Malware Detection Through Memory Forensics and Windows Event Log Analysis

Dinesh Patil
Department of Computer Engineering, Vidyavardhini's College of Engineering and Technology, India
dinesh9371@gmail.com

Akshaya Prabhu
Department of Artificial Intelligence and Machine Learning
Dwarkadas Jivanlal Sanghvi College of Engineering, India
akshaya.prabhu@djsce.ac.in

Abstract: *With the increasing reliance of human society on computer systems in daily life, cybercrime is also on the rise. Malware is increasingly used by cybercriminals to attack, compromise, and steal sensitive information, and more critically, to demand ransom from users of infected systems. Existing antivirus solutions often fall short in detecting and alerting users to attacks carried out by newly developed or evolving malware strains. This highlights the need for a more robust and proactive strategy for malware detection. This paper presents a hybrid approach for advanced malware detection, integrating the identification of suspicious code executing in main memory with the analysis of malware-related events in Windows Event Logs. Experiments were conducted using a code injection technique on Windows 7 and Windows 10 systems, and the corresponding memory images and Event Logs were analyzed to validate the effectiveness of the proposed approach. Training and testing were performed on both code-based and event-based datasets to evaluate detection accuracy. For the detection of suspicious code, we employed the Canadian Institute for Cybersecurity-Malware in Memory 2023 (CIC-MalMem 2023) dataset. For event-based analysis, we utilized the EVTX-ATTACK-SAMPLES and the Windows Event Log dataset. Experimental results using the Random Forest (RF)classifier demonstrate a detection accuracy of 99% based on suspicious code and 95% based on Event Log data.*

Keywords: *Malware, windows event logs, code section, main memory analysis, main memory structures, VAD, volatile memory.*

Received March 10, 2025; accepted July 18, 2025
https://doi.org/10.34028/iajit/22/6/5

1. Introduction

Malicious actors develop and deploy software designed to compromise computers and mobile devices in order to steal sensitive information. The first computer-based malware attack occurred in 1986 with a virus known as 'Brain', which infected floppy disks. The first reported ransomware attack was in 1989, involving the AIDS Trojan, which was also distributed via floppy disk. Victims were required to pay a ransom to regain access to their systems [6]. According to recent reports, malware attacks particularly ransomware have surged by over 105% [7], leading to significant financial losses and data breaches.

The modus operandi of malware often varies between developers. Typically, malware infiltrates a system through phishing emails or watering hole attacks, deceiving users into downloading and executing malicious code. Once loaded into main memory, the malware is scheduled for execution by the Operating System (OS). Each type of malware employs a distinct infection and execution strategy. For example, ransomware begins encrypting the system's hard disk once granted execution privileges. Cybercriminals commonly use encryption algorithms such as Rivest-Shamir-Adleman (RSA) and Advanced Encryption Standard (AES) for this purpose.

A typical ransomware infection follows these steps:

- *Collecting system information*: the malware gathers details about the host system such as computer name, OS, location, and whether it is running in a virtual environment. If a virtual environment is detected, the ransomware may terminate to avoid analysis.
- *Encryption keys*: the malware retrieves encryption keys from a remote server, depending on its implementation.
- *Encryption*: files on the hard disk are encrypted, often renamed or given new extensions. The encryption commonly uses RSA, AES, or a hybrid of both.
- *File deletion*: original files are deleted post-encryption.
- *Network scanning*: the malware scans for system vulnerabilities and login credentials.
- *Ransom message preparation*: the ransomware displays a message often as a README file, altered wallpaper, or popup informing the user of the encryption and demanding payment. These messages typically mention the encryption algorithm used and payment instructions.

Ransomware requires appropriate OS permissions to access and modify files. While Windows allows this to a greater extent, macintosh Operating System (macOS) is comparatively more restrictive, which is why most ransomware incidents target Windows systems.

Malware is now also developed for mobile and handheld devices. Developers continuously innovate new techniques to evade detection by antivirus software.

Signature-based detection mechanisms are only effective if the malware signature has already been recorded in the antivirus database. Newly created malware strains often go undetected by traditional antivirus solutions. Previous studies [9, 13, 28] have shown that malware execution triggers identifiable system events in Windows. For instance, [28] presents a comprehensive list of events linked to malware execution. Detection of unknown code running in main memory, when correlated with suspicious or malicious events recorded in system logs, can strongly indicate the presence of malware. Prior studies [20, 22, 26] have predominantly focused on main memory analysis using a range of techniques. However, they often lack depth in locating and analyzing executable malware code within memory an essential step for identifying newly developed threats. It is evident that execution of malwares triggers distinct system events logged by Windows. This underscores the need to analyze both executable code and the resulting system events for accurate detection.

While previous researches [14, 16] have focused on Windows Event Log analysis to trace malware activity, these efforts often overlook the identification of unknown or suspicious code present in main memory. Our research addresses this gap by locating the executable code in memory and detecting its maliciousness through code comparison and correlated system events.

Given the increasing sophistication of malware and the limitations of traditional signature-based antivirus tools in detecting newly developed or unknown threats, there is a pressing need for more effective detection mechanisms. Previous research has largely focused on main memory or Event Log analysis in isolation, with limited success in identifying novel malware strains. However, when malware executes, it leaves identifiable traces in both memory and system logs. Therefore, we hypothesize that an integrated approach combining memory code analysis with Windows Event Log correlation significantly improves the detection of unknown or new malware.

The primary objective of this research work is to develop a hybrid malware detection approach that integrates main memory analysis with Windows Event Log correlation to identify sophisticated malware attacks. This work aims to enhance detection accuracy by combining main memory inspection techniques with behavioral indicators derived from system logs.

This paper makes the following key contributions:

a) Establishes the relationship between various memory structures to locate the code section of executing programs.
b) Proposes a software architecture for malware detection based on memory code and associated malicious events.
c) Demonstrates improved malware detection using a hybrid approach on the Canadian Institute for Cybersecurity Malware in Memory dataset (CIC-MalMem-2022) dataset [31], EVTX-ATTACK-SAMPLES for malicious events [24], and Windows Event Log dataset [15] for non-malicious events.

This research presents a hybrid approach to detect and alert users about malware presence on Windows-based systems. The proposed method involves two key components:

1. Main memory analysis to locate and evaluate the code of running executables.
2. Event Log analysis to identify correlated malicious system behavior.

The structure of the paper is as follows: Section 1 introduces the threats posed by malware and their societal and financial impacts. Section 2 surveys related work on malware detection based on main memory analysis and Windows Event Logs. Section 3 details the proposed malware detection approach, along with the legal implication of live main memory analysis. Experiments and evaluations conducted, including training and testing, system resource utilization and performance overhead are detailed in section 4. Section 5 presents the conclusions and future work to be carried out.

2. Related Work

This section discusses about the work carried out to detect the malware.

Ahlegren [1] highlights the differences between host-based and network-level ransomware detection, showing that local monitoring of processes and memory often provides faster detection compared to network traffic inspection.

General overviews such as Baker [5] classify malware detection techniques into static, dynamic, and hybrid approaches, providing a foundation for understanding the limitations that motivate more advanced memory and event-based analysis.

Damodaran *et al.* [10] conducted a comparative study of malware detection techniques based on static, dynamic, and hybrid analysis. Their approach involved training Hidden Markov Models (HMM) using both static and dynamic features, primarily Application Programming Interface (API) call sequences and opcode patterns across various malware families. While combining these features showed improved detection capability, the narrow feature scope may fail to capture the full behavioural spectrum of modern malware, thereby limiting generalizability. The dataset used in this study was constructed by combining samples from various malware families along with benign programs. The final dataset comprises a total of 785 samples, including 745 malware instances and 40 benign samples. Evaluation was primarily based on the Area Under the Receiver Operating Characteristic (ROC)

Curve (AUC); however, the inclusion of metrics such as precision, recall, and F1-score would have provided a more comprehensive assessment, particularly for imbalanced datasets.

Moskovitch *et al*. [19] adopted a static analysis approach that utilizes opcode n-grams extracted from disassembled binaries for malware classification. The method assumes consistent opcode patterns across malware families; however, polymorphic and metamorphic variants often generate diverse opcode sequences even within the same family, leading to potential detection failures. The dataset comprises over 30,000 files sourced from Virus Exchange (VX) Heaven and benign campus systems but lacks evaluation against modern, adaptive malware, including Advanced Persistent Threat (APT)-level threats. Furthermore, the study does not incorporate dynamic behaviour analysis such as system calls or runtime behaviour's, which are essential for detecting evasive malware. While some feature selection was applied, issues of computational efficiency and real-time scalability remain unaddressed.

Ucci *et al*. [30] provided a comprehensive survey of machine learning techniques applied to the analysis of Portable Executable (PE) files in Windows. The study systematically reviewed the objectives, data types, and machine learning models employed in prior research. As a survey, it does not conduct empirical benchmarking or propose new models. Although it discusses emerging deep learning architectures (e.g., Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Generative Adversarial Networks (GANs) transformers), the discussion is not exhaustive, particularly considering the rapid advancements in malware representation techniques like graph embeddings. The paper also highlights inconsistencies and closed nature of datasets in malware research, contributing to challenges in reproducibility and benchmarking.

Mohamed and Azher [18] reviewed classical malware detection techniques, including signature-based, heuristic-based, and specification-based approaches. Their analysis highlighted the strengths and limitations of each method but remained largely descriptive. The study did not involve any implementation, experimental validation, or dataset-based comparison. Emerging research trends such as adversarial machine learning and deep learning architectures (e.g., CNNs, RNNs, transformers) were not discussed, despite their relevance in current malware detection. Furthermore, standard performance metrics such as accuracy, precision, recall, F1-score, and latency were absent, impeding any quantitative comparison between techniques.

Singh *et al*. [27] introduced a malware detection framework using machine learning classifiers, specifically decision trees and Random Forests (RFs). The model with the highest accuracy was selected for deployment. Evaluation involved analysing False Positives (FPs) and False Negatives (FNs) using a confusion matrix. However, the model was not tested against obfuscated, packed, or adversarial malware samples common in real-world attacks. The study relies solely on the Microsoft Malware Classification Challenge dataset from Kaggle, which may lead to overfitting or dataset bias. Additionally, it lacks consideration of dynamic behavioural features like API calls and runtime logs, which are crucial for detecting advanced malware. The literature review is also limited, omitting modern approaches such as deep neural networks.

Akbanov *et al*. [3] presented a case-specific analysis of the WannaCry ransomware, proposing a Software Defined Networking (SDN) based method for its detection and mitigation. The approach identifies key system features exploited by WannaCry to encrypt files. However, the proposed solution is highly tailored and may not extend to other ransomware families with distinct propagation or encryption techniques. Detection relies heavily on static and dynamic blacklists (e.g., IP addresses, domains, and ports), with no incorporation of machine learning, behavioural analysis, or anomaly detection. Moreover, standard evaluation metrics such as FP rate, detection delay, throughput, and scalability are not reported.

Vehabovic *et al*. [32] conducted a detailed survey of ransomware detection techniques, focusing on host-based, network-based, forensic characterization, and authorship attribution methods. While many of these strategies utilize machine learning, the paper does not propose new models, datasets, or implementation frameworks. Instead, it synthesizes existing literature without standardized comparisons or unified performance metrics. As a result, its practical contribution is limited, and it does not address the reproducibility or benchmarking challenges prevalent in malware research.

Santangelo *et al*. [23] analyzed ransomware threats targeting Industrial Control Systems (ICS), particularly Ekans and MegaCortex, and proposed a protocol-based detection solution leveraging Windows Management Instrumentation (WMI) and Distributed Computing Environment/Remote Procedure Call (DCE/RPC) tracing. Although the study identifies unique lateral movement behaviour's used by ransomware in ICS environments, it does not explore machine learning or adaptive learning techniques for enhanced detection. Furthermore, the solution has not been benchmarked against industry-standard tools such as Suricata, Snort, or commercial endpoint detection systems.

Subedi *et al*. [29] utilized digital forensic techniques to investigate Dynamic Link Library (DLL) dependencies in ransomware samples, using static reverse engineering. The study included only 450 malware samples, limiting its generalizability. The proposed method does not account for common evasion tactics such as sandbox detection, code injection, process hollowing, or encrypted payloads. Additionally, it lacks comparative benchmarking with established malware analysis platforms like VirusTotal, Snort, or Suricata. The approach is dependent on manually defined rule sets and DLL-function mappings, which may hinder scalability.

Amanowicz and Jankowski [4] proposed a data mining-based framework for detecting and classifying malicious network flows in SDN. The system utilizes native SDN features and machine learning models (e.g., Support Vector Machine (SVM), Multi-Layer Perceptron (MLP)), focusing on automated flow rule generation and classification. Experiments were conducted using synthetic traffic generated by tools like Metasploit, Hydra, and Hping3. Although the classifiers demonstrated high detection accuracy, especially in lab settings, models like MLP incurred significant execution time, raising concerns about real-time performance and scalability.

Hossain and Islam [12] developed a framework to detect obfuscated malware in memory dumps. The process includes data normalization, feature encoding, Synthetic Minority Over-sampling Technique (SMOTE)-based class balancing, and feature selection using statistical methods (e.g., Chi-square, mutual information). Although the framework is effective for selected obfuscation techniques, it has not been validated against hybrid or multi-layered threats, limiting its adaptability.

Nguyen *et al*. [20] proposed a hybrid malware detection system tailored for cloud environments, combining static features (e.g., opcodes, file metadata) with dynamic features (e.g., API call traces, behavioral logs) using deep learning models. While the approach enhances detection coverage, it relies on specific datasets that may not generalize well to broader malware ecosystems. Additionally, the integration of multiple feature sets and complex models results in high computational overhead, potentially limiting real-time deployment.

Maniriho *et al*. [17] introduced an innovative malware detection framework, MeMalDet, which directly leverages memory dump data (RAM images) for analysis. The framework employs deep autoencoders to perform unsupervised feature extraction by reducing the dimensionality of raw memory features, enabling the automatic identification of significant patterns. Although MeMalDet demonstrates impressive performance, achieving an accuracy of 98.82% and an F1-score of 98.72%, the study does not extensively evaluate its robustness against adversarial evasion tactics such as memory injection, obfuscation, or anti-forensic techniques.

A study by Mahanta and Kumar [16] explores malware detection using Windows Event Logs transformed into structured datasets for machine learning analysis. While the method is capable of identifying attack patterns, it focuses on limited malware types and lacks generalization across diverse threat vectors. The framework's effectiveness is therefore constrained in broader applications.

Kalinkin *et al*. [14] investigated the use of Event Tracing for Windows (ETW) data in conjunction with machine learning models to detect ransomware. ETW enables detailed tracking of system and application behavior, facilitating anomaly detection. However, the framework's success is highly dependent on the quality and completeness of the collected ETW data. In addition, implementing the system at scale presents challenges related to real-time processing and performance overhead.

Table 1. A Summary of the existing work.

Ref	Authors	Method	Dataset used	Accuracy/ Evaluation metric
[10]	Damodaran *et al.*	HMMs on static and dynamic features (API, opcode sequences)	A mix of malware family and benign	AUC is obtained for separate malware family
[19]	Moskovitch *et al.*	Static n-gram opcode representation for classification	VX Heaven+campus machine benign files (~30,000)	99% accuracy was observed for 15% of malicious files
[27]	Singh *et al.*	ML classifiers (decision tree, RF); confusion matrix analysis	Kaggle Microsoft malware classification challenge	99% accuracy (potential overfitting)
[3]	Akbanov *et al.*	Static+dynamic analysis with SDN detection (WannaCry ransomware)	WannaCry instance	Not reported
[29]	Subedi *et al.*	Static analysis+reverse engineering of DLL-function mappings	450 ransomware samples	70% accuracy
[4]	Amanowicz and Jankowski	ML classifiers (SVM, MLP) on SDN malicious flow detection	Lab-generated traffic (Metasploit, Hydra, Hping3)	97% TPR
[12]	Hossain and Islam	Memory dump normalization+SMOTE+feature selection+ML	Obfuscated-MalMem2022 dataset	More than 99% accuracy
[20]	Nguyen *et al.*	Static+dynamic features (e.g., API logs, opcodes)+deep learning	Cloud-based Malware Dataset 2024 (CMD_2024 dataset)	99.42% accuracy for Dynamic and Deep Malware Detection (D2MD) model, 86.97% accuracy for multi-class classification
[17]	Maniriho *et al.*	multiple machine learning classifiers (like RF, XGBoost, LightGBM) in a stacked ensemble	MemMal-d2024	accuracy of 98.82% and an F1-score of 98.72%
[14]	Kalinkin *et al.*	ETW+ML for ransomware detection	2 ransomware, 4 benign	Highest precision of 0.98
[8]	Celdran *et al.*	Device behavioral fingerprinting+kernel-level event monitoring+machine learning classifiers	10 distinct malware samples (botnets, rootkits, backdoors, ransomware,	Achieved up to 99.99% accuracy with Artificial Neural Network. (ANN) classifiers in supervised settings

Celdran *et al*. [8] introduced a modular detection framework that integrates device behavioral fingerprinting with machine learning techniques to identify malware in IoT spectrum sensors. The framework was empirically evaluated using the ElectroSense platform, a practical and widely adopted crowdsensing environment. Although the proposed system demonstrated effective performance on ElectroSense devices, its generalizability to other IoT hardware and environments remains an area for future exploration. The anomaly detection component, aimed at identifying zero-day threats, achieved a True Positive Rate (TPR) ranging from 88% to 90%, while the malware classification module, focused on known attacks, attained an F1-score between 94% and 96%.

Shamshirsaz *et al.* [25] propose a process supervision/control-based malware detection mechanism that forces activation of latent code paths and monitors sensitive OS function calls, reporting ~98% accuracy.

Most existing research emphasizes main memory analysis for malware detection but often overlooks the impact of malware on system-level indicators such as Windows Event Logs.

Table 1 presents a summary of earlier work, categorized based on the methodology employed, datasets utilized, and evaluation metrics reported.

It has been observed that the majority of existing malware detection approaches are primarily based on memory analysis. In contrast, relatively few studies focus on utilizing Windows Event Log analysis for malware detection. Moreover, none of the reviewed works have attempted to precisely locate and analyze the code sections of running processes. Given that malware execution often results in artifacts within the Windows Event Logs, there is a critical need to integrate both code-level analysis and Event Log analysis to enhance the effectiveness and reliability of malware detection.

3. Proposed Approach

The proposed malware detection approach adopts a dual-pronged strategy. The first component focuses on identifying malicious code executing within the system by performing code analysis on executable files mapped into the system's main memory. The second component involves monitoring and identifying events indicative of suspicious activity, such as unauthorized file access, changes in file permissions, and previously flagged malware-related behaviors. These two detection mechanisms are integrated to generate alerts that notify the user of a potential malware attack.

This section elaborates on the methodology for locating the code section of executables associated with active processes from the system's main memory. It also discusses the correlation of suspicious system events and the overall system architecture designed to trigger user alerts. Specifically, once the executable code is located and extracted from the memory-resident image of the process, it is compared against a database of verified legitimate code and existing malware code. A match with the existing malware code strongly indicates the presence of malware. But if none of the code in the database is matched then it suggests the presence of potentially malicious or unauthorized code in memory. Following this, system Event Logs particularly those accessed via the Windows Event Viewer are examined for activities consistent with malware behavior. If both suspicious memory-resident code and corroborating system events are identified, the system generates a warning to inform the user of a possible malware intrusion.

3.1. Detecting Code Section

Traditional antivirus software relies on known signatures to detect malicious programs. However, this approach fails when the malware's signature is absent from the antivirus database, allowing novel or obfuscated threats to evade detection. To overcome this limitation, it is essential to analyze the code section of executable programs directly within the system's main memory. When an executable is loaded, the OS generates multiple kernel-level data structures associated with the process, such as the EPROCESS block, Virtual Address Descriptors (VADs), and the Page Table. These structures store critical information, including virtual memory addresses and other attributes necessary for identifying and locating process-specific components in memory. Accessing these memory structures requires translating virtual addresses into corresponding physical addresses, enabling accurate inspection of the data held in main memory and supporting advanced malware detection techniques beyond signature matching.

a) *Key Data structures and their roles*.

1. EPROCESS structure. The EPROCESS structure plays a pivotal role in identifying the code section of a PE file during memory analysis. In a dumped memory image, this structure can typically be located by searching for the American Standard Code for Information Interchange (ASCII) string "pro" or its hexadecimal representation 0x50726FE3. One of the key attributes within the EPROCESS structure is VadRoot, which stores the virtual address of the VAD tree. This tree structure is essential for mapping the memory regions allocated to a process, including those corresponding to the code section of the loaded executable. By traversing the VAD tree, forensic tools can pinpoint the location of the PE code in memory, enabling in-depth analysis for potential malware.
2. VAD tree structure. The VAD tree is an essential data structure used to identify memory-mapped files associated with an active process in main memory.

Each node in the VAD tree contains several attributes that facilitate the interpretation and traversal of process memory regions. Among these attributes,

- StartVpn and EndVpn: StartVpn attribute identify the starting address of the first frame of a memory-mapped file. EndVpn attributes identify the starting address of the last frames of a memory-mapped file.
- FirstProtoPte: this attribute holds the virtual address of the Page Table associated with the process.
- Subsection: this attribute points to the first subsection structure, which represents a section of the executable file mapped in memory.

3. Page Table. The Page Table holds the mapping between virtual and physical memory addresses and is responsible for storing the starting physical address of the first memory frame associated with a memory-mapped file. This address is crucial for accessing and analyzing the physical memory content corresponding to a specific virtual address range.
4. Subsection structures. Each section of an executable file that is mapped into memory is associated with a corresponding subsection structure. These subsection structures are integral to understanding how the executable's sections are organized and managed in memory. Key fields within a subsection include NextSubsection and PteInSubsection.

- NextSubsection: points to the next subsection structure, thereby forming a linked sequence of memory sections
- PteInSubsection: indicates the number of Page Table Entries (PTEs) for the section. This value reflects the number of memory frames allocated to that section where one PTE corresponds to one frame, two PTEs to two frames, and so forth. These structures are essential for reconstructing the memory layout of an executable during memory forensics and for detecting anomalies associated with malicious code injections.

b) *Executable file sections in memory*.

An executable file is composed of multiple sections, each serving a distinct function in the execution and management of the program. Among these, the Header Section contains critical metadata about the file, such as the file type, entry point, and section layout. In a 32-bit Windows environment, this section typically occupies a single memory frame, equivalent to 4 KB. Another crucial component following header section is the code section, which holds the actual machine-level instructions that the microprocessor executes. Identifying this section in main memory is essential for performing code analysis, particularly in the context of malware detection and reverse engineering. Accurate extraction and interpretation of the code section enable analysts to detect unauthorized modifications, embedded malicious routines, or obfuscated instructions within a potentially compromised executable.

c) *Relationship between memory structures*.

Figure 1 illustrates the hierarchical relationship among the EPROCESS, VAD, and Page Table structures in the context of memory management for a running process. When an executable is loaded, its code is mapped into the frames in main memory. The EPROCESS structure, contains the VadRoot attribute a pointer to the root of the VAD tree. This tree is used to track the memory regions allocated to the process. Each VAD node in the tree represents a specific memory space allocated to a file associated with a running process. Each VAD contains attributes that assist in identifying the starting and ending frames of the memory-mapped file. Through these interconnected structures, it becomes possible to locate and analyze the code section of an executable.

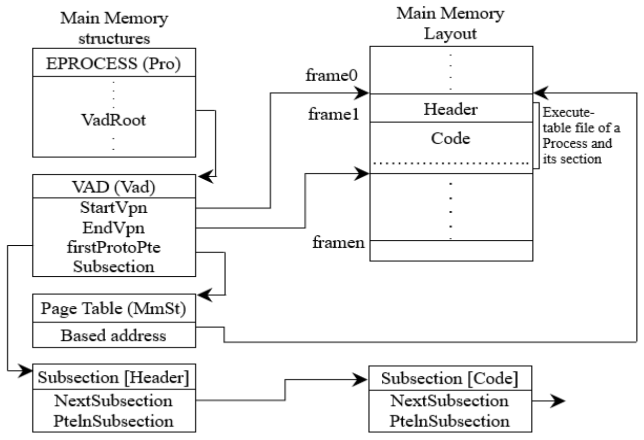


Figure 1. Relationship between various main memory structures.

d) *Extracting executable code*.

After identifying the number of memory frames that contain the executable code and determining the corresponding physical addresses of these frames, the code within the code section can be extracted from memory for detailed analysis.

3.2. Windows Events

Windows Event Logs maintain a comprehensive record of system, security, and application-related events generated by the Windows OS and the applications running on it. These logs provide investigators with critical information, such as the applications involved, user login timestamps, and various system events relevant for forensic analysis. Notably, even if antivirus software fails to detect malware present on the system, evidence of malicious activity can often be inferred from specific Event Log entries.

Microsoft Windows registers certain events triggered by malware infections, which are recorded in the Event Logs. These recorded events serve as valuable indicators for detecting malware presence. Timely monitoring and analysis of these events can significantly mitigate the impact of malware by enabling early detection and response. Table 2 enumerates some common Event IDs and descriptions associated with malware activity.

Table 2. Malware related events.

Event ID	Event
7045	Creation of a new service which enables remote access to the target.
4670	Permission on any object changed.
1116	The anti-malware platform detected malware.
1006	The anti-malware engine detected malware.
1008	The anti-malware platform has attempted to perform an action to protect your system from malicious software or other potentially harmful software.
4798	APT actors have compromised local accounts on the system.

Digital forensic investigators must scrutinize these events to identify suspicious behavior indicative of compromise. When an event associated with malware is detected, its presence can be corroborated by correlating it with suspicious code extracted from the executable file mapped in the running process's memory, as described in section 3.1. If both suspicious code and malware-related events are identified, the presence of malware actively running in main memory can be confidently confirmed.

3.3. Software Architecture

To alert users of potential malware attacks on Windows-based systems, a software architecture has been proposed that integrates two primary detection mechanisms: identification of suspicious code from the executable program in main memory and detection of malware-related events from the Windows Event Logs. The architecture of the proposed system, illustrated in Figure 2, comprises several functional modules, including code extraction, code comparison, event detection, and an alert generation module.

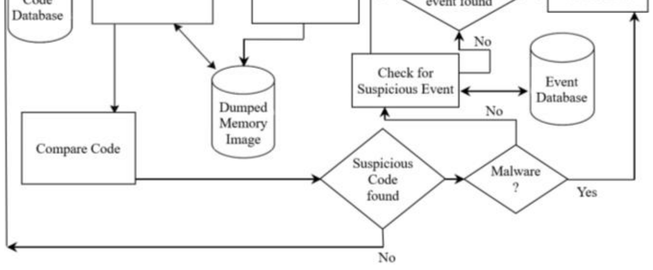


Figure 2. Software architecture of the proposed system.

In addition to these modules, the architecture incorporates two databases: one for storing the known legitimate code of running processes and code of the existing malwares, and another for storing dumped memory images for forensic analysis. This modular design enables the system to perform real-time correlation between memory-resident code anomalies and Event Log patterns indicative of malware activity. A description of each module in the architecture is provided below.

a) Dump memory

This module of the proposed system is responsible for periodically dumping the Windows main memory at regular intervals, specifically every 10 minutes. The resulting memory dump is then stored in the dumped memory image database for subsequent analysis. Tools such as DumpIt are utilized for this task, generating memory images in the raw format, which preserves the entire physical memory content of the system at the time of capture. These memory dumps are essential for enabling offline analysis of running processes, code sections, and potential malware residing in memory.

b) Extract code.

The executable code of the file associated with a running process is identified using the methodology outlined in the relevant subsection on code detection. Once the code section has been successfully located in main memory, this module proceeds to extract the corresponding code for further analysis. The extracted code serves as a critical input for comparison against known legitimate and malicious code patterns stored in the code database, thereby facilitating the detection of anomalies indicative of potential malware.

c) Compare code.

The extracted code is subsequently compared against entries in the code database, which contains both the original code of legitimate running processes and the known code of existing malware. If the extracted code matches that of a legitimate process, the system resumes monitoring and initiates the next memory dump after a 10-minute interval, continuing the periodic analysis cycle. This routine ensures ongoing surveillance of the system's memory state, enabling timely detection of any deviations that may indicate malicious activity. If the extracted code matches that of a malware code, then it triggers alarm.

d) Check for suspicious event.

Events related to malware activity are identified and extracted from the Windows Event Logs database. When the extracted code from memory is determined to be suspicious that is, it does not match any known legitimate process or matches known malware the system proceeds to analyze the Windows Event Logs for any associated suspicious events. This correlation between anomalous code and relevant system events enhances the reliability of malware detection by providing both behavioral and memory-based evidence of compromise.

e) Give alarm.

This module is responsible for alerting the user of the Windows-based computer system in the event of a potential malware attack. If the extracted code is determined to be malicious matching known malware code the system immediately triggers an alarm to notify the user. In cases where the code is suspicious, i.e., it does not match either known legitimate processes or known malware the system conducts a further analysis by checking the Windows Event Log database for malware-related events. If such events are detected in conjunction with the suspicious code, the system raises an alert, indicating a high likelihood of an active malware presence in the main memory.

The pseudocode of the proposed approach is presented in Algorithm (1).

Algorithm 1: *Proposed approach to detect malware.*

```
Initialize:
    Set memory_dump_interval = 10 minutes
    Load legitimate_code_db
    Load malware_code_db
    Load Windows_event_log_db
Loop:
    For each memory_dump_interval do:
      1. Dump main memory→memory_dump.raw
      2. Extract code section
      memory_dump.raw→extracted_code
      3. Compare extracted_code with legitimate_code_db
       If match_found:
         Continue to next interval
        Else:
      4. Compare extracted_code with malware_code_db
           If malware_match_found:
             Trigger ALARM: "Malicious code detected!"
              Continue to next interval
             Else:
      5. Analyze Windows_event_log_db for malware-related
      events.
           If suspicious_events_found:
             Trigger ALARM: "Suspicious code+malicious events
             detected!"
                Else:
                    Log "Suspicious code, no associated events"
Continue to next interval
```

Key Function Descriptions of the pseudocode are as follows:

- *Dump main memory*: uses tools like DumpIt to capture system memory.
- *Extract code section*: identifies and isolates the code section using EPROCESS, VAD, and Page Table structures.
- *Compare extracted_code*: matches binary patterns or hashes with entries in legitimate and malware code databases.
- *Analyze Windows_event_log_db*: looks for Event IDs typically associated with malware activity (e.g., privilege escalation, unauthorized file access, etc.).
- *Trigger ALARM*: notifies the user of potential malware based on detection criteria.

The flowchart of the proposed approach is shown in Figure 3.

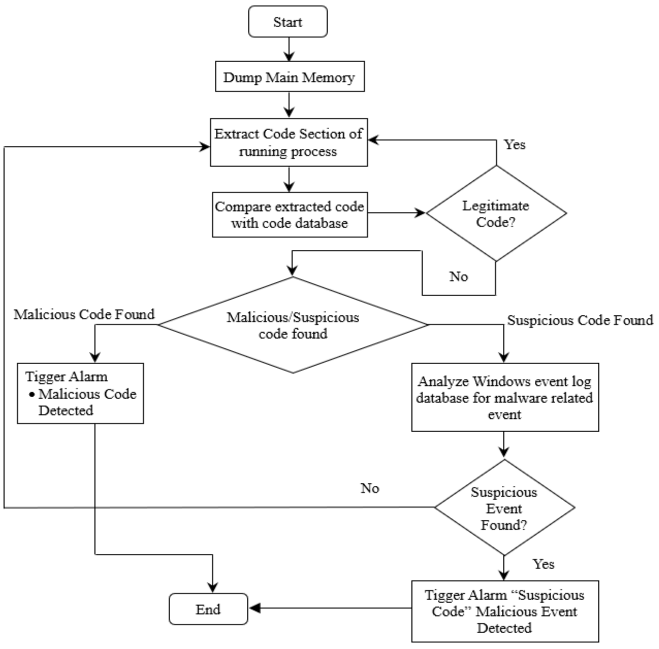

Start
Dump Main Memory
Extract Code Section of running process
Yes
Compare extracted code with code database
Legitimate Code?
No
Malicious Code Found
Malicious/Suspicious code found
Suspicious Code Found
Tigger Alarm
• Malicious Code Detected
Analyze Windows event log database for malware related event
No
Suspicious Event Found?
Yes
End
Tigger Alarm "Suspicious Code" Malicious Event Detected

Figure 3. Flowchart of the proposed approach.

3.4. Legal Implications of Live Memory Acquisition and Analysis

Live memory acquisition and analysis, while crucial for modern digital forensics and malware detection, present several ethical issues most notably concerning privacy and legal implications. During memory acquisition, analysts can inadvertently access sensitive personal data such as login credentials, open communications, cryptographic keys, and private browsing sessions that reside temporarily in main memory. This raises significant privacy concerns, especially when such data pertains to individuals not under investigation or when consent has not been explicitly obtained. Furthermore, the process may conflict with legal rights related to data protection and unauthorized access, particularly under regulations such as the General Data Protection Regulation (GDPR) or the Computer Fraud and Abuse Act (CFAA), Digital Personal Data Protection Act, 2023 (DPDP Act). In some jurisdictions, even well-intentioned forensic investigations might be deemed unlawful if proper legal authorization is not secured beforehand. Thus, it is imperative for practitioners to balance investigative objectives with strict adherence to legal standards and ethical guidelines to ensure that memory acquisition is performed responsibly, transparently, and within the bounds of the law.

4. Result

This section presents the experiments conducted to detect the code sections of executable programs mapped in main memory, as well as the events triggered by malware activity. It also details the training and testing performed to evaluate the accuracy of the proposed approach, employing various classifiers.

4.1. Experimentation

Experiments were conducted on 32-bit Windows 7 and 64-bit Windows 10 systems the most widely used OSs, as identified in the survey presented by Pot [21] to validate the proposed approach. The objective of these experiments was to acquire memory dump images containing both malicious and benign code, and to detect events associated with the execution of malicious code. As Ahmed and Aslam [2] have discussed memory dumps were obtained using the DumpIt tool, which was selected for its ability to reliably and completely capture main memory content to disk. The resulting memory dump files were saved with a raw extension.

To validate the technique for detecting code sections of malware executing on Windows-based systems, experiments were carried out using a sample process hollowing executable, ProcessHollowing.exe, available at [11]. This executable performs a process hollowing attack by hollowing out the code section of the legitimate svchost.exe process and replacing its memory space with the image of helloworld.exe.

The following steps outline the experimental procedure for detecting code sections:

1. ProcessHollowing.exe was executed on a Windows system with several application programs open.
2. A memory dump was created using the DumpIt tool.
3. At the time of dumping, live memory analysis was performed using Windows Debugger (WinDbg).
4. The physical address of memory structures such as the VAD for the running svchost.exe process was extracted using WinDbg.
5. The memory dump image was loaded into the OSForensics tool for offline analysis.
6. The physical address of the VAD obtained from WinDbg was used to locate the corresponding code section within the dumped main memory.

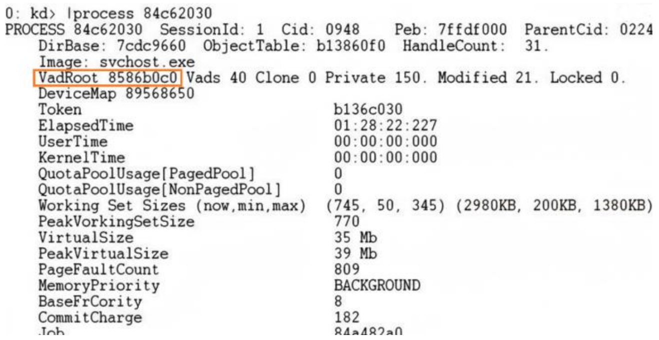
As part of the evaluation, 10 memory dump images were analyzed for each OS (Windows 7 and Windows 10) to establish a correlation between process-specific memory structures and the detection of code sections. Additionally, during the execution of the hollowing process (ProcessHollowing.exe), Windows Event Logs were monitored for any security-relevant events. Table 3 provides the specifications of the test machine used in the experiments.

The DumpIt tool was used to acquire a complete snapshot of the Windows main memory. For live memory analysis during execution, the WinDbg tool was utilized. The acquired memory dumps were further examined using the OSForensics tool.

Table 3. Test system.

OS	Windows 7, 32-bit	Windows 10, 64-bit	Windows 10, 64-bit
Main memory	2GB	8GB	16GB
Processor	Celeron 440 @2.0GHz	Intel(R) Core(TM) i3-6006U CPU @ 2.00GHz	Intel(R) Core(TM) i3-6006U CPU @ 2.00GHz

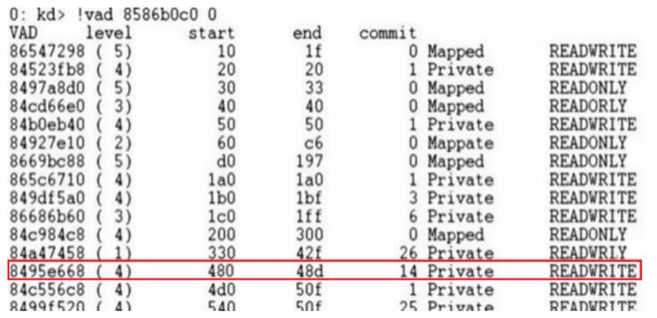
Figure 4 presents the VadRoot virtual address of the svchost.exe process, extracted using WinDbg commands. This address serves as the root of the VAD tree, which represents the memory layout of a running process. Each VAD node corresponds to a memory region allocated to a file or module associated with the process.



```
0: kd> !process 84c62030
PROCESS 84c62030  SessionId: 1  Cid: 0948    Peb: 7ffdf000  ParentCid: 0224
    DirBase: 7cdc9660  ObjectTable: b13860f0  HandleCount:  31.
    Image: svchost.exe
    VadRoot 8586b0c0 Vads 40 Clone 0 Private 150. Modified 21. Locked 0.
    DeviceMap 89568650
    Token                             b136c030
    ElapsedTime                       01:28:22:227
    UserTime                          00:00:00:000
    KernelTime                        00:00:00:000
    QuotaPoolUsage[PagedPool]         0
    QuotaPoolUsage[NonPagedPool]      0
    Working Set Sizes (now,min,max)  (745, 50, 345) (2980KB, 200KB, 1380KB)
    PeakVorkingSetSize                770
    VirtualSize                       35 Mb
    PeakVirtualSize                   39 Mb
    PageFaultCount                    809
    MemoryPriority                    BACKGROUND
    BaseFrCority                      8
    CommitCharge                      182
    Job                               84a482a0
```

Figure 4. WinDbg snapshot of svchost.exe showing VadRoot.

Figure 5 illustrates the VAD entries for the svchost.exe process. The highlighted entry indicates the VAD corresponding to the executable file associated with svchost.exe, which was hollowed out and overwritten by the malicious ProcessHollowing.exe. The memory protection for this region was modified to EXECUTE_READWRITE by the hollowing process to allow code injection.

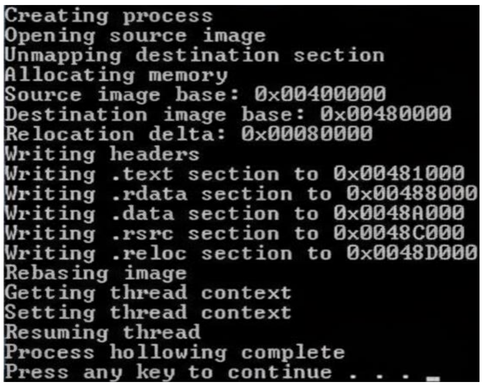


```
0: kd> !vad 8586b0c0 0
VAD     level       start       end    commit
86547298 ( 5)          10        1f         0 Mapped       READWRITE
84523fb8 ( 4)          20        20         1 Private      READWRITE
8497a8d0 ( 5)          30        33         0 Mapped       READONLY
84cd66e0 ( 3)          40        40         0 Mapped       READORLY
84b0eb40 ( 4)          50        50         1 Private      READWRITE
84927e10 ( 2)          60        c6         0 Mappate      READONLY
8669bc88 ( 5)          d0       197         0 Mapped       READONLY
865c6710 ( 4)         1a0       1a0         1 Private      READWRITE
849df5a0 ( 4)         1b0       1bf         3 Private      READWRITE
86686b60 ( 3)         1c0       1ff         6 Private      READWRITE
84c984c8 ( 4)         200       300         0 Mapped       READONLY
84a47458 ( 1)         330       42f        26 Private      READWRLY
8495e668 ( 4)         480       48d        14 Private      READWRITE
84c556c8 ( 4)         4d0       50f         1 Private      READWRITE
8499f520 ( 4)         540       50f        25 Private      READWRITE
```

Figure 5. WinDbg snapshot of svchost.exe VAD tree structure.

Figure 6 shows a snapshot of ProcessHollowing.exe writing to various sections of the svchost.exe executable in main memory. The address of the hollowed-out VAD region was validated through comparative analysis, as demonstrated in Figures 7 and 8.

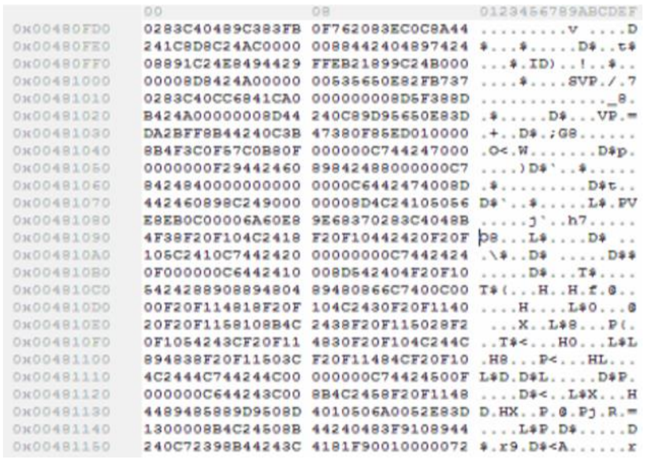
The physical address range corresponding to the specific VAD for the svchost.exe executable is determined to be from 0x00480000 to 0x0048D000, which matches the address range shown in Figure 6. This range is derived by appending four 0000 to the values of the ‘start’ and ‘end’ fields highlighted in Figure 5. The address 0x00480000 marks the beginning of the first memory frame containing the svchost.exe file, while 0x0048D000 indicates the starting address of the last frame associated with the same file. On a 32-bit OS, the first page of a VAD typically consists of a 4KB header section. Consequently, the code section (i.e., the .text section) begins at address 0x00481000. During process hollowing, this original code section is overwritten with malicious code.



```
Creating process
Opening source image
Unmapping destination section
Allocating memory
Source image base: 0x00400000
Destination image base: 0x00480000
Relocation delta: 0x00080000
Writing headers
Writing .text section to 0x00481000
Writing .rdata section to 0x00488000
Writing .data section to 0x0048A000
Writing .rsrc section to 0x0048C000
Writing .reloc section to 0x0048D000
Rebasing image
Getting thread context
Setting thread context
Resuming thread
Process hollowing complete
Press any key to continue . . . _
```

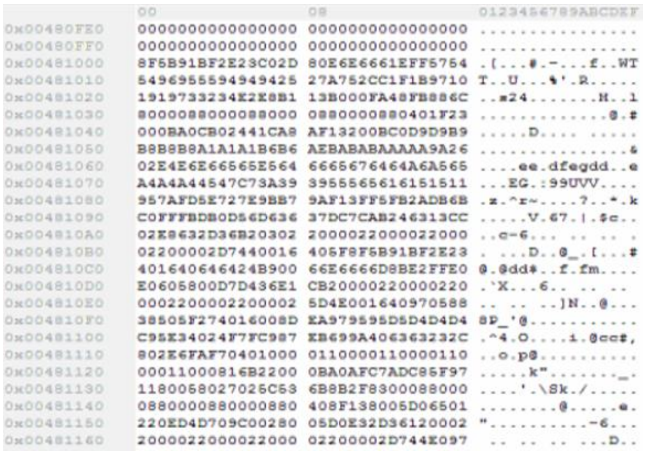
Figure 6. Process hollow snap injecting code in svchost.exe image on Window 7.

Figure 7 presents a snapshot of the first page of the original code section of the svchost.exe executable, starting at address 0x00481000, as captured using the OSForensics tool. In contrast, Figure 8 displays the corresponding snapshot after the code section has been modified through process hollowing. A comparative analysis of the two snapshots reveals clear discrepancies, confirming that the code within the code section of svchost.exe differs significantly before and after the hollowing process. This observation validates the effectiveness of identifying malicious activity through code comparison techniques.



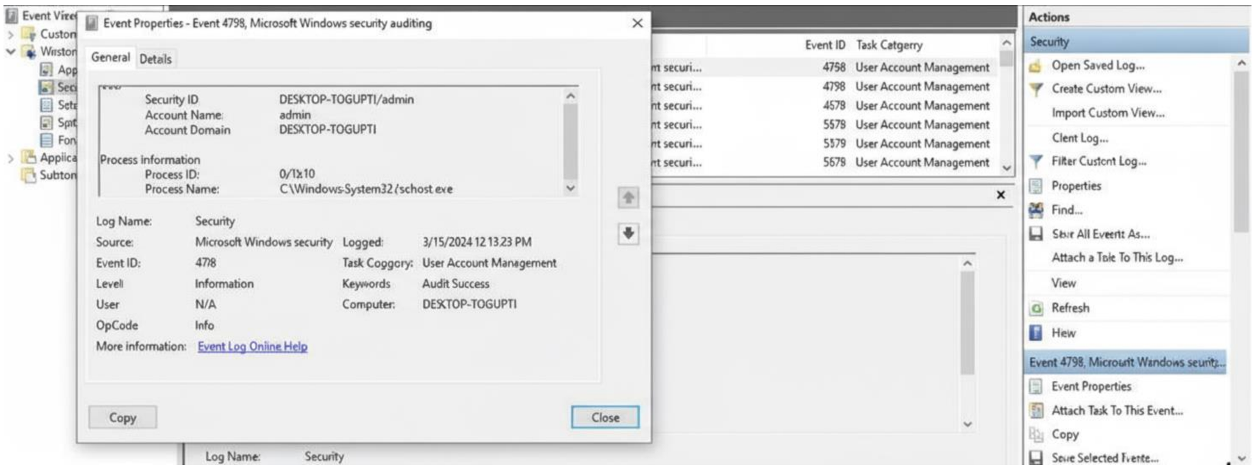
```
             00               08               0123456789ABCDEF
0x00480FD0   0283C40489C383FB 0F762083EC0C8A44 ..........v ....D
0x00480FE0   241C8D8C24AC0000 0088442404897424 $...$......D$..t$
0x00480FF0   08891C24E8494429 FFEB21899C24B000 ...$.ID)..!..$..
0x00481000   00008D8424A00000 00535650E82FB737 ....$....SVP./.7
0x00481010   0283C40CC6841CA0 000000008D5F388D ..............._8.
0x00481020   B424A000000008D44 240C89D95650E83D .$......D$...VP.=
0x00481030   DA2BFF8B44240C3B 47380F85ED010000 .+..D$.;G8......
0x00481040   8B4F3C0F67C0B80F 000000C744247000 .O<.W........D$p.
0x00481050   0000000F29442460 8984248800000C7 .....)D$`..$.....
0x00481060   8424840000000000 0000C6442474008D .$..........D$t..
0x00481070   442460898C249000 00008D4C24105056 D$`..$......L$.PV
0x00481080   E8EB0C00006A60E8 9E68370283C4048B ......j`..h7.....
0x00481090   4F38F20F104C2418 F20F1044242 0F20F O8...L$....D$ ..
0x004810A0   105C2410C7442420 00000000C7442424 .\$..D$ ......D$$
0x004810B0   0F000000C6442410 008D542404F20F10 ......D$...T$....
0x004810C0   5424288908894804 89480866C7400C00 T$(...H..H.f.@..
0x004810D0   00F20F114818F20F 104C2430F20F1140 ....H.....L$0...@
0x004810E0   20F20F1158108B4C 2438F20F115028F2  ...X..L$8...P(.
0x004810F0   0F1054243CF20F11 4830F20F104C244C ..T$<...H0...L$L
0x00481100   894838F20F11503C F20F11484CF20F10 .H8...P<...HL...
0x00481110   4C2444C744244C00 000000C74424500F L$D.D$L......D$P.
0x00481120   000000C644243C00 8B4C2458F20F1148 ....D$<..L$X...H
0x00481130   4489485889D9508D 4010506A0052E83D D.HX..P.@.Pj.R.=
0x00481140   1300008B4C24508B 44240483F9108944 ....L$P.D$.....D
0x00481150   240C72398B44243C 4181F90010000072 $.r9.D$<A......r
```

Figure 7. OSForensic snap of original svchost.exe code section.



```
             00               08               0123456789ABCDEF
0x00480FE0   0000000000000000 0000000000000000 ................
0x00480FF0   0000000000000000 0000000000000000 ................
0x00481000   8F5B91BF2E23C02D 80E6E6661EFF5754 .[...#.-...f..WT
0x00481010   5496955594949425 27A752CC1F1B9710 T..U...%'.R.....
0x00481020   191973323 4E2E8B1 13B000FA48FB886C ..s24........H..l
0x00481030   8000088000088000 0880000880401F23 ..............@.#
0x00481040   000BA0CB02441CA8 AF13200BC0D9D9B9 .....D.... .....
0x00481050   B8B8B8A1A1A1B6B6 AEBABABAAAAA9A26 ...............&
0x00481060   02E4E6E66565E564 6665676464A6A565 ....ee.dfegdd..e
0x00481070   A4A4A44547C73A39 3955565616151511 ...EG.:99UVV....
0x00481080   957AFD5E727E9BB7 9AF13FF5FB2ADB6B .z.^r~....?..*.k
0x00481090   C0FFFBDB0D56D636 37DC7CAB246313CC .....V.67.|.$c..
0x004810A0   02E8632D36B20302 2000022000022000 ..c-6... .. .. .
0x004810B0   02200002D7440016 405F8F5B91BF2E23 .  ...D..@_.[...#
0x004810C0   40164064 6424B900 66E6666D8BE2FFE0 @.@dd$...f.fm....
0x004810D0   E0605800D7D436E1 CB20000220000220 .`X...6.. .. ..
0x004810E0   0002200002200002 5D4E001640970588 .. .. ..]N..@...
0x004810F0   38505F274016008D EA979595D5D4D4D4 8P_'@...........
0x00481100   C95E34024F7FC987 EB699A406363232C .^4.O....i.@cc#,
0x00481110   802E6FAF70401000 0110000110000110 ..o.p@..........
0x00481120   00011000816B2200 0BA0AFC7ADC85F97 ......k"......_.
0x00481130   1180058027025C53 6B8B2F8300088000 ....'.\Sk./.....
0x00481140   0880000880000880 408F138005D06501 ........@.....e.
0x00481150   220ED4D709C00280 05D0E32D36120002 ".........-6...
0x00481160   2000022000022000 02200002D744E097  .. .. .. ...D..
```

Figure 8. OSForensic snap of hollowed svchost.exe code section.



Event Properties - Event 4798, Microsoft Windows security auditing

General | Details

Security ID	DESKTOP-TOGUPTI/admin
Account Name:	admin
Account Domain	DESKTOP-TOGUPTI
Process Information	
Process ID:	0/1x10
Process Name:	C:\Windows-System32/schost.exe

Log Name:	Security		
Source:	Microsoft Windows security	Logged:	3/15/2024 12 13 23 PM
Event ID:	4798	Task Coggory:	User Account Management
Levell	Information	Keywords	Audit Success
User	N/A	Computer:	DESKTOP-TOGUPTI
OpCode	Info		
More information:	Event Log Online Help		

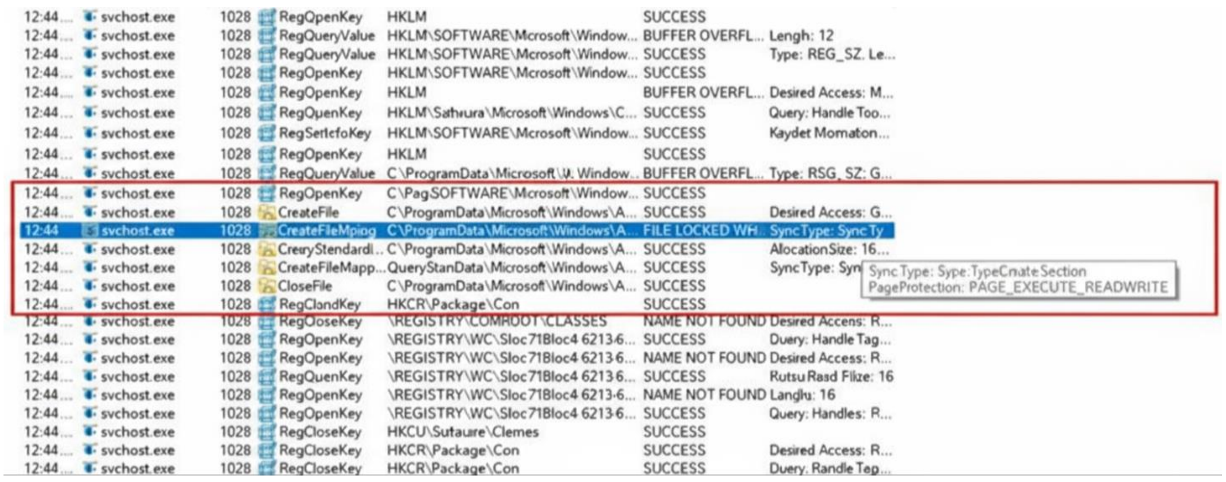
	Event ID	Task Catgerry
nt securi...	4758	User Account Management
nt securi...	4798	User Account Management
nt securi...	4578	User Account Management
nt securi...	5578	User Account Management
nt securi...	5379	User Account Management
nt securi...	5678	User Account Management

Log Name: Security

Figure 9. Event detected in event viewer.

Following the execution of ProcessHollowing.exe, an event with Event ID 4798 is recorded in the Windows Event Viewer, as shown in Figure 9. This event is triggered when a process enumerates the local group memberships associated with a specific user account on the system. The logging of this event is critical in detecting APT actors, as it indicates attempts to investigate compromised user accounts an activity often associated with lateral movement within a target environment.

The ProcessHollowing.exe process has overwritten the code section of the svchost.exe executable associated with an active process in memory by injecting the code of HelloWorld.exe. To facilitate this modification, ProcessHollowing.exe altered the underlying security principles that govern resource access and protection for services and users on the system. Specifically, the memory protection attributes of the physical frames holding the code section of the svchost.exe process were modified to permit write operations. This change in memory protection enabled the injection of the malicious code into the previously protected .text section of the executable.



```
12:44... svchost.exe  1028 RegOpenKey       HKLM                                   SUCCESS
12:44... svchost.exe  1028 RegQueryValue    HKLM\SOFTWARE\Mcrosoft\Window...       BUFFER OVERFL... Lengh: 12
12:44... svchost.exe  1028 RegQueryValue    HKLM\SOFTWARE\Mcrosoft\Window...       SUCCESS          Type: REG_SZ. Le...
12:44... svchost.exe  1028 RegOpenKey       HKLM\SOFTWARE\Mcrosoft\Window...       SUCCESS
12:44... svchost.exe  1028 RegOpenKey       HKLM                                   BUFFER OVERFL... Desired Access: M...
12:44... svchost.exe  1028 RegQuenKey       HKLM\Sathura\Microsoft\Windows\C...    SUCCESS          Query: Handle Too...
12:44... svchost.exe  1028 RegSetlcfoKey    HKLM\SOFTWARE\Mcrosoft\Window...       SUCCESS          Kaydet Momaton...
12:44... svchost.exe  1028 RegOpenKey       HKLM                                   SUCCESS
12:44... svchost.exe  1028 RegQueryValue    C\ProgramData\Microsoft\V. Window...   BUFFER OVERFL... Type: RSG_SZ: G...
12:44... svchost.exe  1028 RegOpenKey       C\PagSOFTWARE\Mcrosoft\Window...       SUCCESS
12:44... svchost.exe  1028 CreateFile       C\ProgramData\Microsoft\Windows\A...   SUCCESS          Desired Access: G...
12:44    svchost.exe  1028 CreateFleMping   C\ProgramData\Microsoft\Windows\A...   FILE LOCKED WH.. SyncType: Sync Ty
12:44... svchost.exe  1028 CreryStendardl...C\ProgramData\Microsoft\Windows\A...   SUCCESS          AlocationSize: 16...
12:44... svchost.exe  1028 CreateFileMapp...QueryStanData\Microsoft\Windows\A...   SUCCESS          SyncType: Syn
                                                                                                    Sync Type: Sype:TypeCnate Section
                                                                                                    PageProtection: PAGE_EXECUTE_READWRITE
12:44... svchost.exe  1028 CloseFile        C\ProgramData\Microsoft\Windows\A...   SUCCESS
12:44... svchost.exe  1028 RegClandKey      HKCR\Package\Con                       SUCCESS
12:44... svchost.exe  1028 RegOoseKey       \REGISTRY\COMROOT\CLASSES              NAME NOT FOUND   Desired Accens: R...
12:44... svchost.exe  1028 RegOpenKey       \REGISTRY\WC\Sloc71Bloc4 62136...      SUCCESS          Duery: Handle Tag...
12:44... svchost.exe  1028 RegOpenKey       \REGISTRY\WC\Sloc71Bloc4 62136...      NAME NOT FOUND   Desired Access: R...
12:44... svchost.exe  1028 RegQuenKey       \REGISTRY\WC\Sloc71Bloc4 62136...      SUCCESS          Rutsu Rasd Filze: 16
12:44... svchost.exe  1028 RegOpenKey       \REGISTRY\WC\Sloc71Bloc4 6213-6...     NAME NOT FOUND   Langlu: 16
12:44... svchost.exe  1028 RegOpenKey       \REGISTRY\WC\Sloc71Bloc4 62136...      SUCCESS          Query: Handles: R...
12:44... svchost.exe  1028 RegCloseKey      HKCU\Sutaure\Clemes                    SUCCESS
12:44... svchost.exe  1028 RegCloseKey      HKCR\Package\Con                       SUCCESS          Desired Access: R...
12:44... svchost.exe  1028 RegCloseKey      HKCR\Package\Con                       SUCCESS          Duery. Randle Tep...
```

Figure 10. A screenshot of process monitor showing svchost.exe memory protection flags altered on 64-bit Window 10.

Experiments were also conducted on a 64-bit Windows 10 systems equipped with an Intel i3 processor (2 cores) and 8GB/16 GB of RAM. The ProcessHollowing.exe was executed on this setup to evaluate its impact. To monitor system behavior during execution, Process Monitor part of the Sysinternals suite was utilized for real-time observation of active processes. As shown in Figure 10, the svchost.exe process was compromised during the execution of ProcessHollowing.exe. Notably, the memory protection flags associated with the code section of svchost.exe were altered to PAGE_EXECUTE_READWRITE, enabling unauthorized modification and execution of injected code within the previously protected memory region.

4.2. System Resource Utilization and Performance Overhead

We evaluated the impact and feasibility of memory dumping and Event Log analysis for detecting malicious activity across 3 different system configurations. The systems included:

1. 32-bit Windows 7 with 2 GB RAM and Celeron 440 processor.
2. 64-bit Windows 10 with 8 GB RAM and an Intel i3 dual-core processor.
3. 64-bit Windows 10 with 16 GB RAM and an Intel i3 dual-core processor.

Memory acquisition was performed using the DumpIt tool, which captured the entire contents of main memory into a binary dump file. The size of the memory dump varied according to the system's physical RAM, ranging from approximately 1.8 GB on the Windows 7 system to 16 GB on the higher-end Windows 10 machine. Following acquisition, analysis was carried out using forensic tools such as WinDbg, and OSForensics to examine the contents of the dump, particularly focusing on the code section of the svchost.exe process to detect signs of process hollowing. Additionally, Windows Event Viewer was employed to analyze security and system logs, capturing relevant Event IDs that correlate with suspicious behavior. Resource consumption, including Central Processing Unit (CPU) usage, memory overhead, and storage impact, was monitored throughout the process to assess system performance.

Table 4. Performance of different system configurations.

Parameter	32-bit Win 7, 2GB RAM, Celeron 440	64-bit Win 10, 8GB RAM, i3 (2-core)	64-bit Win 10, 16GB RAM, i3 (2-core)
Memory dump tool used	DumpIt	DumpIt	DumpIt
Avg. CPU usage (dumping)	20–25%	15–30%	10–20%
Memory overhead (dumping)	~100–150 MB	~200–400 MB	~300–500 MB
Dump file size	~1.5–1.8 GB	~7.5–8.0 GB	~15–16 GB
Analysis tool used	WinDbg/OSForensics/Process monitor	WinDbg/OSForensics/Process monitor	WinDbg/OSForensics/Process monitor
Avg. CPU usage (analysis)	30–40%	25–35%	20–30%
Memory usage (analysis)	~1.2 GB	~1.5–2.0 GB	~2.0–2.5 GB
Event Log tool used	Event Viewer	Event Viewer	Event Viewer
CPU usage (event analysis)	10–15%	10–20%	10–20%
Storage usage (logs)	~100–200 MB	~300–400 MB	~400–600 MB
System responsiveness	Moderate-Low	Moderate	High (minimal lag)

The results confirmed that while lower-end systems experienced noticeable slowdowns and higher CPU loads, mid and high-end configurations maintained stable performance during the memory forensic and log analysis operations. Table 4 shows the comparative performance of 3 different system configurations.

High system responsiveness was observed on the 64-bit Windows 10 system equipped with 16 GB RAM and an Intel i3 dual-core processor. In contrast, the 32-bit Windows 7 system with 2 GB RAM and a Celeron 440 processor exhibited moderate to low responsiveness, while the 64-bit Windows 10 system with 8 GB RAM and an i3 dual-core processor showed moderate performance. These results suggest that the proposed approach is best suited for higher-end system configurations, where adequate memory and processing power ensure smooth execution of memory acquisition and analysis tasks.

4.3. Testing and Training

To validate the effectiveness of the proposed approach, we trained several machine learning models, including SVM, RF, Gaussian naive bayes, and decision tree classifier. The performance of these models was assessed using two key evaluation metrics: Accuracy and F-measure. The primary objective was to identify the most suitable model for malware classification based on these performance indicators.

For experimentation, two datasets were utilized: the CIC-MalMem-2022 dataset [31], which contains labeled malware samples, and a combination of the EVTX-ATTACK-SAMPLES dataset [24] (representing malicious events) and a publicly available Windows Event Log dataset [15] (representing non-malicious events).

The CIC-MalMem-2022 dataset comprises 58,596 instances across 57 features, including 29,231 benign and 28,831 malicious samples. The malicious samples are further categorized into three major classes: Ransomware, Spyware, and Trojan horse, making it a comprehensive resource for malware classification research.

a) *Data pre-processing*.

Standard pre-processing procedures were applied prior to model training. Invariant features those that did not contribute meaningful information to the classification task were eliminated. The remaining numerical features were normalized using min-max scaling to ensure a consistent value range across all inputs. Additionally, the categorical class labels were label-encoded, assigning '0' to benign samples and '1' to malicious samples. The dataset was complete and free of missing values, eliminating the need for imputation.

b) *Evaluation criteria*.

To evaluate the effectiveness of the classifiers, a set of standard performance metrics was employed, including the confusion matrix, accuracy, precision, recall, and F1-score. Each of these metrics provides a unique perspective on different aspects of model performance and is defined as follows:

- Confusion matrix: a tabular representation that summarizes the predicted outcomes versus the actual outcomes for a binary classification task. It comprises four categories:
    - *True Positives (TPs)*: malware instances correctly classified as malware.
    - *True Negatives (TNs)*: benign instances correctly classified as benign.
    - *FPs*: benign instances incorrectly classified as malware.
    - *FNs*: malware instances incorrectly classified as benign.
- Accuracy: represents the ratio of correctly predicted instances (both benign and malicious) to the total number of instances. It is a general measure of overall model performance.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

- Precision: indicates the proportion of correctly identified malware samples among all instances that were predicted as malware. It reflects the model's ability to avoid FPs.

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

- Recall (sensitivity): measures the model's ability to correctly identify actual malware instances. It is the proportion of true malware samples correctly classified as such.

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

- F1-score: a harmonic mean of precision and recall, providing a balanced measure of the classifier's ability to detect malware, particularly when the dataset has imbalanced class distributions.

$$F1 - Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4}$$

Table 5 presents a detailed comparative evaluation of each classifier, reporting the values for accuracy, precision, recall, and F1-score. All performance metrics range from 0 to 1, with values closer to 1 indicating a higher degree of predictive accuracy and reliability. These metrics collectively demonstrate each model's capability to distinguish between benign and malicious instances effectively. Figures 11 and 12 shows the confusion matrix and ROC curve obtained from the malware dataset, respectively.

Table 5. Evaluation of classifier on malware dataset.

Classifier name	Accuracy	Precision	F1-score	Recall
SVM	0.9956	0.99	1.00	1.00
RF classifier	0.9999	1.00	1.00	1.00
Naive bayes classifier	0.9919	0.99	1.00	1.00
Decision tree classifier	0.9992	0.99	1.00	1.00

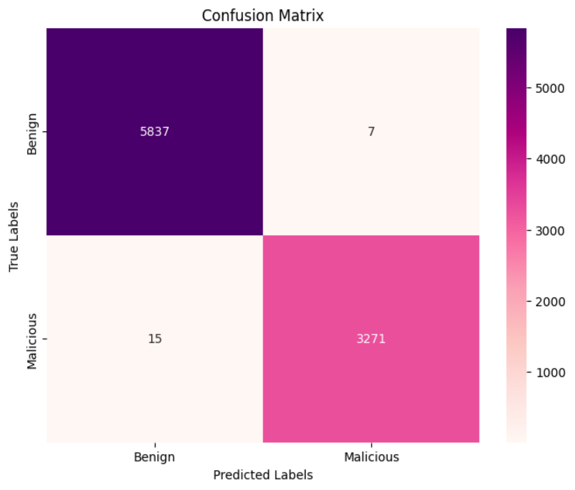
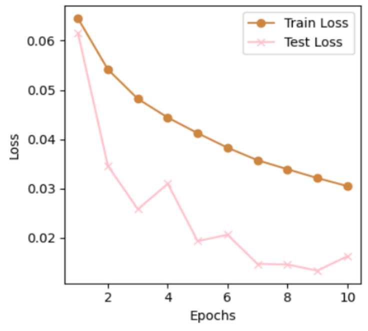
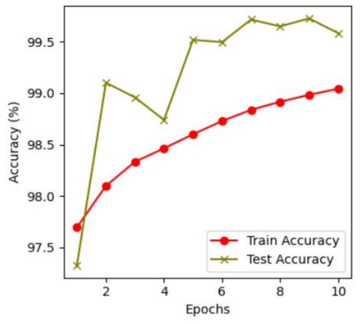


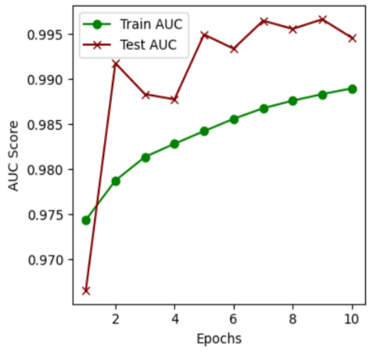
Figure 11. Confusion matrix for malware dataset.



a) Training and testing losses.



b) Training and testing accuracies.



c) Training and testing AUC scores.

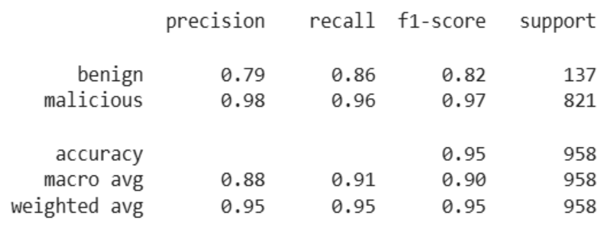
Figure 12. ROC curve on malware dataset.

The results of this study demonstrate the superior performance of the RF classifier in accurately detecting and classifying malware within the CIC-MalMem-2022 dataset. This finding underscores the potential of RF as a reliable model for enhancing cybersecurity systems by facilitating the development of more effective and resilient malware detection mechanisms.

Table 6. Statistics of evtx_attack_samples dataset.

Malicious Event	**In % of dataset**
Execution	13
Persistence	9
DefenseEvasion, execution	7
DefenseEvasion	12
Initial Access	1
NA	1
Discovery	1
Privilege escalation	21
Credential access	11
Lateral movement	16
Command and control	4

Additionally, the EVTX-ATTACK-SAMPLES dataset, comprising 4,633 rows and 326 columns, contains a wide range of Event Log entries associated with malicious activities. These entries capture evidence of various attack behaviors recorded during system execution. Table 6 presents the statistical distribution of key malicious event types identified within this dataset.



```
              precision    recall  f1-score   support

      benign       0.79      0.86      0.82       137
   malicious       0.98      0.96      0.97       821

    accuracy                           0.95       958
   macro avg       0.88      0.91      0.90       958
weighted avg       0.95      0.95      0.95       958
```

Figure 13. Accuracy of RF classifier.

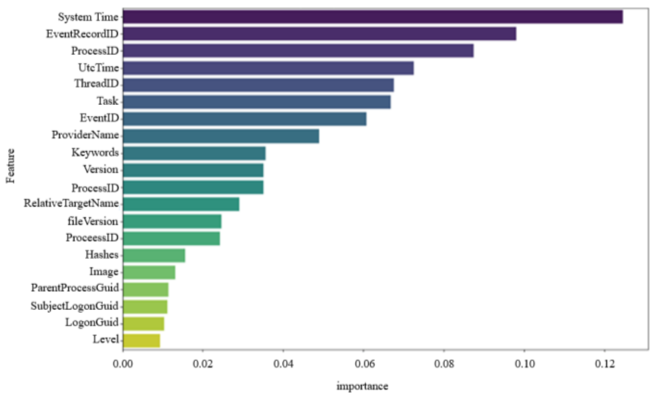
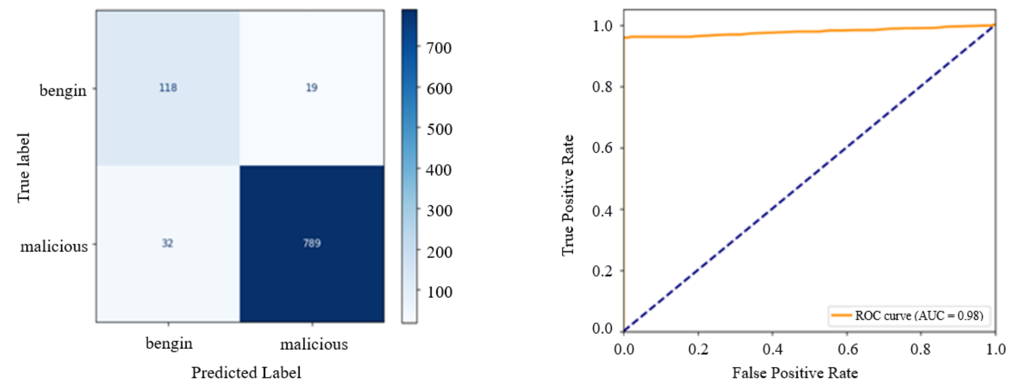


Figure 14. Feature importance.

For effective model training and evaluation, the EVTX_ATTACK_SAMPLES dataset [24] was merged with the Windows Event Log dataset [15], resulting in a preprocessed combined dataset containing 6,960 rows and 338 features. The merged dataset was then partitioned into 60% for training and 40% for testing. A RF classifier was employed to build the malware detection model. The model was trained on the training subset and subsequently evaluated on the test subset. To assess the model's performance comprehensively, key evaluation metrics were utilized, including accuracy, precision, recall, F1-score, and the confusion matrix. Figure 13 shows the accuracy of the RF classifier on event dataset. Figure 14 shows the feature importance.

The RF classifier demonstrated strong overall performance, achieving an accuracy of 95%, precision of 88%, recall of 91%, and an F1-score of 90%. These results indicate that RF is highly effective in distinguishing between benign and malicious events with a high degree of reliability. Its robust performance can be attributed to its ensemble-based architecture, which minimizes overfitting and enhances generalization. The model also offers a well-balanced trade-off between precision and recall, making it a reliable choice for classification tasks where computational cost is acceptable.

In comparison, the decision tree classifier, known for its simplicity and interpretability, yielded slightly lower performance metrics, with an accuracy of 93%, precision of 86%, recall of 87%, and an F1-score of 86%. These results highlight a modest decline in detection capability, underscoring the trade-off between model complexity and classification effectiveness.



a) Confusion matrix (binary). b) ROC curve (binary).

Figure 15. Confusion matrix and ROC curve on event dataset.

The findings from the event-based analysis further support the superior performance of the RF model in accurately detecting both malicious and non-malicious events within the combined dataset. Figure 15 shows the confusion matrix and ROC curve obtained from the merged event dataset. Table 7 provides a comprehensive evaluation of each classifier based on key performance metrics, including accuracy, precision, recall, and F1-score, on the merged dataset comprising malicious and benign Event Logs.

Table 7. Evaluation of classifier on event dataset.

Classifier name	Accuracy	Precision	F1-score	Recall
SVM	0.91	0.83	0.81	0.79
RF classifier	0.95	0.88	0.90	0.91
Naive bayes classifier	0.81	0.75	0.76	0.84
Decision tree classifier	0.93	0.86	0.86	0.87

By analyzing and correlating the outcomes of training and testing phases on both malware code datasets and Windows Event Log datasets, the study demonstrates that the RF classifier exhibits high predictive accuracy in identifying the presence of malware on a computer system.

4.4. Impact of False Positives and True Negatives in Malware Detection

Accurate malware detection is paramount for maintaining system security and preserving user trust. Two critical performance outcomes FPs and TNs have significant implications in practical deployment scenarios.

a) *Impact of FPs*.

FPs occur when benign files or behaviours are incorrectly classified as malicious. Although such errors do not reflect a real threat, they can severely affect user experience and system operations. High FP rates may result in:

- *Operational disruptions*: legitimate applications may be blocked or quarantined, affecting critical system functions and user workflows.
- *Decreased user trust*: users may lose confidence in the detection system, particularly when frequent alerts turn out to be non-malicious, leading to alert fatigue and the potential neglect of real threats.
- *Resource wastage*: investigating false alarms consumes time and resources, both for automated systems and human analysts.

In contrast, state-of-the-art tools like traditional signature-based antivirus software often exhibit relatively low FP rates due to their reliance on known patterns. However, this comes at the cost of limited detection capability for zero-day and obfuscated threats, as such tools fail to generalize effectively to new malware variants.

b) *Importance of TNs*.

TNs represent correctly identified non-malicious instances. While often overlooked in performance reporting, a high TN rate is essential for:

- *Ensuring system stability*: allowing legitimate processes to execute without interruption preserves the integrity and usability of the system.
- *Reinforcing user confidence*: consistently correct classifications build trust in the detection mechanism.
- *Balancing sensitivity and specificity*: high TN rates indicate good specificity, a necessary balance to high

sensitivity TPR to prevent overwhelming the system with false alarms.

State-of-the-art machine learning-based malware detectors often aim to increase sensitivity, sometimes at the expense of specificity, leading to higher FP rates. The proposed hybrid approach in this study, which integrates memory-based evidence with Windows Event Log analysis, aims to reduce FPs while maintaining high TN and TPRs by leveraging contextual behavioural information.

While maximizing detection accuracy is vital, the balance between FPs and TNs determines the real-world usability and reliability of a malware detection system. Compared to existing tools, our proposed approach offers a more nuanced and behaviour-aware detection mechanism, potentially reducing FPs and improving overall user trust without compromising detection capability.

5. Conclusions

The paper described an approach to detect the code section of the running process in the main memory. The various memory structures to locate the code section is identified and their relationship is established. An approach to detect the presence of malware in the main memory is proposed, which is based on the events associated with the malware execution on the computer system along with the suspicious code detected in the main memory. A software architecture based on the detection of the suspicious code in the main memory and the event associated with the malware has been proposed to detect malware presence in the main memory

If the antivirus software is unable to detect malicious code running in the main memory, then the presence of malicious code can be detected by the hybrid approach. The proposed approach has been validated by carrying out the experimentation. The training and testing carried out specified that the hybrid approach gives good accuracy for the detection of the malware using RF classifier.

Future Work Direction

The primary goal of the proposed approach is to accurately detect the presence of malware by extracting executable code from the system's main memory, comparing the code against a curated database of known benign and malicious samples, and correlating it with pertinent Event Log entries. To extend this research into a practical tool, a modular development plan has been envisioned. The development process of the proposed tool will be executed in multiple structured phases, each elaborated as follows.

a) The initial phase involves designing the system architecture, selecting Windows as the target platform for prototype implementation, and identifying essential data sources namely memory dumps and system-generated Event Logs. A critical step in this phase is the creation of a comprehensive code signature database. Malicious code samples are to be sourced from open repositories such as VirusShare and MalwareBazaar, while benign software is collected from trusted platforms like GitHub and SourceForge.
b) Next, the memory acquisition and analysis module are to be developed. System memory will be captured using forensic tools such as DumpIt or WinPMEM, and analyzed using Volatility3. A custom Volatility3 plugin will be implemented to detect the code sections of running processes and extract the corresponding executable code. The code extracted is then hashed using algorithms like SHA-256 to generate unique hash for comparison.
c) In parallel, an Event Log analysis module is to be implemented. Logs are to be collected using tools such as wevtutil or the Windows Event Log API, with a focus on security-relevant events including process creation (Event ID 4688), user logon (4624), PowerShell execution (4104), and service installation (7045). These logs are filtered, normalized, and structured to enable efficient querying and correlation.
d) A correlation engine is to be built to link memory-resident code artifacts specifically, the hash of contents in code section from running processes with relevant Event Log entries. This is achieved by aligning shared metadata such as timestamps and Process Identifiers (PIDs), enabling the reconstruction of execution traces indicative of malicious behavior. A comparison engine will subsequently matche these hashes against the precompiled code signature database. Optionally, machine learning models trained on labeled datasets of malicious and benign code may be integrated to improve classification accuracy.
e) To improve usability, a web-based interface will be developed using frameworks such as Flask (backend) and React (frontend). This interface will enable remote invocation of the memory acquisition and analysis module, Event Log analysis, correlation engine, and comparison engine in a sequential and automated manner. Dashboards will present insights on suspicious processes, memory-event correlations, and detection outcomes. Reporting features will support export in formats such as JSON and PDF, facilitating integration with broader incident response workflows.
f) The final phase will involve comprehensive testing in controlled sandbox environments, using both known malware samples and legitimate applications. This testing will assess detection accuracy, processing efficiency, and the tool's impact on system resources.

In order to minimize the latency between malware execution and detection, the efficiency of key tasks such as locating and extracting code sections from memory, performing log analysis, and executing correlation mechanisms must be improved. Techniques such as multithreading can be employed to perform tasks in parallel, including the detection of code sections from running processes and the identification of malicious events. This parallelization will significantly enhance the responsiveness of the system, enabling more timely alerts and improving the overall effectiveness of real-time detection.

Individuals with malicious intent often deploy malware to steal sensitive information, disrupt system functionality, and conduct various other harmful activities. Traditional antivirus solutions are limited in their effectiveness against newly emerging malware strains, as they rely on frequent updates to their signature databases for detection. The proposed approach aims to address this limitation by enabling the detection of malware based on memory-based analysis and event-based analysis, thereby providing a more robust and proactive defense mechanism independent of malware signature updates.

References

[1] Ahlegren F., Local and Network Ransomware Detection Comparison, Bachelor Thesis, Blekinge Institute of Technology, 2019. http://www.diva-portal.org/smash/get/diva2:1333153/FULLTEXT02.pdf

[2] Ahmed W. and Aslam B., "A Comparison of Windows Physical Memory Acquisition Tools," *in Proceedings of the IEEE Military Communications Conference*, Tampa, pp. 1292-1297, 2015. https://ieeexplore.ieee.org/document/7357623

[3] Akbanov M., Vassilakis V., and Logothetis M., "Ransomware Detection and Mitigation Using Software-Defined Networking: The Case of WannaCry," *Computers and Electrical Engineering*, vol. 76, pp. 111-121, 2019. https://doi.org/10.1016/j.compeleceng.2019.03.012

[4] Amanowicz M. and Jankowski D., "Detection and Classification of Malicious Flows in Software-Defined Networks Using Data Mining Techniques," *Sensors*, vol. 21, no. 9, pp. 1-24, 2021. https://doi.org/10.3390/s21092972

[5] Baker K., CrowdStrike, 10 Malware Detection Techniques, https://www.crowdstrike.com/en-us/cybersecurity-101/malware/malware-detection/, Last Visted, 2025.

[6] Baker K., CrowdStrike, History of Ransomware, https://www.crowdstrike.com/cybersecurity-101/ransomware/history-of-ransomware, Last Visited, 2025.

[7] Beck C., Boumezoued A., Cherkaoui Y., Pradat E., and Fleisher B., "Modeling Financial Losses from a Ransomware Attack Using a Causal Approach," Milliman White Paper, 2023. https://www.milliman.com/en/insight/modeling-financial-losses-from-ransomware-attack

[8] Celdran A., Sanchez P., Castillo M., Bovet G., and et al., "Intelligent and Behavioral-based Detection of Malware in IoT Spectrum Sensors," *International Journal of Information Security*, vol. 22, no. 3, pp. 541-561, 2023. https://doi.org/10.1007/s10207-022-00602-w

[9] Cyber5w, Windows Event Log Analysis, https://blog.cyber5w.com/eventlog-analysis, Last Visited, 2025.

[10] Damodaran A., Troia F., Visaggio C., Austin T., and Stamp M., "A Comparison of Static, Dynamic, and Hybrid Analysis for Malware Detection," *Journal of Computer Virology and Hacking Techniques*, vol. 13, pp. 1-12, 2017. https://doi.org/10.1007/s11416-015-0261-z

[11] GitHub, Process-Hollowing Executables, 2016, https://github.com/m0n0ph1/Process-Hollowing/tree/master/executables, Last Visited, 2025.

[12] Hossain M. and Islam M., "Enhanced Detection of Obfuscated Malware in Memory Dumps: A Machine Learning Approach for Advanced Cyber Security" *Cybersecurity*, vol. 7, pp. 1-23, 2024. https://doi.org/10.1186/s42400-024-00205-z

[13] JPCERT, Event Log Talks a Lot: Identifying Human-Operated Ransomware through Windows Event Logs, https://blogs.jpcert.or.jp/en/2024/09/windows.html, Last Visited, 2025.

[14] Kalinkin A., Golub S., Korkin I., and Pyatovskiy D., "Ransomware Detection Based on Machine Learning Models and Event Tracing for Windows," *IT Security*, vol. 29, no. 3, pp. 82-93, 2024. DOI: 10.26583/bit.2022.3.07

[15] Katara M., Kaggle, Windows Event Log Dataset, https://www.kaggle.com/datasets/mehulkatara/windows-event-log, Last Visited, 2025.

[16] Mahanta R. and Kumar R., "Utilizing Windows Event Logs for Malware Detection Using Machine Learning," *IET Conference Proceedings*, vol. 2024, no. 23, pp. 19-27, 2024. https://doi.org/10.1049/icp.2024.4396

[17] Maniriho P., Mahmood A., and Chowdhury M., "MeMalDet: A Memory Analysis-based Malware Detection Framework Using Deep Autoencoders and Stacked Ensemble Under Temporal Evaluations," *Computers and Security*, vol. 142, pp. 103864, 2024. https://doi.org/10.1016/j.cose.2024.103864

[18] Mohamed K. and Azher M., "Malware Detection Techniques," *in Proceedings of the 4th Novel Intelligent and Leading Emerging Sciences*

Conference, Giza, pp. 349-353, 2022. https://ieeexplore.ieee.org/abstract/document/9942395

[19] Moskovitch R., Feher C., Tzachar N., and Berger E., and et al., "Unknown Malcode Detection Using OPCODE Representation," *in Proceedings of the European Conference on Intelligence and Security Informatics*, Esbjerg, pp. 204-215, 2008. https://doi.org/10.1016/j.cose.2018.11.001

[20] Nguyen P., Huy T., Tuan T., Trung P., and Long H., "Hybrid Feature Extraction and Integrated Deep Learning for Cloud-based Malware Detection," *Computers and Security*, vol. 150, pp. 104233, 2025. https://doi.org/10.1016/j.cose.2024.104233

[21] Pot J., Digital Trends, Windows 10 Leaps Ahead of 7 among Steam Gamers, 2016, https://www.digitaltrends.com/computing/steam-users-windows-10-market-share/, Last Visited, 2025.

[22] Reshma Sri T. and Kumar Yogi M., "An Investigative Study on Malware Signatures," *Journal of Information Security System and Cyber Criminology Research*, vol. 1, no. 2, pp. 20-29, 2024. https://matjournals.net/engineering/index.php/JoISSCCR/article/view/615

[23] Santangelo G., Colacino V., and Marchetti M., "Analysis, Prevention and Detection of Ransomware Attacks on Industrial Control Systems," *in Proceedings of the International Symposium on Network Computing and Applications*, Boston, pp. 1-5, 2021. https://ieeexplore.ieee.org/document/9685713

[24] Sbousseaden, GitHub, EVTX_ATTACK_SAMPLES, https://github.com/sbousseaden/EVTX-ATTACK-SAMPLES, Last Visited, 2025.

[25] Shamshirsaz B., Asghari S., and Marvasti M., "An Improved Process Supervision and Control Method for Malware Detection," *The International Arab Journal of Information Technology*, vol. 19, no. 4, pp. 652-659, 2022. https://doi.org/10.34028/iajit/19/4/9

[26] Shaukat K., Luo S., and Varadharajan V., "A Novel Deep Learning-based Approach for Malware Detection," *Engineering Applications of Artificial Intelligence*, vol. 122, pp. 106030, 2023. https://doi.org/10.1016/j.engappai.2023.106030

[27] Singh P., Kaur S., Sharma S., Sharma G., and et al., "Malware Detection Using Machine Learning," *in Proceedings of the International Conference on Technological Advancements and Innovations*, Tashkent, pp. 11-14, 2021. https://ieeexplore.ieee.org/abstract/document/9673465

[28] Sophos, Interesting Windows Event IDs-Malware/General Investigation, https://support.sophos.com/support/s/article/KBA-000006797?language=en_US, Last Visited, 2025.

[29] Subedi K., Budhathoki D., and Dasgupta D., "Forensic Analysis of Ransomware Families Using Static and Dynamic Analysis," *in Proceedings of the Security and Privacy Workshops*, San Francisco, pp. 180-185, 2018. https://ieeexplore.ieee.org/document/8424649

[30] Ucci D., Aniello L., and Baldoni R., "Survey of Machine Learning Techniques for Malware Analysis," *Computers and Security*, vol. 81, pp. 123-147, 2019. https://doi.org/10.1016/j.cose.2018.11.001

[31] UNB, Malware Memory Analysis CIC-MalMem-2022, https://www.unb.ca/cic/datasets/malmem-2022. html, Last Visited, 2025.

[32] Vehabovic A., Ghani N., Bou-Harb E., Crichigno J., and Yayimli A., "Ransomware Detection and Classification Strategies," *in Proceedings of the IEEE International Black Sea Conference on Communications and Networking*, Sofia, pp. 316-324, 2022. https://ieeexplore.ieee.org/document/9858296

Dinesh Patil is an Associate Professor of Computer Engineering at Vidyavardhini's College of Engineering and Technology, Vasai. He received his PhD in Computer Engineering from Mumbai University, Mumbai, India in 2020. His research interests include: Digital Forensics and Computer Security.

Akshaya Prabhu is an Assistant Professor in the Department of Artificial Intelligence and Machine Learning at D. J. Sanghvi College of Engineering, Mumbai. She received her M.E. in Computer Engineering from Savitribai Phule Pune University (SPPU), Pune, in 2015. Her research interests include: Medical Image Processing, Deep Learning and AI.