An Explainable Memory Forensics Approach for Malware Analysis

Silvia Lucia Sanna[1,*], Davide Maiorca[1] and Giorgio Giacinto[1,2]

[1]University of Cagliari, Italy
[2]Consorzio Interuniversitario Nazionale per l'Informatica (CINI), Italy

Abstract

Memory forensics is an effective methodology for analyzing living-off-the-land malware, including threats that employ evasion, obfuscation, anti-analysis, and steganographic techniques. By capturing volatile system state, memory analysis enables the recovery of transient artifacts such as decrypted payloads, executed commands, credentials, and cryptographic keys that are often inaccessible through static or traditional dynamic analysis. While several automated models have been proposed for malware detection from memory, their outputs typically lack interpretability, and memory analysis still relies heavily on expert-driven inspection of complex tool outputs, such as those produced by Volatility. In this paper, we propose an explainable, AI-assisted memory forensics approach that leverages general-purpose Large Language Models (LLMs) to interpret memory analysis outputs in a human-readable form and to automatically extract meaningful Indicators of Compromise (IoCs), in some circumstances detecting more IoCs than current state-of-the-art tools. We apply the proposed methodology to both Windows and Android malware, comparing full RAM acquisition with target-process memory dumping and highlighting their complementary forensic value. Furthermore, we demonstrate how LLMs can support both expert and non-expert analysts by explaining analysis results, correlating artifacts, and justifying malware classifications. Finally, we show that a human-in-the-loop workflow, assisted by LLMs during kernel-assisted setup and analysis, improves reproducibility and reduces operational complexity, thereby reinforcing the practical applicability of AI-driven memory forensics for modern malware investigations.

Keywords

Memory Forensics, Explainable Forensics, Malware Analysis

1. Introduction

Malware analysis is currently facing different challenges because of the development of stealthy malware e.g. evasive, obfuscated, stegomalware [1, 2], both for Windows desktop computers and Android mobile devices, the most used Operating Systems worldwide. Traditional techniques involving static (i.e. without execution) and dynamic (i.e. runtime monitoring) analysis are sometimes inefficient for such specific malware classes. Firstly, obfuscated malware cannot be detected using static analysis techniques, especially when using pattern matches (i.e. regex) and the API names are not in clear. Even if static and dynamic methodologies have been combined during the years with Artificial Intelligence (AI) algorithms [3], they lost accuracy with the development of adversarial samples, i.e. malware evading AI-based detectors [4]. For this reason, memory analysis is emerging in recent studies [5, 6, 7, 8, 9]. They all demonstrated that memory forensics enables the extraction of specific artifacts that are not retrievable with other techniques. Current works provide different acquisition techniques and different analysis strategies. Acquisition methodologies are divided into target-application analysis and full memory extraction. The acquisition methodology depends on the target OS. For Android, full memory acquisition requires the kernel recompilation in order to load the `LiME` (Linux Memory Extractor) [10] kernel module, the only available tool for complete memory dumping. This approach is helpful to study the effects of the malware on the whole system by analyzing the interaction with the OS and other processes, file access and creation, environmental variable, network traffic, bash commands. On the contrary

1st Workshop on Digital Forensics (WDF 2026), co-located with the 2nd Joint National Conference on Cybersecurity (ITASEC & SERICS 2026), February 09–13, 2026, Cagliari, Italy

*Corresponding author.
silvial.sanna@unica.it (S. L. Sanna); davide.maiorca@unica.it (D. Maiorca); giorgio.giacinto@unica.it (G. Giacinto)
0009-0002-8269-9777 (S. L. Sanna); 0000-0003-2640-4663 (D. Maiorca); 0000-0002-5759-3017 (G. Giacinto)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-14

CEUR-WS.org/Vol-4329/paper4.pdf

the target-app dump in Android uses Fridump, based on instrumentation tools such as `Frida` [11, 12], only extracts the memory allocated to the target process under analysis. For Windows, the complete memory can be extracted with specific tools like `FTK-Imager`, or `VirtualBox Management` when using a Virtual Machine. On the other hand, target-software application can be extracted with tools like `procdump`.

The analysis methodologies are based on specific content extraction such as allocated objects [5], binary-to-image encoding [6], specific IoC matching through strings and regex matching [7, 13, 14]. However, none of the current studies focused on the interpretability of the complete memory extraction and subsequently the full analysis of its content for malware analysis. They merely rely on specific known pattern matching [7]. Additionally, to the best of our knowledge, there is no study based on the differences between the complete memory dump and the target-app dump. Regarding the complete memory analysis, the only available tool is `Volatility` which requires a deep knowledge on its usage and data interpretability. We claim that general-purpose LLMs can be applied in these scenarios even for teaching purposes by giving a human-readable interpretation of the tool output [15, 16]. This concept has already been explored in traditional teaching but to the best of our knowledge there is still no application in Digital Forensics (DF).

In this study, we propose a comparative analysis of the full RAM extraction and the target-process dump. In our case, the analysis is guided by general-purpose LLMs and Natural Language Processing (NLP) algorithms. In fact, by analyzing both the `Volatility` [17] outputs and the text encoding (i.e. reading the strings of the dump) from the memory extracted using both approaches, we can give to the human analyst a comprehensive report on the interpretation of *(i)* the `Volatility` outputs, *(ii)* the details of the found artifacts and *(iii)* an explanation according to the given classification of the algorithm. If the memory is captured over time, our approach can also evaluate the differences in the dumps evolving during runtime app execution. We opt for general-purpose LLMs to adapt their vast ground truth for detection (i.e., an Large Language Model (LLM) like ChatGPT would be more efficient than a self-hosted LLM trained on a few under-representative data). In fact the dataset is a huge problem in DF research and investigations, that because of the lack of public databases, cases, files and evidences, research goes slightly slow [18, 19]. Additionally, real cases cannot be used if not properly managed according to privacy preservation [20, 19]. In this way, synthetic datasets are a plausible solution. Our analysis is supported by the release of a unified methodology for full RAM extraction. We ask the following research questions:

- **RQ1**: Can AI be applied as an assistant in Digital Forensics analysis?
- **RQ2**: What is the difference between complete memory and target-process analysis?
- **RQ3**: Can general-purpose LLMs help the analyst in malware detection?

Our main contributions are threefold. First, we introduce an explainable, AI-assisted memory forensics pipeline that leverages general-purpose LLMs to interpret volatile memory artifacts and analysis outputs in a human-readable and forensically meaningful manner. Second, we provide a systematic comparative analysis between full RAM acquisition and target-process memory dumping, highlighting their differences in malware visibility, forensic coverage, and the evolution of malicious behavior over time during execution. Third, we explore and demonstrate the effectiveness of general-purpose LLMs as analysis assistants for malware memory forensics, showing how they can support both expert and non-expert analysts in understanding complex memory artifacts. In addition, we address the practical challenges of volatile memory acquisition by adopting a human-in-the-loop methodology during the kernel-assisted setup phase. To streamline this complex and error-prone process, we used ChatGPT as an interactive assistant, requesting step-by-step guidance tailored to the target kernel and iteratively providing compilation and runtime error logs to resolve toolchain mismatches, missing headers, module loading failures, and symbol extraction issues. This approach significantly reduced setup time while preserving methodological transparency, as all corrective actions were driven by explicit error traces. To further support reproducibility and independent validation, we release the

compiled kernel artifacts and associated build metadata used in our experiments. Finally, we emphasize the critical role of memory forensics in modern malware detection by comparing memory-based analysis results with traditional approaches, such as VirusTotal [21], Drebin [22], and Entroplyzer [23], thereby illustrating the added value of volatile-memory analysis against stealthy and evasive malware.

The remainder of this paper is structured as follows. Section 2 highlights the main concepts for memory forensics and the current techniques used in malware analysis, while Section 3 introduces the used Methodology. The memory analysis results of Windows and Android using LLMs and NLP are presented respectively in Section 4.1 and Section 4.2. Section 5 closes the paper answering to the previously posed Research Questions.

2. Background and Related Works

Memory forensics is a fundamental component of modern malware analysis, enabling investigators to observe malicious behavior at runtime and recover evidence that is deliberately concealed from persistent storage. Unlike static disk-based analysis, memory forensics captures the live execution state of a system, exposing transient artifacts such as decrypted malware payloads, unpacked shellcode, injected dynamic libraries, reflective loaders, credential material, encryption keys, command-and-control configurations, and active network sessions. These properties make memory analysis particularly effective against fileless malware, packers, droppers, and multi-stage attacks, which increasingly dominate both Windows and Android threat landscapes [24, 25].

Memory Forensics for Windows Malware Analysis Windows memory forensics represents the most mature and operationally deployed domain of volatile analysis. Early foundational work demonstrated that malware frequently resides exclusively in memory, motivating the systematic analysis of RAM to detect rootkits, injected code, and kernel-level manipulation [26, 27]. Modern Windows malware routinely employs runtime-only techniques such as process hollowing, reflective DLL injection, APC injection, direct system call invocation, and kernel rootkits, all of which leave minimal or no traces on disk [28, 29]. Memory acquisition on Windows is typically performed using kernel drivers or trusted execution environments, enabling full physical memory capture with high fidelity. Tools such as `WinPmem`, `DumpIt`, and `FTK Imager` are widely used in live-response scenarios, while hypervisor-based acquisition enables repeatable malware analysis in sandboxed environments [30, 31]. Compared to mobile platforms, Windows benefits from a more stable kernel architecture and symbol infrastructure, facilitating reliable reconstruction of kernel objects and process internals. Post-acquisition analysis is commonly conducted using frameworks such as `Volatility` and `Rekall`, which reconstruct malware-relevant artifacts including hidden processes, injected memory regions, anomalous virtual address descriptors (VADs), malicious kernel modules, registry hives, credential material, and in-memory PE files [17, 32]. These techniques are central to the detection of advanced persistent threats (APTs), ransomware, and fileless malware campaigns that evade traditional signature-based defenses [28, 29].

Memory Forensics for Android Malware Analysis Memory forensics has also become increasingly important for Android malware analysis, particularly in response to widespread use of dynamic loading, runtime unpacking, and native code exploitation. Android malware often decrypts payloads only at execution time, dynamically loads DEX files or native libraries, and manipulates ART or JNI components, behaviors that are frequently invisible to static APK analysis [5, 6]. Android memory forensics inherits many principles from Linux but introduces additional complexity due to kernel fragmentation, device heterogeneity, and security mechanisms such as SELinux, Verified Boot, and application sandboxing [33]. As a result, full physical memory acquisition is less commonly achievable than on Windows systems, and many approaches rely on partial acquisition strategies, including process dumps, heap analysis, and runtime instrumentation [34, 13]. Despite these constraints, memory-based analysis enables the recovery of crucial malware artifacts, including decrypted DEX code, native shellcode, runtime-generated strings, cryptographic keys, and command-and-control endpoints [5, 35, 14]. These capabilities are essential for understanding modern Android malware that actively evades static and network-based detection.

textbfAI for Memory Forensics The rapid growth in data volume and attack sophistication has driven the adoption of artificial intelligence (AI) and machine learning (ML) across digital forensics [36, 3]. ML techniques have been successfully applied to malware classification, anomaly detection, forensic triage, and artifact prioritization across disk, network, and memory data sources. Memory forensics, in particular, has emerged as a promising domain for AI-driven analysis due to the highly entropic and unstructured nature of RAM dumps. Statistical and ML-based approaches classify memory regions using features such as entropy, byte-frequency distributions, n-grams, and structural regularities, enabling detection of injected or packed malware components even when semantic parsing fails [23, 37]. Recent work explores representation learning to improve scalability and generalization. Image-based encodings transform memory or binary data into grayscale or RGB matrices suitable for convolutional neural networks, an approach initially proposed for Windows malware binaries [38] and later extended to memory dumps and Android artifacts [6, 39]. These approaches enable featureless learning and visual explainability but may introduce artificial spatial correlations due to the reshaping of linear memory into two-dimensional layouts [40]. Complementary approaches include sequence-based deep learning, entropy-flow modeling, and audio-based encodings of binaries [41, 42]. However, most existing AI-based memory malware detection systems still operate on partial memory views or predefined regions of interest, limiting their ability to capture multi-stage or cross-process attacks. Explainability is increasingly recognized as a core requirement for forensic AI systems. In investigative and legal contexts, opaque predictions are insufficient; analysts must understand why a memory region is classified as malicious and how it relates to observable runtime behavior [43, 44]. Explainable AI (xAI) techniques address this need by highlighting influential memory regions, byte patterns, or learned features that drive classification decisions. By integrating xAI into memory-based malware analysis, AI models transition from black-box classifiers to decision-support systems that expose actionable Indicators of Compromise (IoCs), such as suspicious memory segments, injected code regions, or anomalous runtime structures [23]. This capability is particularly valuable in volatile memory forensics, where evidence is ephemeral and must be interpreted rapidly and defensibly.

3. Approach

In the following we show the detailed methodology used for this work. The pipeline reported in Figure 1 is designed to run a target program in the correct sandbox, extract the memory during execution, save the memory dump. If requested it is possible to repeat the execution over time, with a preferred time interval in order to have a sequential memory dump to study the behavior of the target malware sample during execution, sometimes with automatic or manual interaction to study the events. Subsequently, the dump is analysed with `Volatility` in case of full RAM extraction, otherwhise with `strings` to extract UTF-8 readable characters as no public tool is available for target-process memory analysis. At this point, we apply LLMs and NLP algorithms to extract specific IoCs from the dump, classify the target application with an explanation on the result. Additionally, we use the general-purpose LLMs to interpret and explain to the analyst the `Volatility` output, a fundamental step for non-expert users. At the end we compare the extracted IoCs and classification report with common state-of-the-art tools to stress the power of AI-based memory forensics for malware analysis.

3.1. Dataset Creation

We downloaded real malware samples for Android and Windows (the most used OS respectively for Desktop and Mobile) and executed in protected sandbox to not damage real environments as they are real malware. Despite some malware detect the presence of a debug environment, we excluded such samples otherwise the malware does not start. We considered dropper and fake APKs for Android because are one of the most dangerous categories to be detected. While for Windows we considered the ransomware as it is the main threat. We expect to extract important IoCs from the memory dump and improve the analysis of current state-of-the-art tools.

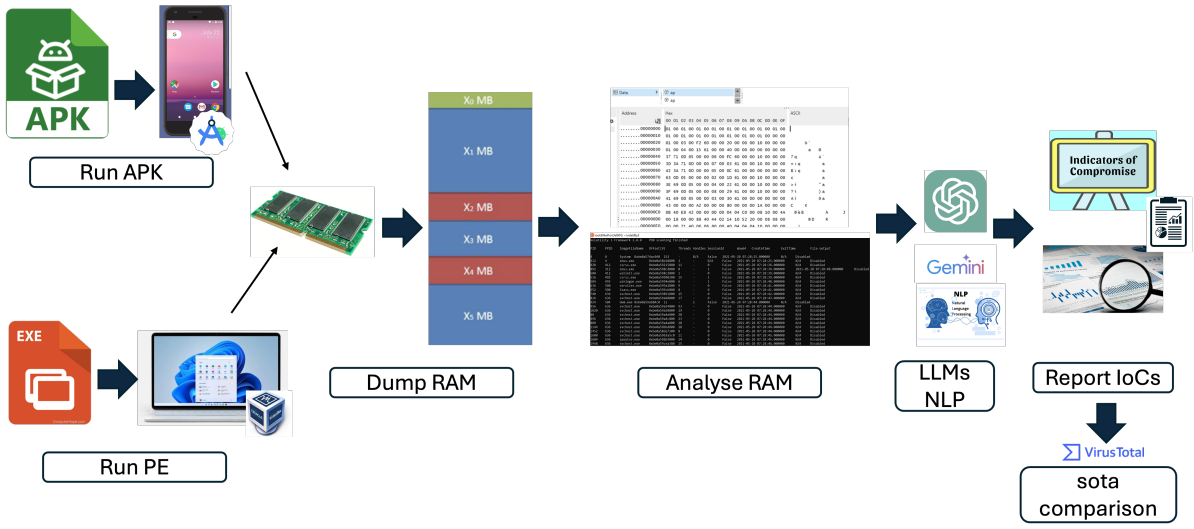


Figure 1: Execution pipeline. Run the target program (APK or PE) in the correct sandbox, extract the RAM related to the target process and the full RAM, analyse the dump and extract IoCs with LLMs and NLP algorithms, producing a human-readable report explaining the content of the dump, listing the found IoCs and explaining the classified executed program. The extracted results are compared with popular state-of-the-art tools

3.2. Memory Extraction

Due to the different architectures, the memory extraction is strictly dependant on the target OS.

Windows Memory Extraction For Windows experiments, full RAM acquisition was performed using a hypervisor-assisted approach that does not require kernel instrumentation or elevated privileges within the guest operating system. All malware samples were executed inside a controlled VirtualBox virtual machine, and while the malware was running, the complete physical memory of the guest was captured directly from the host using the native `VBoxManage` command. This method exports a snapshot of the virtual machine's RAM without interacting with the guest kernel or altering its runtime state, thereby ensuring acquisition transparency and minimizing observer effects. Because memory extraction is performed externally at the hypervisor level, this approach bypasses in-guest security mechanisms such as driver signing and kernel patch protection, while still providing a forensically complete memory image suitable for subsequent analysis with standard memory forensics frameworks.

Android Memory Extraction We adopted a kernel-assisted methodology for full RAM acquisition on Android, as complete volatile memory extraction necessarily requires execution in kernel context. Unlike desktop operating systems, Android enforces strict isolation between user space and kernel space through SELinux, application sandboxing, and Verified Boot, which prevents unprivileged processes from accessing physical memory or kernel data structures. As a consequence, super-user (root) privileges are mandatory to load kernel instrumentation and read physical page frames. Furthermore, common consumer and emulator kernels either disable loadable modules or enforce strict module signature verification, making third-party forensic modules incompatible with stock builds. For this reason, kernel recompilation is required to enable loadable module support, ensure ABI compatibility with the running kernel, and allow the insertion of the LiME acquisition module without altering runtime semantics. Our procedure follows the official Volatility Android guideline[1], which prescribes target characterization, kernel source selection, configuration alignment, LiME out-of-tree compilation, and symbol/profile generation for memory analysis. To streamline this complex and error-prone process, we used ChatGPT as an interactive assistant, requesting step-by-step guidance tailored to the target kernel and iteratively providing compilation and runtime error logs to resolve toolchain mismatches, missing headers, module loading failures, and symbol extraction issues. This human-in-the-loop approach reduced setup time while preserving methodological transparency, as all corrective steps were driven by explicit error traces. Finally, to ensure reproducibility and facilitate independent validation, we release the compiled

[1] https://github.com/volatilityfoundation/volatility/wiki/Android

kernel artifacts and associated build metadata used in our experiments.

3.3. Memory Analysis

In the following we present the memory analysis methodology details for full RAM extraction and target-process dumps.

Full Memory Analysis Memory analysis was performed using `Volatility`, which requires kernel-specific metadata to correctly interpret raw memory images. While `Volatility` automatically resolves symbols and analysis contexts for Windows memory dumps, Android memory analysis mandates the explicit creation of a kernel-matching profile due to the absence of standardized debugging symbols and the extensive fragmentation of Android kernels across versions and vendors. In this work, Volatility profiles were generated directly from the same custom kernel build used during memory acquisition, ensuring exact ABI consistency between the running system and the analysis framework. Specifically, kernel recompilation produced the unstripped `vmlinux` binary and the corresponding `System.map` file, which together encode symbol addresses and structure layouts. For Volatility 2, we extracted DWARF debugging information from `vmlinux` to generate a `module.dwarf` file and packaged it with `System.map` into a profile archive, which was then loaded explicitly during analysis. For Volatility 3, the same kernel artifacts were processed using `dwarf2json` to generate a JSON-based symbol file that encapsulates kernel types, offsets, and symbols in a structured format. This symbol file was supplied to Volatility at runtime to enable kernel and process reconstruction. The profile-generation step is mandatory on Android, as even minor kernel configuration changes alter structure offsets and invalidate generic assumptions. Although this procedure introduces additional complexity compared to Windows, it guarantees analytical correctness by ensuring that all memory objects are parsed using symbols derived from the exact kernel image responsible for the acquired RAM dump.

Target Memory Analysis Process-level memory acquisition is performed using platform-specific tools. On Android, the memory of the target application was extracted using `Fridump`, which attaches to the running process via Frida and dumps its mapped memory regions at runtime. On Windows, process and system memory acquisition was carried out using `WinPmem`, a kernel-level acquisition tool that enables reliable extraction of volatile memory while the malware sample is executing. These targeted dumps complement full RAM acquisition by providing finer-grained visibility into application-specific runtime artifacts.

3.4. AI-assisted Memory Analysis

The main contribution of this paper is the application of general-purpose LLMs to analyse memory dumps while executing a malware. First of all, general-purpose LLMs are used to interpret the output of the analysis conducted in the complete memory dump with `Volatility` as it produces outputs in table format, readable only by expert users and analysts. We apply LLMs to this output to help people understanding the output of this tool, hence for teaching improvements. Additionally, as general-purpose LLMs are trained on a large amount of data, we use them to extract automatically IoCs from the memory dump. This part is fundamental to understand if there is a malware running in the target device and because of memory forensics extract artifacts not detectable with traditional approaches as every data in RAM resides in plain text (e.g. passwords, encryption keys). After extracting the IoCs, the LLM produces a report explaining the `Volatility` output, the extracted IoCs and the classification of the program running on RAM with its explanation. We also compared the LLMs results with ad-hoc NLP algorithms and for target-application memory dump.

4. Evaluation Results and Discussion

In this set of experiments, we perform full RAM analysis following the methodology described in Section 3 and illustrated in Figure 1. Memory dumps are examined using `Volatility` version 2 (v2.6.1) and version 3 (v2.27.0), which extract forensic artifacts such as process trees, network connections,

loaded libraries, memory mappings, and file system activity through their respective plugins. The analysis workflow is common to both Windows and Android: once memory is acquired, Volatility parses kernel and process structures and produces structured outputs that must be interpreted to assess malicious behavior.

To evaluate the difference between full RAM acquisition and target-process memory dumping, we designed experiments that repeatedly capture memory during execution. For each sample, memory was dumped five times within the first 60 seconds of execution (i.e., one dump every 10 seconds), allowing us to study the temporal evolution of malicious behavior. Full RAM analysis provides a system-wide view of attack effects, including file access, network activity, inter-process interaction, and injected code, while target-process dumping offers a more focused perspective on application-specific behavior. This experimental design is applied consistently across both operating systems, although the acquisition mechanisms differ by platform.

After acquisition, all memory dumps were parsed using a predefined set of Volatility plugins (reported in Table 4 for both Volatility 2 and Volatility 3), and their outputs were stored for subsequent analysis. Because Volatility produces large, table-based outputs that typically require expert interpretation, we investigated whether general-purpose LLMs, specifically ChatGPT-4o-mini and Gemini 2.0-flash-lite, could assist analysts in understanding these results. The prompts (reported in Table 5) instruct the LLMs to *(i)* read Volatility plugin outputs, *(ii)* translate them into human-readable explanations, *(iii)* extract relevant Indicators of Compromise (IoCs), *(iv)* correlate evidence across plugins to identify suspicious or malicious processes, and *(v)* generate a final report explaining the classification decision.

In parallel, we implemented a custom rule-based NLP pipeline to analyze both Volatility outputs and raw strings extracted from memory dumps. This NLP component relies on deterministic regular-expression matching and entropy-based heuristics to detect forensic clues such as command execution patterns, network indicators, privilege abuse, code loading, and obfuscation artifacts. The resulting features are aggregated into interpretable vectors and classified as malicious, suspicious, benign, or unknown. Unlike embedding-based or statistical NLP approaches, this system is fully explainable and knowledge-driven, enabling direct attribution of decisions to specific textual evidence. The comparison between LLM-based interpretation and deterministic NLP analysis is summarized in Figure 4.

4.1. Windows

The Windows experiments were conducted in a controlled virtualized environment using `VirtualBox`. A Windows 11 guest virtual machine was executed on the host system, and full memory acquisition was performed externally via `VBoxManage guestcontrol`. This approach allows reliable and repeatable memory snapshots without requiring kernel drivers or in-guest acquisition tools, thereby minimizing interference with the system under analysis.

Memory Extraction. For each experiment, the full RAM image was collected at runtime and subsequently analyzed using `Volatility 3`. Unlike Android experiments, no kernel recompilation or profile tuning was required, as `Volatility 3` natively supports modern Windows memory structures. Each memory dump corresponds to a single execution snapshot, ensuring consistency between forensic artifacts and observed runtime behavior. No analysis could be conduct with `Volatility 2` because when running the plugins to extract data from the memory dump a profile is required and no match is available for Windows 11.

Memory Analysis. The forensic analysis on Windows memory dumps focused on interpreting the outputs of a comprehensive set of `Volatility 3` plugins (shown in Table 3), including process enumeration, memory allocation structures, loaded modules, kernel callbacks, registry artifacts, and persistence-related mechanisms. These plugins provide low-level visibility into system behavior that cannot be reconstructed from static binaries alone. The analysis was performed using three approaches: *(i)* a rule-based heuristic NLP engine, *(ii)* general-purpose Large Language Models (ChatGPT-4o-mini and Gemini 2.0-flash-lite), and *(iii)* external reputation-based analysis via `VirusTotal`. Each approach was evaluated on the same set of Windows memory dumps to ensure comparability. A key advantage of LLM-based analysis lies in its ability to transform low-level forensic outputs into human-readable

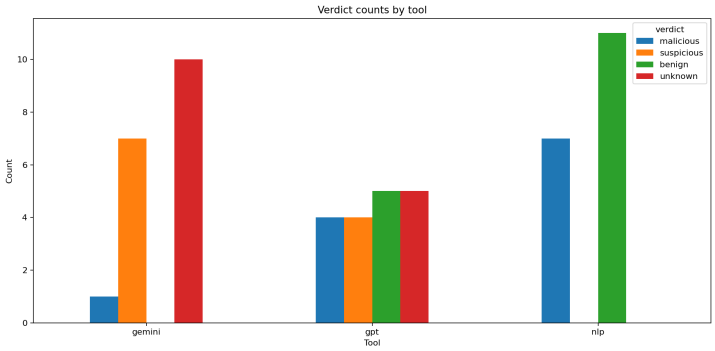


Figure 2: Evaluation results on Windows

explanations. Both ChatGPT and Gemini were able to contextualize complex `Volatility` plugin outputs, such as suspicious memory regions, anomalous pool allocations, hidden processes, or injected code segments, and explain their forensic relevance in natural language. In contrast, the heuristic NLP approach produces interpretable but limited explanations based on predefined rules and regex-based IoC extraction. While effective at highlighting known patterns (e.g. URLs, IP addresses, encoded blobs), it lacks the contextual reasoning required to relate multiple artifacts across different plugins. Overall, LLMs significantly improve forensic explainability by bridging the semantic gap between raw memory artifacts and analyst-level understanding.

Comparison with LLMs and VirusTotal. The comparative analysis highlights that LLM-based memory forensics and VirusTotal address fundamentally different forensic problems. VirusTotal primarily operates on static artifacts and reputation signals, relying on antivirus engine consensus, file hashes, and known indicators. As a result, VirusTotal provides limited insight into runtime behavior and does not expose memory-resident artifacts such as injected code, live threads, kernel callbacks, or transient filesystem activity. Conversely, LLM-based analysis of Windows memory dumps is capable of extracting a broader and more diverse set of Indicators of Compromise directly from RAM. These include evidence related to process injection, anomalous memory regions, filesystem interactions (temporary files, caches, databases), registry modifications, and kernel-level structures. Our experiments show that LLMs consistently extract more IoCs than VirusTotal, particularly those related to dynamic behavior and system-level manipulation. Between the two LLMs, ChatGPT demonstrates more stable behavior in terms of verdict consistency and explanation quality, while Gemini shows higher variance and a greater tendency toward conservative or unknown classifications. This difference is reflected in the lower agreement scores between Gemini and the other approaches. The quantitative results confirm that the three approaches produce substantially different verdict distributions. The heuristic NLP classifier tends to polarize results toward benign or malicious based on explicit patterns, whereas LLMs exhibit a more nuanced distribution across malicious, suspicious, benign, and unknown categories. Pairwise agreement analysis further shows limited overlap between VirusTotal and LLM-based verdicts, indicating that reputation-based detection and memory forensics capture complementary aspects of malicious behavior. Higher agreement is observed between ChatGPT and the NLP approach, suggesting that rule-based IoC extraction partially aligns with LLMs reasoning when analyzing memory artifacts.

Importantly, disagreement does not indicate incorrectness but rather reflects the fact that each method targets a distinct forensic layer: VirusTotal focuses on known malware signatures, NLP emphasizes explicit textual indicators, and LLMs synthesize heterogeneous memory evidence into higher-level forensic interpretations.

4.2. Android

Since the target device runs an Android system on a Linux kernel, the correct Linux plugin and matching profile must be selected to properly interpret kernel structures. However, available Linux plugins (particularly across Volatility 2 and 3) are limited, and kernel-dependent challenges make full RAM forensics less explored in prior work. Recent works such as `VolMemDroid` focus on similar Android-

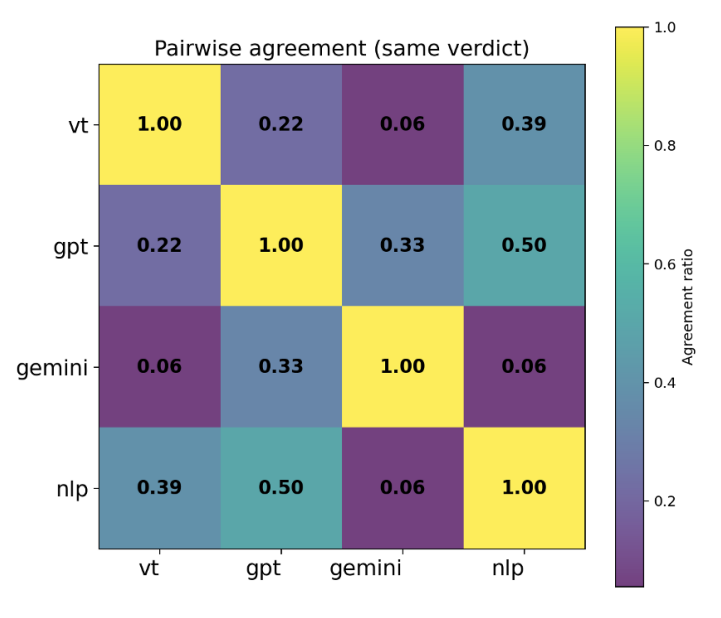


Figure 3: Comparison results on Windows

specific guidelines and rely on a Goldfish 4.4 kernel, consistent with Android 9 (API 28). `VolMemDroid` extracts IoCs using `awk` and regex and converts them into feature vectors for analysis. We adopted the same execution environment, also because Goldfish kernel is still used and adopted by different devices even if release many years ago. In contrast, our methodology parses `Volatility`'s plugin outputs and leverages general-purpose LLMs (ChatGPT-4o-mini and Gemini 2.0-flash-lite) to interpret results and generate human-readable summaries. This approach allows the automatic extraction of significant patterns (i.e. IoCs, strings, and plugin-specific indicators) providing analysts with clear, interpretable reports.

Memory Extraction. The complete RAM extraction requires more resources because of the kernel recompilation. At the same time, the full RAM analysis gives more details on the effects of the target Android Package (APK) in the whole system, for example monitoring written/accessed files and network connection. For this reason, we extracted the outputs of the selected `Volatility` plugins (Table 4) for each dump collected over time.

For each APK we dumped 5 times during the first 60 seconds of execution (i.e. every 10 seconds which is also the time our machine needs to dump the memory of 2GB, notably not corresponding to current real physical Android devices). We considered only a restricted subset of APK, i.e. dropper and fake APK (e.g.`Netflix`, `Twitter`, `Whatsapp`, `Instagram`, `TikTok`, `Facebook`) compared with the real official version. It is fundamental to recognize fake applications that could contain malicious behaviors while mimiking real legitimate applications not recognizable by users. One example is the recent Android stegomalware [45, 2, 46].

Memory Analysis, Interpretability and Explanation (Android focus). As shown in Figure 4, `chatGPT` is strictly performant in analysing the output of `Volatility` and giving a human-readable description but also classifying them correctly, with an average confidence (i.e. accuracy) of 0.93 for Volatility 3. The LLM can classify the dumps in goodware/malware by also providing an explanation on the string pattern and its filename that lead to such classification (e.g. malware because of the process *com.quasar.bistocook2* found in linux.pstree), as shown in Table 1 with a restricted dataset of fake APK. Similarly, the NLP algorithm extracts specific IoC based on regex and classifies the APK according to the content of the memory dump. We made our own NLP algorithm to parse both `Volatility` outputs and the extracted strings both from the full RAM methodology (i.e.`LiMe`) and the target-process (i.e.`Fridump`), as displayed in Figure 4. The NLP component uses rule-based regular expressions and entropy checks to detect domain-specific forensic clues, producing interpretable feature vectors to classify memory-related text as malicious, suspicious, benign, or unknown. It is a knowledge-driven,

Fake APK	GPT Detection	Gemini Detection
Netflix	process name, malfind	process name, process entries
Twitter	connected IP, process name	process name, malfind
Tiktok	process name, hidden process, malfind	process name, hidden process, malfind
Facebook	process name, domain	manipulation, domain

Table 1
Comparison of GPT and Gemini string-based detections across a subset of Android fake applications

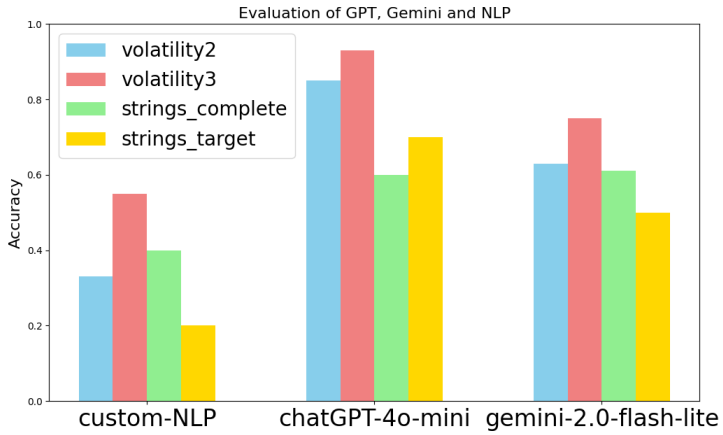


Figure 4: Comparison of the Android evaluation accuracy of the three models (from left to right: custom NLP, chatGPT-4o-mini, gemini-2.0-flash-lite) with the four different analysis (from left to right: volatility2, volatility3, strings extracted from the complete dump with LiME, and with the target dump with Frida

explainable system that avoids statistical or embedding-based models.

Moreover, the models recognize the differences between fake and real APK by highlighting differences when asked (Table 1). In particular, fake APK can be detected because of the presence of IoCs regarding repacking, which is the technique to falsify APK starting from existing ones [47]. Moreover, thanks to the plugin `malfind`, it can retrieve injected malicious payloads, especially when comparing fake samples against legitimate counterparts. Both LLMs, `chatGPT` and `Gemini`, can identify the name of the package exhibiting malicious behavior.

Overall, LLM are more efficient than our NLP algorithm, plausibly due to the broader knowledge base of general-purpose models, as reflected in Figure 4. `Volatility` analysis with respect to `Fridump` is more complete and the LLM can better explain the effects in the whole system. On the other hand, `Fridump` analysis allows a more focused understanding of the target application behavior, especially if analyzed over time, by capturing actions such as runtime permissions, API calls, encryption, and dynamically loaded code.

Comparative Analysis. Finally, we compared our memory forensics analysis with current literature tools, i.e.`VirusTotal`, `Drebin` and `Entroplyzer`. First of all, `Drebin` and `Entroplyzer` could analyze only some of them due to, respectively, anti-static analysis measures for anti-repacking in Androguard and anti-dynamic analysis for function hooking protection with Frida, as shown in Figure 5. The remaining APK, less than 30% were successfully detected by the tools. With respect to `VirusTotal` we compared the extracted IoC. Our memory forensics approach could find all IoC retrieved by VirusTotal, except for some IP addresses and domains that have been disabled. Additionally, the memory forensics approach could detect more IoCs for example regarding threads, file management (e.g. db, tmp, cache files), filesystem interaction, syscalls, native libraries, hidden modules, as highlighted in Table 2.

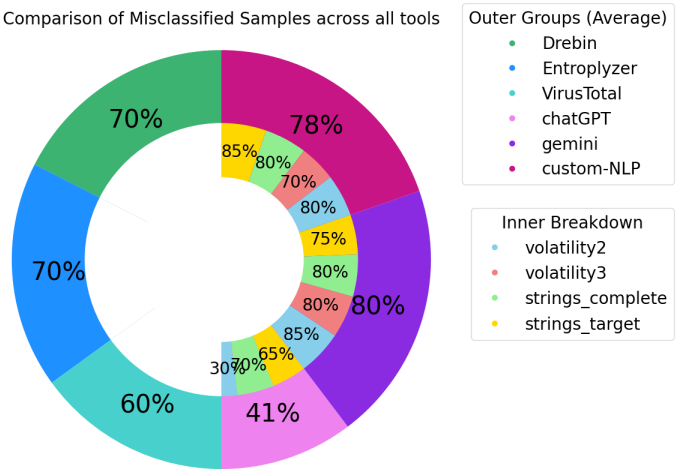


Figure 5: Average percentage of misclassified samples for the different Android detectors, i.e. Drebin, Entroplyzer, VirusTotal (notably average percentage of AV engines that fail to detect on VT) and the LLM-RAM approach with the different LLMs and RAM analysis methodologies (volatility2, volatility3, strings from the complete dump or the target process dump)

IoC / Evidence Category	VirusTotal	LLM–RAM (Memory Forensics)
URLs/IPs/Domains	✓	✓
Bundled files / archives	✓	✓
Files / artifacts on disk	✗	✓
Executable code / processes	✗	✓
System & kernel structures	✗	✓
User activity / environment	✗	✓
Malware / manipulation indicators	✗	✓
Device & memory interfaces	✗	✓
Tracing events	✗	✓

Table 2
Comparison of IoC categories supported by VirusTotal vs. Android LLM–RAM.

5. Conclusions

In this work, we propose a cross-platform memory analysis and interpretation methodology applicable to both Windows and Android systems. We introduce an analysis framework for malware detection and explanation based on the integration of general-purpose LLMs with volatile memory forensics. Given either structured outputs produced by `Volatility` from Windows and Android full RAM images, or unstructured textual artifacts (e.g. extracted strings) from memory dumps, the proposed approach enables human-readable interpretation, automatic extraction of Indicators of Compromise (IoCs), and explainable malware classification.

The text-based analysis (i.e. string extraction and semantic interpretation) is applicable to both complete system memory snapshots and target-process memory regions, and is therefore suitable for desktop-class OS as well as mobile environments. Through a comparative evaluation of these acquisition strategies, we show that full RAM analysis provides a more comprehensive view of system-wide attack effects on the OS (particularly in Windows hosts), while target-process memory analysis can expose a higher density of application-specific IoCs, particularly in mobile scenarios.

We generate analyst-oriented reports in which the LLM explains why the examined memory artifacts derived from `Volatility` plugin outputs or string-based analysis, should be considered benign, suspicious, or malicious, and explicitly links each classification decision to the extracted IoCs. The results are compared against established tools such as VirusTotal, demonstrating that memory-centric analysis reveals substantially richer forensic evidence than file-based reputation services alone, while

remaining complementary to them.

We also show that a human-in-the-loop approach, supported by general-purpose LLMs during Android kernel-assisted memory acquisition and analysis, can significantly reduce setup complexity while preserving transparency and reproducibility, as demonstrated by the release of the compiled kernel artifacts and build metadata used in our experiments.

This study examines the applicability of Artificial Intelligence as a supportive tool in Digital Forensics analysis. **RQ1** investigates whether AI can assist critical phases of memory forensics, encompassing Android kernel recompilation for memory extraction and the analysis of acquired memory dumps through Volatility output interpretation, Indicators of Compromise (IoCs) extraction, malware classification, and result explanation [48]. **RQ2** focuses on the distinction between complete memory analysis and target-process analysis, where the former emphasizes operating system–level interactions and system-wide behavior, while the latter enables a more granular inspection of application-specific activities. **RQ3** evaluates the contribution of general-purpose LLMs to malware detection, assessing their capability to perform knowledge-driven IoC extraction and to generate human-readable explanations that enhance the interpretability of classification outcomes for forensic analysts.

While the proposed approach demonstrates promising results, several aspects warrant further investigation. The use of general-purpose LLMs is exploratory and primarily aimed at enhancing the interpretability of memory forensics outputs; a more extensive analysis of response consistency and reliability would be necessary to fully assess their suitability for forensic applications. Additionally, LLMs to be used in real Digital Forensics investigations must protect user data and commercial LLMs do not guarantee it, especially as data in memory dumps is in plain text. Hence, one solution would be the use of offline LLMs but we need specific data for training and ad-hoc tests on their performances must be conducted in the future.

The framework builds upon well-established tools and techniques, and its contribution lies mainly in their integration within a unified workflow rather than in the development of novel acquisition mechanisms. The experimental evaluation is conducted on a limited set of samples and within an emulator-based environment using a single kernel configuration, which may restrict the generalization of the findings to other kernels, modern Generic Kernel Images, or physical devices. In addition, the requirement for kernel recompilation introduces practical constraints that should be considered when applying the approach in real-world forensic scenarios. Future work will focus on expanding the dataset, validating the methodology across diverse platforms, and further strengthening the evaluation of the LLM-based analysis component. The data used for this work is published online [2].

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 in order to: Grammar and spelling check, Paraphrase and reword. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content. As part of the methodology described in the paper, the authors also used the general-purpose Large Language Models ChatGPT and Gemini to interpret Volatility reports, produce human-readable descriptions and reports, and extract Indicators of Compromise (IoCs).

[2] https://github.com/slsanna/LLM-chatbot-MFMA

Data and Code Availability

The dataset of the work can be found at the following repository https://github.com/slsanna/LLM-RAM-MalwareAnalysis.

References

[1] A. Ruggia, D. Nisi, S. Dambra, A. Merlo, D. Balzarotti, S. Aonzo, Unmasking the veiled: A comprehensive analysis of android evasive malware, in: Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 383–398. URL: https://doi.org/10.1145/3634737.3637658. doi:10.1145/3634737.3637658.

[2] 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.

[3] G. Apruzzese, et al., Machine learning for cybersecurity: A systematic review, ACM Computing Surveys (2022).

[4] 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.

[5] A. Ali-Gombe, S. Sudhakaran, A. Case, G. G. R. III, DroidScraper: A tool for android In-Memory object recovery and reconstruction, in: 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), USENIX Association, Chaoyang District, Beijing, 2019, pp. 547–559. URL: https://www.usenix.org/conference/raid2019/presentation/ali-gombe.

[6] A. Ali-Gombe, S. Sudhakaran, R. Vijayakanthan, G. G. Richard, crgb_mem: At the intersection of memory forensics and machine learning, Forensic Science International: Digital Investigation 45 (2023) 301564. URL: https://www.sciencedirect.com/science/article/pii/S2666281723000732. doi:https://doi.org/10.1016/j.fsidi.2023.301564.

[7] S. Khalid, F. B. Hussain, Volmemdroid—investigating android malware insights with volatile memory artifacts, Expert Systems with Applications 253 (2024) 124347. URL: https://www.sciencedirect.com/science/article/pii/S0957417424012132. doi:https://doi.org/10.1016/j.eswa.2024.124347.

[8] 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.

[9] J. Bellizzi, E. Losiouk, M. Conti, C. Colombo, M. Vella, Vedrando: A novel way to reveal stealthy attack steps on android through memory forensics, Journal of Cybersecurity and Privacy 3 (2023) 364–395. URL: https://www.mdpi.com/2624-800X/3/3/19. doi:10.3390/jcp3030019.

[10] 504ensicsLabs, Lime: Linux memory extractor (loadable kernel module for volatile memory acquisition), https://github.com/504ensicsLabs/LiME, 2012. GitHub repository.

[11] L. Yang, B. Zhang, Y. Wang, Process-based memory extraction via frida for android forensics, in: Proceedings of the DFRWS EU, 2019.

[12] Frida documentation, Frida, . URL: https://frida.re/docs/home/, Online; Accessed 02/2025.

[13] J. Bellizzi, A. Bruno, F. Martinelli, Jit-mf: Just-in-time memory forensics for android devices, Digital Investigation 42 (2022) 301450. URL: https://arxiv.org/abs/2103.04891.

[14] 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.

[15] M. Liu, F. M'Hiri, Beyond traditional teaching: Large language models as simulated teaching

assistants in computer science, in: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1, SIGCSE 2024, Association for Computing Machinery, New York, NY, USA, 2024, p. 743–749. URL: https://doi.org/10.1145/3626252.3630789. doi:10.1145/3626252.3630789.

[16] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, J.-R. Wen, A survey of large language models, 2025. URL: https://arxiv.org/abs/2303.18223. arXiv:2303.18223.

[17] V. Foundation, Volatility: The memory forensics framework, https://www.volatilityfoundation.org/, ???? Open source memory forensics tool.

[18] F. Breitinger, J.-N. Hilgert, C. Hargreaves, J. Sheppard, R. Overdorf, M. Scanlon, Dfrws eu 10-year review and future directions in digital forensic research, Forensic Science International: Digital Investigation 48 (2024) 301685. doi:10.1016/j.fsidi.2023.301685.

[19] T. Göbel, H. Baier, F. Breitinger, Data for digital forensics: Why a discussion on "how realistic is synthetic data" is dispensable, Digital Threats: Research and Practice 4 (2023) 1–18.

[20] S. Mombelli, J. R. Lyle, F. Breitinger, Fairness in digital forensics datasets' metadata – and how to improve it, Forensic Science International: Digital Investigation 48 (2024) 301681. doi:10.1016/j.fsidi.2023.301681.

[21] Virustotal, https://www.virustotal.com, ????

[22] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, Drebin: Effective and explainable detection of android malware in your pocket., in: NDSS, The Internet Society, 2014. URL: http://dblp.uni-trier.de/db/conf/ndss/ndss2014.html#ArpSHGR14.

[23] D. S. Keyes, B. Li, G. Kaur, A. H. Lashkari, F. Gagnon, F. Massicotte, Entroplyzer: Android malware classification and characterization using entropy analysis of dynamic characteristics, in: 2021 Reconciling Data Analytics, Automation, Privacy, and Security: A Big Data Challenge (RDAAPS), 2021, pp. 1–12. doi:10.1109/RDAAPS48126.2021.9452002.

[24] D. Bhusal, N. Rastogi, Adversarial patterns: Building robust android malware classifiers, ACM Comput. Surv. 57 (2025). URL: https://doi.org/10.1145/3717607. doi:10.1145/3717607.

[25] M. Gaber, M. Ahmed, H. Janicke, Malware Detection with Artificial Intelligence: A Systematic Literature Review, ACM Computing Surveys 56 (2024). doi:10.1145/3638552.

[26] H. Carvey, Windows Forensic Analysis, Syngress, 2007.

[27] A. Schuster, Searching for processes and threads in microsoft windows memory dumps, in: Digital Investigation, 2006.

[28] J. Okolica, G. Peterson, Windows operating system agnostic memory analysis, Digital Investigation (2011).

[29] R. Gianni, et al., Memory forensics for advanced malware detection, in: ACSAC, 2016.

[30] J. Sylve, A. Case, L. Marziale, G. G. Richard, Acquisition and analysis of volatile memory from android devices, in: DFRWS, 2012.

[31] A. Case, J. Levy, Automated memory forensics, Digital Investigation (2013).

[32] Rekall Project, Rekall Memory Forensic Framework, 2017. https://www.rekall-forensic.com.

[33] V. Afonso, et al., Identifying android malware using dynamically obtained features, NDSS (2016).

[34] J. Zhang, E. Chengyuan, A. Hu, A method of android application forensics based on heap memory analysis, 2018. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85056819842&doi=10.1145%2f3207677.3277934&partnerID=40&md5=c42d0dd2e9a55805b2105bc27b5990f1. doi:10.1145/3207677.3277934, cited by: 0.

[35] F. Franzen, T. Holl, M. Andreas, J. Kirsch, J. Grossklags, Katana: Robust, automated, binary-only forensic analysis of linux memory snapshots, in: Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses, RAID '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 214–231. URL: https://doi.org/10.1145/3545948.3545980. doi:10.1145/3545948.3545980.

[36] S. García, et al., An empirical study of machine learning for cybersecurity, IEEE Security & Privacy (2018).

[37] Y. Song, et al., Entropy-based detection of packed malware, SIGSAC (2018).

[38] L. Nataraj, S. Karthikeyan, G. Jacob, B. S. Manjunath, Malware images: visualization and automatic classification, in: Proceedings of the 8th International Symposium on Visualization for Cyber Security, VizSec '11, Association for Computing Machinery, New York, NY, USA, 2011. URL: https://doi.org/10.1145/2016904.2016908. doi:10.1145/2016904.2016908.

[39] J. Daoudi, T. F. Bissyandé, J. Klein, Dexray: Simple, efficient and effective android malware detection via bytecode gray-scale imaging, 2021. arXiv:2108., arXiv preprint.

[40] V. Malhotra, K. Potika, M. Stamp, A comparison of graph neural networks for malware classification, Journal of Computer Virology and Hacking Techniques 20 (2024) 53 – 69. doi:10.1007/s11416-023-00493-y.

[41] F. Mercaldo, A. Santone, Audio signal processing for android malware detection and family identification, Journal of Computer Virology and Hacking Techniques 17 (2021) 139–152. URL: https://doi.org/10.1007/s11416-020-00376-6. doi:10.1007/s11416-020-00376-6.

[42] O. E. Kural, E. Kiliç, C. Aksaç, Apk2audio4andmal: Audio based malware family detection framework, IEEE Access 11 (2023) 27527–27535. doi:10.1109/ACCESS.2023.3258377.

[43] M. Ribeiro, S. Singh, C. Guestrin, Why should i trust you? explaining the predictions of any classifier, KDD (2016).

[44] R. Guidotti, et al., A survey of explainable artificial intelligence, ACM Computing Surveys (2019).

[45] D. Soi, S. L. Sanna, A. Liguori, M. Zuppelli, L. Regano, D. Maiorca, L. Caviglione, G. Manco, G. Giacinto, On the Feasibility of Android Stegomalware: A Detection Study, Italian Conference on CyberSecurity (2025). https://ceur-ws.org/Vol-3962/paper9.pdf.

[46] 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.

[47] A. Ruggia, D. Nisi, S. Dambra, A. Merlo, D. Balzarotti, S. Aonzo, Unmasking the veiled: A comprehensive analysis of android evasive malware, in: Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 383–398. URL: https://doi.org/10.1145/3634737.3637658. doi:10.1145/3634737.3637658.

[48] S. L. Sanna, L. Regano, D. Maiorca, G. Giacinto, Improving cybercrime detection and digital forensics investigations with artificial intelligence, 2025. URL: https://arxiv.org/abs/2510.14638. arXiv:2510.14638.

Windows Category	Volatility 3 Plugins
Process & Execution Analysis	windows.pslist.PsList, windows.psscan.PsScan, windows.pstree.PsTree, windows.cmdline.CmdLine, windows.envars.Envars, windows.handles.Handles, windows.sessions.Sessions, windows.joblinks.JobLinks
Memory Injection & Malware Techniques	windows.malfind.Malfind, windows.hollowprocesses.HollowProcesses, windows.processghosting.ProcessGhosting, windows.malware.pebmasquerade.PebMasquerade, windows.malware.hollowprocesses.HollowProcesses, windows.malware.processghosting.ProcessGhosting
Loaded Modules & Drivers	windows.dlllist.DllList, windows.modules.Modules, windows.ldrmodules.LdrModules, windows.unloadedmodules.UnloadedModules, windows.driverscan.DriverScan, windows.drivermodule.DriverModule, windows.malware.drivermodule.DriverModule
Kernel & Low-Level Structures	windows.ssdt.SSDT, windows.kpcrs.KPCRs, windows.callbacks.Callbacks, windows.driverirp.DriverIrp, windows.timers.Timers
Memory Layout & VAD Analysis	windows.memmap.Memmap, windows.vadinfo.VadInfo, windows.vadwalk.VadWalk, windows.vadregexscan.VadRegExScan, windows.virtmap.VirtMap, windows.bigpools.BigPools
Persistence & Autostart Mechanisms	windows.registry.scheduled_tasks.ScheduledTasks, windows.scheduled_tasks.ScheduledTasks, windows.registry.userassist.UserAssist, windows.shimcachemem.ShimcacheMem, windows.amcache.Amcache, windows.registry.amcache.Amcache
Registry Analysis	windows.registry.hivelist.HiveList, windows.registry.hivescan.HiveScan, windows.registry.printkey.PrintKey, windows.registry.getcellroutine.GetCellRoutine, windows.registry.certificates.Certificates
Filesystem & Object Artifacts	windows.filescan.FileScan, windows.mutantscan.MutantScan, windows.symlinkscan.SymlinkScan, windows.poolscanner.PoolScanner
Networking & System Context	windows.info.Info, windows.statistics.Statistics, windows.getsids.GetSIDs, windows.getservicesids.GetServiceSIDs, windows.svclist.SvcList, windows.svcscan.SvcScan, windows.svcdiff.SvcDiff
User Interface & Session Objects	windows.deskscan.DeskScan, windows.desktops.Desktops, windows.windowstations.WindowStations, windows.windows.Windows
Miscellaneous & Special-Purpose	windows.mbrscan.MBRScan, windows.truecrypt.Passphrase, windows.devicetree.DeviceTree, windows.strings.Strings

Table 3
Categorization of Windows Volatility 3 plugins grouped by forensic analysis domain

Linux Category	Volatility 2 Plugins	Volatility 3 Plugins
Process & Execution Analysis	linux_pslist, linux_psscan, linux_pstree, linux_paux, linux_psenv, linux_psxview, linux_threads	linux.pslist.PsList, linux.psscan.PsScan, linux.pstree.PsTree, linux.psaux.PsAux, linux.pscallstack.PsCallStack, linux.ptrace.Ptrace, linux.kthreads.Kthreads
Kernel & System Structures	linux_pidhashtable, linux_iomem, linux_lsmod, linux_ldrmodules, linux_library_list, linux_banner	linux.kallsyms.Kallsyms, linux.pidhashtable.PIDHashTable, linux.iomem.IOMem, linux.module_extract.ModuleExtract, linux.lsmod.Lsmod, linux.modxview.Modxview, linux.library_list.LibraryList
Malware & Rootkit Detection	linux_malfind, linux_hidden_modules, linux_check_creds, linux_check_idt, linux_check_syscall, linux_check_modules, linux_check_tty, linux_plthook	linux.malfind.Malfind, linux.hidden_modules.Hidden_modules, linux.check_creds.Check_creds, linux.check_idt.Check_idt, linux.check_syscall.Check_syscall, linux.check_modules.Check_modules, linux.check_afinfo.Check_afinfo, linux.netfilter.Netfilter, linux.keyboard_notifiers.Keyboard_notifiers, linux.ebpf.EBPF, linux.tty_check.tty_check
Filesystem & Page Cache	linux_enumerate_files, linux_recover_filesystem, linux_mount, linux_kernel_opened_files, linux_lsof	linux.pagecache.Files, linux.pagecache.InodePages, linux.pagecache.RecoverFs
Networking & Communication	linux_ifconfig, linux_netstat, linux_netscan, linux_route_cache, linux_sk_buff_cache	linux.ip.Addr, linux.ip.Link, linux.sockstat.Sockstat, linux.netfilter.Netfilter
Tracing & Instrumentation	linux_info_regs	linux.tracing.ftrace.CheckFtrace, linux.tracing.perf_events.PerfEvents, linux.tracing.tracepoints.CheckTracepoints
User Environment & Capabilities	linux_bash, linux_bash_env, linux_bash_hash, linux_dynamic_env	linux.bash.Bash, linux.capabilities.Capabilities, linux.envars.Envars
Miscellaneous & Device Interfaces	linux_truecrypt_passphrase, linux_dmesg	linux.kmsg.Kmsg, linux.graphics.fbdev.Fbdev, linux.vmaregexscan.VmaRegExScan, linux.vmayarascan.VmaYaraScan

Table 4
Comparison between Linux plugins in Volatility 2 and Volatility 3, grouped by functional typology

Level	Goal	Full prompt content (preamble + numbered tasks)	JSON keys (output)
Plugin level	Explain one plugin/file; extract IoCs	*System role:* "You are a forensic analyst." Given the text file from a memory dump plugin, perform three tasks: 1) Briefly explain what the plugin/file output shows (describe plugin and summarize output; focus on findings). 2) Extract all observable, investigation-relevant Indicators of Compromise (IoCs) from the file. 3) Keep the explanation neutral and concise.	`file_summary`; `iocs[{value,type,evidence}]`
Dump level	Correlate files, infer verdict & rationale	*System role:* "You are a forensic analyst." You receive (1) multiple short per-file summaries and IoCs (JSON objects), and (2) truncated excerpts from the same dump (for context). *Tasks:* 1) Produce one explanation for the dump (aggregate/interpret the content). 2) Extract all IoCs across everything provided. 3) Decide a `dump_verdict` = `malicious|suspicious|benign|unknown` and explain why using IoCs only; provide `recommended_next_steps`.	`summary`; `iocs`; `dump_verdict`; `confidence`; `rationale`; `recommended_next_steps`
App level	Aggregate dump verdicts & IoCs across volumes into a single app verdict	*No LLM prompt (deterministic aggregation). Inputs:* per-dump results. *Steps:* 1) Count dump verdicts & list dumps by verdict; 2) aggregate IoCs *(type,value,count,dumps,examples)* and keep top N (30–40); 3) decide `app_verdict`: `malicious` if any malicious dumps, else `suspicious` if any suspicious, else `benign` if any benign, else `unknown`; 4) write recap + sections (app Verdict, Dump Verdicts, Top IoCs, Notes).	`app_verdict`; `verdict_counts`; `by_verdict[dumps]`; `top_iocs[type,value,dumps]`; `summary/recap`; `notes`

Table 5
Full prompt hierarchy: from per-plugin and dump-level GPT analyses to App-level aggregation. *Constraint:* Ignore all forensic/tooling artefacts (Volatility, LiME, plugin names, `insmod`/`modprobe`, tracebacks, warnings, errors); they must not appear as IoCs or influence verdicts.