A Temporal Study on Memory Forensics Artifacts Extraction: the Volatility Reliability

Aurora Arrus[1,2,*,†], Silvia Lucia Sanna[1,†], Antonio Aracri[1] and Giorgio Giacinto[1,3]

[1]University of Cagliari, Italy
[2]IMT School for Advanced Studies, Lucca, Italy
[3]Consorzio Interuniversitario Nazionale per l'Informatica (CINI), Italy

Abstract

Memory forensics is a crucial branch of digital investigation focusing on the acquisition and analysis of a system's volatile memory to extract valuable artifacts such as encryption keys, process traces, and in-memory payloads. It is particularly relevant in malware analysis, where volatile data often reveals behaviors and capabilities invisible to traditional disk-based methods. Over the years, numerous acquisition and analysis tools have been developed ranging from full memory imaging to targeted process extraction. Different frameworks have been developed, like Volatility serving as the standard for artifact retrieval and interpretation. However, despite its widespread adoption, little attention has been devoted to understanding how acquisition timing and system dynamics influence the consistency and reliability of extracted artifacts. This paper presents a systematic evaluation of Volatility's performance across multiple temporal memory acquisitions on both Windows and Linux systems. By repeatedly capturing system memory under controlled conditions and comparing the artifacts recovered over time, we quantify inconsistencies in file and process recovery. When known artifacts fail to appear in Volatility's structured output, additional analysis is performed through manual inspection and carving techniques to assess their persistence in raw memory. The results reveal that volatile memory extraction is inherently non-atomic, leading to temporal inconsistencies and partial artifact loss due to ongoing system activity and page overwrites. These findings highlight fundamental challenges in volatile memory forensics and underscore the need for improved acquisition atomicity, enhanced Linux support, and standardized evaluation methods to ensure reliability in forensic investigations and malware analysis.

Keywords

Memory Forensics, Memory Analysis, Volatility Reliability, Artifacts Retrieval, Memory Artifacts

1. Introduction

Malware analysis refers to the study of malicious programs to determine their functionality, behavior, and potential impact, and it has become increasingly essential for protecting systems and organizations against evolving cyber threats. Over the years, a wide range of analysis techniques has been developed, broadly categorized into *static* and *dynamic* approaches, depending on whether the malware is examined without or with execution, respectively [1]. More recently, Artificial Intelligence (AI) driven methods have been introduced to enhance detection speed and accuracy, leveraging Machine Learning (ML) and Deep Learning (DL) models for automated classification [2]. However, several studies have demonstrated that AI-based systems can be unreliable in adversarial settings, as malware can intentionally manipulate features or behavior to evade ML/DL detectors [3, 4, 5]. In parallel, increasingly stealthy malware categories, (e.g. obfuscated malware, stegomalware, and fileless malware) continue to emerge, further complicating detection [6, 7, 8, 9].

To address these challenges, recent research has proposed detection strategies grounded in *memory forensics*, which involve extracting and analyzing the volatile memory of a system or a specific target process, often using tools such as `procdump` or `fridump` [10]. The resulting memory dumps can

Joint National Conference on Cybersecurity (ITASEC & SERICS 2026), February 09-13, 2026, Cagliari, IT
[†]These authors contributed equally.
aurora.arrus@unica.it (A. Arrus); silvial.sanna@unica.it (S. L. Sanna); MAIL.Antonio@unica.it (A. Aracri); giorgio.giacinto@unica.it (G. Giacinto)
0000-0002-0877-7063 (A. Arrus); 0009-0002-8269-9777 (S. L. Sanna); 0000-0002-9421-8566 (A. Aracri); 0000-0002-5759-3017 (G. Giacinto)

then be examined through various techniques, including image-based representations [11], pattern or string extraction [10], or dedicated memory-forensics frameworks such as `Volatility` [12]. These methodologies have been applied across heterogeneous platforms, including desktop and mobile systems running Windows, Linux, or Android [10].

`Volatility` is the memory analysis tool most used worldwide that allows to monitor processes, files, Operating System (OS) interaction, network connection at user and kernel level at the time of capturing the complete RAM of the target device. Based on the authors' prior experience in memory forensics malware analysis, Digital Forensics investigations, teaching activities, and Capture The Flag (CTF) competitions, it has been observed that `Volatility` exhibits inconsistent behavior in extracting and recovering the files of interest, with certain executions resulting in no recoverable file despite its confirmed presence in memory. For this reason, an over-time analysis was performed, beginning with dumping the memory of Windows and Linux systems immediately after startup, with no user interaction with the system. Subsequently, specific files in the system (i.e. text documents, images) were opened, and additional memory dumps were collected while such artifacts remained active in memory. It was observed that some OS-based activities, programs and tasks appear and disappear from the list of running processes retrieved by `Volatility` when comparing memory dumps acquired at different points in time while the process has not been altered by the user. This behaviour is observed also in user files. In order to understand the nature of this pattern and to assess if this depends on the OS management or to `Volatility` capabilities, the opened user artifacts were retrieved through carving tools and techniques.

An additional layer of complexity arises from how volatile memory behaves during extraction. The acquisition of volatile memory is not atomic: RAM is captured page by page while the OS continues to allocate, free, and overwrite memory regions. As a result, different portions of the dump represent different temporal states of the system. Pages may be overwritten by background tasks, fragmented across non-contiguous memory regions, or swapped to disk. The OS may also reclaim memory used by a process even if the user still sees the file as "open", causing the corresponding artifact to disappear from the dump. This introduces inconsistencies, e.g. residual data from previous processes, incomplete structures, missing regions, or corrupted mappings. Hence, the extraction of meaningful evidence is complicated. In some cases, an artifact exists in memory but is impossible to reconstruct because only part of its content was captured. In other cases the tool reports no results due to the misalignment between page tables and actual memory pages. Therefore, failures in file recovery are not always attributable to the forensic tool, but may depend on the inherent volatility, fragmentation, and dynamic reuse of system memory during acquisition. Hence, we asked the following Research Questions:

- How complete and consistent are memory dumps acquired by current tools in Windows and Linux?
- Are failures in artifact recovery due to limitations of current forensic tool?
- How do OS-level mechanisms such as paging, caching, and memory reuse impact the integrity of captured data?

The main contribution of this paper is the study of artifact retrieval in different OS versions and distributions, under different conditions (i.e. files managed by the OS or by the user). This study advances the field of volatile memory forensics by providing a systematic and empirical assessment of acquisition reliability across different OS. The study introduces a cross-platform experimental framework that enables reproducible evaluation of memory forensic tools under different memory states and configurations. Through extensive testing on Windows and Linux systems, the study identifies critical inconsistencies in memory snapshot integrity (i.e. caused by non-atomic acquisition, OS-specific memory management, and tool-dependent behavior) that directly affect the recovery and reliability of forensic artifacts. The work further evaluates how paging, atomicity and OS memory management influence the completeness of captured data, exposing gaps in existing acquisition methodologies and artifacts retrieval and extraction. By analyzing the performance of widely used frameworks such as `Volatility` 2 and 3, it highlights how current tools may yield inconsistent or partial results depending on acquisition conditions. Finally, the study establishes methodological guidelines for standardized

testing and suggests improvements toward more reliable, OS-aware, and automation-assisted forensic tools, ultimately strengthening the foundation for trustworthy volatile memory acquisition and analysis.

The remainder of this paper is structured as follows. Section 2 introduces some important concepts on the OS memory management, while Section 3 describes the current methodologies and tools adopted for memory forensics. Our methodology is introduced in Section 4 and subsequent results are presented in Section 5. A discussion on future advancements and suggestions is presented in Section 6, while Section 7 closes the paper.

2. Background

Volatile memory (i.e. main memory, RAM) plays a critical role in Digital Forensics because it contains data that exists only while power is supplied and when the OS needs such data. When a system shuts down or loses power, its contents are lost. Therefore, memory forensics aims to capture a live snapshot of RAM during system operation. RAM is organized into fixed-size pages, which are mapped to physical memory using page tables. This design enables efficient allocation, swapping, and sharing of memory across processes through virtual memory management. However, because RAM's contents constantly change, the state captured in a memory dump strictly depends on the precise acquisition moment.

Memory acquisition is the foundational step in memory forensics. Acquisition can occur through hardware-based approaches using Direct Memory Access (DMA) devices that read physical memory directly, bypassing the OS and reducing the risk of interference by malware. Alternatively, software-based tools (e.g. WinPmem, LiME, OSXPmem, FTK Imager, Magnet Acquire, VBoxManage) run on the target system and capture memory using kernel drivers or system-level APIs while preserving data integrity through read-only access and cryptographic hashing. In both cases, the acquisition tools extract RAM page by page in ascending physical address order, which is efficient but not simultaneous. This leads to one of the fundamental difficulties of memory forensics: memory acquisition is *not atomic*. Because pages are copied sequentially while the OS and active processes continue to modify memory, different pages in the dump correspond to different points in time. As a result, memory dumps may contain outdated or overwritten data, or data belonging to processes that have terminated during acquisition. Residual data from previously freed memory pages can be misinterpreted and incomplete or inconsistent regions may appear, making certain analyses misleading or impossible. This problem, called *page smearing*, occurs when the page tables are recorded at one moment but the physical memory which they point is recorded later. The mismatch in timing can cause discrepancies between the system's view of memory and the actual contents of RAM. Additional sources of inconsistency include memory overwriting, memory fragmentation, and OS interference (e.g. process scheduling, page allocation, or memory paging). Anti-forensic techniques further increase complexity; rootkits or memory-wiping routines may actively modify or hide memory regions during acquisition. Page swapping and demand paging (i.e. pages are temporarily moved to disk or loaded into memory only when needed) can prevent acquisition of user-level activity such as browsing history, DNS records, email content or executable code relevant to malware analysis.

To interpret memory dumps and mitigate these issues, investigators rely on advanced memory forensic tools. The most widely adopted open-source framework is `Volatility` [13], which provides modular plugins capable of extracting detailed artifacts from Windows, Linux, and macOS systems, including processes, DLLs/shared objects, encryption keys, network sessions, and injected code. It also detects hidden or terminated processes and can carve residual data from partially overwritten memory. `Rekall` [14] provides deeper inspection and internal consistency checks, supporting live analysis and improved parsing of corrupted or non-linear memory structures. `Redline` [15], a commercial tool developed by FireEye, focuses on volatile data collection and integrates memory artifacts with system context to support incident response and threat hunting. Such modern tools incorporate heuristics, cross-validation between memory structures e.g. comparing kernel object tables and process lists, and data reconstruction techniques to reduce the effect of smearing, fragmentation, or overwritten pages. Virtualization further improves acquisition reliability: hypervisors such as `VirtualBox` allow memory

snapshots to be captured while pausing system execution, improving consistency than live acquisition.

Despite these advances, mitigating non-atomic acquisition remains an open challenge. Current research explores timestamp correlation across pages, profiling memory access patterns to identify overwritten or modified data, and applying AI/ML to detect anomalies within dumps [10]. However, these techniques may introduce new risks, such as susceptibility to adversarial manipulation. Post-acquisition validation tools help identify incomplete or unreliable memory regions and guide analysts toward trustworthy artifacts.

While Windows memory dumps are immediately supported by `Volatility`, Linux memory forensics introduces additional complexity. In Windows, `Volatility` includes pre-built kernel profiles. In contrast, Linux analysis requires the generation of a specific Linux profile, i.e. a file containing kernel symbol tables and DWARF debugging data describing internal data structures. As each Linux kernel version is unique, with its own structure definitions, `Volatility` must be provided with a matching profile to correctly parse memory dumps. Creating a profile requires extracting the kernel's System.map file and compiling a kernel module with debugging information. Furthermore, Linux acquisition tools such as `LiME` must be built specifically for the running kernel version and inserted into the kernel at runtime, introducing operational overhead and potential instability. As a result, Linux memory forensics is powerful but considerably more challenging due to kernel variability, profile generation requirements, and reliance on kernel-specific acquisition modules.

3. Related Works

Volatile memory forensics has become a central approach for digital investigations and malware analysis, as RAM contains execution-time artifacts that never appear on disk. Early foundational work defined the three criteria that characterize a high-quality forensic snapshot, i.e. correctness, atomicity, and integrity [16]. Acquisition tools frequently fail to capture the entire physical address space and sometimes produce misaligned dumps, with atomicity degrading as memory size increases [17]. General surveys reinforce that memory forensics exposes unique artifacts (e.g. cryptographic material, network connections, process injection residues, and unpacked malware payloads) that are impossible to obtain through filesystem analysis [18**?**]. In malware research, memory inspection has proven particularly effective for fileless and memory-resident malware, where malicious code exists only in memory [19]. Memory-based detection techniques reconstruct injected code, hollowed processes, or runtime-loaded payloads using frameworks such as `Volatility` and `Rekall` [13**?**]. More recently, researchers showed that malware can embed payloads inside images loaded in memory (stegomalware), and such payloads can be recovered only with memory forensics techniques [7].

Despite ongoing progress, unreliable acquisition continues to hinder the accuracy and completeness of memory forensics. Non-atomic page-by-page extraction leads to page smearing, while demand paging and swap activity cause missing or partial evidence [20]. On Windows 10, hypervisor-based protections such as Device Guard and the shift toward `swapfile.sys` limit memory access [20]. Linux introduces a different challenge: every kernel version requires a matching acquisition module and `Volatility` profile [21]. Thus, maintaining full kernel coverage at scale is operationally unrealistic [22]. Application-aware forensics remains underdeveloped as most frameworks lack plugins to extract applications' data, pushing investigators toward carving or manual triage [23]. Hybrid environments such as the Windows Subsystem for Linux (WSL) further complicate analysis because Linux data structures appear inside Windows memory, breaking single-OS assumptions [24].

Recent research has extended memory forensics into new domains: *(i)* network-centric reconstruction (e.g., `VolNet`) enables recovery of communication artifacts directly from RAM [25]; *(ii)* firmware-level acquisition studies examine which tools can capture memory even when protected by rootkits [26]; and *(iii)* dedicated frameworks for Android memory forensics have now emerged [27, 28, 29]. Notably, `VolMemDroid` [30] enables Android memory acquisition and analysis directly within `Volatility`, overcoming limitations of ad-hoc Android dumping scripts and advancing mobile memory forensics. Additional studies explore IoT memory acquisition [31], and AI-supported malware detection that

Figure 1: Methodology pipeline from VM creation to artifact analysis elaboration, including all five main steps: VM creation, VM memory acquisition, Volatility Analysis, Report Generation, Result Filtering

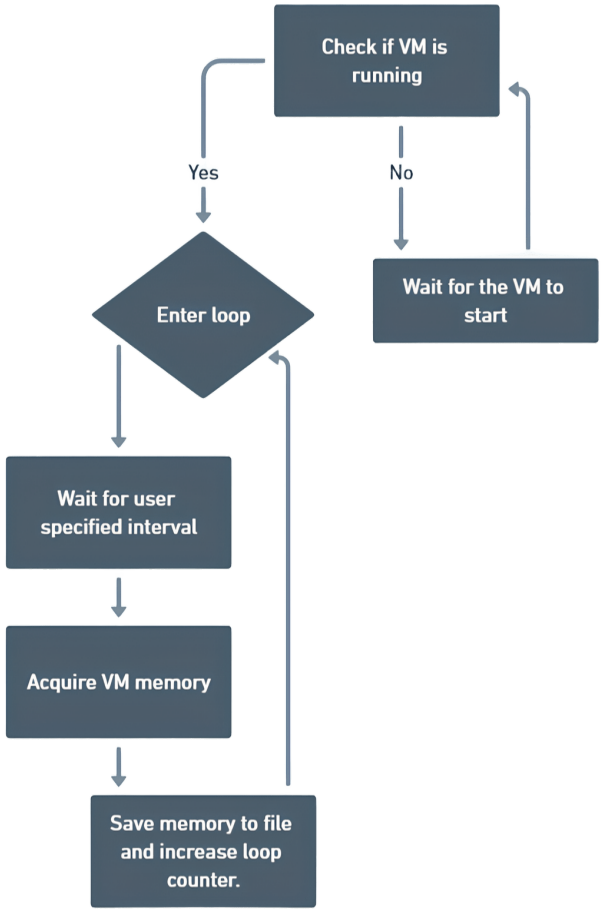


Figure 2: Acquisition pipeline for VM startup, user interaction and final memory dump. First check the VM status, if it is running the program enters a loop where it waits for the time interval specified by the user and then acquires the RAM. Otherwise, it waits for the specified VM to power on

operates on memory features rather than static files [11]. Across platforms, the persistent limitations (i.e. non-atomic capture, paging effects, kernel diversity, virtualization barriers, sparse application-aware plugins, and limited support for mobile/IoT systems) show that achieving a complete and consistent memory snapshot remains an unsolved problem. Our work contributes by *(i)* systematically assessing acquisition reliability across tools and OS versions, *(ii)* evaluating the effects of compression/encryption on recoverable evidence, and *(iii)* providing empirical evidence that supports the need for standardized acquisition methodology.

4. Methodology

This research adopts a systematic experimental and comparative methodology to evaluate the reliability and completeness of volatile memory acquisition and forensic analysis across different OS. The objective was to determine how acquisition tools, OS behaviors, and the inherent volatile RAM nature, influence the accuracy and reproducibility of digital evidence. The entire methodology, shown in Figure 1, was structured around four main stages: environment preparation, memory acquisition, forensic analysis, and comparative evaluation. Each phase was carefully designed to ensure repeatability, minimize environmental interference, and maintain data integrity.

The first stage involved the creation of controlled experimental environments. Virtualization was chosen as the experimental foundation because it provides a reproducible, isolated, and safe setting for forensic testing. Virtual Machines (VMs) were configured to emulate the behavior of different platforms, including Windows 7 Ultimate, Windows 10 Pro, and Kali Linux, e.g. using Oracle VirtualBox. Each VM was assigned uniform hardware specifications to guarantee comparability across experiments. Network connectivity was disabled to eliminate unpredictable traffic and background processes that could introduce unrelated volatile artifacts to this research focus. The use of virtualization also allows the creation of consistent system snapshots, enabling the repetition of identical tests under controlled conditions. These measures ensured that observed differences in results could be attributed to memory acquisition behavior rather than uncontrolled system variations.

The second stage focused on the acquisition of volatile memory, representing a critical step because it determines the quality and reliability of all subsequent analyses. Memory dumps were obtained while the systems were actively running to capture the state of processes, files, and system interactions as they existed in real time. The VirtualBox management utility (`VBoxManage debugvm dumpvmcore`) was employed to extract the contents of physical memory into an ELF-formatted dump. Acquisitions were repeated at different temporal intervals (i.e. 30 seconds, 1-5-10 minutes) to simulate varying system activity and observe the evolution of artifacts over time. This temporal dimension was essential for evaluating the effects of non-atomic acquisition, as memory is captured incrementally and different pages represent different system states. This approach enables the identification of inconsistencies due to process scheduling, memory overwriting, paging, and other OS activities that occur during the acquisition phase.

The third stage involved forensic analysis using the `Volatility` framework, one of the most widely adopted and extensible tools in memory forensics. `Volatility` plugins allow targeted extraction of specific forensic artifacts, such as running processes, open handles, kernel modules, network sessions, and user-level interactions. For Windows systems, plugins such as `windows.pslist`, `windows.filescan`, `windows.dumpfiles`, `windows.netscan` were used to reconstruct the state of the system and identify user activity. For Linux, plugins including `linux.pslist`, `linux.lsof`, `linux.proc.Maps`, `linux.lsmod` were used to identify process hierarchies, memory-mapped files, and loaded kernel modules. All extracted artifacts were exported into standardized text and HTML reports to support a rigorous comparison between acquisition instances and OS. This structured representation, in Figure 3, enabled correlational analysis between interdependent elements, such as matching open network connections with initiating processes, or associating memory-mapped files with extracted binaries or document remnants. The report format provides evidence suitable for peer review or judicial review, ensuring the transparency, reproducibility, and traceability of the forensic procedure.

To further enhance the analytical process, a dedicated report generation and result filtering phase was introduced to manage the volume and complexity of the comparative outputs. The initial HTML reports produced by the side-by-side comparison of plugin outputs included both unchanged and modified artifacts, resulting in large files that limited efficient manual inspection. To address this issue, a custom filtering procedure was applied to automatically extract only evidence-relevant differences while preserving contextual information, such as the associated `Volatility` plugin and the original report structure. By retaining visual markers and formatting while discarding redundant data, the filtered reports significantly reduced noise and improved readability. This step enables analysts to focus

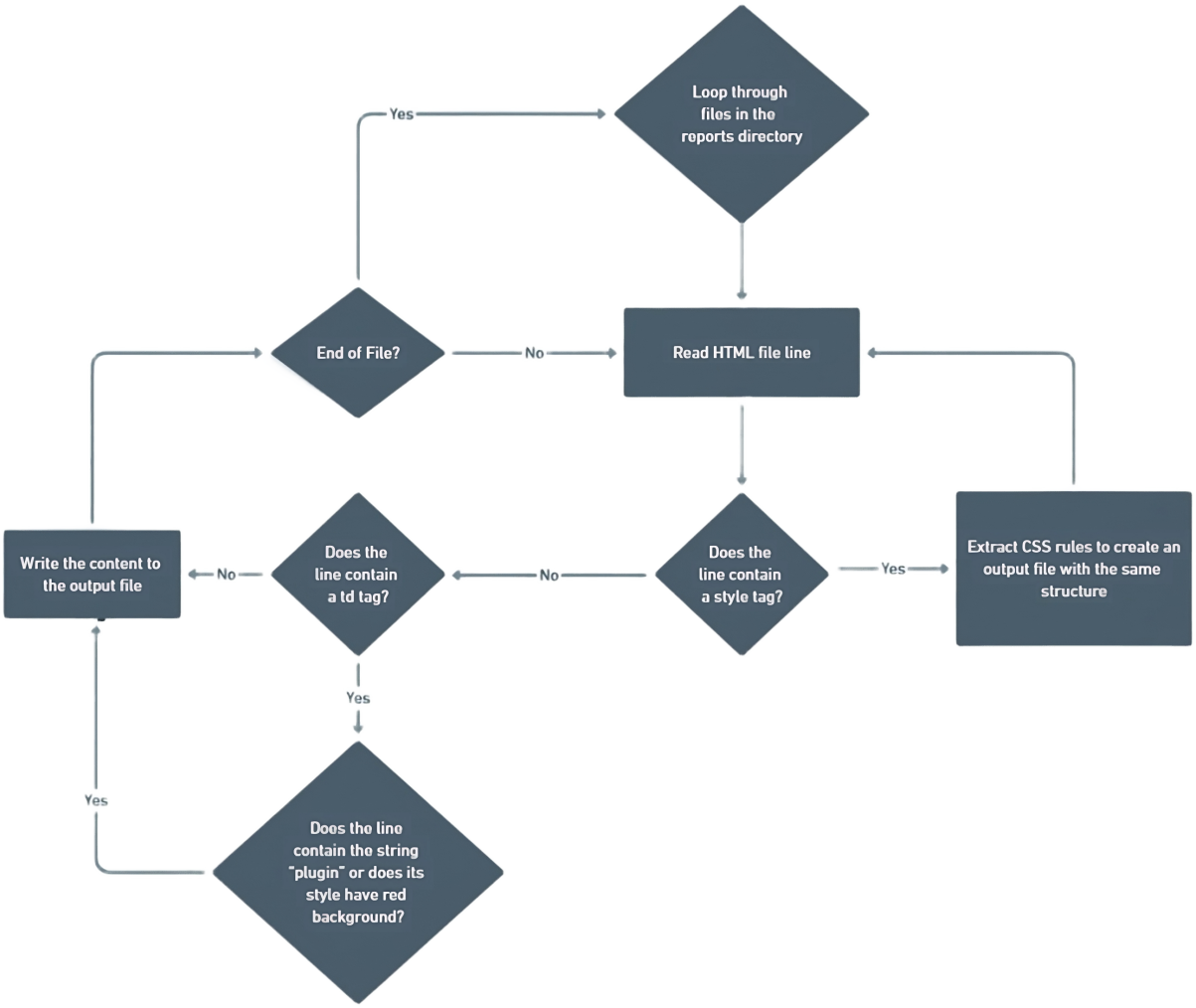


Figure 3: Result Filtering Diagram. Workflow on the report elaboration to filter evidence-related results. Loop through each file in the folder containing the reports. Read the content of each file on a line-by-line basis. Perform a series of checks and operations based on different conditions related to the content present in the read line

on meaningful temporal variations across memory snapshots, facilitating systematic comparison and reducing the risk of overlooking subtle but forensically relevant discrepancies.

To validate the forensic analysis, additional verification techniques were used. When `Volatility` failed to extract expected artifacts (e.g. files opened by the user or transient processes), manual inspection and file carving techniques were applied to the raw dumps using utilities like strings, grep, and dedicated carving tools. This dual-layer approach made it possible to distinguish between *(i)* artifacts genuinely lost due to the volatile nature of RAM and *(ii)* those that the tool failed to recognize due to parsing or profile limitations. The use of cross-validation between automated and manual methods increased the robustness of the results and provided insight into the tool's strengths and weaknesses.

The final stage involved comparative evaluation and interpretation. Each memory dump was independently analyzed and then compared with previous acquisitions to assess the persistence, structural consistency, and temporal stability of the artifacts. Differences in recovered processes, open files, and kernel modules between consecutive snapshots were used as indicators of inconsistency or incompleteness. The results were examined to quantify the frequency and type of discrepancies observed across OS, acquisition intervals, and tool. Qualitative assessments were also performed to interpret the potential causes of the observed differences (i.e. non-atomic acquisition, paging mechanisms, or tool limitations). This comparative approach allowed the study to correlate observed inconsistencies

with known challenges in memory forensics, such as page smearing, memory overwriting, and OS interference.

Through this multi-stage and repeatable methodology, the study provides a comprehensive framework for understanding the operational and technical challenges of volatile memory acquisition. It bridges theoretical aspects of memory forensics (e.g. atomicity, correctness, and integrity) with practical experimentation and cross-validation. The results generated through this methodology offer empirical evidence to support the improvement of acquisition tools and to advance the standardization of forensic procedures in volatile memory analysis.

5. Experimental Results

The experimental evaluation was carried out on three VM configured using Oracle VirtualBox: Windows 7 Ultimate SP1 (64-bit), Windows 10 Pro (64-bit), and Kali Linux 2023.1. Each system was provisioned with 4 GB of RAM and two CPU cores, executed in safe mode to reduce noise from background services, and isolated from network connectivity to prevent unsupervised updates or traffic. The adoption of virtualization ensured reproducible acquisition conditions, stability in resource allocation, and controlled rollback capabilities. Memory was acquired using hypervisor-level dumping procedures, and all resulting images were fully readable. However, while acquisition succeeded universally, the extracted evidentiary content varied markedly across platforms and over time, demonstrating that the fidelity and interpretability of volatile memory are influenced not only by the analysis tool but also by OS design, acquisition timing, and the inherent non-atomicity of memory capture.

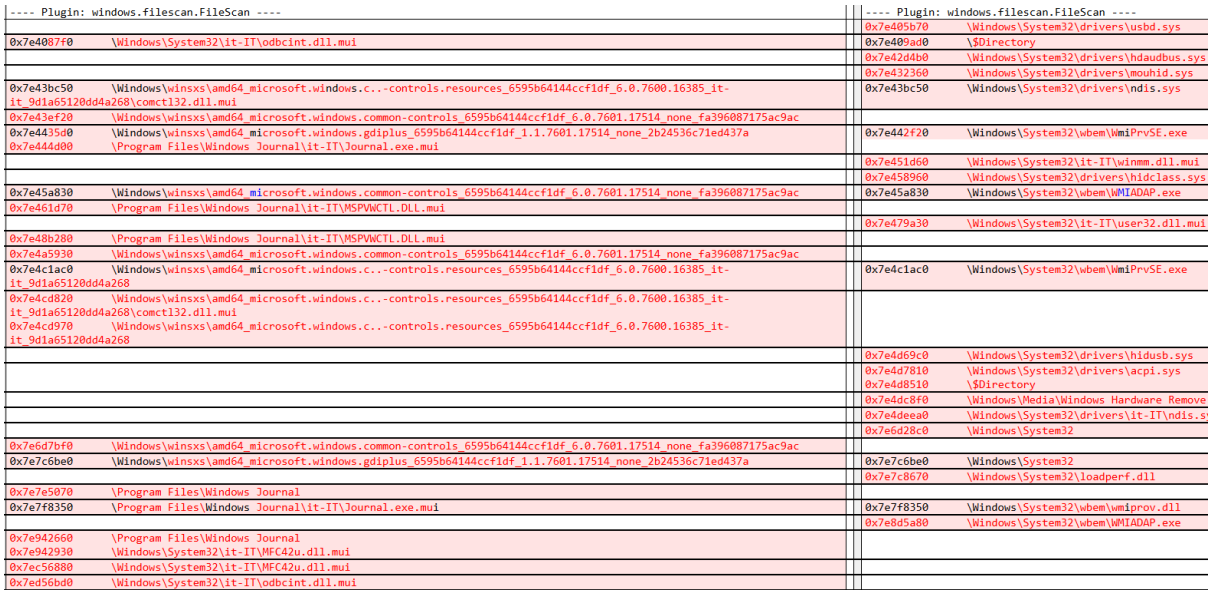
A clear divergence emerged in how each OS retained file artifacts during execution. Windows 7 consistently preserved complete document buffers in memory. The opened files remained reconstructable even after the application was close, reflecting a memory caching strategy that delays reclamation of user-level data. `Volatility`'s file scanning and dumping plugins repeatedly located intact structures and extracted binary content with full integrity. In contrast, Windows 10, despite exposing comparable metadata (e.g. file handles, filenames, object headers), frequently yielded incomplete or corrupted file payloads. Extractable content often exhibited overwritten segments, which implies that the presence of a file object in memory does not guarantee the continued retention of its readable content. These observations indicate a shift toward more rapid clearance or reallocation of cache buffers, consistent with modern Windows design policies prioritizing reduced exposure of sensitive memory regions. Kali Linux differed more radically. Even when documents were visibly opened by the user, attempts to use `Volatility` to extract file-backed memory regions failed to retrieve any substantive artifacts. File descriptors and process mappings could occasionally be identified, yet no recoverable buffers corresponding to user-level documents were present. Analysis of memory-mapped regions revealed that file contents were either transiently rendered into short-lived buffers, encrypted or handled externally, or purged immediately following demand fulfillment. This behaviour contrasts with legacy Windows caching, indicating that Linux minimizes long-term exposure of file data within RAM, thereby greatly reducing forensic visibility despite successful acquisition.

Temporal instability emerged as a dominant factor that affects forensic interpretation. Across all systems, early memory dumps captured immediately after system stabilization displayed the highest integrity, whereas dumps taken after prolonged interaction exhibited greater fragmentation or loss of artifacts. On Windows, running processes, open handles, and DLL structures remained largely reproducible using `Volatility` process-inspection plugins, with only minor discrepancies appearing between consecutive snapshots. Nonetheless, as the time between acquisitions increased, approximately $8\tilde{}15\%$ of processes were no longer consistently identified across dumps, reflecting scheduler variance and page-level overwriting. File-related artifacts showed even sharper degradation over time. If a document was opened and closed between acquisition cycles, its buffer persistence varied widely; in Windows 7 it frequently remained intact, whereas in Windows 10 it decayed rapidly or became partially overwritten. On Linux, the effects were more severe: even with accurate `Volatility` profiles, kernel diversity and minor build differences produced inconsistent parsing of task lists, module tables, and

```
---- Plugin: windows.filescan.FileScan ----

Volatility 3 Framework 2.8.0



Offset  Name

0x7e400050      \Windows\System32\LogFiles\WMI\RtBackup\EtwRTMsMpPsSession7.etl

0x7e401d20      \Windows\ShellNew\Journal.jnt

0x7e401e70      \Windows\System32\it-IT\advapi32.dll.mui

0x7e403bc0      \Users\antonio\AppData\Roaming\Microsoft\Windows\Recent\AutomaticDestinations\9ffcc573e4a429c.automaticDestinations-ms

0x7e406070      \Windows\System32\wscproxystub.dll

0x7e406660      \Windows\System32\werconcpl.dll

0x7e406bf0      \Windows\winsxs\amd64_microsoft.windows.c..-controls.resources_6595b64144ccf1df_6.0.7600.16385_it-
it_9d1a65120dd4a268\comctl32.dll.mui

0x7e407070      \Windows\System32\netprofm.dll

0x7e407a30      \Windows\System32\cryptbase.dll

0x7e4087f0      \Windows\System32\it-IT\odbcint.dll.mui
```

```
---- Plugin: windows.filescan.FileScan ----

Volatility 3 Framework 2.8.0



Offset  Name

0x7e400050      \Windows\System32\LogFiles\WMI\RtB

0x7e401d20      \Windows\ShellNew\Journal.jnt

0x7e401e70      \Windows\System32\it-IT\advapi32.d

0x7e403bc0      \Users\antonio\AppData\Roaming\Mic

0x7e405b70      \Windows\System32\drivers\usbd.sys

0x7e406070      \Windows\System32\wscproxystub.dll

0x7e406660      \Windows\System32\werconcpl.dll

0x7e406bf0      \Windows\winsxs\amd64_microsoft.wi
it_9d1a65120dd4a268\comctl32.dll.mui

0x7e407070      \Windows\System32\netprofm.dll

0x7e407a30      \Windows\System32\cryptbase.dll

0x7e409ad0      \$Directory
```

(a) Unfiltered comparison of two consecutive memory dumps, where differences highlight non-atomic acquisition and time-smeared artefacts.



```
---- Plugin: windows.filescan.FileScan ----
0x7e4087f0      \Windows\System32\it-IT\odbcint.dll.mui
0x7e43bc50      \Windows\winsxs\amd64_microsoft.windows.c..-controls.resources_6595b64144ccf1df_6.0.7600.16385_it-
it_9d1a65120dd4a268\comctl32.dll.mui
0x7e43ef20      \Windows\winsxs\amd64_microsoft.windows.common-controls_6595b64144ccf1df_6.0.7601.17514_none_fa396087175ac9ac
0x7e4435d0      \Windows\winsxs\amd64_microsoft.windows.gdiplus_6595b64144ccf1df_1.1.7601.17514_none_2b24536c71ed437a
0x7e444d00      \Program Files\Windows Journal\it-IT\Journal.exe.mui
0x7e45a830      \Windows\winsxs\amd64_microsoft.windows.common-controls_6595b64144ccf1df_6.0.7601.17514_none_fa396087175ac9ac
0x7e461d70      \Program Files\Windows Journal\it-IT\MSPVWCTL.DLL.mui
0x7e48b280      \Program Files\Windows Journal\it-IT\MSPVWCTL.DLL.mui
0x7e4a5930      \Windows\winsxs\amd64_microsoft.windows.common-controls_6595b64144ccf1df_6.0.7601.17514_none_fa396087175ac9ac
0x7e4c1ac0      \Windows\winsxs\amd64_microsoft.windows.c..-controls.resources_6595b64144ccf1df_6.0.7600.16385_it-
it_9d1a65120dd4a268
0x7e4cd820      \Windows\winsxs\amd64_microsoft.windows.c..-controls.resources_6595b64144ccf1df_6.0.7600.16385_it-
it_9d1a65120dd4a268\comctl32.dll.mui
0x7e4cd970      \Windows\winsxs\amd64_microsoft.windows.c..-controls.resources_6595b64144ccf1df_6.0.7600.16385_it-
it_9d1a65120dd4a268
0x7e6d7bf0      \Windows\winsxs\amd64_microsoft.windows.common-controls_6595b64144ccf1df_6.0.7601.17514_none_fa396087175ac9ac
0x7e7c6be0      \Windows\winsxs\amd64_microsoft.windows.gdiplus_6595b64144ccf1df_1.1.7601.17514_none_2b24536c71ed437a
0x7e7e5070      \Program Files\Windows Journal
0x7e7f8350      \Program Files\Windows Journal\it-IT\Journal.exe.mui
0x7e942660      \Program Files\Windows Journal
0x7e942930      \Windows\System32\it-IT\MFC42u.dll.mui
0x7ec56880      \Windows\System32\it-IT\MFC42u.dll.mui
0x7ed56bd0      \Windows\System32\it-IT\odbcint.dll.mui
```

```
---- Plugin: windows.filescan.FileScan ----
0x7e405b70      \Windows\System32\drivers\usbd.sys
0x7e409ad0      \$Directory
0x7e42d4b0      \Windows\System32\drivers\hdaudbus.sys
0x7e432360      \Windows\System32\drivers\mouhid.sys
0x7e43bc50      \Windows\System32\drivers\ndis.sys
0x7e442f20      \Windows\System32\wbem\WmiPrvSE.exe
0x7e451d60      \Windows\System32\it-IT\winmm.dll.mui
0x7e458960      \Windows\System32\drivers\hidclass.sys
0x7e45a830      \Windows\System32\wbem\WMIADAP.exe
0x7e479a30      \Windows\System32\it-IT\user32.dll.mui
0x7e4c1ac0      \Windows\System32\wbem\WmiPrvSE.exe
0x7e4d69c0      \Windows\System32\drivers\hidusb.sys
0x7e4d7810      \Windows\System32\drivers\acpi.sys
0x7e4d8510      \$Directory
0x7e4dc8f0      \Windows\Media\Windows Hardware Remove
0x7e4deea0      \Windows\System32\drivers\it-IT\ndis.s
0x7e6d28c0      \Windows\System32
0x7e7c6be0      \Windows\System32
0x7e7c8670      \Windows\System32\loadperf.dll
0x7e7f8350      \Windows\System32\wbem\wmiprov.dll
0x7e8d5a80      \Windows\System32\wbem\WMIADAP.exe
```

(b) Filtered report showing only mismatched entries, enabling clearer analysis of artefact persistence and disappearance.

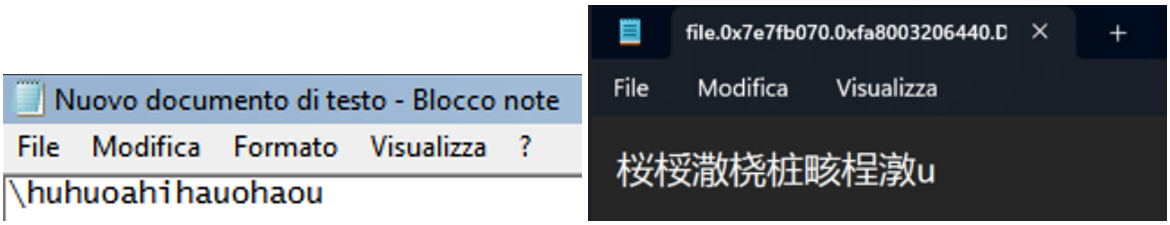
Figure 4: Comparison between raw and filtered RAM reports, demonstrating how `Volatility` affects evidence visibility and why structured filtering is required for forensic analysis.

memory maps. Some artifacts visible in raw dumps could not be reconstructed by automated plugins, indicating that discrepancies were often due to parser limitations rather than the absence of data. When LiME-style acquisition was simulated, parsing fidelity improved, but the requirement for version-locked kernel modules introduced practical constraints that hinder real-world scalability.

Although hypervisor-based dumping imposed minimal operational impact on the running systems, its sequential, page-oriented capture remained non-atomic. Different sections of the resulting memory images corresponded to slightly different system moments, resulting in time-smeared artifacts that reflect a composite of quick but independent states. Cross-validation using file carving tools such as `foremost` [1] and `bulk_extractor` [2] confirmed that `Volatility`'s structured output represented only a partial view of the available evidence. In several instances, fragments of user documents and binary data absent from `Volatility`'s reports were still recoverable through carving, albeit often with damaged

[1] https://www.kali.org/tools/foremost/
[2] https://www.kali.org/tools/bulk-extractor/

headers or misaligned offsets, consistent with partially overwritten pages. These findings demonstrate that automated memory analysis tools may under-represent the evidentiary content preserved in RAM and that memory carving remains necessary to reveal artifacts overlooked by structure-dependent extractors. Quantitatively, process recoverability reached between 80% and 95% for Windows systems and between 65% and 85% on Linux, while file artifact recovery fluctuated between 50% and 70% depending on workload, timing, and OS caching behaviour. The disparity between metadata persistence and content recoverability was particularly pronounced in Windows 10, where file structures persisted long after the readable content had been cleared. Additionally, plugin reliability varied significantly, particularly for complex kernel objects or hybrid architectures such as WSL, which produced ambiguous or unreadable `Volatility` output due to mixed memory semantics.



(a) Original written note (Windows 10) (b) Volatility retrieved note (Volatility 3)

Figure 5: Comparison of original text document artifact (left) and retrieved document from memory dump (right) in Windows 10 using Volatility 3

Collectively, these results show that the forensic reliability of volatile memory is constrained by three interdependent factors: *(i)* non-atomic capture, *(ii)* OS-driven memory reuse, and *(iii)* `Volatility` analysis tool. The absence of an artifact in memory cannot be interpreted as evidence that no corresponding user interaction occurred; instead, it may reflect intentional design choices that minimize data persistence. Older architectures, such as Windows 7, display behavior favourable to forensic analysis by maintaining caches beyond immediate use, while modern systems reduce their evidentiary footprints through rapid buffer clearance, increased abstraction, or restrictive memory mapping. Despite these constraints, substantial evidence still persists beyond the visibility of standard tools, motivating the development of acquisition and analysis frameworks capable of reconciling raw memory content with incomplete or ambiguous structural interpretation.

6. Discussion

The findings from the evaluation highlight the inherent complexity of achieving reliable and reproducible results in volatile memory forensics. Despite the evolution of acquisition tools and analysis frameworks such as `Volatility`, the fundamental limitation remains the non-atomic nature of memory capture, preventing investigators from obtaining a perfectly consistent system snapshot. Memory pages are acquired sequentially while the OS continues to execute background processes, perform paging operations, and update kernel structures. Consequently, different segments of a memory dump represent slightly different temporal states, introducing inconsistencies that manifest as missing, partially recovered, or misaligned artifacts during analysis. This structural dynamism explains why identical experiments can yield variable outcomes even under controlled conditions.

The results reinforce previous theoretical models that emphasize the importance of correctness, atomicity, and integrity in acquisition quality. As shown in the experiments, these three dimensions are closely interdependent: loss of atomicity inevitably affects correctness, whereas integrity is influenced by both system dynamism and tool intrusiveness. For example, although hypervisor-based acquisition via `VirtualBox` minimized interference, it did not eliminate smearing entirely because memory writes can still occur between page reading. Software-level tools, while more accessible, amplified the issue by interacting with the target OS kernel during acquisition, sometimes overwriting or locking memory

regions in the target process. These effects collectively reduce the evidential reliability of volatile data and must be considered during forensic interpretation, especially when using memory evidence in court.

Another key insight concerns the OS behavior. Windows systems, with more standardized kernel structures and integrated debugging symbols, consistently produced cleaner and more interpretable dumps. Linux environments, by contrast, displayed broader variability due to kernel version fragmentation and the necessity of matching version-specific `Volatility` profiles. This difference not only underscores the need for continuous profile generation and validation, but also exposes an operational gap: forensic practitioners cannot feasibly maintain exhaustive module libraries for every Linux distribution. The reliance on manually compiled kernel modules, such as those used by `LiME`, further complicates real-world investigations, particularly in large-scale or heterogeneous environments. These findings corroborate existing literature suggesting that Linux memory forensics still lacks the level of standardization and automation achieved for Windows platforms.

The experiments also highlight the limitations of automated analysis tools. `Volatility` successfully recovered the majority of process- and kernel-level artifacts, but failed to detect several user-space objects (e.g. opened files, images, and cached text fragments) that were later identified by manual carving. This discrepancy reveals that even when acquisition is forensically sound, analysis tools may fail to interpret all relevant data due to structural mismatches, incomplete symbol information, or unhandled data fragmentation. Consequently, comprehensive investigations require a hybrid approach that combines automated parsing with low-level raw inspection and carving, especially when dealing with high-value artifacts or obfuscated malware. From a broader perspective, these results reveal a persistent gap between forensic theory and practical implementation. While existing frameworks focus on reconstructing stable system states, live systems are inherently unstable, with constant changes in memory allocation, process execution, and I/O activity. The presence of anti-forensic techniques (e.g. in-memory code injection, data wiping, and encryption) further reduces the predictability of volatile data capture. The study confirms that failures in artifact recovery cannot be attributed solely to tool inefficiency; rather, they reflect the dynamic nature of the digital evidence itself. This recognition should inform future research directions that emphasize acquisition resilience and multi-snapshot correlation to approximate atomicity through temporal reconstruction.

Finally, these findings hold broader implications for standardization and forensic reliability. Current memory acquisition and analysis practices lack uniform testing protocols and quality benchmarks. As shown, factors such as timing, tool implementation, and OS architecture all contribute to result variability. Establishing standardized evaluation procedures would help forensic laboratories measure acquisition correctness and integrity more transparently. Moreover, integrating timestamp synchronization, memory access profiling, and AI-assisted anomaly detection could enhance consistency assessment across repeated acquisitions. In conclusion, the study demonstrates that while modern tools like `Volatility` provide a powerful foundation for volatile memory forensics, their outputs must be interpreted within the context of inherent system dynamism and acquisition issue. Improving atomicity, strengthening Linux compatibility, and developing standardized validation frameworks are essential next steps. These enhancements will move the field closer to achieving reliable, reproducible, and legally defensible memory forensic analysis across diverse computing environments.

7. Conclusions

This study contributes to improve the understanding of reliability and completeness in volatile memory acquisition by empirically assessing the behavior of the leading forensic tools between different OS and acquisition conditions. The results confirm that the accuracy of recovered artifacts is inherently limited by the dynamic and non-atomic nature of system memory, where continuous background activity alters data structures during capture. While modern tools such as `Volatility` provides strong foundations for analysis, it remains affected by OS constraints, kernel variability, and analysis dependency on version-specific profiles. The experiments demonstrate that inconsistencies observed during artifact

extraction are not necessarily symptomatic of tool failure, but reflect intrinsic properties of volatile memory and its OS management.

The key conclusion is that forensic reliability depends on both tool capability and acquisition context. Differences in kernel design, paging mechanisms, and process scheduling significantly influence the integrity of captured data. Cross-validation through carving revealed that a substantial portion of evidence remains present but is inaccessible to standard plugins, suggesting that future tools must combine structured analysis with heuristic or ML-assisted reconstruction to identify fragmented or hidden data. These findings reinforce the need for greater transparency and methodological rigor in forensic practice, especially where memory evidence is used in judicial proceedings.

However, several limitations constrain the generalization of these results. The experimental setup focused on specific OS versions and controlled workloads, which, while representative, cannot capture the full variability of real-world environments. The use of virtualized systems, though advantageous for reproducibility, may not fully reflect timing characteristics of bare-metal acquisitions. Furthermore, while carving confirmed data persistence beyond structured parsing, quantitative measures of artifact completeness remain approximate, as there is no universal baseline defining what constitutes a complete volatile snapshot. Tool versions and plugin dependencies also evolve rapidly, i.e. results are bounded by the `Volatility` version at the time of experimentation. Finally, the study did not explore anti-forensic or adversarial conditions in depth (such as encrypted, obfuscated, or intentionally corrupted memory regions) which may exacerbate acquisition inaccuracies.

Building on these findings, several future research directions emerge. First, there is a need to develop standardized evaluation frameworks that can objectively quantify snapshot correctness, atomicity, and integrity across tools and platforms. Such frameworks should incorporate real-time metrics of page updates, timing differentials, and OS activity levels during capture to better model non-atomicity effects. Second, advancing cross-platform memory analysis is crucial, particularly for Linux, Android, and hybrid environments such as WSL, where heterogeneous memory structures challenge traditional analysis pipelines. Third, integrating ML and automated anomaly detection into forensic workflows could enhance artifact reconstruction by identifying patterns in fragmented or unreferenced memory regions. Lastly, collaboration between academia, tool developers, and law enforcement is essential to establish reproducible testing environments, public datasets, and open benchmarks to drive methodological transparency and long-term tool validation.

In summary, this work underscores that volatile memory forensics remains a technically demanding and evolving discipline. The interplay between acquisition fidelity, OS behavior, and analytical precision continues to shape the field's progress. By highlighting the sources of inconsistency and proposing paths toward standardized, adaptive, and verifiable acquisition methods, this study contributes to strengthening the scientific and evidentiary foundations of digital forensics.

Acknowledgments

This work was partially supported by Project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU. This work was carried out while Silvia Lucia Sanna was enrolled in the Italian National Doctorate on Artificial Intelligence run by Sapienza University of Rome in collaboration with the University of Cagliari.

Declaration on Generative AI

During the preparation of this work, the authors used Grammarly to check grammar and spelling, Paraphrase, and reword. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] A. Damodaran, F. D. Troia, V. A. Corrado, T. H. Austin, M. Stamp, A comparison of static, dynamic, and hybrid analysis for malware detection, arXiv preprint arXiv:2203.09938 (2022).

[2] H. K. Singh, J. P. Singh, A. S. Tewari, Static malware analysis using machine and deep learning, in: A. K. Bashir, G. Fortino, A. Khanna, D. Gupta (Eds.), Proceedings of International Conference on Computing and Communication Networks, Springer Nature Singapore, Singapore, 2022, pp. 437–446.

[3] L. Demetrio, S. E. Coull, B. Biggio, G. Lagorio, A. Armando, F. Roli, Adversarial exemples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection, ACM Trans. Priv. Secur. 24 (2021). URL: https://doi.org/10.1145/3473039. doi:10.1145/3473039.

[4] J. W. Stokes, D. Wang, M. Marinescu, M. Marino, B. Bussone, Attack and defense of dynamic analysis-based, adversarial neural malware detection models, MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM) (2018) 1–8. URL: https://api.semanticscholar.org/CorpusID:26928273.

[5] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, F. Roli, Yes, machine learning can be more secure! a case study on android malware detection, IEEE Trans. Dependable Secur. Comput. (2019) 711–724. doi:10.1109/TDSC.2017.2700270.

[6] I. Kara, Fileless malware threats: Recent advances, analysis approach through memory forensics and research challenges, Expert Systems With Applications (2023).

[7] D. Soi, S. L. Sanna, G. Benedetti, A. Liguori, L. Regano, L. Caviglione, G. Giacinto, Analysis and detection of android stegomalware: the impact of the loading stage, in: Proceedings of the 2025 ACM Workshop on Information Hiding and Multimedia Security, IH&MMSEC '25, Association for Computing Machinery, New York, NY, USA, 2025, p. 35–45. URL: https://doi.org/10.1145/3733102.3733122. doi:10.1145/3733102.3733122.

[8] D. Dell'Orco, G. Bernardinetti, G. Bianchi, A. Merlo, A. Pellegrini, Would you mind hiding my malware? building malicious android apps with stegopack, Pervasive and Mobile Computing 111 (2025) 102060. URL: https://www.sciencedirect.com/science/article/pii/S1574119225000495. doi:https://doi.org/10.1016/j.pmcj.2025.102060.

[9] S. Aonzo, A. Merlo, Y. Fratantonio, A. Ruggia, Obfuscapk: An open-source obfuscation framework for android apps, in: IEEE EuroS&P Workshops, 2020. doi:10.1109/EuroSPW51379.2020.00046, used to study environmental deception and evasion.

[10] Y. Dehfouli, A. Habibi Lashkari, Memory analysis for malware detection: A comprehensive survey using the oscar methodology, ACM Comput. Surv. 58 (2025). URL: https://doi.org/10.1145/3764580. doi:10.1145/3764580.

[11] A. S. Bozkir, E. Tahillioglu, M. Aydos, A malware detection approach through memory forensics and computer vision, Computers & Security (2021).

[12] Volatility, 2018. Open-source memory forensics framework.

[13] The Volatility Foundation, The volatility framework, n.d. URL: https://volatilityfoundation.org/the-volatility-framework/, accessed: 2025-01-13.

[14] Google Rekall Project, Rekall: The advanced memory forensics framework, n.d. URL: https://github.com/google/rekall, accessed: 2025-01-13.

[15] A. Sekibaala, Incident response and threat hunting with redline: Power of live memory analysis (part 1), 2019. URL: https://medium.com/@abel.sekibaala/incident-response-and-threat-hunting-with-redline-power-of-live-memory-analysis-part-1-74f11929c6bc, accessed: 2025-03-17.

[16] S. Vömel, F. C. Freiling, Correctness, atomicity, and integrity: defining criteria for forensically-sound memory acquisition, Digital Investigation 9 (2012) 125–137. doi:10.1016/j.diin.2012.01.002.

[17] S. Vömel, J. Stüttgen, An evaluation platform for forensic memory acquisition software, Digital Investigation 10 (2013) S30–S40. doi:10.1016/j.diin.2013.06.004.

[18] H. Nyholm, K. Monteith, S. Lyles, M. Gallegos, M. DeSantis, J. Donaldson, C. Taylor, The evolution of volatile memory forensics, Journal of Cybersecurity and Privacy 2 (2022) 556–572. URL: https://doi.org/10.3390/jcp2030028. doi:10.3390/jcp2030028.

[19] B. Sanjay, D. Rakshith, R. Akash, V. Hegde, An approach to detect fileless malware and defend its evasive mechanisms, in: Proceedings of the 2018 3rd International Conference on Computational Systems and Information Technology for Sustainable Solutions (CSITSS), Bengaluru, India, 2018, pp. 234–239. doi:10.1109/CSITSS.2018.8768769.

[20] A. Case, G. G. Richard, Memory forensics: The path forward, Digital Investigation 20 (2017) 23–33. URL: https://www.sciencedirect.com/science/article/pii/S1742287616301529. doi:10.1016/j.diin.2016.12.004.

[21] L. Team, Lime: Linux memory extractor, 2017. URL: https://github.com/eviltikz/LiME, accessed: 2025-01-23.

[22] J. Stüttgen, M. Cohen, Robust linux memory acquisition with minimal target impact, Digital Investigation 11 (2014). doi:10.1016/j.diin.2014.03.014.

[23] I. Hamid, A. Alabdulhay, M. M. Rahman, A Systematic Literature Review on Volatility Memory Forensics, 2023, pp. 589–600. doi:10.1007/978-981-19-9819-5_42.

[24] N. Lewis, A. Case, A. Ali-Gombe, G. G. Richard III, Memory forensics and the windows subsystem for linux, Digital Investigation 26 (2018) S3–S11. doi:10.1016/j.diin.2018.04.018.

[25] V. Roussev, P. Richard III, Volnet: A framework for analyzing network-based artifacts from volatile memory, in: Digital Forensics Research Workshop (DFRWS), 2017.

[26] J. Taylor, B. Turnbull, G. Creech, Volatile memory forensics acquisition efficacy: A comparative study towards analysing firmware-based rootkits, in: Proceedings of the 13th International Conference on Availability, Reliability and Security (ARES), ACM, Hamburg, Germany, 2018. doi:10.1145/3230833.3232810.

[27] J. Bellizzi, M. Vella, C. Colombo, J. Hernandez-Castro, Responding to targeted stealthy attacks on android using timely-captured memory dumps, IEEE Access 10 (2022) 35172–35218. doi:10.1109/ACCESS.2022.3160531.

[28] J. Bellizzi, M. Vella, C. Colombo, J. Hernandez-Castro, Real-time triggering of android memory dumps for stealthy attack investigation, in: M. Asplund, S. Nadjm-Tehrani (Eds.), Secure IT Systems, Springer International Publishing, Cham, 2021, pp. 20–36.

[29] J. Bellizzi, M. Vella, C. Colombo, J. Hernandez-Castro, Responding to living-off-the-land tactics using just-in-time memory forensics (jit-mf) for android, 2021. URL: https://arxiv.org/abs/2105.05510. arXiv:2105.05510.

[30] S. Khalid, F. B. Hussain, VolMemDroid—Investigating Android Malware Insights with Volatile Memory Artifacts, Expert Systems with Applications 253 (2024). doi:10.1016/j.eswa.2024.124347.

[31] G. Grispos, H. Studiawan, S. Alrabaee, Internet of things (iot) forensics and incident response: The good, the bad, and the unaddressed, Forensic Science International: Digital Investigation 48 (2024) 301671. URL: https://www.sciencedirect.com/science/article/pii/S2666281723001907. doi:https://doi.org/10.1016/j.fsidi.2023.301671.

A. Volatility Plugins

Table 1
Volatility commands used for each OS. The left column lists Windows plugins, while the right column lists Linux plugins.

Windows commands	Linux commands
windows.bigpools.BigPools	linux.bash.Bash
windows.cachedump.Cachedump	linux.capabilities.Capabilities
windows.callbacks.Callbacks	linux.check_afinfo.Check_afinfo
windows.cmdline.CmdLine	linux.check_creds.Check_creds
windows.devicetree.DeviceTree	linux.check_idt.Check_id
windows.dlllist.DllList	linux.check_modules.Check_modules
windows.driverirp.DriverIrp	linux.check_syscall.Check_syscall
windows.drivermodule.DriverModule	linux.elfs.Elfs
windows.driverscan.DriverScan	linux.envvars.Envvars
windows.envvars.Envvars	linux.iomem.IOMem
windows.filescan.FileScan	linux.lsmod.Lsmod
windows.getsids.GetSIDs	linux.lsof.Lsof
windows.handles.Handles	linux.proc.Maps
windows.iat.IAT	linux.pslist.PsList
windows.joblinks.JobLinks	linux.psscan.PsScan
windows.kpcrs.KPCRs	linux.pstree.PsTree
windows.ldrmodules.LdrModules	linux.sockstat.Sockstat
windows.lsadump.Lsadump	linux.tty_check.tty_check
windows.mbrscan.MBRScan	linux.keyboard_notifiers.Keyboard_notifiers
windows.pslist.PsList	linux.kmsg.Kmsg
windows.psscan.PsScan	linux.mountinfo.MountInfo
windows.pstree.PsTree	linux.psaux.PsAux