures	4 GB (Windows 10)	Volatility3 framework, during execution of benign and malicious applications	N/A
Panker (2021)	Detecting unknown malware in Linux cloud environments using machine learning	10,900 benign, 10,900 malicious	Ransomware, Trojans, cryptominers, worms, fileless malware	171, Volatility	Behavioral traces from volatile memory	4 GB (Ubuntu 16.04)	Querying hypervisor during execution, while VM is temporarily frozen	Public
Sihwail (2021)	Detect and classify malware using memory-based features extracted from memory images	966 benign, 2502 malware	Backdoor, Keylogger, Downloader, Adware, Ransomware	8898, Volatility	API calls, DLLs, process handles, privileges, networking, code injection	4 GB (Windows 7)	Cuckoo Sandbox, memory dump at the end of sample execution	Public
Habibi Lashkari (2021)	Classify malware using feature engineering on volatile memory dumps	1127 benign, 773 malware	BE2, CoreFlood, Sality, Tigger, Prolaco, Laqma, Zeus	36, VolMemLyzer	Processes, DLLs, handles, modules, code injections, connections, sockets, services, drivers, callbacks	4 GB (Windows VM)	Volatility framework, during live malware infection, analyzed offline	N/A
Arfeen (2021)	Detecting ransomware using memory dumps and process-based features	300 benign, 600 malware	WannaCry, HiddenTear, Cerber, TeslaCrypt, Vipasana	15, Volatility with custom batch script	Handles, DLLs, privileges, process statuses	1.18 GB (Windows 8 VM)	Memory snapshots are taken every 5 minutes for each process	Public
Sihwail (2019)	Combines memory forensics with dynamic analysis for improved malware detection	Malware: 1200, Benign: 400	Over 80 malware families, including Zeus, TrickBot, DarkComet, Darkhotel, Ursnif	6270, Volatility	API calls	24 GB (Windows 10)	VMware's createSnap operation, executed for two minutes after rollback	N/A
Mosli (2016)	Detect malware using heuristic analysis of artifacts in forensic memory images	100 benign, 400 malware	Various malware including Adware, Ransomware, Trojans, Spyware, Rootkits	59,270, Volatility	Registry activity, DLL imports, API function calls	2 GB (Windows 7)	Cuckoo Sandbox, memory dumps after running malware samples, reverted to clean state	N/A
Duan (2015)	Automatically identify and analyze malware processes in forensic scenarios via DLLs	3370 malware, 226 benign	Ransomware, Trojans, Backdoors, Adware, Spyware, Rootkits	121, Volatility and Weka	Loaded DLLs, n-tuples of DLL sequences, HNB classification	4 GB (Windows 7)	Cuckoo Sandbox, processed with Volatility, reverted to clean state	N/A

Table 12: Raw Memory Dump Datasets

Name	Purpose	Sample Numbers	Malware Families	Source	Sandbox Details	Acquisition Method	Availability
Liu (2023)	Detect memory-resident malware using memory forensics and deep neural networks	1010 benign, 1050 malware	Adware, Ransomware, Keylogger, Downloader, Backdoor	Authors' private collection, VirusShare	1 GB (Windows 7)	Volatility (Impscan) tool, memory dumps post-execution	Private
Zhang (2023)	Proposes a malware detection approach based on deep learning and memory forensics	4896 samples total	Fileless malware, Trojans, Ransomware, Keyloggers, Backdoors	VirusShare, MalShare, Microsoft	16 GB (Windows 7/XP)	Custom algorithm, memory dumps every 10 min	Private
Dumpware 10 (2021)	Detect malware using memory forensics, manifold learning, and computer vision	608 benign, 3686 malware	Adware, Worms, Trojans, Virus	Malware: an information security company; Benign: Windows	16 GB (Windows 10)	Procdump v9.0, memory dumps post-execution with 5000 ms delay	Public
Tran (2021)	Detect malware using memory forensics, machine learning, and computer vision techniques	2,750 samples from 8 malware types	Loki-Bot, Agent Tesla, FormBook, StormKitty, OskiStealer, RedlineStealer, SnakeKeylogger	MemGen-generated dataset from MalwareBazar	4 GB (Windows 10)	ProcDump utility, memory dumps post-execution	Public
Nissim (2019)	Detect malware in VMs using MinHash	1,500 malware, 1,500 benign	Ransomware: Vipasana, TeslaCrypt, Chimera, Cerber, Hidden Tear; RATs: DarkComet, Pandora, SpyGate, Comet, Babylon	Real-world malware samples	1 GB (Windows Server 2012R2)	Secure snapshot process, during execution	Public
Sadek (2019)	Study detection of obfuscated malware in volatile memory using deep learning	1,200 malware, 330 benign	Obfuscated malware from reverse shells, Shellter, Hyperion, PEScrambler	VirusTotal	1 GB (Windows 10)	Rekall WinPmem tool, memory snapshots during execution	Public

Table 14: Memory Dump Dataset Scores

Dataset Name	**Sample Numbers**	**Malware Diversity**	**Feature Number**	**Year of Release**	**Memory Dump Size**	**Accessibility**	**Total**	**[0-10] Score**
MemMal-D2024	30	15	12	10	10	5	82/100	8.2
CIC-MalMem-2022	30	15	12	7	10	5	79/100	7.9
Sihwail *et al.* 2021	20	15	20	7	10	5	77/100	7.7
Panker *et al.*	30	15	12	7	10	0	74/100	7.4
Dumpware10	20	15	N/A	7	10	5	57/80	7.1
Nissim *et al.*	15	25	N/A	6	6	5	57/80	7.1
Duan *et al.*	20	25	12	4	10	0	71/100	7.1
Zhang *et al.*	20	15	N/A	10	10	0	55/80	6.8
Sihwail *et al.* 2019	15	15	20	7	10	0	67/100	6.7
Tran *et al.*	15	15	N/A	7	10	5	52/80	6.5
Sadek *et al.*	15	15	N/A	6	6	5	47/80	5.9
Liu *et al.*	15	15	N/A	10	6	0	46/80	5.7
Aljabri *et al.*	15	8	12	10	10	0	55/100	5.5
Lashkari *et al.*	15	15	8	7	10	0	55/100	5.5
Mosli *et al.*	5	15	20	4	6	0	50/100	5.0
Lyles *et al.*	5	15	8	7	10	0	45/100	4.5
Arfeen *et al.*	10	15	5	10	6	0	46/100	4.6
Khalid *et al.*	5	15	8	7	6	0	41/100	4.1

5 Proposed Model

The overview of our methodology is depicted in the figure 15.

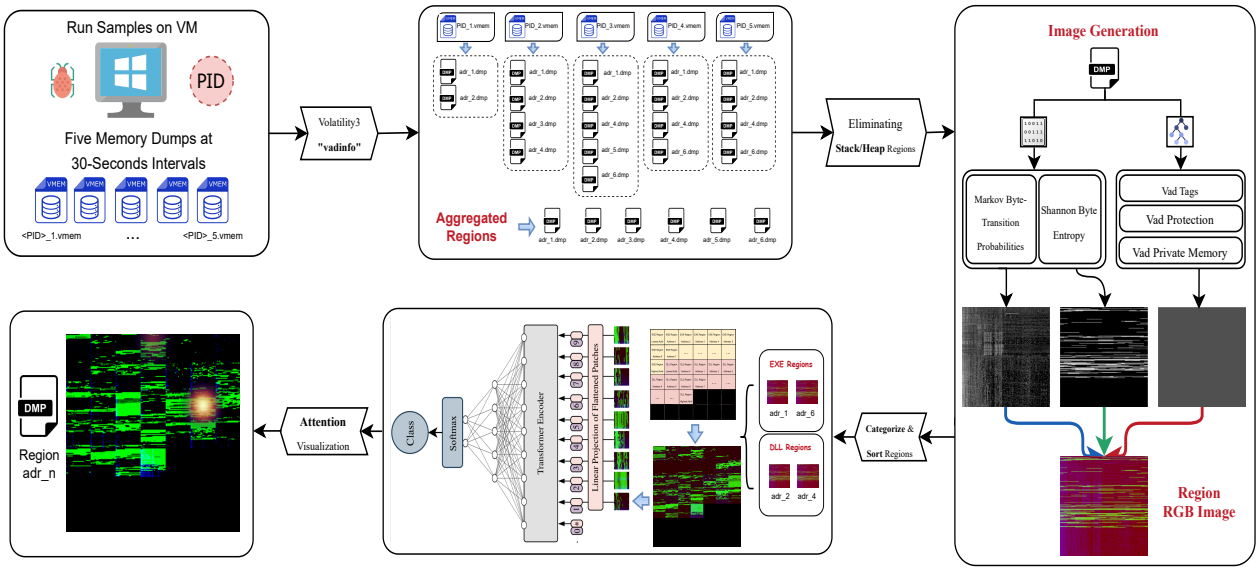


Figure 15: VADViT Methodology Overview

5.1 VAD Region Extraction

The initial step in our methodology involves extracting VAD regions from a known malicious process by utilizing its PID. These regions are obtained directly from dump files generated during execution. As discussed in the 4 section, five dump files are available for each malware sample where the PID of its malicious process was captured during its runtime. However, only a timeout dump file is present when the process is not executed as expected. To ensure the relevance of our dataset, we exclude samples with only one timeout dump file, as they likely represent malicious processes that did not execute properly, making PID retrieval unreliable. Consequently, we refine our dataset to include only samples with five dump files.

We traverse the VAD tree for each selected sample to extract each region by executing the Volatility **vadinfo** command across all five dump files. This step is essential for capturing the malware's behavior at different intervals. The Vadinfo plugin facilitates the traversal of the VAD tree to locate and analyze memory regions efficiently. The traversal process begins by accessing the 'VadRoot' member of the '_EPROCESS' structure, the root node of the self-balancing binary tree used to manage virtual memory allocations. Each node in this tree represents a memory range and con-

tains fields such as ‘StartgVpn‘ and ‘EndVpn‘, which define the starting and ending virtual page numbers. These values must be multiplied by the system page size to determine the actual virtual addresses in memory [94].

As demonstrated in figure 16, the vadinfo command, executed with the dump flag, extracts each VAD region and saves it to a designated folder. Additionally, Volatility provides comprehensive details on these regions. By iterating over the nodes, Vadinfo can identify critical attributes such as protection flags, commit charge values, and whether an area is private or mapped to a file. For example, when analyzing a compromised system, a researcher might extract all VAD nodes and search for suspicious memory regions with ‘PAGE_EXECUTE_READWRITE‘ permissions, often indicative of injected code. Volatility provides direct access to these attributes, allowing analysts to filter VAD entries based on specific characteristics [94]. These attributes are extracted into CSV files for further analysis in our workflow. As a result, for each sample, we generate five separate folders containing both raw memory region files and their corresponding CSV reports.

To refine our data further, we categorized VAD regions based on their protection settings. Most dynamically linked library (DLL) files possess the PAGE_EXECUTE_WRITECOPY protection attribute and are typically file-backed with a ‘.dll‘ extension. These regions are identified and grouped as DLL regions associated with the process. Similarly, the malware executable is file-backed with a ’.exe’ extension, which usually provides the same protection. By examining the file-backed column in the CSV generated by Volatility’s vadinfo command, we separate the malware executable and DLLs into two groups. However, identifying whether a VAD region corresponds to specific process memory segments, such as the heap or stack, is more complex. While file-backed areas can be classified, non-file-backed regions require additional evaluation. Regions lacking the EXECUTE flag in their protection settings are assumed to be non-executable, typically containing heap and stack data. As these constitute a substantial portion of the dataset and are less indicative of malware activity, we exclude them from our analysis. On the other hand, some non-file-backed regions possess execution permissions, making them potential indicators of malicious behavior, including code injections, transient memory allocations, and embedded shellcode. Given their higher probability of containing malicious activity, these regions are grouped alongside the malware executable.

One problem with the protection field from the VAD flags is that it is only the initial protection specified for all pages in the range when they were first reserved or committed. Thus, the current

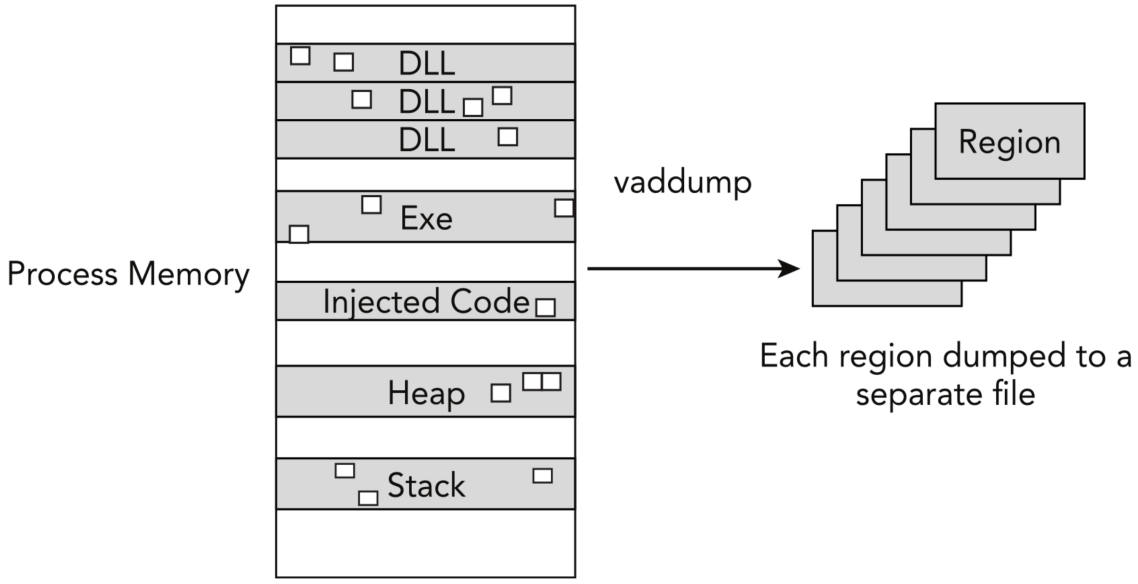


Figure 16: vadinfo plugin with –dump flag extracting each region to separate files

protection can be different. For example, you can call VirtualAlloc and reserve ten pages with PAGE_NOACCESS. Later, you can commit three of the pages as PAGE_EXECUTE_READWRITE and four others as PAGE_READONLY, but the Protection field still contains PAGE_NOACCESS [94]. For an authoritative look at page permissions, we need to consult the bits in the page table, which is not feasible in the scope of our work. Therefore, we assume the protections do not vary from the mentioned protection flags.

By implementing this methodology, we systematically extract and categorize critical memory regions from raw dump files, ensuring a focused analysis on the most relevant aspects of malware execution. This approach prioritizes executable regions and DLL mappings while filtering out less informative data, such as heap and stack regions, thereby optimizing the effectiveness of our memory forensic analysis.

5.2 VAD Region Aggregation

After extracting and categorizing the VAD regions into DLLs, executables, and non-executables across all five memory dumps, the next step involves consolidating these regions to provide the most informative representation of the process. This consolidation ensures that we capture a comprehensive view of the process throughout its runtime and execution period.
To achieve this, we begin by analyzing the categorized regions from the first dump, the initial view of the process. We then examine the areas extracted from the next dump, specifically within the executable and DLL categories, to identify any modifications or updates. At this stage, we encounter three possible scenarios for each region in the next dump:

- **Region Repetition**: If a region in the second dump was already present in the first dump, we replaced the earlier instance with the more recent one. This approach allows us to observe the latest state of the file. It provides more time for the malware to exhibit its complete behavior, ensuring that later modifications are reflected in our analysis.
- **New Region Appearing**: If a region exists in the second dump but was absent in the first dump, we add it to the set of VAD regions and position it according to its memory address. This scenario often occurs because the operating system dynamically allocates new memory regions during execution in response to malware activity. Consequently, newly allocated regions typically appear in subsequent dumps.
- **Region Removal**: If a region in the first dump is absent in the second dump, we retain the removed region in our analysis. This decision comprehensively represents the entire process runtime, capturing active and previously allocated memory regions.

This process is repeated for all five memory dumps, progressively integrating and updating the VAD regions to construct a consolidated memory representation. Once all five dumps are processed, the final VAD region dataset is prepared for the image-generation step of our methodology.

5.3 Image Generation

The memory regions obtained from the vadinfo output in Volatility consist of the memory allocations made by malware processes. Malware prevents the direct disassembly of VAD regions by

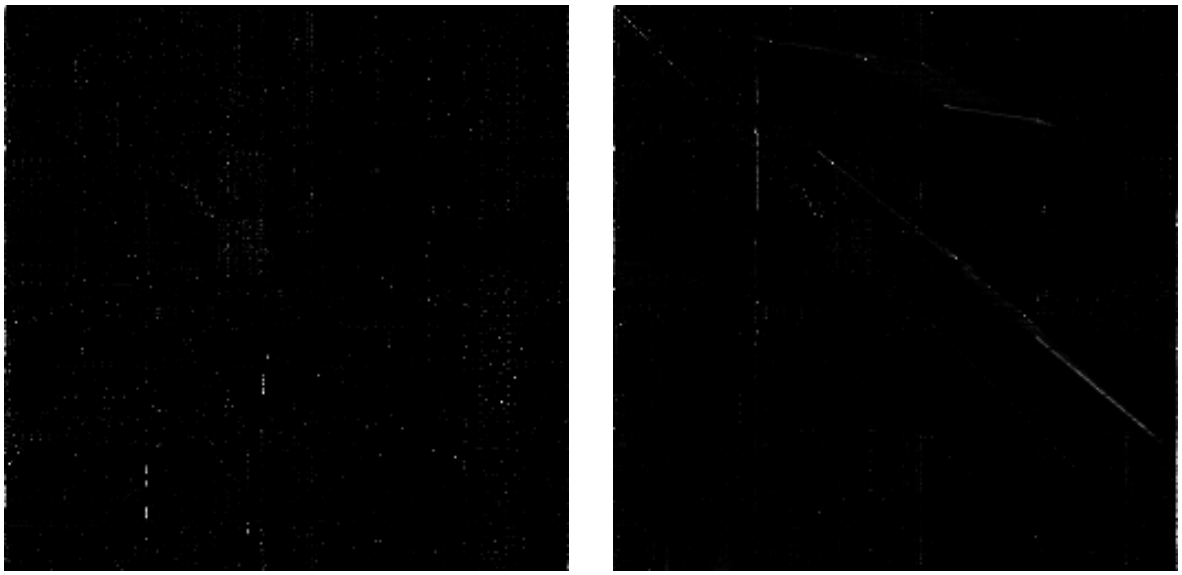
modifying or stripping MZ and PE headers, corrupting structures to evade recognition. Furthermore, feature extraction from raw binaries is challenging due to structural variability, packing, encryption, and header manipulation. Malware obfuscates code, injects dynamically, self-modifies, and disrupts disassembly with junk code and control flow tricks. Therefore, to analyze the extracted memory regions further, our methodology incorporates a visualization approach as the next step. This transformation allows us to capture the raw byte-level representation of these regions in a format suitable for further analysis.

As highlighted in the literature review, visualization has been recognized as an effective technique for malware detection. Prior studies have demonstrated its success in classifying malware executables through various visualization-based approaches. Building on this foundation, we generate three distinct image types for each memory region: Markov, entropy, and a custom intensity image. The selection of these image types is motivated by the wide variation in memory region sizes, ranging from a few kilobytes to over 20 megabytes. Directly reshaping binary content into a uniform size for machine learning could introduce inconsistencies and result in substantial information loss. Although some degree of loss is unavoidable, our approach seeks to minimize it while preserving the essential structural characteristics of the memory regions.

5.3.1 Markov Images

Markov images are among the various applications of Markov chains, particularly in image processing and analysis. Markov chains are stochastic processes. A stochastic process is a collection of random variables representing a system's evolution over time, where future states depend probabilistically on the current state and potentially on past states. Markov chains are a specific type of stochastic process characterized by the Markov property, which asserts that the future state depends only on the present state, not on the preceding sequence of events. This property makes Markov chains particularly useful in various applications, including optimization problems and decision-making scenarios, where states represent different conditions or decisions, and transitions between states are governed by probabilities [128, 129].

In many current malware binary classification methods based on malware images, the bytes of malware binaries are directly used as pixel points in gray images. The influence of information loss in preprocessing the truncation or scaling of malware images on the final accuracy is not considered. Moreover, it does not consider the negative impact of a large amount of information



(a) Exploit malware executable VAD region

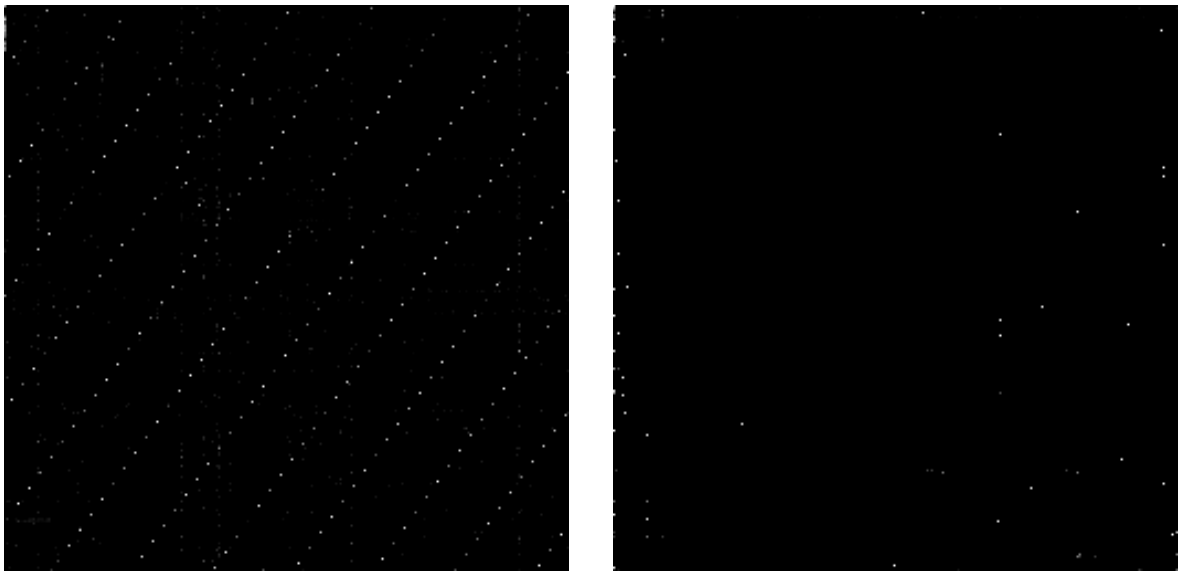
(b) Hoax malware executable VAD region

Figure 17: Markov images of Malware executable VAD Regions for Two Different Families

redundancy brought by this direct conversion [130]. Markov images offer advantages over traditional gray-scale methods by preserving spatial relationships between pixels, enhancing texture representation, and structural analysis. This property is crucial for edge detection and image classification, where connectivity between pixels improves accuracy [131]. Additionally, the probabilistic nature of Markov models enables robustness against noise and artifacts, making them suitable for complex image structures [131].

Malicious processes typically allocate VAD regions with distinct structural and byte-level similarities compared to benign processes. These similarities stem from the consistent techniques employed by malware developers, leading to recognizable memory allocation patterns even among malware from the same family. Figure 17 presents Markov images from the VAD regions of two malware processes in our dataset. It illustrates the distinct visual patterns observed with the naked eye, demonstrating the potential to distinguish between the two categories.

This is not exclusive to the file-backed regions, as the figure 18 shows the visual difference between two regions with PAGE_EXECUTE_READWERITE protection allocated by the same two malware families. Furthermore, the raw VAD regions exhibit significant variability in size, ranging from a few Kilobytes to more than 20 megabytes. We mitigate inconsistencies and standardize the data format by converting these regions into a fixed-size image representation.



(a) Exploit VAD Region (b) Hoax VAD Region

Figure 18: Markov Images of Non-File-Backed VAD Regions for Two Different Families

5.3.1.1 Generating the Markov Images

The bytes of VAD region raw files could be viewed as a byte stream representing a stochastic process [130] defined below in the formula 1
where N refers to the length of the VAD region binary:

$$Byte_i, \quad i \in \{0, 1, \ldots, N-1\} \tag{1}$$

Assuming that the probability of $Byte_i$ is only related to $Byte_{i-1}$, then the *random variable* $Byte_i$ is a *markov chain* defined in the equation 2:

$$P(Byte_{i+1}|Byte_0, \ldots, Byte_i) = P(Byte_{i+1}|Byte_i) \tag{2}$$

In Markov chains, transition probabilities between states are represented in a transition matrix, encapsulating the likelihood of moving from one state to another. This framework allows for the analysis of long-term behavior, such as steady-state distributions and ergodicity, which are crucial for understanding the stability and performance of systems modeled by Markov chains [132]. The probability matrix is the normalized transition frequency matrix that counts the occurrences of byte value changes [131].

Since the value of each byte ranges from 0 to 255, $Byte_i$ has 256 possible states, $Byte_i \in \{0, 1, \ldots, 255\}$. The Markov transfer probability matrix is generated based on the transfer probability of each state [130]. If $P_{m,n}$ indicates the transfer probability that the subsequent byte of m is n, then the formula for $P_{m,n}$ is as follows in the 3:

$$p_{m,n} = P(n|m) = \frac{f(m,n)}{\sum_{n=0}^{255} f(m,n)} \tag{3}$$

where $f(m,n)$ indicates the frequency at which m is followed by n, and $m,n \in \{0,1,\ldots,255\}$. Considering the probability that all bytes are transferred to each other, the transfer probability matrix M is formed [130]-:

$$M = \begin{bmatrix} p_{0,0} & p_{0,1} & \cdots & p_{0,255} \\ p_{1,0} & p_{1,1} & \cdots & p_{1,255} \\ \vdots & \vdots & \ddots & \vdots \\ p_{255,0} & p_{255,1} & \cdots & p_{255,255} \end{bmatrix} \tag{4}$$

Malware Markov images are generated using matrix M. In Markov images, each transfer probability is mostly dark images to a pixel point at $M_{m,n}$.

5.3.1.2 Enhancing the Markov Images

Enhancing raw Markov images improves visual clarity in memory forensics and malware detection. Specific 2-byte transitions (usually zeros) can occur more frequently than others, causing an imbalance in the probability distribution after normalization. As a result, the more common transitions dominate the visualization, while rarer byte transitions contribute little to the overall image structure. This leads to mostly dark images, with only small scattered details, reducing their effectiveness for analysis. Also, the integration of machine learning techniques, as discussed by [18], demonstrates that enhanced image clarity can lead to better classification outcomes in malware detection scenarios. Additionally, anti-forensic techniques can obscure critical data, making refined visual representation essential for reliable forensic investigations [2].
We applied the following enhancements to overcome this issue in our proposed methodology.

•Logarithmic Frequency Scaling

This enhances Markov image generation by effectively modeling image data across various scales. This approach addresses the complexity and variability inherent in image data, providing a more

balanced distribution of data, which is crucial for accurately modeling low-frequency components [133]. Therefore, the initial raw frequency counts of byte transitions are transformed using a logarithmic function 5, which helps compress large values while preserving distinctions among smaller ones.

$$\text{log-scaled frquencies} = \log(1+x) \tag{5}$$

where x represents the Markov byte frequency matrix elements.

•Contrast Stretching

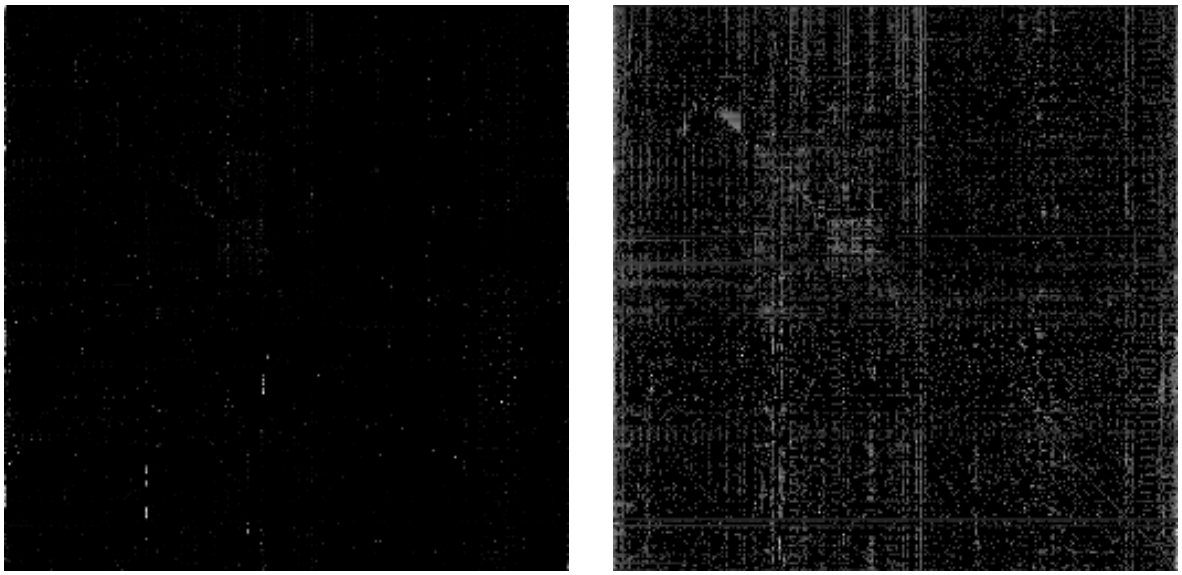
Contrast stretching enhances image quality in medical imaging and microscopy by improving luminance distribution and addressing poor lighting conditions [134]. Combined with adaptive thresholding, it refines image quality in retinal imaging [135]. High dynamic range (HDR) imaging enhances morphological features for better detail identification [136]. Additionally, it improves ultrasound image clarity, aiding in accurate diagnostics [137]. As this technique is effective in various domains, we apply it to enhance our Markov images' clarity and overall quality, ensuring a more informative representation for analysis. This prevents data from being too concentrated within a small range and enhances contrast in the final visualization. After computing the probability table, the values are stretched between their minimum and maximum values, as shown in the equation 6.

$$S(i,j) = \frac{P(i,j) - P_{\min}}{P_{\max} - P_{\min}} \tag{6}$$

where:

- $S(i,j)$ is the contrast-stretched probability at position (i,j).
- $P(i,j)$ represents the original probability at position (i,j) in byte frequency matrix.
- $P_{\min} = \min(P(i,j))$ is the minimum probability value in the matrix.
- $P_{\max} = \max(P(i,j))$ is the maximum probability value in the matrix.

•Square Root Enhancement



(a) Exploit Executable VAD Region (b) Enhanced Exploit Executable VAD Region

Figure 19: Markov images of non-file backed VAD Regions for Two Different Families

Integrating square-root techniques allows for more accurate estimations in Markov models, leading to improved image quality and detail retention, especially in applications involving complex signals and images[138]. This enhancement technique improves the visual quality of images. It contributes to better performance in classification tasks, as demonstrated in various studies where Markov image enhancement methods significantly improved classification accuracies across different datasets[139, 140]. Thus, we used this non-linear adjustment to boost lower probability transitions, making subtle details more visible without overly distorting the high probability transitions.

$$E(i,j) = \sqrt{S(i,j)} \tag{7}$$

where $E(i,j)$ is the square root enhanced probability at position (i,j).
At the final step, histogram equalization is applied to re-normalize the enhanced probabilities to the range [0,1] to ensure the transformed values span the full dynamic range of intensity levels. Figure 19 presents the effects of the visual enhancements on the Markov images.

5.3.2 Entropy Images

Entropy is a key metric in byte stream analysis as it measures uncertainty, randomness, or disorder in a system. In particular, Shannon entropy is a specific formulation of entropy in information theory shown in the equation 8 and is commonly used to quantify this randomness and disorder in data.

$$H(X) = -\sum_{i} p(x_i) \log p(x_i) \tag{8}$$

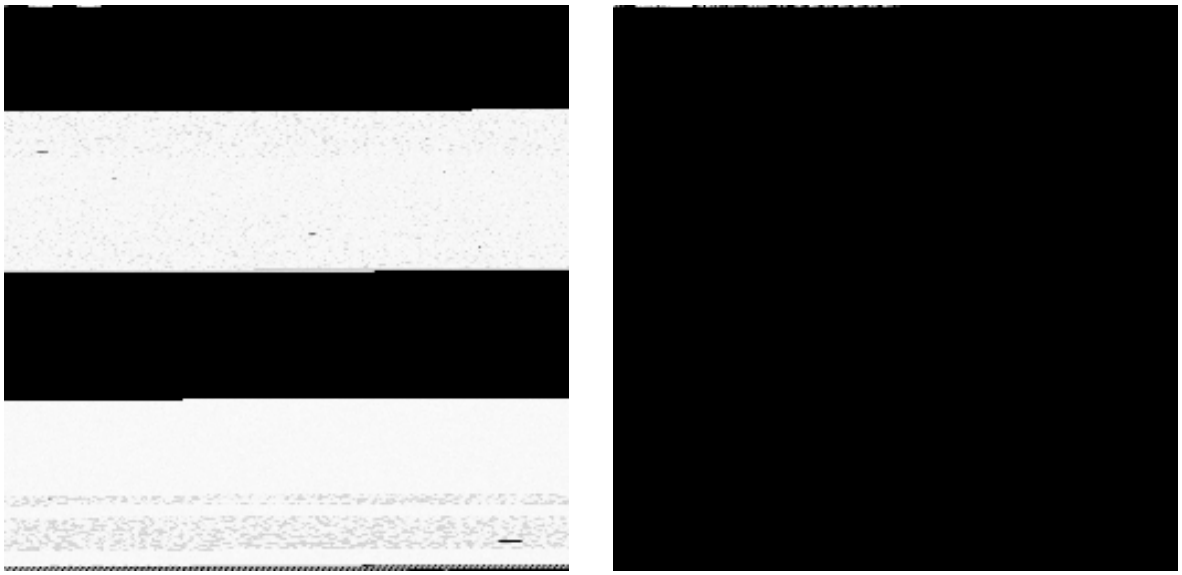
where:

- $H(X)$ is the Shannon entropy, representing the uncertainty in the dataset.
- X is a discrete random variable with possible outcomes x_i.
- $p(x_i)$ is the probability of occurrence of the outcome x_i.

This makes it an effective tool for evaluating obfuscation techniques such as packing and encryption in malware detection. Low-entropy packing schemes compress malware to evade traditional static analysis [141], while high-entropy values result from complex packing methods that obscure malicious intent [142, 143]. Entropy profiles segment files into chunks for entropy calculation, effectively distinguishing between benign and polymorphic malware [144, 145]. As a result, entropy analysis remains a crucial tool for enhancing adaptive malware detection techniques against evolving threats [146, 147, 148].

The comparison of entropy images in Figure 20 highlights the contrast between the binary file contents of a VAD region associated with a malware executable and one allocated for a benign system network DLL (ntdll.dll). This distinction underscores the structural and informational differences between malicious and legitimate memory allocations. The prevalence of high-intensity pixel values in the entropy image of the malware VAD region indicates greater randomness and uncertainty in its byte contents compared to the benign image, which appears almost entirely black. This contrast suggests the presence of obfuscation or packing techniques commonly employed by malicious processes to evade detection.

Entropy images can not only differentiate between benign and malicious memory regions, but they can also distinguish between different malware families. As shown in Figure 21, the various



(a) Rootkit Malware Executable VAD Region

(b) Benign Network DLL VAD Region

Figure 20: Entropy Images of Malware Executable VAD Region and Benign DLL Region

patterns in these images are evident, making variations between malware families recognizable even to the naked eye.

5.3.2.1 Generating Entropy Images

To generate entropy images from the VAD regions, we calculated Shannon entropy on the byte stream of the binary content within each region. Since the final images must be uniform in size, we employed a dynamic window size approach that adjusts based on the VAD region's byte length and the required fixed image dimensions. A fixed window size approach was considered but was found to introduce inconsistencies, as it resulted in images of varying dimensions that required resizing, potentially distorting entropy values.

Instead, we adopted a dynamic window size instead of a fixed-size byte window. To generate an $n \times n$ entropy image, we require exactly n^2 entropy values. To achieve this, we first extract the byte length of the VAD region binary and divide it by n^2 to determine the appropriate window size for entropy calculation. If the byte stream length is not a multiple of n^2, we pad it with null bytes to ensure even division. This method eliminates the need for resizing, as the entropy values are computed directly to fit the final $n \times n$ image dimensions, resulting in uniformly sized images across all VAD regions.

(a) Rootkit Malware Executable VAD Region

(b) Exploit Malware Executable VAD Region

Figure 21: Entropy Images of Different Malware Executable VAD Regions

5.3.3 Custom Intensity Images

The VAD regions allocated by a malware process possess high-level attributes such as tags, protection settings, and commitcharge. While the previously generated images focused on capturing the byte-level contents of these regions, providing a detailed low-level representation, they do not account for the type of regions or how they are allocated. To address this, we introduce a third image that visualizes these high-level attributes, ensuring a more comprehensive representation of the VAD regions.

5.3.3.1 VAD Tags

As discussed in Section 3, each memory region can be assigned one of three VAD tags: Vad, VadS, and VadF, which define the type of memory allocation associated with a particular VAD region. Memory regions that are file-backed almost always carry the Vad tag. According to Table 9, the Vad tag indicates that the region follows a _MMVAD_LONG structure allocated when a process requires a relatively large memory range. This is particularly relevant for file-backed regions, as they usually demand additional space in memory for mapping and management [94].
In contrast, VadS and VadF tags indicate the allocation of shorter memory ranges, typically reserved for transient or dynamically allocated regions. Since VadS and VadF tags often indicate

small, non-file-backed regions that may contain injected shellcode, they serve as critical indicators for identifying potential malicious memory modifications. This makes them valuable for forensic investigators in detecting code injection attempts [94]. While VadS and VadF tags are linked to the _MMVAD_SHORT structure and are often used for non-file-backed memory allocations, they exhibit key differences. VadS (Short VAD) regions are commonly seen in memory allocations for thread stacks, heaps, and, in some cases, injected shellcode. These regions lack additional metadata, resulting in a more compact structure, making them well-suited for thread stack memory allocations [94]. In contrast, VadF (Short VAD with Flags) regions include additional flags that provide insights into commit status, protection levels, and memory privacy. Because of these flags, VadF-tagged regions are more frequently associated with code injection cases, where a malicious shellcode is injected into a process's address space [94].

5.3.3.2 Protection

The protection attribute of VAD memory regions is a crucial feature for forensic analysis as it provides insight into the access permissions assigned when the memory was initially allocated. Table 10 outlines eight different protection types that can be applied to VAD regions. Given our classification of regions into DLLs, malware executables, and heap/stack regions, we focus exclusively on executable regions while filtering out those without EXECUTE permissions. As a result, the protection values relevant to our workflow are:

- PAGE_EXECUTE
- PAGE_EXECUTE_READ
- PAGE_EXECUTE_WRITECOPY
- PAGE_EXECUTE_READWRITE

As shown in Table 10, the PAGE_EXECUTE_WRITECOPY protection type is never allocated using VirtualAlloc or VirtualAllocEx, which are commonly used for dynamic memory allocation. This is further illustrated in Figure 8, where the diagram demonstrates how different Windows APIs interact with memory allocation protections. Almost all DLL files in our dataset exhibit PAGE_EXECUTE_WRITECOPY protection, confirming that this setting is typical for file-backed library modules rather than dynamically allocated or injected memory regions. Similarly, Table 10

highlights that PAGE_EXECUTE_READWRITE protection is frequently associated with injected code, allowing regions to be read, written, and executed—a strong indication of potential malware behavior. In conclusion, the regions with the PAGE_EXECUTE_READWRITE can be considered more interesting in a post-incident memory forensic approach.

5.3.3.3 Private Memory

Private memory refers to memory regions allocated by a process that are not shared with other processes and are not file-backed. Such regions are typically created using system calls like 'VirtualAlloc()' or 'HeapAlloc()' and are commonly used for managing stacks, heaps, and dynamically allocated memory. However, they also serve as a prime target for malware attempting to inject code stealthily[94]. Since private memory is often assigned PAGE_EXECUTE_READWRITE permissions, it provides an ideal environment for injecting and executing malicious payloads without leaving disk artifacts. As a result, memory regions with the private memory bit set highly indicate potential malware activity and warrant closer examination.

5.3.3.4 Generating Custom Intensity Images

The previous subsections analyzed individual VAD region attributes separately. However, forensic investigations require a holistic approach, as no single attribute alone can flag a memory region as suspicious. Each attribute offers a partial view of the memory's structure, but malicious activity is often identified by examining combinations of attributes rather than isolated properties. For instance, a private memory region may be entirely benign, while an executable region could belong to a legitimate application. Furthermore, VadS and VadF tags often indicate non-file-backed memory regions. Still, without additional context, it is difficult to determine whether a region is a standard allocation or contains injected code [94].

To address this, we ranked the VAD attribute values based on their probability of indicating malicious behavior and assigned a custom pixel intensity to each attribute value. Table 16 shows that all of these numbers were chosen as prime numbers so that subsequent combinations of the attributes do not result in similar pixel intensities for different combinations.

Table 17 details these pixel intensity values and their corresponding VAD attribute combinations, which are the main source for generating our custom pixel intensity images. This approach allows us to visualize high-level VAD attributes (rather than byte-level contents) in a structured manner,

Table 16: Pixel Intensity Values for Different VAD Region Attributes

Attribute	Values	Pixel Intensity
VAD Tag Type	Vad (Standard)	0
	VadS (Short VAD)	47
	VadF (Flagged VAD)	59
Memory Protection	PAGE_EXECUTE_WRITECOPY	0
	PAGE_EXECUTE_READ	61
	PAGE_EXECUTE	61
	PAGE_EXECUTE_READWRITE	91
Private Memory	No (Mapped or Shared)	0
	Yes (Private Allocation)	23

providing a multi-source feature set for the next step of our methodology.

If the pixel values in this channel are too high, they can dominate the overall intensity of the final image, reducing the relative contribution of details from the other channels. To mitigate this effect, we normalized the values in Table 17 to 0–120, reducing the maximum possible value from 173 to 120.

5.3.4 Final Image Representation

In the last step of the image generation step of our methodology, we have three gray-scale images visualizing the complete low-level and high-level characteristics of VAD memory regions of the malicious process. Inspired by the GEM images discussed in [149], we fuse them into a 3-channel image illustrated in figure 22. By stacking the Markov, Entropy, and Custom Intensity images as the Blue, Green, and Red channels of an RGB image, respectively, we make a compact representation of our region data.

Once each memory region has been visualized, the corresponding region images must be concatenated to construct a process-level visualization of the target process. The virtual addresses of these regions facilitate this process, as the images inherently follow a sequential order. By sorting the patch images based on their respective memory addresses, we can effectively represent the flow of memory allocations requested by the process, arranging the images sequentially from the lowest to the highest memory address. However, it is essential to note that the positions of these memory ranges are not static, particularly on systems utilizing Address Space Layout Ran-

Table 17: Sorted Pixel Intensity Summation for Attribute Combinations

VAD Tag Type	Memory Protection	Private Memory	Total Pixel Intensity
VadF (Flagged VAD)	PAGE_EXECUTE_READWRITE	Yes (Private Allocation)	173
VadS (Short VAD)	PAGE_EXECUTE_READWRITE	Yes (Private Allocation)	161
VadF (Flagged VAD)	PAGE_EXECUTE_READWRITE	No (Mapped or Shared)	150
VadF (Flagged VAD)	PAGE_EXECUTE	Yes (Private Allocation)	143
VadF (Flagged VAD)	PAGE_EXECUTE_READ	Yes (Private Allocation)	143
VadS (Short VAD)	PAGE_EXECUTE_READWRITE	No (Mapped or Shared)	138
VadS (Short VAD)	PAGE_EXECUTE	Yes (Private Allocation)	131
VadS (Short VAD)	PAGE_EXECUTE_READ	Yes (Private Allocation)	131
VadF (Flagged VAD)	PAGE_EXECUTE	No (Mapped or Shared)	120
VadF (Flagged VAD)	PAGE_EXECUTE_READ	No (Mapped or Shared)	120
Vad (Standard)	PAGE_EXECUTE_READWRITE	Yes (Private Allocation)	114
VadS (Short VAD)	PAGE_EXECUTE	No (Mapped or Shared)	108
VadS (Short VAD)	PAGE_EXECUTE_READ	No (Mapped or Shared)	108
Vad (Standard)	PAGE_EXECUTE_READWRITE	No (Mapped or Shared)	91
Vad (Standard)	PAGE_EXECUTE	Yes (Private Allocation)	84
Vad (Standard)	PAGE_EXECUTE_READ	Yes (Private Allocation)	84
VadF (Flagged VAD)	PAGE_EXECUTE_WRITECOPY	Yes (Private Allocation)	82
VadS (Short VAD)	PAGE_EXECUTE_WRITECOPY	Yes (Private Allocation)	70
Vad (Standard)	PAGE_EXECUTE	No (Mapped or Shared)	61
Vad (Standard)	PAGE_EXECUTE_READ	No (Mapped or Shared)	61
VadF (Flagged VAD)	PAGE_EXECUTE_WRITECOPY	No (Mapped or Shared)	59
VadS (Short VAD)	PAGE_EXECUTE_WRITECOPY	No (Mapped or Shared)	47
Vad (Standard)	PAGE_EXECUTE_WRITECOPY	Yes (Private Allocation)	23
Vad (Standard)	PAGE_EXECUTE_WRITECOPY	No (Mapped or Shared)	0

domization (ASLR). In such environments, thread stacks may be located either below or above the process executable, and mapped file regions can be dispersed throughout the entire process memory space rather than being grouped contiguously [94], as illustrated in Figure 7. To address the challenge posed by ASLR, we standardize the process image representation, ensuring a consistent visualization across all process VAD regions. This is achieved by consistently prioritizing executable VAD regions over DLL regions.

To implement this uniform representation, we define a $m \times m$ grid representing the entire process

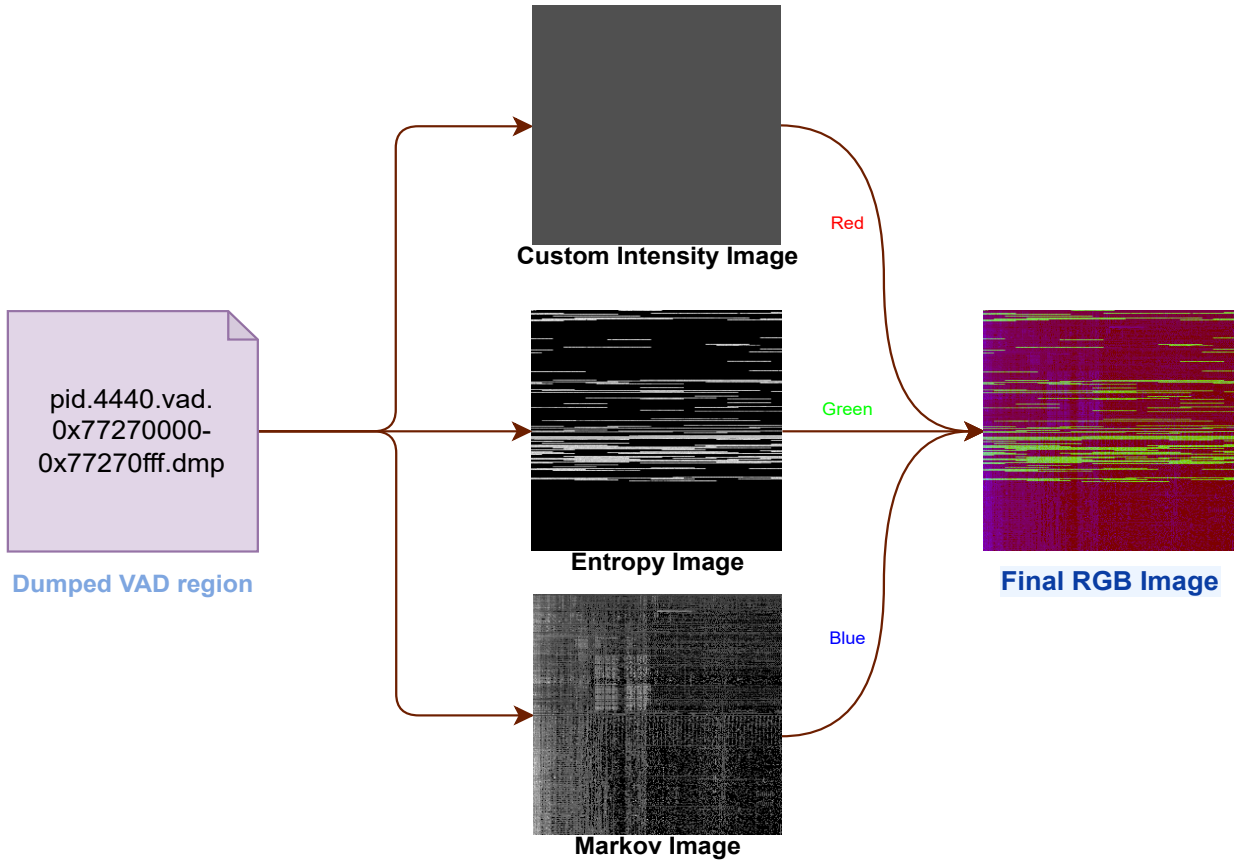


Figure 22: Visualizing Method for Dumped VAD Regions

memory, where each cell corresponds to a VAD region. The grid is structured, beginning with the executable regions, which are inserted based on the order of their memory addresses. Once all executable regions have been placed, the DLL regions are added sequentially, following their respective address order. By adopting this approach, we establish a standardized representation of processes across different systems and varying memory allocation patterns.

This approach presents specific bottlenecks. One of the primary challenges is ensuring that the final image used for the model maintains a fixed size, which dictates the total number of cells in the grid and the pixel dimensions of each cell. Various combinations of these parameters are systematically tested and analyzed to address this. The results of these experiments, along with a discussion of their impact, are presented in Section 6.

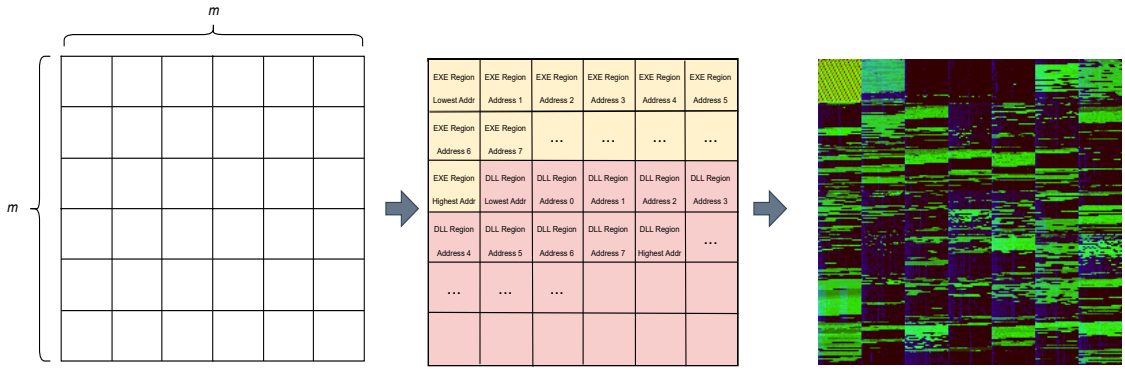


Figure 23: Sorting the Visualized Regions and Generating Process Image

5.4 Proposed Model

The model we use to classify the VAD regions of the target process is the Vision Transformer (ViT) proposed by Dosovitskiy *et al.* [150]. It is shown in [150] that the reliance on Convolutional Neural Networks is unnecessary, and a pure transformer applied directly to sequences of image patches can perform very well on image classification tasks.

5.4.1 Model Selection Motivation

Vision Transformer framework applies the Transformer architecture directly to sequences of image patches, eliminating the need for convolutional networks. The model captures global dependencies through multi-head self-attention by treating images as a series of non-overlapping patches, each linearly embedded into a fixed-dimensional space. Unlike convolutional architectures that rely on local receptive fields and translation equivariance, ViT learns spatial relationships from data, leveraging large-scale pre-training for improved generalization. This approach has demonstrated strong performance on image classification benchmarks while maintaining computational efficiency, remarkably when fine-tuned on downstream tasks [150].

Now that each VAD region has been transformed into an RGB image, the next step is concatenating these region images to represent the entire process rather than isolated memory segments. This is crucial in selecting and proposing the appropriate model as the nature of our conceptual model aligns with the patch-based representation used in ViTs. Unlike CNNs, which rely on convolutions that assume spatial locality and continuity, ViTs process images as sequences of discrete patches without imposing locality constraints [150]. This characteristic enables ViTs to

capture long-range dependencies between patches more effectively, whereas CNNs may struggle when spatial relationships in the data are inconsistent or non-continuous [150]. The operating system VAD defines contiguous yet separate regions in the virtual address space representing the whole process memory. However, these regions may not always be physically continuous and sometimes non-adjacent, even in the virtual space [94]. Similarly, ViT treats image patches as independent tokens during initial embedding, without assuming spatial continuity, yet still learns to represent them as part of a unified structure through self-attention[150]. Hence, by structuring VAD region images in a manner compatible with ViTs, we leverage their ability to model arbitrary relationships between patches to generate meaningful patch embeddings.

Another important reason for choosing Vision Transformers (ViTs) for this analysis is the structured order of VAD regions. Each dumped memory file from a process has a specific offset in the physical memory address and a corresponding range of virtual page numbers, creating a meaningful sequence that ViT's attention mechanism can effectively leverage. Unlike CNNs, which rely on localized feature extraction through convolutional layers, ViTs use a global self-attention mechanism that allows each image patch to relate to all others, regardless of their spatial positioning [150]. This makes ViTs particularly effective in capturing long-range dependencies, whereas CNNs inherently prioritize spatial locality by progressively aggregating features from neighboring regions [150]. By applying ViTs, we can analyze malware behavior beyond individual memory regions, focusing on how regions interact rather than being limited to a sequential or neighboring structure. This broader perspective allows the model to understand malware's overall allocation patterns, leading to improvement in distinguishing between benign and malicious memory activities.

5.4.2 Vision Transformer

The proposed model follows the Transformer framework introduced by Vaswani et al. (2017) [151] and extends its application to vision tasks through the Vision Transformer, as presented by Dosovitskiy et al. (2021) [150]. Unlike conventional convolutional neural networks (CNNs), which process images through hierarchical feature extraction, the Vision Transformer treats an image as a sequence of non-overlapping patches and processes them using a self-attention-based encoder. The model overview is demonstrated in the figure 24.

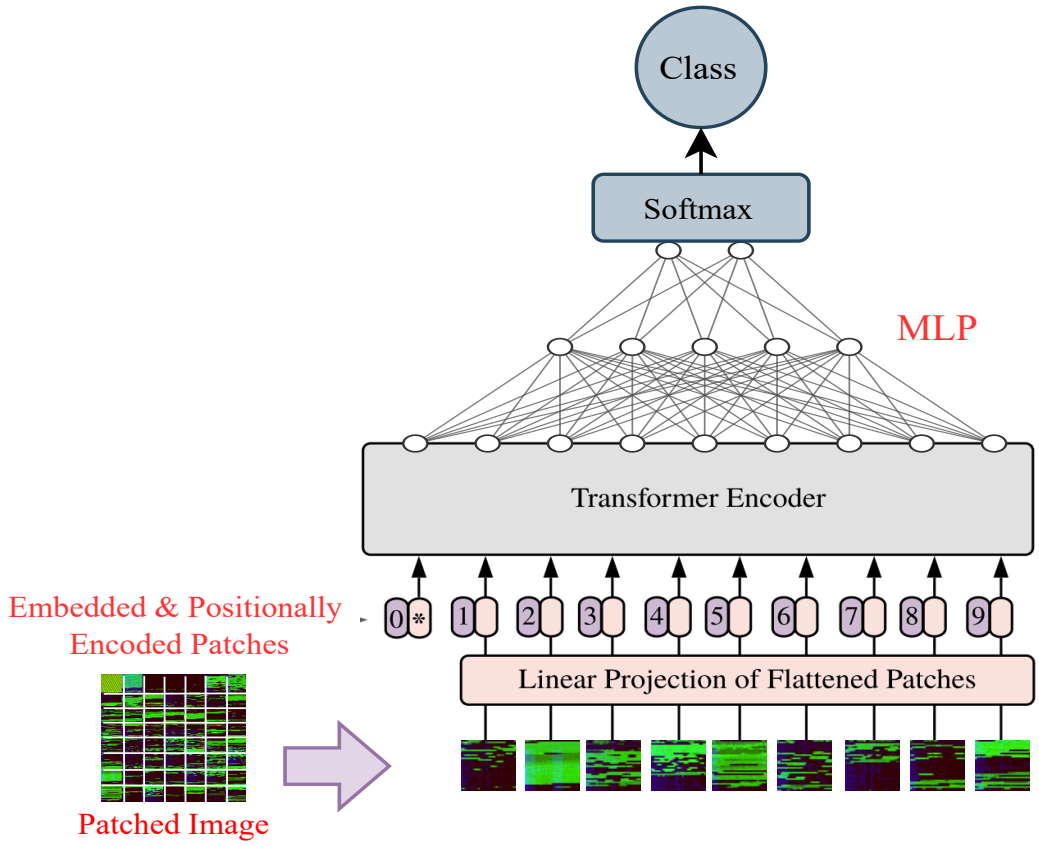


Figure 24: Vision Transformer Model Overview

5.4.2.1 Tokenizing the Input Image

Traditional CNNs use convolutional filters to progressively extract spatial features, but ViT restructures the input differently. Instead of applying local filters, the input image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ is split into non-overlapping patches of size $P \times P$ as defined in the equation 9:

$$N = \frac{H \times W}{P^2} \tag{9}$$

where:

- H, W are the height and width of the image.
- C is the number of channels (e.g., RGB: $C = 3$).
- $P \times P$ is the patch size.
- N is the total number of patches, each treated as a token in a Transformer model [150].

The total number of patches (N) also serves as the effective input sequence length for the Transformer encoder. Each patch is then flattened into a vector and mapped into a lower-dimensional embedding space using a trainable linear projection defined in the equation 10 where $\mathbf{E} \in \mathbb{R}^{(P^2C)\times D}$ is a learnable embedding matrix and D is the Transformer's hidden dimension. In most transformer architectures, this hidden dimension is 768

$$\mathbf{x}_p = \mathbf{x}\mathbf{E} \tag{10}$$

Since Transformers do not inherently capture positional information, a learned positional embedding $\mathbf{E}_{\text{pos}}$ is added to the patch embeddings as in 11. $\mathbf{x}_{\text{class}}$ is a learnable classification token that aggregates information across all patches and $\mathbf{E}_{\text{pos}}$ is a trainable positional encoding matrix, allowing the model to capture spatial structure [150]. At this stage, the image is a sequence of token embeddings similar to a sentence and is ready for processing by the Transformer blocks.

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_1\mathbf{E}; \mathbf{x}_2\mathbf{E}; \ldots; \mathbf{x}_N\mathbf{E}] + \mathbf{E}_{\text{pos}} \tag{11}$$

5.4.2.2 Processing Through the Transformer Encoder

The ViT model uses a Transformer encoder, introduced in [151] to process the input sequence. The transformer encoder consists of alternating layers of multi-headed self-attention(MSA) and Multi-Layer Perceptron (MLP) blocks with normalization (LN) applied before every block and residual connections after them. The MLP contains two layers with a GELU non-linearity [150]. As illustrated in Figure 25, the Transformer block processes the input data through multiple layers, following the equations (12, 13, 14, 15) presented alongside the figure, ultimately producing the block's output.

$$\mathbf{z}_0 = \left[\mathbf{x}_{\text{class}}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots; \mathbf{x}_p^N\mathbf{E}\right] + \mathbf{E}_{\text{pos}} \tag{12}$$
$$\mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}, \quad \mathbf{E}_{\text{pos}} \in \mathbb{R}^{(N+1) \times D}$$

$$\mathbf{z}'_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \quad \ell = 1 \ldots L \tag{13}$$

$$\mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, \quad \ell = 1 \ldots L \tag{14}$$

$$\mathbf{y} = \text{LN}(\mathbf{z}_L^0) \tag{15}$$

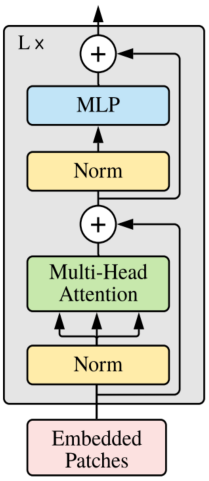


Figure 25: Transformer Encoder Layers

5.4.2.3 Multi-Head Self-Attention

At each layer, the Transformer determines the importance of each token (patch) about others in the sequence using the multi-head self-attention (MSA) mechanism. This is computed through the query (Q), key (K), and value (V) matrices [151]. These matrices represent the embedded and positionally encoded patches, which are multiplied by learnable weight matrices $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V$ to project them into the respective query, key, and value vector spaces [151]. Figure 26 illustrates this process, where three linear blocks (corresponding to the three aforementioned matrices) transform the input before passing it through the scaled dot-product attention block.

$$\mathbf{Q} = \mathbf{z}\mathbf{W}^Q, \quad \mathbf{K} = \mathbf{z}\mathbf{W}^K, \quad \mathbf{V} = \mathbf{z}\mathbf{W}^V \tag{16}$$

The attention scores are computed using scaled dot-product attention. The input consists of the set of queries packed together into a matrix Q, and the keys and values are also packed together into matrices K and V. The dot-product is a similarity measure used by this architecture to identify the most relevant K to the Q [151].

The computed **MatMul** in Figure 26 is then divided by $\sqrt{d_k}$ to prevent exploding gradients. A softmax function is applied to obtain the weights on the values and normalize the scores so they sum to 1 [151].

The final attention values (**2nd MatMul**) are generated in such a way that each patch (V) contributes proportionally according to its relationship strength to Q. This operation is performed for each Q in the input image patch sequence.

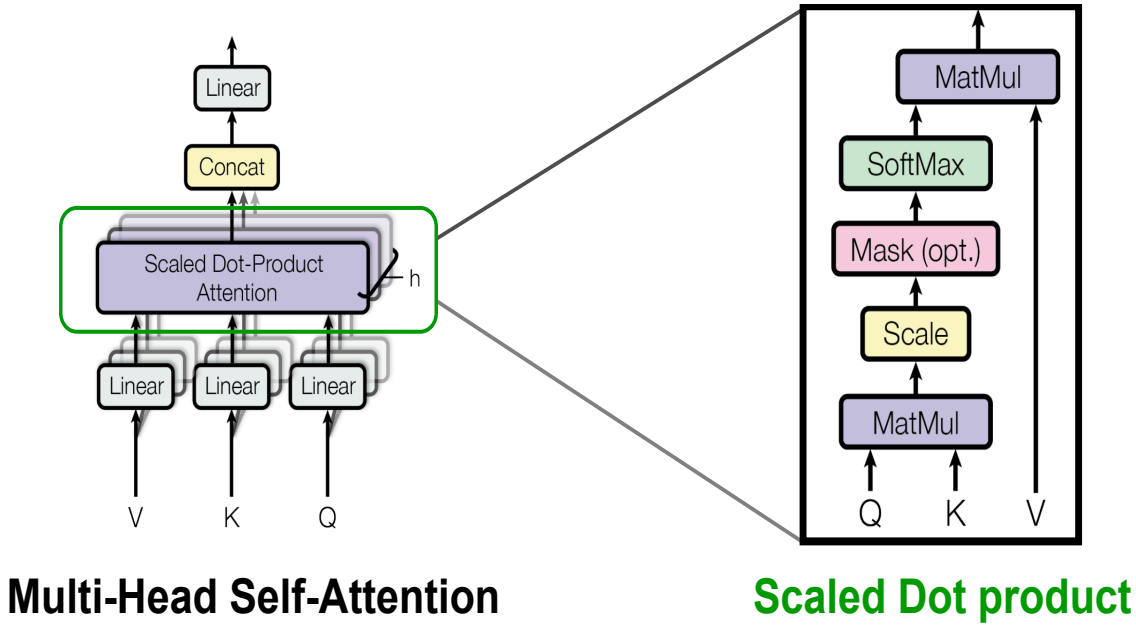


Figure 26: Multi-Head Self-Attention Block

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{17}$$

It is worth noting that figure 26 shows multiple attention heads (h) running in parallel, where each attention head learns a different representation of the patches [151]. Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions [151]. The attention scores from each head are concatenated and once again projected with another learnable matrix ($\mathbf{W}^O$), resulting in the final values.

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}^O \tag{18}$$

5.4.2.4 Feed-Forward Network and Classification

After the attention mechanism has aggregated information across the input tokens, each token undergoes further transformation independently through a position-wise feed-forward network (FFN), allowing per-patch transformations. This step applies a two-layer fully connected network

where $\mathbf{W}_1, \mathbf{W}_2$ are learnable weight matrices with a RELU non-linearity [151] (**??**).

$$\text{FFN}(x) = \max(0, x\mathbf{W}_1 + b_1)\mathbf{W}_2 + b_2 \tag{19}$$

At the final Transformer layer, the classification token $\mathbf{x}_{\text{class}}$, which has aggregated contextual information across all patches, is extracted as the global representation of the entire image. To ensure numerical stability, layer normalization (LN) is applied where L is the total number of Transformer layers [150] (**??**).

$$\mathbf{y} = \text{LN}(\mathbf{z}_{\text{class}}^{L}) \tag{20}$$

A fully connected classification head maps this final representation to output class probabilities. The $\mathbf{W}$ in 21 is a learnable weight matrix, and the softmax function converts raw logits into a probability distribution over the possible output classes [150].

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{y}\mathbf{W}) \tag{21}$$

6 Experiments and Results

This section presents the experimental setup, design, and evaluation of our VADViT model on the dataset introduced in Section 4. We then provide a detailed explanation of each experiment and analyze the corresponding results.

6.1 Dataset Details

As outlined in Section 4, we introduce the BCCC-MalMem-SnapLog-2025 dataset to address the limitations of existing datasets in the field. Our dataset comprises 2,000 malware samples, each obtained from the VirusTotal malware repository [152]. These samples are categorized into eight distinct types: Backdoor, Hoax, HackTool, Trojan, Worm, Virus, Rootkit, and Exploit.

Additionally, 250 benign samples were collected from various sources, including Windows system executables (System32 binaries), the Sysinternals Suite, GitHub Releases, and a selection of benign portable executable files from the VirusTotal and CNET [153] repository. The inclusion of diverse sources ensures that our model does not develop a bias toward distinguishing Microsoft applications as benign, thereby preventing a simplistic "Microsoft vs. non-Microsoft" classification approach.

As detailed in Section 4, a time window of 600 seconds was allocated for a malware process to be detected as an active PID. However, some malware samples did not execute within the given timeframe, so a memory dump was created at the end of the period to ensure comprehensive data collection. If a process does not appear as an active PID during execution, creating a system memory dump ensures that we still capture the state of the machine where the malware ran. This approach allows us to retain valuable data even for inactive or unobservable processes. While our primary focus is on tracking instances through their PIDs, preserving a complete system snapshot ensures that the dataset remains useful for various types of research beyond just PID-based analysis.

In some cases, malware processes appeared only once within the given timeframe, triggering an immediate capture of the first memory snapshot as soon as they were detected in the active process list. However, many malware samples behave unpredictably, running only briefly before disappearing. As a result, some samples are captured only in the initial snapshot, while others persist across multiple memory dumps. Table 18 provides an overview of the BCCC-MalMem-

SnapLog-2025 dataset, showing how malware and benign samples are distributed—whether they timed out, appeared in just one memory dump, or were captured in multiple snapshots. This behavior is not exclusive to malware as most of the benign samples also appeared only in the first dump and were not there in the subsequent ones 18.

Class	Timeout	Single	Multiple	Total (Non-Timeout)
Benign	28	177	39	216
Backdoor	15	188	47	235
Exploit	0	108	142	250
HackTool	69	68	112	180
Hoax	39	4	207	211
Rootkit	38	16	196	212
Trojan	13	92	145	237
Virus	11	106	133	239
Worm	16	97	137	234

Table 18: Distribution of samples based on process persistence across memory snapshots

6.1.1 Data Cleaning

Timeout samples were excluded from our analysis because the process never appeared in the active process list, leaving no extractable VAD regions. After this data cleaning step, a total of **2014** samples, comprising both benign and malicious instances, remained for use in the VADViT framework. Following the visualization of each process discussed in 5, the dataset was split into training, validation, and test sets using an 8:1:1 ratio. The exact distribution of these samples is presented in Table 19.

Class	Train	Test	Validation
Backdoor	185	21	18
Benign	171	25	17
Exploit	197	35	22
HackTool	143	20	22
Hoax	167	18	25
Rootkit	164	18	23
Trojan	176	21	27
Virus	204	25	19
Worm	189	18	26
Total Benign Samples	171	25	17
Total Malicious Samples	1425	176	182
Total Samples	1596	201	199

Table 19: Data distribution of samples in the Train, Test, and Validation sets

6.1.2 Augmentation

The relatively small size of our dataset raises concerns about overfitting. We applied various data augmentation techniques to the training dataset to mitigate this issue and enhance model generalization. However, it is important to consider certain constraints when implementing these techniques. As outlined in section 5, visualized VAD regions are arranged across the grid image according to their memory address order. Consequently, any augmentation technique that disrupts this order cannot be used, as it would compromise the model's ability to generate meaningful and consistent embeddings for patches in the ViT model. When the order of patches varies across the training samples with the same label, the model may struggle to maintain coherent feature representations. Therefore, augmentation techniques such as flipping, rotating, cutmix [154], and similar transformations were excluded. Instead, we employed augmentation methods that introduce in-place variations within patches without altering their spatial arrangement.

These transformations include modifications to color balance, contrast, and sharpness, as well as adjustments to pixel distributions. Additionally, we incorporated noise injection and randomized filtering effects to further enhance dataset diversity. Table 20 provides a detailed overview of the augmentation techniques applied in the training pipeline.

Augmentation Technique	**Description**
Random Posterization	Reduces bit depth per channel (applied with $p = 0.4$)
Random Solarization	Inverts pixel values above a threshold ($p = 0.4$)
Random Inversion	Randomly inverts pixel colors ($p = 0.3$)
Random Equalization	Adjusts histogram for contrast improvement ($p = 0.4$)
Autocontrast	Enhances contrast by stretching pixel range ($p = 0.4$)
Sharpness Adjustment	Modifies image sharpness by a factor of 2 ($p = 0.4$)
Color Jittering	Adjusts contrast, saturation, and hue ($p = 0.2$, $s = 0.2$, $h = 0.05$)
Random Grayscale	Converts images to grayscale with probability $p = 0.2$
Gaussian Blur	Applies a blur with sigma range $(0.1, 1.0)$ ($p = 0.65$)
Gaussian Noise	Injects noise ($\mu = 0.0$, $\sigma = 0.02$, $p = 0.4$)

Table 20: Overview of Applied Data Augmentation Techniques

Following augmentation, all images were converted to tensors and is normalized to ImageNet [155] mean ($[0.485, 0.456, 0.406]$) and standard deviation ($[0.229, 0.224, 0.225]$) to ensure compatibility with the ViT model pretrained on ImageNet dataset. Unlike the training set, the validation and test datasets were not augmented, ensuring consistency during evaluation.

6.2 Experiment Setup

Our execution environment was implemented on an ASUS GL502VM physical machine equipped with an Intel Core i7-7700U processor running at 2.40 GHz, 16 GB of RAM, and an NVIDIA

GeForce GTX 1060, utilizing CUDA Toolkit 11.8 for GPU acceleration. To run the malware samples and generate memory dump files, we employed QEMU version 9.0.4 on a VM configured with 4 GB of RAM and five CPU cores, running Windows 11 Home Edition. The primary programming environment used for our implementation was Python 3.8.

6.2.1 Image Size Variations

The varying patch sizes and overall image dimensions for the ViT input require a systematic comparison of different configurations. Since processes contain a varying number of VAD regions, the grid size ($m \times m$) directly influences the number of VAD regions that can be included in the image representation, as illustrated in Figure 23. A smaller patch size allows for more VAD regions to be included in a fixed-size image, providing a broader view of the entire process. However, reducing patch size may result in a loss of fine-grained details in each patch, which could affect the representation of byte-level information in visualized regions. On the other hand, increasing the image size allows for more fixed-size patches without reducing their scale. However, since many dataset samples appear only in the first memory dump, this can lead to images that are mostly empty, with just a few scattered patches. This sparsity, combined with the limited dataset size, might make it harder for the model to learn meaningful patterns, potentially affecting its performance.

This trade-off between maximizing the number of included VAD regions and maintaining sufficient detail in each patch necessitates experimentation with different values of grid image size m. The ViT model input must be a fixed, square image size, and specific dimensions are required depending on the architecture. Many ViT architectures, such as ViT-Base [150], ViT-Large [28], ViT-Huge [28], DeiT [156], and Swin Transformer [157], have a default input size of 224×224, with some supporting a higher resolution of 384×384 for finer details. Additionally, each of these architectures employs a fixed patch size. Table 21 presents a detailed comparison of these configurations across different ViT architectures.

Given the limited dataset size, large models such as ViT-Large and ViT-Huge were excluded due to their high parameter count, which increases the risk of overfitting. Similarly, the Swin Transformer architecture was not used, as its hierarchical patch merging mechanism (starting with a 4×4 patch size) disrupts VAD region independence by merging regions and also leads to information loss during resizing to such small image size. Instead, we opted for the ViT-Base architecture,

ViT Model	Input Size	Patch Size	Num Patches ($m \times m$)
ViT-Base/Large/Huge [150, 28]	224×224	16×16	$14 \times 14 = 196$
	224×224	32×32	$7 \times 7 = 49$
	384×384	16×16	$24 \times 24 = 576$
	384×384	32×32	$12 \times 12 = 144$
Swin Transformer [157]	224×224	4×4 (Hierarchical)	$56 \times 56 = 3136$
	384×384	4×4 (Hierarchical)	$96 \times 96 = 9216$
DeiT [156]	224×224	16×16	$14 \times 14 = 196$
	384×384	16×16	$24 \times 24 = 576$

Table 21: ViT Architectures, Patch Sizes, and Number of Patches

which features a default patch size of 16×16 but also supports 32×32 patches. In addition to these considerations, if the number of regions in a process still exceeds the number of available slots in the grid image, we select only the first $m \times m$ patches that fit within it. Since the order of these regions is crucial, we avoided applying any region selection methods (e.g., entropy-based sorting), as they would disrupt the original sequence. Given that our model is transformer-based, we prioritized maintaining the correct order of the sequence over preserving its full length.

6.2.2 Model Fine-Tuning

For our experiments, we utilize the ViT-Base model consisting of 12 Transformer layers, each with 12 self-attention heads, and a hidden dimension of 768, while the MLP hidden size is 3072. The classification head uses the [CLS] token, processed through a fully connected layer. As the number of our samples are insufficient we used the pretrained model that was trained on the ImageNet-1K [155] dataset with 1000 classes and 88 million model parameters.

When fine-tuning a pre-trained ViT model on a limited dataset, layer unfreezing plays a crucial

role in balancing feature transferability and model adaptation. Recent efforts have shown that the front layers primarily extract general features of the raw data (e.g., the shape of objects in an image) and often become well-trained much earlier, while deeper layers are more task-specific and capture complicated features output from front layers [158]. When fine-tuning a pre-trained model on a new task, we can freeze (that is, fix the weights of the layers and not update them) the front layers or only fine-tune them for a few iterations and focus on training the deep layers on the new dataset (unfreezing the layers) [158]. This also reduce the training cost by 45% leading to faster training time for fine-tuning tasks [158]. Mostly, the idea is that we freeze the layers when their performance is stable and unfreeze them when the learning rate decreases [158]. To explore the effect of gradual layer unfreezing, we experiment with four different layer freezing strategies presented in the table 22. This setup allows us to directly compare how gradual layer unfreezing impacts accuracy, training stability, and the model's ability to generalize with a small dataset.

Strategy	Frozen Layers	Unfreezing Steps
1	4 Frozen Layers	2 Steps (Unfreeze two layers in each step)
2	6 Frozen Layers	3 Steps (Unfreeze two layers in each step)
3	9 Frozen Layers	3 Steps (Unfreeze three layers in each step)

Table 22: Layer Freezing and Unfreezing Strategies

6.2.3 Stochastic Weight Averaging

Stochastic Weight Averaging (SWA) enhances generalization by averaging model weights over training iterations, leading to flatter optima and improved test accuracy [**?**]. Unlike standard SGD, which converges to sharper minima, SWA shifts the solution toward a broader, more robust region, reducing overfitting. SWA is implemented by maintaining a running average of weights while using a cyclical or constant learning rate. It approximates Fast Geometric Ensembling (FGE) but with a single model, avoiding ensemble overhead [**?**]. In our work, Stochastic Weight Averaging (SWA) is applied during the final four epochs of training to improve generalization and stability. Once the training reaches epoch 28, we initialize the SWA model by creating an AveragedModel

instance that maintains a running average of the model's parameters. The learning rate for SWA is set to half of the last adjusted learning rate obtained from ReduceLROnPlateau, ensuring a smooth transition. Using a cosine annealing strategy over five epochs, the SWALR scheduler gradually refines the learning rate, facilitating convergence to a flatter optimum.

6.2.4 Evaluation Metrics

The evaluation metrics utilized in this study have been widely adopted in the literature and offer a detailed assessment of the proposed approach [25, 23, 21, 24]. We employed standard classification metrics, including accuracy, precision, recall, and F1-score, to quantify different aspects of model performance. These metrics are derived from the fundamental classification outcomes:

- True Positives (TP): The number of correctly classified instances belonging to the positive class.
- False Positives (FP): The number of cases where the model incorrectly predicts a positive class when the actual label is negative.
- True Negatives (TN): The count of correctly classified instances belonging to the negative class.
- False Negatives (FN): The number of instances where the model misclassifies actual positive cases as negative.

Accuracy measures the overall correctness of the model, computed as the proportion of correctly classified instances:

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \tag{22}$$

Precision evaluates the reliability of positive predictions by calculating the fraction of correctly predicted positive instances among all predicted positives:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{23}$$

Recall (Sensitivity) measures the model's ability to identify actual positive cases, defined as the proportion of true positives detected out of all actual positives:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{24}$$

F1-score is the harmonic mean of precision and recall, providing a balanced measure that accounts for both false positives and false negatives:

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{25}$$

We employ additional measures, namely the Receiver Operating Characteristic (ROC) curve and the confusion matrix, to gain deeper insights into classification performance. The ROC curve visualizes the trade-off between the true positive rate (TPR) and the false positive rate (FPR) across different decision thresholds:

$$\text{TPR} = \frac{TP}{TP + FN}, \quad \text{FPR} = \frac{FP}{FP + TN} \tag{26}$$

The area under the curve (AUC-ROC) quantifies the classifier's ability to discriminate between classes, with values closer to 1 indicating better performance. Additionally, the confusion matrix provides a granular breakdown of the model's predictions, detailing the counts of TP, FP, TN, and FN. This allows for direct observation of classification errors and helps in identifying biases toward certain classes.

6.3 Results

After multiple evaluations and refinements, the final hyperparameter configuration for the ViT model used in our experiments is presented in Table 23. In this binary classification task with an imbalanced dataset, macro-averaged precision, recall, and F1-score are reported to mitigate bias toward the majority class. By calculating each metric per class and taking their arithmetic mean, the macro-averaging approach fairly represents both benign and malicious samples.

6.3.1 Binary Classification

Table 24 summarizes how three hyperparameters—patch size, image resolution, and layer-freezing strategy—significantly influence the model's detection capabilities.

Hyperparameter	Value
Epochs	36
Learinng Rate	0.0002
Batch Size	1
Model Checkpoint	$monitor =' ValidationLoss', mode =' min'$

Table 23: Overview of Hyperparameter Selection

6.3.1.1 Patch Size

A larger patch size (32×32) consistently yields higher F1-scores and precision across multiple configurations by preserving the structural integrity of VAD regions, ensuring critical forensic features like executable code, injected payloads, and mapped DLLs remain intact. In contrast, smaller patches (16×16) require excessive downscaling, leading to information loss that can obscure entropy variations and byte-level representation of the regions. This, combined with the increased token (patch) count, raises computational overhead and the risk of overfitting to irrelevant artifacts.

6.3.1.2 Image Size

Increasing the resolution from 224×224 to 384×384 does not consistently improve performance, with only minor variations in F1-score, recall, and precision. While larger images accommodate more patches, they also introduce sparsity, particularly when many samples appear only in the first memory dump, potentially reducing the model's ability to capture meaningful patterns. From a forensic point of view, excessive sparsity can dilute the visibility of critical VAD regions, making it harder to identify malicious allocations. However, 384×384 remains competitive with 224×224 across most setups, except in the 224×224, 32×32 configuration with six frozen layers and three-step unfreezing, which yields the best overall performance.

Image Size	Patch Size	Freezing Strategy	Accuracy	F1 Score	Precision	Recall
224×224	16×16	4 Frozen, 2 Steps	0.91	0.76	0.71	0.73
		6 Frozen, 3 Steps	0.93	0.72	0.66	0.90
		9 Frozen, 3 Steps	0.92	0.83	0.84	0.82
	32×32	4 Frozen, 2 Steps	0.96	0.90	0.91	0.90
		6 Frozen, 3 Steps	**0.99**	**0.96**	**0.95**	**0.97**
		9 Frozen, 3 Steps	0.97	0.90	0.91	0.90
384×384	16×16	4 Frozen, 2 Steps	0.89	0.71	0.70	0.73
		6 Frozen, 3 Steps	0.92	0.69	0.77	0.66
		9 Frozen, 3 Steps	0.91	0.75	0.75	0.76
	32×32	4 Frozen, 2 Steps	0.96	0.90	0.90	0.86
		6 Frozen, 3 Steps	0.98	0.93	0.99	0.88
		9 Frozen, 3 Steps	0.98	0.94	0.97	0.93

Table 24: Evaluation Metrics for Different Training Configurations

6.3.1.3 Layer Freezing Strategy

No single layer freezing strategy is universally optimal. Freezing four layers and unfreezing them in two steps often leads to lower recall, suggesting unstable early feature extraction. Freezing nine layers shows mixed results—strong precision in some cases but lower F1-scores and recall in others, indicating possible constraints on adaptability. Freezing six layers and unfreezing them in three steps offers the best balance, achieving the highest accuracy (0.99) and F1-score (0.96) in the 224×224, 32×32 configuration. This approach retains general early-layer features while enabling deeper layers to learn domain-specific patterns, such as suspicious VAD regions. However,

the nine-layer strategy remains competitive in some scenarios, making it an alternative. Figure 27 visualizes these performance variations across different configurations.

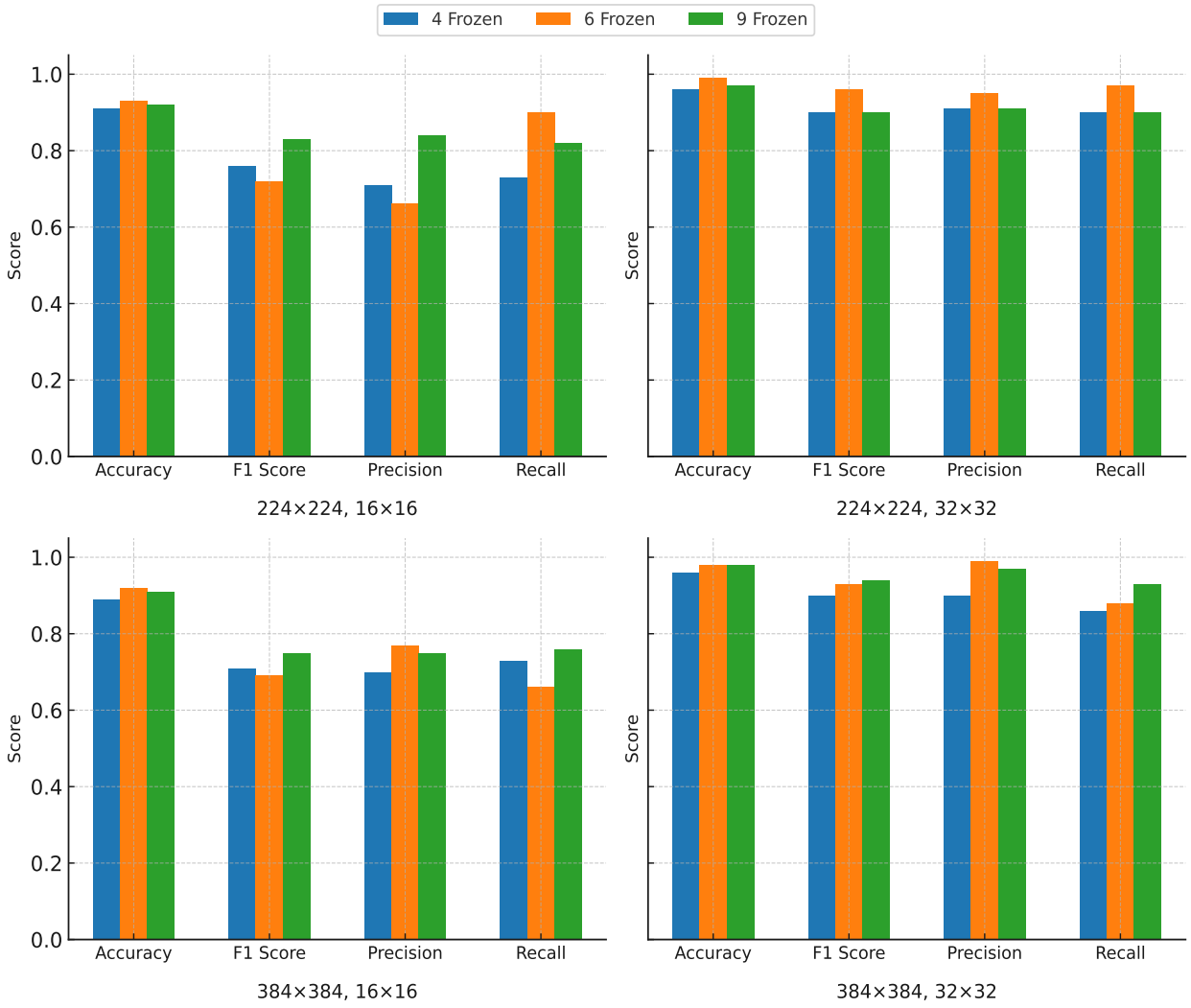
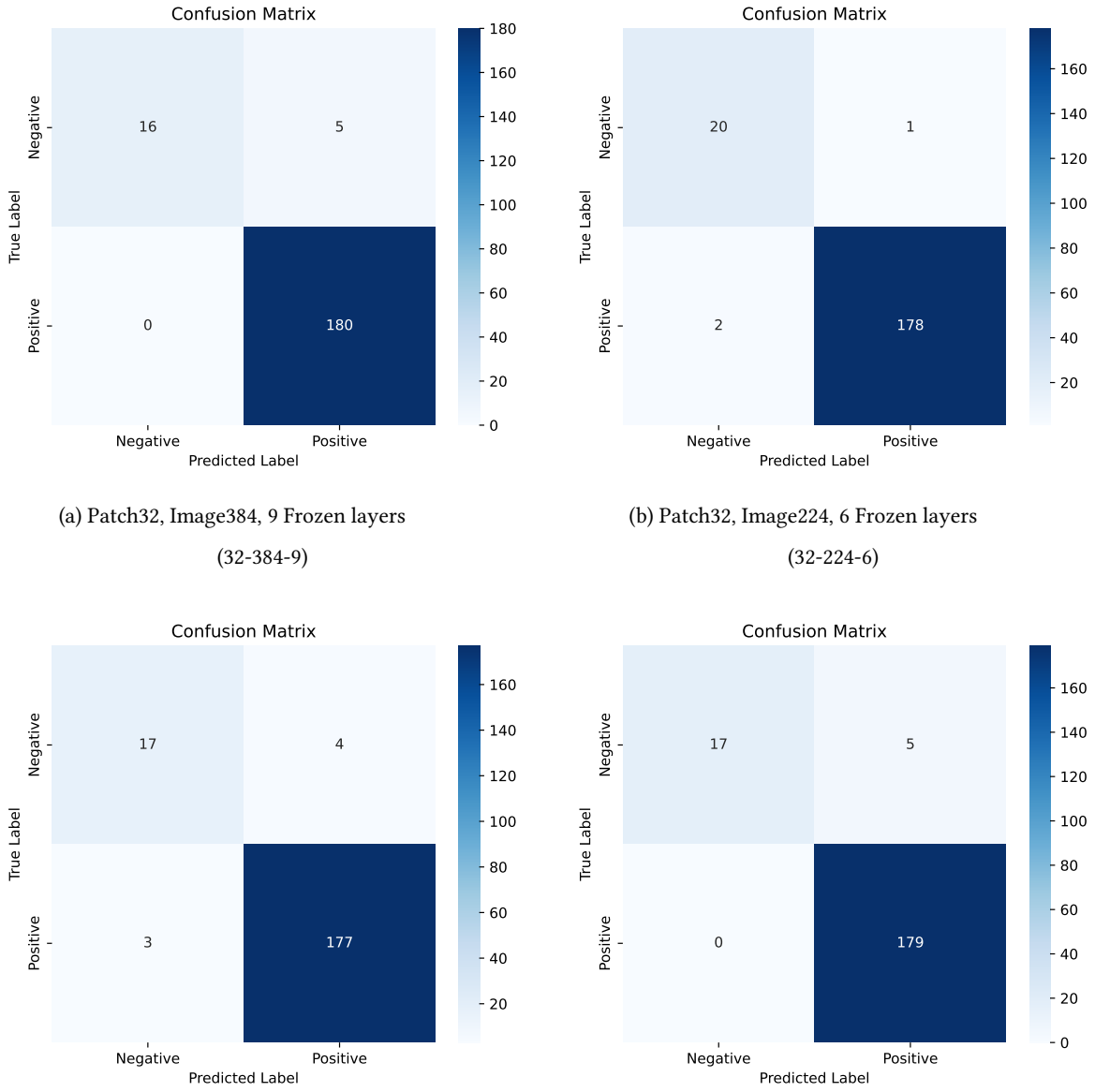


Figure 27: Comparison of the Different Image Sizes and Patch Sizes Across the Freezing Strategy

6.3.1.4 Additional Metrics

Due to the relatively similar performances of these configurations, the confusion matrices of the top four models are depicted in figure 28. Model configuration of ”32-384-9f” did not produce false negatives, which means it correctly identified all malicious samples, but all misclassified instances were benign samples mistakenly labeled malicious. Another model ”32-224-6f” had two false negatives, meaning it missed two malicious cases, but its misclassification was distributed across both benign and malicious classes. These results indicate different patterns in how errors

occur, with one model strictly overestimating maliciousness while the other misclassifies in both directions.



(a) Patch32, Image384, 9 Frozen layers (32-384-9)

(b) Patch32, Image224, 6 Frozen layers (32-224-6)

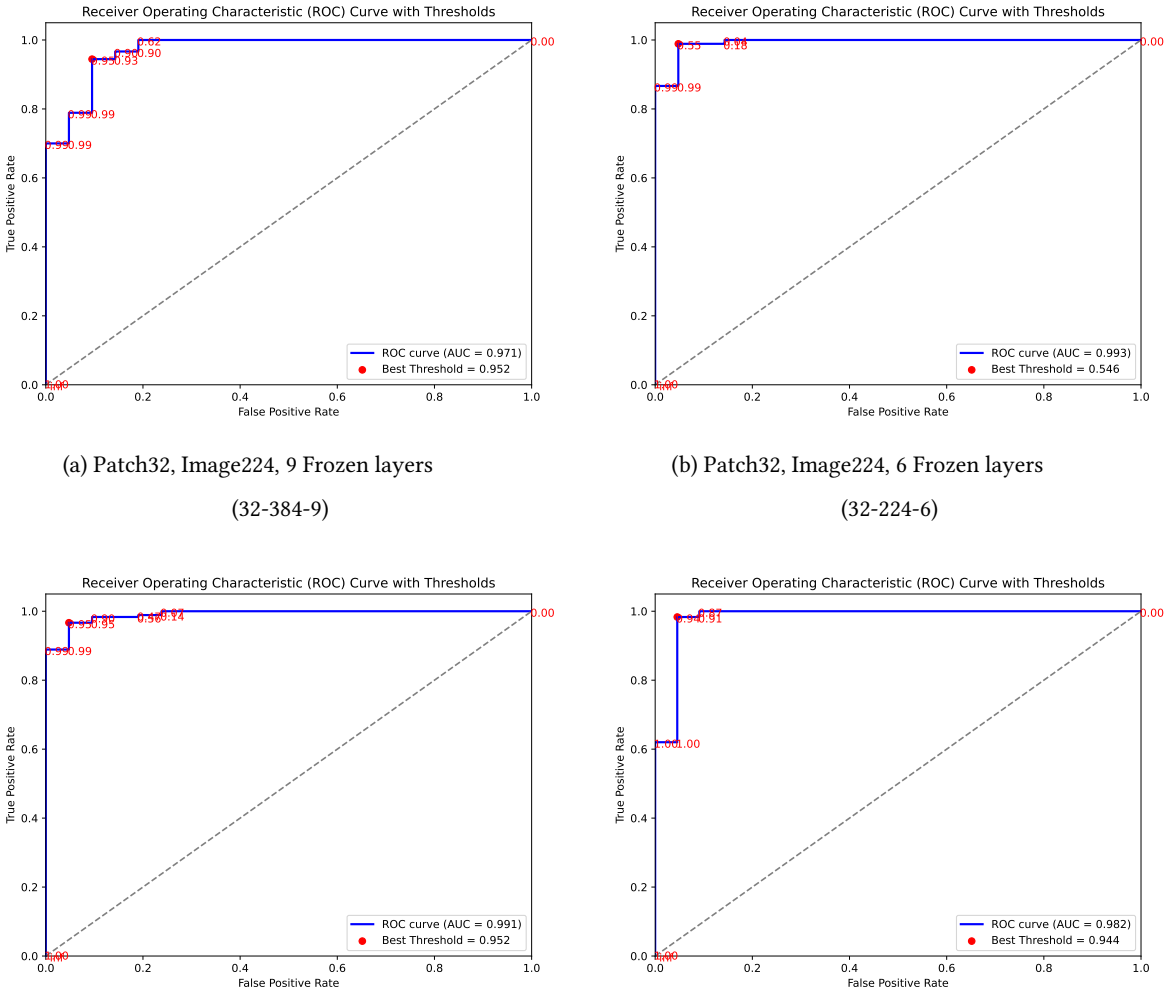
(c) Patch32, Image224, 9 Frozen layers (32-224-9)

(d) Patch32, Image384, 6 Frozen layers (32-384-6)

Figure 28: Confusion Matrices of the Top 4 Best-Performing Models

It is worth noting that most benign samples did not appear in multiple memory dumps, making their behavioral patterns less distinctive. In their initial memory footprints, both benign and malicious processes rely on common system DLLs, such as ”kernel32.dll” and ”ntdll.dll”, for essential operations like network communications in their initial memory footprint. Consequently, pro-

cesses that appear in only a single dump may not have exhibited their full memory allocation patterns or specific DLL usage yet. This lack of distinctiveness makes it difficult to differentiate them based solely on the presence of these common libraries, as malware often exploits the same DLLs for persistence or evasion. Despite challenges in early-stage classification, the VADViT framework shows exceptional accuracy, achieving a macro-averaged F1-score of 0.96 in detecting malicious memory artifacts.



(a) Patch32, Image224, 9 Frozen layers (32-384-9)

(b) Patch32, Image224, 6 Frozen layers (32-224-6)

(c) Patch32, Image384, 9 Frozen layers (32-224-9)

(d) Patch32, Image384, 6 Frozen layers (32-384-6)

Figure 29: ROC Curves of the Top 4 Best-Performing Models

We plotted their Receiver Operating Characteristic (ROC) curves and calculated their Area Under the Curve (AUC) scores to provide a more detailed and clearer comparison. Figure 29 shows that the 32-224-6 model achieved the highest AUC at 0.993, while the configuration with the 384

image size and nine frozen layers followed closely with an AUC of 0.991. The 32-224-9 model with recorded an AUC of 0.982, slightly outperforming the 32-384-6, which had an AUC of 0.971. Moreover, figure 30 presents the training and validation metrics in all epochs, highlighting the points where gradual layer unfreezing and stochastic weight averaging were applied.

Figure 30: Training and validation metrics over 36 epochs for the best-performing model

6.3.2 Multi-Class Classification

To further assess the effectiveness of our framework in analyzing VAD regions, we evaluate its performance in classifying malicious samples according to their respective malware families. The dataset consists of eight distinct labels, each corresponding to a specific malware family (Backdoor, Hoax, HackTool, Trojan, Worm, Virus, Rootkit, and Exploit) along with an additional benign class. For this classification task, we selected the two best-performing configurations from the binary classification experiments, ”32-224-6f” and ”32-384-9f”, as they achieved the highest AUC scores in their respective settings. The classification results are summarized in Table 25, while

their confusion matrices are illustrated in Figure 31.

	Patch32, 224, 6 frozen layers			Patch32, 384, 9 frozen layers		
Class	**Precision**	**Recall**	**F1-Score**	**Precision**	**Recall**	**F1-Score**
Backdoor	0.81	1.00	0.89	0.70	0.93	0.80
Benign	0.93	1.00	0.96	0.78	0.90	0.84
Exploit	1.00	0.83	0.91	0.96	0.96	0.96
HackTool	1.00	0.90	0.95	0.95	0.82	0.88
Hoax	0.95	1.00	0.97	1.00	0.96	0.98
Rootkit	0.95	1.00	0.97	1.00	0.93	0.96
Trojan	0.81	0.81	0.81	0.82	0.85	0.84
Virus	0.96	0.88	0.92	0.90	0.86	0.88
Worm	0.84	0.89	0.86	1.00	0.91	0.95
Macro Avg	0.92	0.92	0.92	0.90	0.90	0.90
Weighted Avg	0.92	0.92	0.92	0.91	0.90	0.90
	Accuracy		**0.92**	**Accuracy**		**0.90**

Table 25: Classification performance comparison for two model configurations on various malware classes.

6.3.2.1 Performance Comparison and Observations

The table shows that the 32-224-6f configuration achieves an overall accuracy of 92%, while 32-384-9f slightly lags at 90% accuracy. Both models exhibit strong performance across most classes, but ”32-224-6f” achieves higher precision, recall, and F1-scores in most malware categories.

Higher Recall for Smaller Image Size (224×224): The 32-224-6f configuration achieves perfect recall (1.00) for Benign, Rootkit, and Hoax, ensuring minimal false negatives. In contrast, the 32-384-9f configuration struggles with recall drops in HackTool (0.82), Backdoor (0.93), and Trojan (0.85), indicating that the larger input size does not necessarily enhance feature extraction.

Trojan Classification Remains Challenging: Both models have a lower recall for Trojan sam-

ples (0.81 and 0.85, respectively), suggesting that distinguishing Trojans from similar attack patterns remains difficult. However, 32-224-6f maintains a balanced F1-score of 0.81, while 32-384-9f slightly improves it to 0.84.

Exploit and Virus Classification: The 32-224-6f model struggles with recall for Exploit (0.83), whereas 32-384-9f achieves a perfect 0.96 recall, making it more effective at detecting Exploit-based malware. However, for Virus classification, 32-224-6f achieves a higher F1-score (0.92) compared to 0.88 in 32-384-9f, suggesting that the smaller model is better at distinguishing viral patterns from other malware types.

Overall, the 32-224-6f model slightly outperforms the 32-384-9f model in general classification accuracy, recall, and macro-averaged F1-score. The smaller image size seems to strike a better balance between context and granularity, while the larger model suffers from some inconsistencies.

6.3.2.2 Misclassifications and Potential Underlying Causes

The confusion matrix (Figure 31) highlights key misclassification trends. Since our framework classifies malware using stacked Markov, entropy, and intensity-based images of raw byte structures from VAD regions, misclassifications may occur due to the following reasons:

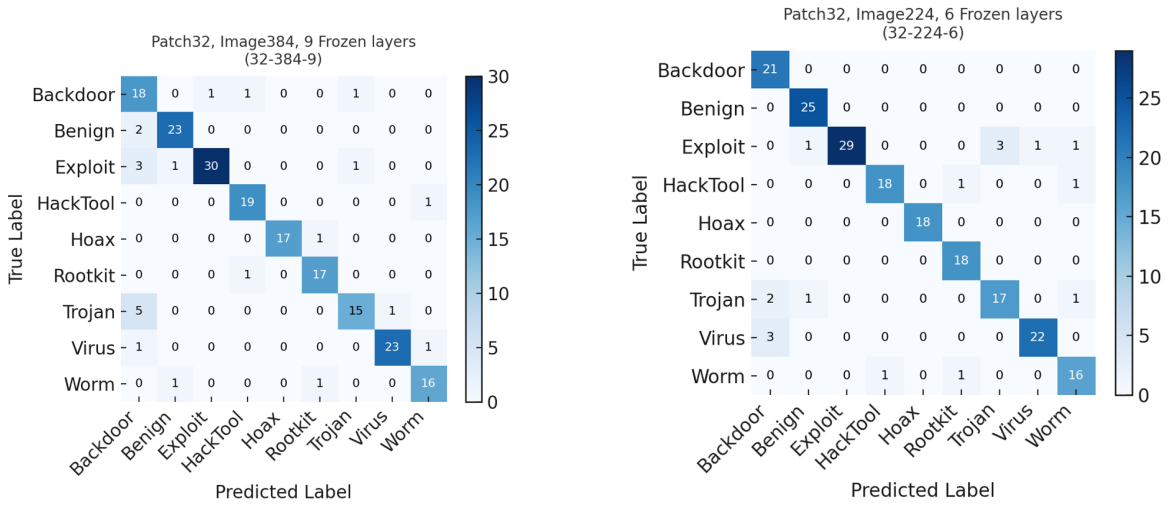
- **Exploit and HackTool Misclassified as Trojan**: In both models, Exploit samples are occasionally misclassified as Trojans (3 samples in each model), and HackTool samples exhibit the same pattern (3 samples across both models). One possible explanation for this misclassification is that Exploits and HackTools often involve code injection, which results in VAD regions marked with ”PAGE_EXECUTE_READWRITE” protections—a characteristic they share with Trojans. This similarity in memory protection attributes, along with potential overlaps in entropy-based patterns, may influence the model’s classification.

  Additionally, given that single memory snapshots might not fully capture the distinguishing traits of Exploits and HackTools, the model may predominantly observe injected code regions that, in terms of memory characteristics, bear a resemblance to those typically seen in Trojans.

- **Impact of Sparse Snapshots** Since some malware families exhibit transient behavior, their memory footprints might only appear in one snapshot. Therefore, single snapshots can

miss exploit or hacktool–specific memory regions, so the model primarily sees short-lived injection footprints that resemble Trojans at the byte and region-flag level.

- **Dataset Limitations and Trojan as a Catch-All Category**: Another factor is that the dataset size may not capture enough variants of each malware type, and the Trojan label often becomes a "catch-all" category for injection-heavy payloads. When multiple families converge on similar memory usage and high-entropy allocations, the classifier focuses on those overlapping VAD features—resulting in mislabeling. Even unique behaviors like privilege escalation or scanning can leave Trojan-like footprints if they rely on shared packers, encryption layers, or injection stubs, especially in a purely VAD-centric approach with limited sample coverage.



(a) Patch32, Image384, 9 Frozen layers (32-384-9)

(b) Patch32, Image224, 6 Frozen layers (32-224-6)

Figure 31: Confusion Matrices of the Multiclass Best-Performing Models

Similar to the binary classification, figure 32 presents the training and validation metrics in all epochs, highlighting the points where gradual layer unfreezing and stochastic weight averaging were applied.

6.4 Attention Visualization

In our work, we implemented attention visualization to enhance the interpretability of the ViT model's decision-making process. These attention values indicate the focus each image patch

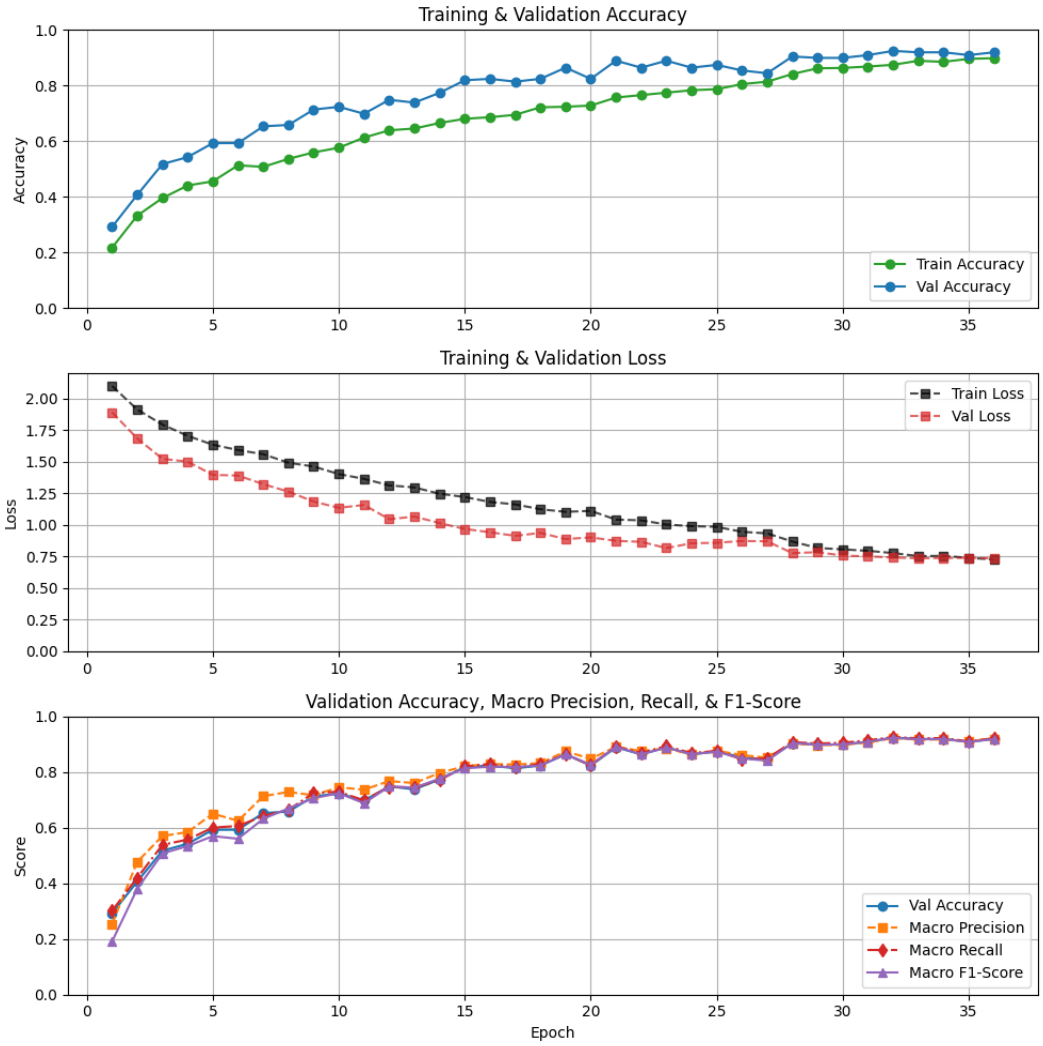
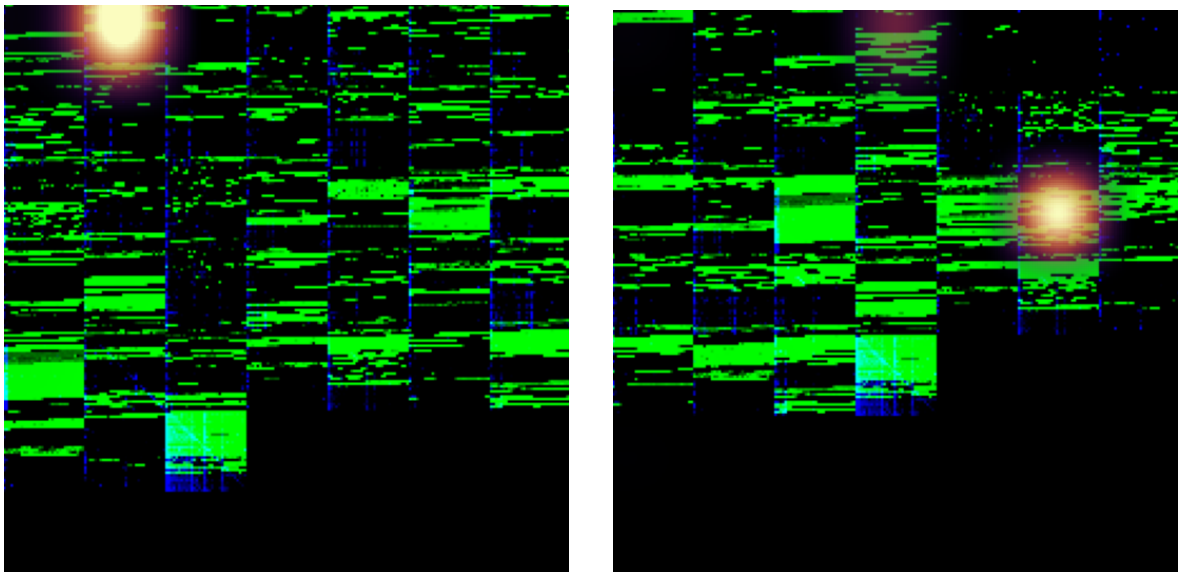


Figure 32: The Training Progress for Multi-Class Classification Across 36 Epochs

receives during classification. Since each patch in the processed image corresponds to a VAD region with a unique address in the memory, we can further enhance forensic analysis by narrowing down the search scope to the most attended regions in the process image. Thus, The VADViT can pinpoint critical memory areas where malicious activity is most likely to reside. This approach is particularly valuable for forensic engineers, allowing them to quickly identify relevant VAD regions and efficiently locate malicious payloads or injected code.

By default, the ViT-Base model does not store attention values, as they are computed dynamically during the forward pass. To address this, we manually integrated attention extraction by registering a forward hook on the final self-attention block. This hook captures the Query-Key-Value projections, computes the scaled dot-product attention, and applies a softmax function to derive



(a) Highly attended region (High attention color intensity)

(b) Moderately attended region (Moderate attention color intensity)

Figure 33: Visualizig the most attended region in final classification result

attention scores (Attention computation is described in section 5).

We overlay the attention maps onto the input images, as illustrated in Figure 33. The image on the left highlights a strongly attended patch, corresponding to a crucial region in the process that likely contains traces of malware behavior valuable for analysis. The image on the right displays a moderately important region that plays a key role in the classification result. Additionally, another less-attended region (fourth patch in the first row) contributes to the output, suggesting another potential area of interest for forensic investigation. During the inference phase, we map these image patches to their actual virtual addresses by running the test file with an ”explain” flag, allowing integration with forensic tools such as Volatility for memory analysis or IDA Pro for disassembly.

To demonstrate the effectiveness of our model in narrowing the forensic investigation scope, we analyzed the most attended region in a case where the ViT-based model identified a process as malicious. This region was extracted from a known HackTool malware sample. Since many of these VAD regions are memory-resident and not file-backed (e.g., injected code or dynamically allocated segments), they often lack the headers required for disassembly by traditional tools like IDA Free or Ghidra. We used the Capstone disassembly framework in Python, which is well-suited for analyzing raw binary data. We dumped the first 160 bytes of the memory region for inspection

and applied an offset of 0x1000 before disassembly. This offset accounts for potential misalignment or non-instructional data at the beginning of the region, helping ensure that decoding begins at a valid instruction boundary. The contents of this highly attended VAD region are shown in Figure 34.

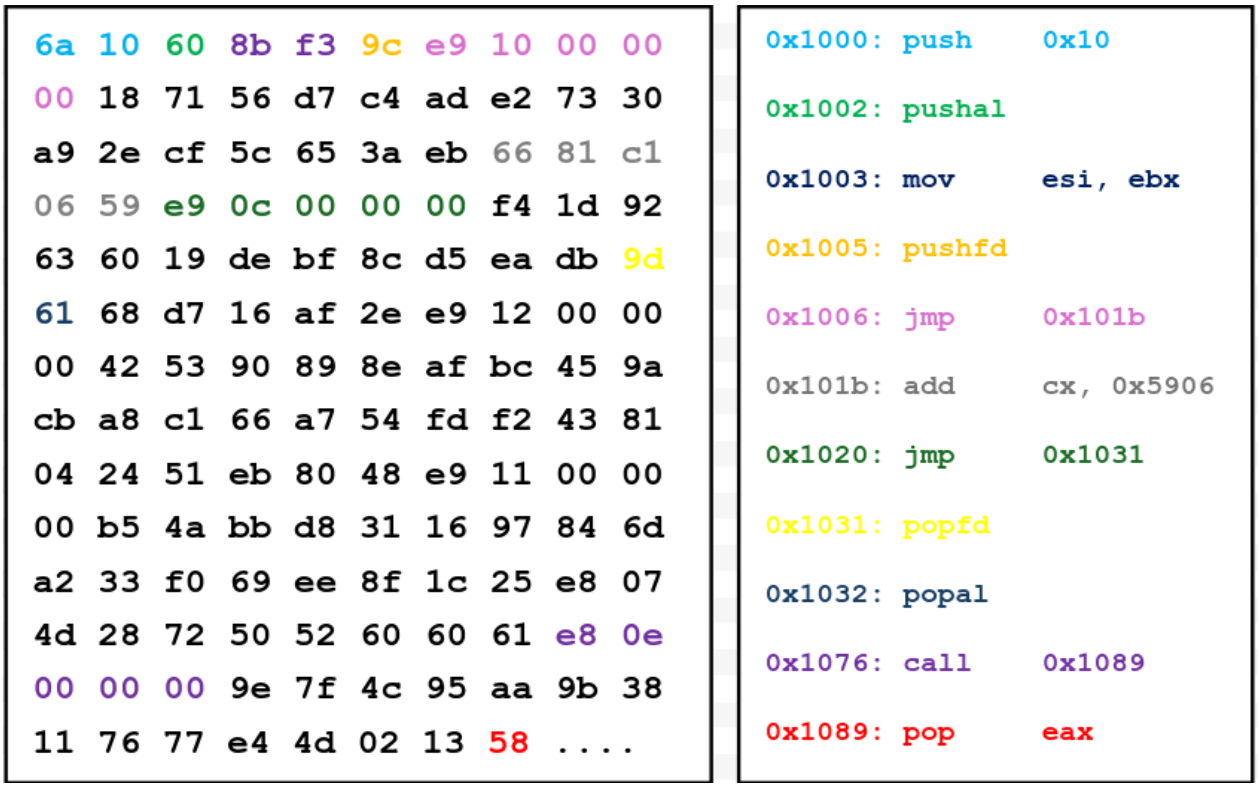


Figure 34: Most Attended VAD Region of a HackTool Malware Sample

The region begins with a recognizable shellcode stub, starting with **pushad**, **pushfd**, and a **jmp** over a block of embedded high-entropy data which is a characteristic of state-saving and control flow redirection techniques often used by malware to evade static detection. This is followed by a **call** and **pop eax**, implementing the widely-used call-pop pattern that enables the shellcode to resolve its own memory base address at runtime, facilitating position-independent access to embedded or decrypted payloads. To maintain clarity in Figure 34, the disassembled view on the right panel is focused on this initial stub. However, several additional raw bytes are clearly visible in the hex dump on the left (e.g., 33 c6 69 ee 8f 1c 25 e8 28 72 50). These correspond to further instructions such as **xor esi, eax**, which are indicative of a decryption in unpacking routines. This analysis validates that the attention mechanism of VADViT can correctly localize memory-resident self-decoding shellcode loaders which highlights the forensic interpretability of

our framework.

7 Conclusion

This thesis introduced VADViT, a vision transformer-based malware detection model that uses VAD analysis to detect malicious processes in memory dumps. We developed BCCC-MalMem-SnapLog-2025, a dataset containing more than 10,000 periodic memory snapshots collected at 30-second intervals per sample, recording PIDs, network activity, and system logs, allowing the analysis of malware execution. VADViT extracts VAD regions of the process, generating Markov images that encode transition probabilities of memory bytes, entropy images that highlight randomness and potential encryption, and custom intensity images representing security attributes such as protection flags and private memory mappings. These images are stacked generating an RGB image representing the process. Experiments were evaluated across two image resolutions (224×224, 384×384) two patch sizes (16×16, 32×32), and three layer freezing strategies. The best-performing configuration utilized a 224×224 image resolution with a 32×32 patch size. Initially, 6 layers were frozen, and they were progressively unfrozen in three steps until the model was fully unfrozen. The transformer enables the model to identify malicious memory regions by localizing the most attended image patches and mapping them back to their corresponding VAD entries. This approach significantly narrows the forensic search space, directing investigators to specific memory regions containing injected code or unauthorized modifications.

8 Future work

Future work can explore treating memory dumps independently rather than aggregating them, using temporal modeling approaches such as time-based graphs or sequential analysis to capture the evolution of process behavior over time. Incorporating alternative backbone models—such as CNNs, ResNets, or Swin Transformers—for initial feature extraction may further enhance robustness and representation quality. Addressing stealth techniques like Direct Kernel Object Manipulation (DKOM) also remains essential; comparing VAD structures with page table entries (PTEs) could help reveal hidden injections, hollow processes, and other evasive behavior [94]. Other promising directions include adaptive unfreezing schedules based on validation loss, expanded datasets, overlapping or content-aware patching, and regularization or data balancing methods such as SMOTE [159] or ADASYN [160].

Bibliography

[1] "Check point research reports highest increase of global cyber attacks seen in last two years – a 30% increase in q2 2024 global cyber attacks," 2024. Check Point Blog, https://blog.checkpoint.com/research/check-point-research-reports-highest-increase-of-global-cyber-atta

[2] D. Gibert, C. Mateu, and J. Planes, "The rise of machine learning for detection and classification of malware: Research developments, trends and challenges," *Journal of Network and Computer Applications*, vol. 153, p. 102526, Mar. 2020.

[3] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh, "A survey on heuristic malware detection techniques," in *The 5th Conference on Information and Knowledge Technology*, IEEE, May 2013.

[4] B. Sanjay, D. Rakshith, R. Akash, and D. V. Hegde, "An approach to detect fileless malware and defend its evasive mechanisms," in *2018 3rd International Conference on Computational Systems and Information Technology for Sustainable Solutions (CSITSS)*, IEEE, Dec. 2018.

[5] M. Botacin, A. Grégio, and M. A. Z. Alves, "Near-memory amp; in-memory detection of fileless malware," in *The International Symposium on Memory Systems*, vol. 1 of *MEMSYS 2020*, p. 23–38, ACM, Sept. 2020.

[6] A. Michalas and R. Murray, "Memtri: A memory forensics triage tool using bayesian network and volatility," in *Proceedings of the 2017 International Workshop on Managing Insider Security Threats*, CCS '17, p. 57–66, ACM, Oct. 2017.

[7] A. Case, R. D. Maggio, M. Firoz-Ul-Amin, M. M. Jalalzai, A. Ali-Gombe, M. Sun, and G. G. Richard, "Hooktracer: Automatic detection and analysis of keystroke loggers using memory forensics," *Computers amp; Security*, vol. 96, p. 101872, Sept. 2020.

[8] Z. Zhang, Z. Liu, and J. Bai, "Network attack detection model based on linux memory forensics," in *2022 14th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA)*, vol. 21, p. 931–935, IEEE, Jan. 2022.

[9] S. Mele Pottaraikkal and A. Sujeer Sugatha, "Effectiveness of multiple memory-images in detecting fileless malware," in *2023 11th International Symposium on Digital Forensics and Security (ISDFS)*, vol. 7, p. 1–5, IEEE, May 2023.

[10] A. Thakar, R. S. Kunwar, H. Thakar, K. Kumar, and C. Patel, "Enhanced malware detection method using baseline comparison in memory forensics," in *2023 IEEE 11th Region 10 Humanitarian Technology Conference (R10-HTC)*, vol. 68, p. 153–156, IEEE, Oct. 2023.

[11] M. M. Alani, A. Mashatan, and A. Miri, "Xmal: A lightweight memory-based explainable obfuscated-malware detector," *Computers amp; Security*, vol. 133, p. 103409, Oct. 2023.

[12] M. Aljabri, F. Alhaidari, A. Albuainain, S. Alrashidi, J. Alansari, W. Alqahtani, and J. Alshaya, "Ransomware detection based on machine learning using memory features," *Egyptian Informatics Journal*, vol. 25, p. 100445, Mar. 2024.

[13] M. Al-Qudah, Z. Ashi, M. Alnabhan, and Q. Abu Al-Haija, "Effective one-class classifier model for memory dump malware detection," *Journal of Sensor and Actuator Networks*, vol. 12, p. 5, Jan. 2023.

[14] A. Arfeen, M. Asim Khan, O. Zafar, and U. Ahsan, "Process based volatile memory forensics for ransomware detection," *Concurrency and Computation: Practice and Experience*, vol. 34, Oct. 2021.

[15] T. Carrier, P. Victor, A. Tekeoglu, and A. Lashkari, "Detecting obfuscated malware using memory feature engineering," in *Proceedings of the 8th International Conference on Information Systems Security and Privacy*, SCITEPRESS - Science and Technology Publications, 2022.

[16] M. Dener, G. Ok, and A. Orman, "Malware detection using memory analysis data in big data environment," *Applied Sciences*, vol. 12, p. 8604, Aug. 2022.

[17] S. M. Rakib Hasan and A. Dhakal, "Obfuscated malware detection: Investigating real-world scenarios through memory analysis," in *2023 IEEE International Conference on Telecommunications and Photonics (ICTP)*, p. 01–05, IEEE, Dec. 2023.

[18] M. A. Hossain and M. S. Islam, "Enhanced detection of obfuscated malware in memory dumps: a machine learning approach for advanced cybersecurity," *Cybersecurity*, vol. 7, Jan. 2024.

[19] M. R. Naeem, M. Khan, A. M. Abdullah, F. Noor, M. I. Khan, M. A. Khan, I. Ullah, and S. Room, "A malware detection scheme via smart memory forensics for windows devices," *Mobile Information Systems*, vol. 2022, p. 1–16, Oct. 2022.

[20] H. Naeem, S. Dong, O. J. Falana, and F. Ullah, "Development of a deep stacked ensemble with process based volatile memory forensics for platform independent malware detection and classification," *Expert Systems with Applications*, vol. 223, p. 119952, Aug. 2023.

[21] S. S. H. Shah, A. R. Ahmad, N. Jamil, and A. u. R. Khan, "Memory forensics-based malware detection using computer vision and machine learning," *Electronics*, vol. 11, p. 2579, Aug. 2022.

[22] S. S. H. Shah, N. Jamil, and A. u. R. Khan, "Memory visualization-based malware detection technique," *Sensors*, vol. 22, p. 7611, Oct. 2022.

[23] A. S. Bozkir, E. Tahillioglu, M. Aydos, and I. Kara, "Catch them alive: A malware detection approach through memory forensics, manifold learning and computer vision," *Computers amp; Security*, vol. 103, p. 102166, Apr. 2021.

[24] Y. Dai, H. Li, Y. Qian, R. Yang, and M. Zheng, "Smash: A malware detection method based on multi-feature ensemble learning," *IEEE Access*, vol. 7, p. 112588–112597, 2019.

[25] J. Liu, Y. Feng, X. Liu, J. Zhao, and Q. Liu, "Mrm-dldet: a memory-resident malware detection framework based on memory forensics and deep neural network," *Cybersecurity*, vol. 6, Aug. 2023.

[26] A.-D. Tran, N.-H. Vo, Q.-K. Tran, H.-D. Nguyen, and M.-T. Tran, "Os-independent malware detection: Applying machine learning and computer vision in memory forensics," in *2021 17th International Conference on Computational Intelligence and Security (CIS)*, p. 616–620, IEEE, Nov. 2021.

[27] O. Khalid, S. Ullah, T. Ahmad, S. Saeed, D. A. Alabbad, M. Aslam, A. Buriro, and R. Ahmad, "An insight into the machine-learning-based fileless malware detection," *Sensors*, vol. 23, p. 612, Jan. 2023.

[28] B. Wu, C. Xu, X. Dai, A. Wan, P. Zhang, Z. Yan, M. Tomizuka, J. Gonzalez, K. Keutzer, and P. Vajda, "Visual transformers: Token-based image representation and processing for computer vision," *arXiv preprint*, vol. arXiv:2006.03677v4, 2020.

[29] S. Zhang, C. Hu, L. Wang, M. Mihaljevic, S. Xu, and T. Lan, "A malware detection approach based on deep learning and memory forensics," *Symmetry*, vol. 15, p. 758, Mar. 2023.

[30] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic malware analysis in the modern era - a state of the art survey," *ACM Computing Surveys*, vol. 52, pp. 1–48, Sept. 2019.

[31] M. Brengel and C. Rossow, "Memscrimper: time- and space-efficient storage of malware sandbox memory dumps," *Lecture Notes in Computer Science*, pp. 24–45, 2018.

[32] I. G. Kiachidis and D. Baltatzis, "Comparative review of malware analysis methodologies," *International Journal of Network Security Amp; Its Applications*, vol. 13, pp. 105–122, 2021.

[33] P. Black, I. Gondal, A. M. Bagirov, and M. Moniruzzaman, "Malware variant identification using incremental clustering," *Electronics*, vol. 10, p. 1628, 2021.

[34] S. Silva, S. Lima, R. Pinheiro, L. Abreu, R. Lima, and S. Fernandes, "Antivirus solution to iot malware detection with authorial next-generation sandbox," 2023.

[35] F. Wang, L. Hu, J. Hu, and K. Zhao, "Computer forensic analysis model for the reconstruction of chain of evidence of volatile memory data," *Multimedia Tools and Applications*, vol. 75, p. 10097–10107, July 2015.

[36] T. Rupprecht, X. Chen, D. H. White, J. T. Mühlberg, H. Bos, and G. Lüttgen, "Poster: Identifying dynamic data structures in malware," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS'16, p. 1772–1774, ACM, Oct. 2016.

[37] A. Case and G. G. Richard, "Memory forensics: The path forward," *Digital Investigation*, vol. 20, p. 23–33, Mar. 2017.

[38] D. Kim and T. Shon, "Future of kernel object-based memory forensics," in *2023 International Conference on Platform Technology and Service (PlatCon)*, vol. 11, p. 64–66, IEEE, Aug. 2023.

[39] D. Abu Sheihka, "Windows memory forensics," tech. rep., Arab American University, 2023.

[40] R. M. Stevens and E. Casey, "Extracting windows command line details from physical memory," *Digital Investigation*, vol. 7, p. S57–S63, Aug. 2010.

[41] J. Okolica and G. L. Peterson, "Extracting the windows clipboard from physical memory," *Digital Investigation*, vol. 8, p. S118–S124, Aug. 2011.

[42] X. Fu, X. Du, and B. Luo, "Data correlation-based analysis methods for automatic memory forensic," *Security and Communication Networks*, vol. 8, p. 4213–4226, Sept. 2015.

[43] "Yara." `https://yara.readthedocs.io/en/latest/`.

[44] "Autopsy, sleuthkit." `https://www.sleuthkit.org/autopsy/`.

[45] "Belkasoft evidence center 2018," 2018. `https://belkasoft.com/ec`.

[46] A. Schuster, "Searching for processes and threads in microsoft windows memory dumps," *Digital Investigation*, vol. 3, p. 10–16, Sept. 2006.

[47] A. Schuster, "Pool allocations as an information source in windows memory forensics," in *IT-Incident Management IT-Forensics - IMF 2006*, pp. 104–115, Bonn: Gesellschaft für Informatik e. V., 2006.

[48] R. Thantilage and N. Jeyamohan, "A volatile memory analysis tool for retrieval of social media evidence in windows 10 os based workstations," in *2017 National Information Technology Conference (NITC)*, p. 86–88, IEEE, Sept. 2017.

[49] M. Cohen, "Scanning memory with yara," *Digital Investigation*, vol. 20, p. 34–43, Mar. 2017.

[50] A. Case, M. M. Jalalzai, M. Firoz-Ul-Amin, R. D. Maggio, A. Ali-Gombe, M. Sun, and G. G. Richard, "Hooktracer: A system for automated and accessible api hooks analysis," *Digital Investigation*, vol. 29, p. S104–S112, July 2019.

[51] V. Distler, C. Lallemand, and V. Koenig, "Making encryption feel secure: Investigating how descriptions of encryption impact perceived security," in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroSamp;PW)*, vol. 41, p. 220–229, IEEE, Sept. 2020.

[52] V. Khandelwal, A. K. Chaturvedi, and C. P. Gupta, "Bidding strategies for amazon ec2 spot instances-a comprehensive review," in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, p. 1–5, IEEE, Aug. 2018.

[53] L. Liebler and F. Breitinger, "mrsh-mem: Approximate matching on raw memory dumps," in *2018 11th International Conference on IT Security Incident Management amp; IT Forensics (IMF)*, p. 47–64, IEEE, May 2018.

[54] "Memparser: A tool for memory forensics," 2005. DFRWS 2005 Memory Forensics Challenge, `https://sourceforge.net/projects/memparser/`.

[55] "Dfrws 2005 forensics challenge," 2005. GitHub Repository, `https://github.com/dfrws/dfrws2005-challenge`.

[56] "Kntlist analysis tool," 2005. DFRWS 2005 Forensics Challenge, `https://github.com/dfrws/dfrws2005-challenge/blob/main/submissions/kntlist-analysis-tool.md`.

[57] F. Wang, L. Hu, J. Hu, and K. Zhao, "Computer forensic analysis model for the reconstruction of chain of evidence of volatile memory data," *Multimedia Tools and Applications*, vol. 75, p. 10097–10107, July 2015.

[58] P. Sun, R. Han, M. Zhang, and S. Zonouz, "Trace-free memory data structure forensics via past inference and future speculations," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, vol. 5 of *ACSAC '16*, p. 570–582, ACM, Dec. 2016.

[59] T.-A. Tran-Quoc, C. Huynh-Minh, and A.-D. Tran, "Towards os-independent memory images analyzing: Using paging structures in memory forensics," in *2021 14th International Conference on Security of Information and Networks (SIN)*, vol. 1, p. 1–8, IEEE, Dec. 2021.

[60] J. Zeng and Z. Lin, "Towards automatic inference of kernel object semantics from binary code," in *International Symposium on Recent Advances in Intrusion Detection*, pp. 538–561, Springer, 2015.

[61] "The volatility framework: Advanced memory forensics framework," 2014. `https://www.volatilityfoundation.org/`.

[62] "Volatility 3 documentation." https://volatility3.readthedocs.io/en/stable/.

[63] "Rekall memory forensic framework," 2014. Google, http://www.rekall-forensic.com/.

[64] J. Stadlinger, A. Dewald, and F. Block, "Linux memory forensics: Expanding rekall for userland investigation," in *2018 11th International Conference on IT Security Incident Management amp; IT Forensics (IMF)*, vol. 7, p. 27–46, IEEE, May 2018.

[65] "Redline," 2012. FireEye, https://www.fireeye.com/services/freeware/redline.html.

[66] "Memoryze," 2012. FireEye, https://www.fireeye.com/services/freeware/memoryze.html.

[67] M. I. Cohen, "Characterization of the windows kernel version variability for accurate memory analysis," *Digital Investigation*, vol. 12, p. S38–S49, Mar. 2015.

[68] F. Block and A. Dewald, "Linux memory forensics: Dissecting the user space process heap," *Digital Investigation*, vol. 22, p. S66–S75, Aug. 2017.

[69] W. Song, H. Yin, C. Liu, and D. Song, "Deepmem: Learning graph neural network models for fast and robust memory forensic analysis," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, p. 606–618, ACM, Oct. 2018.

[70] Q. Zhang, Y. Wu, and C. Cui, "A new method of live tracking of process memory," in *Proceedings of the 2nd International Conference on Cryptography, Security and Privacy*, vol. 2015 of *ICCSP 2018*, p. 154–158, ACM, Mar. 2018.

[71] R. D. Maggio, A. Case, A. Ali-Gombe, and G. G. Richard, "Seance: Divination of tool-breaking changes in forensically important binaries," *Forensic Science International: Digital Investigation*, vol. 37, p. 301189, July 2021.

[72] J. A. Duke, *Memory forensics comparison of apple m1 and intel architecture using volatility framework*. Louisiana State University and Agricultural & Mechanical College, 2021.

[73] Z. Qi, Y. Qu, and H. Yin, "Logicmem: Automatic profile generation for binary-only memory forensics via logic inference," in *Proceedings 2022 Network and Distributed System Security Symposium*, NDSS 2022, Internet Society, 2022.

[74] P. Fernández-Álvarez and R. J. Rodr´iguez, "Module extraction and dll hijacking detection via single or multiple memory dumps," *Forensic Science International: Digital Investigation*, vol. 44, p. 301505, Mar. 2023.

[75] M. Daghmehchi Firoozjaei, A. Habibi Lashkari, and A. A. Ghorbani, "Memory forensics tools: a comparative analysis," *Journal of Cyber Security Technology*, vol. 6, pp. 149–173, July 2022.

[76] Mital Parekh and Snehal Jani, "Memory forensic: Acquisition and analysis of memory and its tools comparison," *International Journal of Engineering Technologiesand Management Research*, 2018.

[77] J. Ottmann, F. Breitinger, and F. Freiling, "An experimental assessment of inconsistencies in memory forensics," *ACM Transactions on Privacy and Security*, vol. 27, p. 1–29, Dec. 2023.

[78] A. Case and G. G. Richard, "Detecting objective-c malware through memory forensics," *Digital Investigation*, vol. 18, p. S3–S10, Aug. 2016.

[79] N. Nissim, O. Lahav, A. Cohen, Y. Elovici, and L. Rokach, "Volatile memory analysis using the minhash method for efficient and secured detection of malware in private cloud," *Computers amp; Security*, vol. 87, p. 101590, Nov. 2019.

[80] M. Ajay Kumara and C. Jaidhar, "Leveraging virtual machine introspection with memory forensics to detect and characterize unknown malware using machine learning techniques at hypervisor," *Digital Investigation*, vol. 23, p. 99–123, Dec. 2017.

[81] R. Sihwail, K. Omar, and K. Akram Zainol Ariffin, "An effective memory analysis for malware detection and classification," *Computers, Materials amp; Continua*, vol. 67, no. 2, p. 2301–2320, 2021.

[82] R. Mosli, R. Li, B. Yuan, and Y. Pan, "Automated malware detection using artifacts in forensic memory images," in *2016 IEEE Symposium on Technologies for Homeland Security (HST)*, IEEE, May 2016.

[83] P. Maniriho, A. N. Mahmood, and M. J. M. Chowdhury, "Memaldet: A memory analysis-based malware detection framework using deep autoencoders and stacked ensemble under temporal evaluations," *Computers amp; Security*, vol. 142, p. 103864, July 2024.

[84] A. H. Lashkari, B. Li, T. L. Carrier, and G. Kaur, "Volmemlyzer: Volatile memory analyzer for malware classification using feature engineering," in *2021 Reconciling Data Analytics, Automation, Privacy, and Security: A Big Data Challenge (RDAAPS)*, IEEE, May 2021.

[85] T. Panker and N. Nissim, "Leveraging malicious behavior traces from volatile memory using machine learning methods for trusted unknown malware detection in linux cloud environments," *Knowledge-Based Systems*, vol. 226, p. 107095, Aug. 2021.

[86] A. Mezina and R. Burget, "Obfuscated malware detection using dilated convolutional network," in *2022 14th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, IEEE, Oct. 2022.

[87] S. Sazzed and S. Ullah, "Enhancing efficiency and privacy in memory-based malware classification through feature selection," in *2023 International Conference on Machine Learning and Applications (ICMLA)*, p. 550–557, IEEE, Dec. 2023.

[88] R. Sihwail, K. Omar, K. Zainol Ariffin, and S. Al Afghani, "Malware detection approach based on artifacts in memory image and dynamic analysis," *Applied Sciences*, vol. 9, p. 3680, Sept. 2019.

[89] I. Sharafaldin, A. Gharib, A. H. Lashkari, and A. A. Ghorbani, "Botviz: A memory forensic-based botnet detection and visualization approach," in *2017 International Carnahan Conference on Security Technology (ICCST)*, vol. 3, p. 1–8, IEEE, Oct. 2017.

[90] V. E. Y. H. . L. Z. Prakash, A., "On the trustworthiness of memory analysis—an empirical study from the perspective of binary execution," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 5, pp. 557–570, 2015.

[91] S. S. K. Y. . L. C. Jung, S., "Memory layout extraction and verification method for reliable physical memory acquisition," *Electronics*, vol. 10, no. 12, p. 1380, 2021.

[92] A. Nugraha and J. Zeniarja, "Malware detection using decision tree algorithm based on memory features engineering," *Journal of Applied Intelligent System*, vol. 7, no. 3, pp. 206–210, 2022.

[93] H. Nyholm, K. Monteith, S. Lyles, M. Gallegos, M. DeSantis, J. Donaldson, and C. Taylor, "The evolution of volatile memory forensics," *Journal of Cybersecurity and Privacy*, vol. 2, pp. 556–572, July 2022.

[94] M. H. Ligh, A. Case, J. Levy, and A. Walters, *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley, 2014.

[95] A. Socala and M. Cohen, "Automatic profile generation for live linux memory analysis," *Digital Investigation*, vol. 16, pp. S11–S24, 2016.

[96] L. A. . W. N. Wibowo, D., "Investigation of fake insider threats on private cloud computing services," *International Journal of Science Technology Management*, vol. 3, no. 5, pp. 1484–1491, 2022.

[97] W. Neweva, "Forensic analysis of live ransomware attacks on linux-based laptop systems: techniques and evaluation," 2024.

[98] "Malmem2022: Malware memory forensics dataset," 2022. Canadian Institute for Cybersecurity, University of New Brunswick, `https://www.unb.ca/cic/datasets/malmem-2022.html`.

[99] S. Lyles, M. Desantis, J. Donaldson, M. Gallegos, H. Nyholm, C. Taylor, and K. Monteith, "Machine learning analysis of memory images for process characterization and malware detection," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, vol. 8, p. 162–169, IEEE, June 2022.

[100] Y. Duan, X. Fu, B. Luo, Z. Wang, J. Shi, and X. Du, "Detective: Automatically identify and analyze malware processes in forensic scenarios via dlls," in *2015 IEEE International Conference on Communications (ICC)*, p. 5691–5696, IEEE, June 2015.

[101] I. Sadek, P. Chong, S. U. Rehman, Y. Elovici, and A. Binder, "Memory snapshot dataset of a compromised host with malware using obfuscation evasion techniques," *Data in Brief*, vol. 26, p. 104437, Oct. 2019.

[102] J. Palsa, N. Ádám, J. Hurtuk, E. Chovancová, B. Madovs, M. Chovanec, and S. Kocan, "Mlmd—a malware-detecting antivirus tool based on the xgboost machine learning algorithm," *Applied Sciences*, vol. 12, p. 6672, 2022.

[103] L. Jiang, Y. Zhang, and Y. Shi, "Visual fileless malware classification via few-shot learning," *International Conference on Cyber Security, Artificial Intelligence, and Digital Economy (CSAIDE 2023)*, 2023.

[104] Y. Oyama, "Trends of anti-analysis operations of malwares observed in api call logs," *Journal of Computer Virology and Hacking Techniques*, vol. 14, pp. 69–85, 2017.

[105] E. Debas, N. Alhumam, and K. Riad, "Unveiling the dynamic landscape of malware sandboxing: a comprehensive review," *International Journal of Advanced Computer Science and Applications*, vol. 15, 2024.

[106] QEMU Project Developers, *About QEMU*, 2025. Accessed: 2025-01-23.

[107] Virtiofs Project Developers, *Virtiofs: Shared File System for Virtual Machines*, 2025. Accessed: 2025-01-23.

[108] M. M. Shafi, A. H. Lashkari, V. Rodriguez, and R. Nevo, "Toward generating a new cloud-based distributed denial of service (ddos) dataset and cloud intrusion traffic characterization," *Information*, vol. 15, p. 195, 2024.

[109] W. Elbakri, "Adaptive cloud intrusion detection system based on pruned exact linear time technique," *Computers Materials & Continua*, vol. 79, no. 3, pp. 3725–3756, 2024.

[110] K. Moore, T. Bihl, K. Bauer, and T. Dube, "Feature extraction and feature selection for classifying cyber traffic threats," *The Journal of Defense Modeling and Simulation Applications Methodology Technology*, vol. 14, no. 3, pp. 217–231, 2016.

[111] R. Davis, "Classifying malware traffic using images and deep convolutional neural network," *IEEE Access*, vol. 12, pp. 58031–58038, 2024.

[112] I. Sembiring, . Suharyadi, A. Iriani, J. Ginting, and J. Ginting, "A novel approach to network forensic analysis: combining packet capture data and social network analysis," *International Journal of Advanced Computer Science and Applications*, vol. 14, no. 3, 2023.

[113] X. Song, "An intelligent nic design," in *Proceedings of the 2nd International Conference on Advances in Mechanical Engineering and Industrial Informatics (AMEII 2016)*, Advances in Engineering Research, pp. 1149–1154, Atlantis Press, 2016.

[114] J. Hwang, C. Hong, and H. Suh, "Dynamic inbound rate adjustment scheme for virtualized cloud data centers," *IEICE Transactions on Information and Systems*, vol. E99.D, no. 3, pp. 760–762, 2016.

[115] J. Vallejo, D. Feferman, and C. Rothenberg, "Network address translation using a programmable dataplane processor," in *Proceedings of the Workshop on Performance Evaluation of Computer and Telecommunication Systems (WPerformance)*, 2018.

[116] M. Savi, A. Banfi, A. Tundo, and M. Ciavotta, "Serverless computing for nfv: Is it worth it? a performance comparison analysis," in *2022 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, IEEE, 2022.

[117] R. Mahmoud, "Redefining malware sandboxing: enhancing analysis through sysmon and elk integration," *IEEE Access*, vol. 12, pp. 68624–68636, 2024.

[118] R. Portase, "Leda—layered event-based malware detection architecture," *Sensors*, vol. 24, no. 19, p. 6393, 2024.

[119] M. Assi, A. Fahad, and B. Al-Sarray, "Root cause analysis and improvement in windows system based on windows performance toolkit wpt," *Iraqi Journal of Science*, pp. 5046–5057, 2022.

[120] D. Bhanage, "Failure detection using semantic analysis and attention-based classifier model for it infrastructure log data," *IEEE Access*, vol. 11, pp. 108178–108197, 2023.

[121] T. Glass-Vanderlan, M. Iannacone, M. Vincent, and R. Bridges, "A survey of intrusion detection systems leveraging host data," 2018.

[122] F. Rivera-Ortiz and L. Pasquale, "Automated modelling of security incidents to represent logging requirements in software systems," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–8, 2020.

[123] F. Rivera-Ortiz and L. Pasquale, "Towards automated logging for forensic-ready software systems," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 157–163, 2019.

[124] K. Bezas and F. Filippidou, "Comparative analysis of open source security information & event management systems (siems)," *Indonesian Journal of Computer Science*, vol. 12, no. 2, pp. 443–468, 2023.

[125] M. Shibani, "Automated threat hunting using elk stack - a case study," *Indian Journal of Computer Science and Engineering*, vol. 10, no. 5, pp. 118–127, 2019.

[126] C. Smiliotopoulos, G. Kambourakis, and K. Barmpatsalou, "On the detection of lateral movement through supervised machine learning and an open-source tool to create turnkey datasets from sysmon logs," 2023.

[127] A. Pérez-Sánchez and R. Hielscher, "Evaluation of local security event management system vs. standard antivirus software," *Applied Sciences*, vol. 12, p. 1076, 2022.

[128] K. Chatterjee and L. Doyen, "Stochastic processes with expected stopping time," *Logical Methods in Computer Science*, vol. 17, no. 2, pp. 1–27, 2021.

[129] H. Costin and O. Geman, "Parkinson's disease prediction based on multistate markov models," *International Journal of Computers Communications & Control*, vol. 8, no. 4, pp. 525–537, 2013.

[130] B. Yuan, J. Wang, D. Liu, W. Guo, P. Wu, and X. Bao, "Byte-level malware classification based on markov images and deep learning," *Computers & Security*, vol. 92, p. 101740, 2020.

[131] F. Al-Anziand and D. AbuZeina, "A survey of markov chain models in linguistics applications," in *Proceedings of the 6th International Conference on Computer Science and Information Technology (CSIT)*, 2016.

[132] Y. Mao, "Ergodicity for the $gi/g/1$-type markov chain," 2012.

[133] S. Salahuddin, E. Porter, P. Meaney, and M. O'Halloran, "Effect of logarithmic and linear frequency scales on parametric modelling of tissue dielectric data," *Biomedical Physics & Engineering Express*, vol. 3, no. 1, p. 015020, 2017.

[134] M. Nasir, R. Yusof, A. Rehman, and A. Hussain, "Global contrast stretching for enhancement of microscopic malaria images," *2012 International Conference on Biomedical Engineering (ICoBE)*, pp. 357–361, 2012.

[135] A. Desiani, "Contrast stretching and adaptive thresholding for retinal blood vessel segmentation," *Journal of Physics: Conference Series*, vol. 2193, no. 1, p. 012003, 2022.

[136] K. Toh and N. Isa, "Enhancement of high dynamic range images using contrast stretching," in *2016 IEEE International Conference on Signal and Image Processing Applications (ICSIPA)*, pp. 443–447, 2016.

[137] M. Akkala and Y. Kumar, "Enhancement of ultrasound images using contrast stretching techniques," in *2014 International Conference on Advances in Electronics Computers and Communications*, pp. 1–4, 2014.

[138] S. B. Sa¨id, N. L. Bihan, and J. H. Manton, "Hidden markov chains and fields with observations in riemannian manifolds," 2021.

[139] Z. Tang, J. Wang, B. Yuan, H. Li, J. Zhang, and H. Wang, "Markov-gan: markov image enhancement method for malicious encrypted traffic classification," *IET Information Security*, vol. 16, pp. 442–458, 2022.

[140] M. Catak, "Application of markov chains on image enhancement," *Neural Computing and Applications*, vol. 25, pp. 1119–1123, 2014.

[141] A. Mantovani, S. Aonzo, X. Ugarte-Pedrero, A. Merlo, and D. Balzarotti, "Prevalence and impact of low-entropy packing schemes in the malware ecosystem," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2020.

[142] E. Raff, C. Nicholas, and M. McLean, "A new burrows wheeler transform markov distance," in *Proceedings of the ArXiv Conference*, 2019.

[143] F. Biondi, M. Enescu, T. Given-Wilson, A. Legay, L. Noureddine, and V. Verma, "Effective, efficient, and robust packing detection and classification," *Computers & Security*, vol. 85, pp. 436–451, 2019.

[144] H. Menéndez and J. Llorente, "Mimicking anti-viruses with machine learning and entropy profiles," *Entropy*, vol. 21, no. 5, p. 513, 2019.

[145] R. OMACHI and Y. Murakami, "Packer identification method for multi-layer executables using entropy analysis with k-nearest neighbor algorithm," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E106.A, no. 3, pp. 355–357, 2023.

[146] S. Shokr and H. Bahig, "A fast multicore-based window entropy algorithm," *International Journal of Advanced Computer Science and Applications*, vol. 13, no. 11, 2022.

[147] D. Keyes, B. Li, G. Kaur, A. Lashkari, F. Gagnon, and F. Massicotte, "Entroplyzer: Android malware classification and characterization using entropy analysis of dynamic characteristics," in *2021 IEEE International Conference on Research in Adaptive and Convergent Systems (RDAAPS)*, pp. 1–12, IEEE, 2021.

[148] H. Lee, S. Kim, D. Baek, D. Kim, and D. Hwang, "Robust iot malware detection and classification using opcode category features on machine learning," *IEEE Access*, vol. 11, pp. 18855–18867, 2023.

[149] M. Conti, S. Khandhar, and P. Vinod, "A few-shot malware classification approach for unknown family recognition using malware feature visualization," *Computers & Security*, vol. 122, p. 102887, 2022.

[150] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *Proceedings of the International Conference on Learning Representations (ICLR)*, OpenReview, 2021.

[151] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Lukasz Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS)*, pp. 6000–6010, Curran Associates, Inc., 2017.

[152] VirusTotal, "VirusTotal - Free Online Virus, Malware and URL Scanner," 2025. Accessed: February 17, 2025.

[153] CNET$_{D}ownloads$, "$CNETDownloads-FreeSoftware, Apps, andGames,"$ 2025. $Accessed$: $February 18, 2025$.

[154] S. Yun, D. Han, S. J. Oh, S. Chun, J. Choe, and Y. Yoo, "Cutmix: Regularization strategy to train strong classifiers with localizable features," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pp. 6023–6032, 2019.

[155] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, 2009.

[156] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou, "Training data-efficient image transformers & distillation through attention," *arXiv preprint*, vol. arXiv:2012.12877v2, 2021.

[157] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," *arXiv preprint*, vol. arXiv:2103.14030v2, 2021.

[158] Y. Wang, D. Sun, K. Chen, F. Lai, and M. Chowdhury, "Egeria: Efficient dnn training with knowledge-guided layer freezing," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, ACM, 2023.

[159] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[160] H. He, Y. Bai, E. A. Garcia, and S. Li, "ADASYN: Adaptive synthetic sampling approach for imbalanced learning," in *Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN)*, pp. 1322–1328, 2008.

Vita

Candidate's full name: Yasin Dehfouli

University attended (with dates and degrees obtained):

- Amirkabir University of Technology, Bachelor of Applied Science in Electrical Engineering (Control Systems), 2017 - 2022

Publications:

- Dehfouli, Y., & Lashkari, A. H. Memory Analysis for Malware Detection: A Comprehensive Survey Using the OSCAR Methodology. Submitted to ACM Computing Surveys on September 16, 2024. Received a major revision on March 20, 2025.
- Dehfouli, Y., & Lashkari, A. H. VADViT: A Novel Vision Transformer-Driven Malware Detection Approach for Malicious Process Detection and Explainable Threat Attribution Using Virtual Address Descriptor Regions. Submitted to Computers & Security on April 2nd, 2025. Manuscript under review.