Hochschule für angewandte Wissenschaften München

Fakultät für Informatik und Mathematik

Master's Thesis
for obtaining the academic degree of
Master of Science
in course of studies IT Security

Optimizing Memory Forensics: A Proof of Concept for Automating Queries and Integrating Large Language Models

---

Author:	Jan-Hendrik Lang
Matriculation number:	16300323
Submission Date:	May 25, 2025
Supervisor:	Prof. Dr.-Ing. Thomas Schreck
Advisor:	Prof. Dr.-Ing. Thomas Schreck Prof. Dr.-Ing. Erik Krempel

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, May 21, 2025

Jan-Hendrik Lang

Abstract

Memory forensics plays an important role in modern digital investigations in terms of detecting stealthy, fileless malware, and advanced persistent threats. Moreover, large language models (LLMs) have shown promise in different cybersecurity tasks. However, their integration into structured forensic workflows remains underexplored, highlighting the need to evaluate whether LLMs can improve the efficiency and accessibility of memory forensic analysis. In this thesis, we designed and implemented MemoryInvestigator to integrate intelligence based on an LLM into memory forensic workflows and evaluate multiple LLMs, including *OpenAI GPT4o*, *OpenAI o1*, *Gemini 2.0 Flash*, *Gemini 2.0 Flash-Thinking*, *Grok 3*, *and Grok 3 with thinking mode enabled*. We collected memory dumps encompassing a variety of attack scenarios: *process injection (via msfvenom)*, a *PowerShell Empire-based attack*, and real-world malware including *Quasar RAT*, *MassLogger*, *DarkCloud*, *LockBit*, *and LockiBot*. Accuracy, precision, recall, and F1 score metrics were used to assess detection performance, supported by statistical analyses such as analysis of variance (ANOVA) and correlation tests. The findings show that the reasoning-based ('thinking') LLM model outperforms standard models. OpenAI o1 and Gemini Flash-Thinking excel at decoding base64 obfuscated payloads, while Grok3 leads in detecting network anomalies. All LLM-based approaches suffer from high false-positive (FP) rates, reflected in low precision (often <20%). This tendency appears to stem from the precautionary principle in AI safety orientation, leading to models erring on the side of caution and occasionally hallucinating plausible threats when faced with ambiguous or incomplete evidence. The LockBit indicator of compromise (IoC) could not be detected with the LLM because the IoCs lie beyond the *Volatility3* modules used. Due to this reason and the limited size of the context window from the LLM, it is essential to select appropriate data. This thesis demonstrates the practical viability of integrating LLMs into memory forensics and supports their potential to improve accessibility, reduce manual effort, and guide less-experienced analysts through complex forensic data. These results lay the groundwork for hybrid forensic systems that combine symbolic analysis, domain-specific heuristics, and LLM-driven intelligence to enhance the effectiveness and user-friendliness of forensic investigations.

Contents

1 Introduction **1**
1.1 Motivation and Background . . . 1
1.2 Problem Statement . . . 1
1.3 Research Gap . . . 1
1.4 Research Questions . . . 2
1.5 Hypotheses . . . 2
1.6 Objectives . . . 3
1.7 Scope and Limitations . . . 3
1.8 Methodological Approach . . . 4

2 Foundations **5**
2.1 Digital Forensics and the Role of Memory Forensics . . . 5
2.2 Windows Memory Management Structures . . . 5
2.3 Volatility3: Architecture and Plugin System . . . 6
2.4 Large Language Models . . . 6

3 Related Work **7**
3.1 Large Language Models in Cybersecurity . . . 7
3.2 Large Language Models in Digital Forensics . . . 8
3.3 Large Language Models in Volatile Memory Forensics . . . 10
3.4 Challenges and Limitations of Large Language Models . . . 11
3.5 Study Contributions . . . 13

4 Design and Implementation of MemoryInvestigator **15**
4.1 Requirements and Goals of the System . . . 15
4.2 Architecture Overview . . . 15
4.3 Technology Choices . . . 17
4.4 User Interface Frontend . . . 18
4.5 Orchestration . . . 19
4.6 Forensic Engine . . . 19
4.7 Data Structuring: Tree-of-Table Algorithm . . . 20
4.8 Data Structuring: Retrieval-Augmented Generation . . . 22
4.9 Integration with Large Language Model Processing . . . 22
4.10 Large Language Models Used . . . 23
4.11 System Overview . . . 23

5 Experimental Setup **25**
5.1 Environment Configuration and Tools . . . 25
5.2 Malware Sample Selection . . . 25
5.3 Test Cases Overview . . . 26
5.4 Test Case: Process Injection With msfvenom . . . 26

5.5 Test Case: PowerShell Empire Attack 28
5.6 Test Cases: Real-World Malware Samples 30

6 Evaluation **34**
6.1 Evaluation Metrics: Accuracy, Precision, Recall, and F1 Score 34
6.2 Evaluation Metrics: Statistical Analysis 35
6.3 Overall Results 35
6.4 Overall Detection Performance 36
6.5 Model-Wise Performance Differences 37
6.5.1 Reasoning ('Thinking') vs Standard Modes 38
6.5.2 OpenAI Models 38
6.5.3 Notable Strengths 38
6.5.4 Efficiency 39
6.6 Malware Family-Specific Results 40
6.6.1 Quasar RAT 41
6.6.2 MassLogger 44
6.6.3 Process Injection (msfvenom) 44
6.6.4 PowerShell Empire 45
6.6.5 DarkCloud 45
6.6.6 LockiBot 45
6.6.7 LockBit Ransomware 45
6.7 False Positives 46
6.8 False Negatives 47
6.9 Statistical Significance 47

7 Discussion **49**
7.1 Key Findings 49
7.2 Implications for Forensics and LLM Integration 50
7.3 Limitations of the Study 51
7.4 Recommendations for Practitioners 53

8 Conclusion and Future Work **56**
8.1 Summary of Contributions 56
8.2 Further Research 57
8.3 Final Thoughts 58

Bibliography **63**

A Design and Implementation of MemoryInvestigator **74**
A.1 Third-Party Dependencies Overview 74
A.2 MemoryInvestigator Project Structure 75
A.3 Standard Installation 76
A.4 Additional Linux Steps 76

B MemoryInvestigator Screenshots **77**
B.1 Data Input 77
B.2 Display Data as a Table 77
B.3 Display Data as a Graph 78
B.4 Search Data in the Graph 78
B.5 Extract Data 79
B.6 Tree-of-Table Analysis 79
B.7 Standard RAG 80
B.8 Experimental Forensic RAG 80
B.9 Temporary Folder Structure 81

C Tree-of-Table Algorithm **82**
C.1 Mapped Volatility3 Modules to Their Specific Fields 82
C.2 Pseudocode Tree-of-Table Builder 82
C.3 Tree-of-Table Algorithm Graphic 84

D Excerpt of Trees **85**
D.1 Basic-build Tree 85
D.2 Custom-build Tree 86

E Malware Behavior **87**
E.1 MassLogger 87
E.2 DarkCloud 88
E.3 LockiBot 88
E.4 LockBit (aka Worm.m0yv) 89

F Evaluation Summary of Experimental Results **90**
F.1 Tabular Summary of Experimental Results 90
F.2 Summary of Statistical Tests 91
F.3 Tukey HSD Test Results for LLM (F1 Score Comparisons) 91

G Examples MemoryInvestigator Results **92**
G.1 Volatility3 windows.dlllist Quasar RAT Output of PID 2032 92
G.2 Volatility3 windows.netscan Quasar RAT Output 96
G.3 QuasarRAT GPT4o 101
G.4 QuasarRAT Gemini 1.5 Pro 102

1 Introduction

1.1 Motivation and Background

Digital forensics is indispensable for modern cybersecurity operations, e.g. incident response and threat hunting. Every desired interaction between a user and their computer automatically leads to data and processes in the random access memory (RAM). Given that the same applies to malware, gaining insights into memory forensics is critical for detecting advanced threats such as fileless malware and process injection techniques [50], [104]. *Volatility3* is the de facto standard for volatile memory forensics.

Furthermore, large language models (LLMs) have seen notable advances, showing potential in tasks ranging from summarizing complex forensic artifacts to correlating evidence across diverse data sets [81], [96]. In particular, Oh et al. [64] described how they automated and transmitted Volatility 2.5.2 pslist, vadinfo and malfind modules to the GPT-3.5 API through prompt engineering, which enabled them to successfully triage ransomware-infected memory.

This thesis investigates the integration of LLMs in a memory forensic workflow, employing the design science research process of Peffers et al. [68].

1.2 Problem Statement

Memory forensic investigations are crucial for detecting advanced cyber threats such as fileless malware, process injection, and remote access trojans. However, performing in-depth memory forensics with tools such as *Volatility3* requires significant expertise and manual effort. The complexity of interpreting plugin outputs, correlating different forensic artifacts, and identifying subtle indicators of compromise (IoCs) can overwhelm even experienced analysts.

Moreover, while LLMs have shown promise in assisting with forensic analysis, their effective integration into operational workflows remain underexplored, especially for structured data such as memory forensic outputs. There is a lack of empirical evidence on whether LLMs can meaningfully support or automate memory forensic triage without compromising the accuracy and reliability of threat detection.

Thus, a solution is necessary that automates parts of the forensic memory workflow, leverages the reasoning capabilities of modern LLMs to assist in triage, and improves the accessibility and efficiency of memory analysis without introducing significant new risks.

1.3 Research Gap

This research aims to systematically investigate the strengths and weaknesses of LLMs in the context of volatile memory forensics. Specifically, it focuses on evaluating the ability of LLMs to analyze structured outputs generated by *Volatility3* plugins. It is important

to note that this study does not benchmark LLMs against traditional manual forensic analysis workflows. While previous studies have demonstrated that LLMs can assist in digital forensic tasks, empirical evaluations comparing different models in the analysis of volatile memory artifacts are still lacking. This research seeks to address this gap by providing a comparative analysis of multiple LLMs across diverse malware scenarios.

1.4 Research Questions

Our leading research question—initially proposed in the master's thesis application—is: "Can the automation of *Volatility3* be used in combination with an LLM as a proof of concept to make queries in memory forensics more efficient and user-friendly?"

We derive the following specific research sub-questions to explore this overarching question in a structured and detailed manner:

1. *How do different LLMs perform in detecting malware based on memory forensic data generated by Volatility3?*
   We assess a variety of LLMs from different providers, including standard and reasoning-enhanced configurations, comparing their detection performance using accuracy, precision, recall, and F1 scores in diverse malware scenarios.
2. *Do reasoning ('thinking mode') LLMs improve forensic detection performance?*
   We investigate whether reasoning-based or normal LLMs lead to statistically significant improvements in detection metrics.
3. *What are the main limitations of using LLMs for automated memory forensic triage?*
   We analyze the causes of false positives (FPs) and false negatives (FNs) in LLM decisions and discuss whether integrating baseline system knowledge could mitigate this problem.

1.5 Hypotheses

This research is guided by the following two core hypotheses, substantiated by an experimental evaluation using multiple LLMs applied to structured memory forensic data derived from *Volatility3*.

1. Reasoning ('thinking mode') LLMs outperform standard LLMs in detecting malware in memory dumps, reflected in significantly higher F1 scores and accuracy metrics observed in multiple malware families. A two-way analysis of variance (ANOVA) confirmed statistically significant main effects and interactions between the LLM reasoning mode and the type of malware, while post-hoc t-tests showed that the 'thinking mode' leads to an improved F1 score ($p < 0.05$) and accuracy ($p < 0.01$). These findings suggest that reasoning capabilities help LLMs to better correlate weak signals across various plugin outputs.
2. Natural language interfaces—when paired with LLMs—enhance the usability and accessibility of memory forensic tools, enabling even non-specialists to conduct simplified, preliminary analyses. However, this accessibility comes with the risk that

LLM-based evaluations are inherently limited by the forensic scope provided. Our study reveals that a high LLM reported accuracy does not necessarily guarantee full coverage of relevant IoCs, particularly when evidence lies outside the default *Volatility3* modules (e.g., Windows Registry, EVTX event logs, or raw memory strings). Therefore, careful curation and expansion of forensic input data are essential to reduce false negatives and support comprehensive memory triage.

1.6 Objectives

The primary objective of this thesis is to design and implement a functional MemoryInvestigator prototype that enhances the efficiency and accessibility of memory forensic analysis. This includes developing a wrapper that automates selected *Volatility3* Version 2.8 modules and transforms their output into a structured format suitable for analysis by LLMs. The tool aims to streamline the forensic workflow by reducing manual steps and supporting the IT expert in finding possible IoCs. Furthermore, the thesis seeks to evaluate the effectiveness of various LLMs (e.g., *OpenAI GPT-4o*, *Gemini*, *Grok*) in detecting IoCs across multiple malware scenarios. The objectives also include analyzing the strengths and weaknesses of reasoning-based versus standard LLMs, quantifying their detection performance, and identifying sources of FPs and FNs. A final goal is to improve the usability of memory forensics for non-specialists by offering a natural language interface that supports intuitive interaction with complex forensic datasets.

1.7 Scope and Limitations

This thesis aims to address specific aspects of memory forensics within a controlled and focused experimental framework. The analysis is limited to the Windows operating system as it remains the most frequently targeted platform for malware and provides a well-documented memory architecture that is suitable for forensic investigations. The toolset is centered on *Volatility3*—the current de facto standard for volatile memory analysis—due to its modularity, active development, and widespread use in professional investigations. To ensure practical feasibility, the study is restricted to a selection of representative use cases, including both self-generated and real-world malware samples that demonstrate a range of forensic challenges such as process injection, keylogging, and remote access trojans. In terms of LLM integration, the focus lies on the application of known language models from major vendors. This approach enables a realistic evaluation of LLMs in forensics without the computational overhead of full-scale retraining. These boundaries ensure a manageable research scope while still yielding meaningful insights.

1.8 Methodological Approach

The design science research methodology from Peffers et al. [68] is used for this proof of concept.

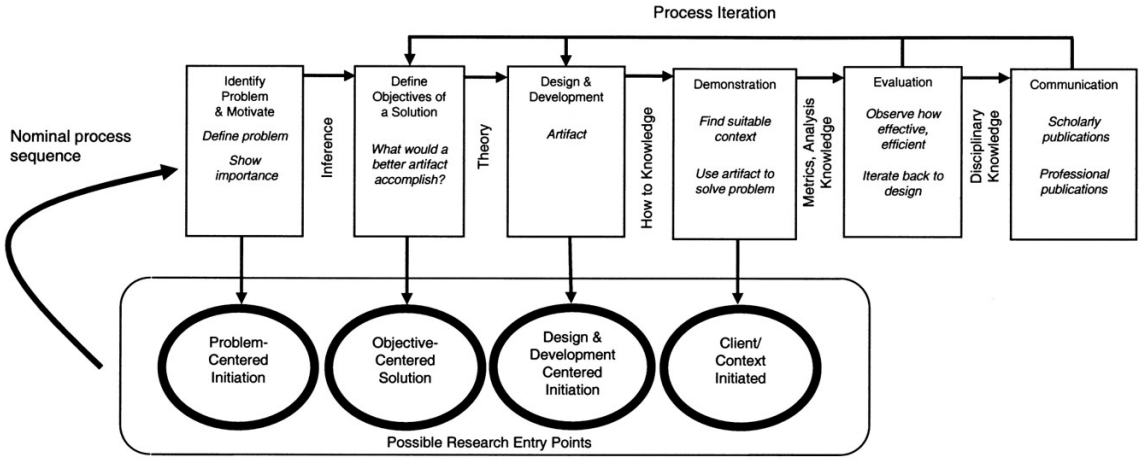


Figure 1.1: *Design Science Research Process [68]*

The focus is placed on developing a functional MemoryInvestigator prototype that automates *Volatility3* Version 2.8 and LLMs to simplify memory forensics. As shown in Figure 1.1, the process is as follows:

1. Problem identification: Put simply, memory forensics is highly complex even for IT experts and based on labor-intensive manual processes. Given that modern attack techniques make memory forensics essential, a support of the IT expert is essential.
2. Target definition: Developing a prototype that automates *Volatility3* Version 2.8 and simplifies analysis through the use of an LLM, which should instruct the IT expert and find IoCs by itself.
3. Development of the artifact: Creating a *Volatility3* wrapper to simplify forensic investigation and enable interaction with LLMs.
4. Evaluation: Comparison of different LLMs based on accuracy, precision, recall and F1 score metrics, and statistical analyses (ANOVA and correlation tests).
5. Communication: Presentation of the results in the master's thesis.

2 Foundations

2.1 Digital Forensics and the Role of Memory Forensics

Memory forensics is a core discipline within digital forensics that focuses on capturing and analyzing RAM to identify artifacts left behind by benign and malicious programs. Memory forensics captures a snapshot of its volatile state at a particular time, thus with no guarantee that the evidence is complete or consistent. Because all active processes, network connections, and user interactions are in memory during exectuion, RAM captures are a gold mine for uncovering evasive threats such as fileless malware, code injection, and advanced persistent threats (APTs). Therefore, volatile memory often contains running processes and their hierarchies, loaded dynamic link libraries (DLLs), active network connections, executed command-line arguments, security identifiers (SIDs) and handle tables, injected code or hooks that indicate malware presence. SANS [93] provides a broader set of recommendations, including (1) identifying invalid processes, (2) analyzing process DLLs and handles, (3) checking network artifacts, and (4) searching for indications of code injections.

2.2 Windows Memory Management Structures

Windows organizes memory around several kernel-maintained structures [89] that facilitate memory parsing. The Kernel Processor Control Region (KPCR) is a per-processor data structure that holds a processor-specific state, including points to debugging and scheduling metadata. Every logical processor has its own KPCR, and each KPCR contains a pointer to the Kernel Debugger Block (KDBG). The KDBG holds a reference to the PsActiveProcessHead, the entry point to a circular doubly linked list of all currently active processes. Each process on a Windows system has a corresponding _EPROCESS BLOCK structure, containing information such as process ID (PID), image name, creation time, and pointers to related objects. The ActiveProcessLinks field in _EPROCESS BLOCK forms a doubly linked list using FLINK (forward links) and BLINK (backward links) pointers. Traversing this list allows tools to enumerate all active (running) processes. The list begins with the system process (PID 4), followed by essential system processes such as smss.exe, csrss.exe, and winlogon.exe. Each _EPROCESS BLOCK also contains:

1. Process environment block (PEB): A pointer for accessing the executable path, command-line arguments, loaded DLLs, and environment variables.
2. Virtual address descriptor (VAD): Lists memory regions used by the process.
3. Handle table: For file, registry, and thread handles.
4. Threads: A list of threads. A process without threads is an exited process.
5. Tokens: Contains information such as SIDs and privileges.

Direct kernel object manipulation (DKOM) is a common anti-forensics technique where malware unlinks its _EPROCESS BLOCK from the active process list. Consequently, it

becomes invisible to standard enumeration tools such as Task Manager, while still running in memory. This requires forensic tools that rely on alternative process detection methods such as pool scanning.

2.3 Volatility3: Architecture and Plugin System

Volatility3 [27] is an open-source memory forensic framework and the de facto standard for performing volatile memory forensics. *Volatility3* uses the KDBG to ensure correct symbol resolution, among other things. *Volatility3* enables analysts to extract and analyze volatile memory artifacts from memory dumps. Normal scans use the doubly linked list and directly parse the memory structures to retrieve information about running processes, loaded DLLs, active network connections, and virtual memory mappings. Common plugins are pslist, dlllist, netstat, and handles. Pool scans are used to analyze kernel memory pools and extract hidden or unlinked objects, scanning the raw memory—for example—for an _EPROCESS BLOCK. Common plugins are psscan and netscan. Alternative tools are MemProcFS [32] and MemProcFS Analyzer [98]. MemProcFS is a real-time memory analysis framework that mounts memory dumps as virtual file systems, which makes navigating through volatile memory structures easier and allows analysts to examine memory artifacts interactively. The MemProcFS Analyzer enhances the standard MemProcFS by providing additional automation. The Rekall project is no longer maintained.

2.4 Large Language Models

LLMs are transformer-based deep learning models trained on massive corpora of text data to perform a wide range of natural language understanding and generation tasks. While LLMs are proficient in activities such as text creation, summarization, translation, and understanding context, they lack inherent logical reasoning or profound conceptual comprehension. Their answers come from probabilistic patterns rather than clear rule-based reasoning. By contrast, reasoning-based systems depend on explicit logic, symbolic AI, and rule-based processing to reach conclusions. These systems employ formal logic (such as propositional logic and first-order logic), knowledge graphs, and expert systems to derive answers via reasoning rather than relying on statistical forecasts. They perform exceptionally well in domains that require structured reasoning and clarity, including proving theorems, decision-making, and legal or medical evaluations [39].

3 Related Work

Research into the intersection of LLMs in combination with cybersecurity, digital forensics, and memory forensics is still emerging.

3.1 Large Language Models in Cybersecurity

Various cybersecurity tasks have adopted LLMs, demonstrating their ability to assist analysts and parse complex information. Traditional cybersecurity tools often rely on signature-based detection and static rules, and they struggle to keep up with the pace of new attacks. In a systematic literature review, Zhang et al. [107] provide a broad overview of LLMs in cybersecurity. Recent work has explored the use of LLMs for a variety of cybersecurity tasks, including cyber threat intelligence (CTI), (in)secure code generation, vulnerability, malware, and anomaly detection. Sai et al. [73] also highlight how Generative AI, including LLMs, is being integrated into real-world cybersecurity tools by industry leaders, aiming to predict and proactively counter sophisticated threats.

In CTI, LLMs are used to extract indicators and summarize unstructured text from threat reports. For example, Siracusano et al. [85] developed a framework that employs LLMs to parse raw threat reports and convert them into structured intelligence, while improving the efficiency of handling large volumes of CTI. Similarly, Hu et al. [43] fine-tuned an LLM to construct threat intelligence knowledge graphs from textual reports, effectively mapping relationships between IoCs. Clairoux-Trepanier et al. [17] built an LLM-based process to extract key CTI variables from cybercrime forums with a high precision (~90%) and recall (~88.2%). Mitra et al. [61] showed that it is also possible to fuse global (CVE, CWE) and local (customized data) knowledge databases in LocalIntel. Researchers have also combined LLMs with retrieval techniques to enhance analysis. Fayyazi et al. [23] introduce a retrieval-augmented generation approach that supplies an LLM with relevant context from the MITRE ATT&CK database and significantly improving the model's ability to explain adversary tactics. Shafee et al.'s. [82] evaluations on an open-source intelligence dataset reveal that while LLMs (e.g., ChatGPT, GPT-4All) perform well in classifying threat data, they still struggle with precise cybersecurity entity recognition, thus highlighting the need for further domain-specific tuning [82]. These advances in CTI automation suggest that language models can distill complex threat data into actionable insights. This capability can be analogously useful for interpreting volatile memory artifacts containing traces of attacker activity.

LLMs have also shown promise in code-focused security tasks such as vulnerability discovery and malware analysis. Gao et al. [33] showed progress in using LLMs to detect software vulnerabilities, indicating that LLMs can identify certain vulnerability patterns in code with high-quality training data. In malware detection, LLMs act as intelligent assistants for static and dynamic analysis. Pearce et al. [67] demonstrated that models such as OpenAI's Codex can aid reverse engineering by generating explanations for decompiled C code and thus help analysts to understand the software functionality from the specific

binary. Tan et al. [90] built on this and created a specialized LLM for decompilation (LLM4Decompile) that translates machine code back into readable source code, which is valuable for dissecting malware found in executables. This capabilities directly support malware analysis workflows. Zahan et al. [105] used an LLM-powered pipeline to detect malicious JavaScript packages in the npm ecosystem, employing techniques such as iterative self-refinement and chain-of-thought prompting. Their approach achieved higher precision and lower false positives than conventional static scanners. Lu et al. [56] introduced the MALSIGHT framework in which an LLM iteratively summarizes binary malware by comparing malicious machine code with benign pseudocode, producing human-readable descriptions of malware behavior.

For system log anomaly detection, Karlsen et al. [49] fine-tuned over 60 language models (BERT, RoBERTa, GPT variants), finding that LLM-based models can effectively adapt to diverse log types. Frameworks such as LogGPT [38] and LogPrompt [52] use prompt engineering (including chain-of-thought reasoning) to parse unstructured logs and identify suspicious events, moving beyond rigid regex-based parsers. Beyond research, the industry is developing LLMs specialized for security operations. In 2023, Microsoft launched the Microsoft Security Copilot, pairing OpenAI's GPT-4 with the Defender ecosystem to assist incident response [74]. The combination of LLMs with SIEM systems reflects that providers are trying to use generative AI for event correlation, drafting remediation steps or generating detection rules from natural language [100]. Instead of focusing on a known IoC, an LLM relies on semantic knowledge (understanding attacker TTPs described in text). This flexibility enables detecting patterns that traditional analysis might have missed [100].

These few examples of existing research illustrate how integrating LLMs into cybersecurity can streamline labor-intensive analyses, from scanning code for vulnerabilities to explaining malware logic. This foundation is highly relevant to memory forensics, where similar pattern recognition and summarization capabilities can help to make sense of raw memory dumps.

3.2 Large Language Models in Digital Forensics

Digital forensics is traditionally very labor-intensive and expert-driven. Experts run various tools (e.g. *Volatility3*, *The Sleuth Kit (TSK)*, *Bulk Extractor)* and manually interpret the findings to generate a report including IoCs. The full range of digital forensic investigations is promising, from evidence analysis and artifact triage to reporting. Researchers have now begun exploring LLMs to assist with investigative and documentation processes. Yin et al. [103] provide a systematic overview of how LLMs have transformed digital forensics. Wickramasekara et al. [96] explored AI agents in digital forensics, underscoring LLMs' capacity to reduce technical barriers—especially in the analysis phase—and streamline investigative workflows. Built around ChatGPT-like models, their framework aimed to make specialized forensic tasks accessible to less-experienced analysts. Given that LLMs traditionally consume significant analyst time, one research area involves automating the generation of forensic reports.

Michelet and Breitinger [60] propose a method for LLMs to draft common sections of a digital forensic report by learning from the structure and content of past cases. In their study, both online and local models (e.g. ChatGPT and LLaMA) had the task to writing report segments (overview, methodology, findings, etc.) based on case data. The results showed that LLMs can produce well-structured narratives for portions of the report, whereby summarizing evidence or writing a background overview is comparable to human-written text. However, the findings also suggest that LLM-generated content can lack precision in technical details and sometimes include subtle inaccuracies, especially in sections requiring deep analysis of evidence [60]. LLMs can reduce the burden of writing boilerplate or straightforward descriptive content, although expert review and editing is still necessary to ensure correctness. The use of LLMs for documentation indicates potential for forensic reporting, enabling investigators to focus more on analysis while the model handles first-draft writing in routine areas. Deploying LLMs as interactive assistants during forensic examinations is another research area.

Wickramasekara and Scanlon [97] propose a framework using AutoGen AI agents and LLMs to automate digital forensic investigations based on natural language inputs, potentially streamlining repetitive subtasks and easing the analyst's workload. Scanlon et al. [81] assessed ChatGPT's capability to assist in digital forensic investigations by having it analyze and answer questions about evidence from a case. In their experiments, the model was given details from a forensic scenario, including file system metadata, log entries, browser history, and memory artifacts. With this information, it asked investigative questions (e.g., "Was a particular file downloaded and executed by the suspect?") [80]. The LLM demonstrated an ability to infer user activity from such data, correctly identifying instances such as file downloads or program executions from narrative clues in logs [81]. This suggests that an LLM can synthesize disparate pieces of forensic evidence into a coherent explanation of events. On the other hand, Scanlon et al. [81] report challenges, including the fact that the model sometimes misinterpreted artifacts or gave confident answers that were incorrect due to ambiguity or noise in the input data, reflecting a known limitation of current LLMs. This underlines the need for fine-tuning LLMs on forensic-specific data or developing better prompt strategies to improve accuracy. Beyond direct analysis of case data, LLMs are also used to integrate external knowledge into investigations.

Rajapaksha et al. [71] describe a retrieval-augmented system that can answer an investigator's query by fetching relevant information from either a knowledge base, containing curated information about attribution cyber-attack investigations or outside resources provided by the users and feeding it into the LLM. Such a system can help to attribute an attack by linking artifacts to known threat actor techniques, combining forensic evidence with global cybersecurity knowledge. The use of LLMs as report drafters, analytical assistants, and intelligent query responders is poised to accelerate forensic workflows. However, due to the reliability required in digital forensics, an LLM's output must still be validated by a human analyst.

3.3 Large Language Models in Volatile Memory Forensics

Volatile memory forensics deals with capturing and analyzing a system's RAM to uncover runtime artifacts such as running processes, injected code, credentials, and malware that run only in memory. As the field of volatile memory forensics is very small, most existing work has focused on its application for cybersecurity in general. Integrating LLMs into the forensic process is an emerging research area that aims to support the analyst in interpreting massive and complex memory dumps. A key potential use is malware detection and triage in RAM snapshots. Oh et al. [64] provided a prominent early application of LLMs in volatile memory forensics, introducing volGPT using GPT-3.5 to triage ransomware processes in memory captures with Volatility 2.5.2. In volGPT, the output from Volatility plugins (e.g., pslist, vadinfo, malfind) is prompted to the LLM. With minimal manual intervention, the authors demonstrated that volGPT is able to identify ransomware patterns. A recent systematic literature review by Hamid and Rahman [37] underscores the rising importance of volatile memory forensics in cybersecurity, pointing out the need for innovative techniques, such as LLMs, to handle increasingly complex threats.

Examining memory-derived data (disassembled code, strings, or process lists from a memory image) and identifying patterns suggestive of malicious activity could also be a capability of an LLM, supported by current advances in using LLMs for code analysis support. A novel fuzz tuning technique that enhances LLM-based code understanding by integrating dynamic program behavior derived from fuzzing test cases was introduced by Zhao et al. [108]. It achieved state-of-the-art results in clone detection and code classification. Zahan et al. [105] showed that an LLM-based workflow can detect malicious code in software packages, implying that a similar approach might flag malicious code present in a memory dump. Scanlon et al. [81] experimented with the application of ChatGPT to generate Python scripts for AES and RSA key extraction in volatile memory. Later, ChatGPT recommended using Volatility and the Python subprocess library for that specific purpose. To translate low-level runtime behavior, Yan et al. [102] utilized GPT-4 to generate explanations for sequences of API calls during malware execution. Applying this idea to memory forensics, LLMs could take an extracted sequence of system calls or function traces from memory and try to explain what the program was doing (accessing files, injecting threads, initiating network connections), thus helping an analyst to quickly understand suspicious behavior.

LLMs could assist in identifying known malware traces in memory by comparing the patterns found in artifacts with its learned knowledge of malware techniques. An LLM could recognize an unusual kernel module or code segment in a memory dump by matching the behavior or code patterns of a known rootkit, and suggest its malicious nature along with a description. Lu et al. [56] have taken steps in this direction with MALSIGHT, which summarizes binary malware functionality. Such capabilities are directly relevant to memory analysis. The analyst could give a piece of machine code extracted from memory and the LLM could translate it into a human-readable form [56]. With this capability, the analyst receives support in understanding the role of that code without the manual and labor-intensive effort of reverse engineering it from scratch.

Researchers have also explored how expanding the knowledge with an external knowledge base can improve an LLM. One interesting approach is the use of retrieval-augmented generation (RAG) in volatile memory analysis. Fayyazi et al. [23] demonstrated that RAG can improve the accuracy of an LLM's output by grounding it in relevant data. While their work focused on explaining attack techniques by retrieving MITRE ATT&CK entries, based on the same idea, an LLM could retrieve information such as OS-specific data structures, known malware signatures, or library documentation when encountering an unknown memory artifact, and present an explanation or assessment that references this retrieved knowledge. This could reduce hallucinations and therefore enhance the trustworthiness of the LLM's analysis of memory content. Instruction-tuning or fine-tuning LLMs on memory forensics tasks is another idea. Domain-specific tuning can significantly improve the performance of an LLM in specialized tasks. In vulnerability detection, Du et al. [22] achieved better generalization by multi-task fine-tuning a model (VulLLM) on security datasets, which suggests that training on labeled memory forensic data could offer a similar benefit. Fine-tuning an LLM on memory dumps annotated with ground truth (e.g., which processes are malicious, which code injections occurred) would likely be more adept at recognizing those patterns in new dumps. Sharma et al. [83] tailored a domain-specific model in ForensicLLM for digital forensic applications. The authors found that ForensicLLM outperformed general LLMs in accuracy and relevance within the digital forensics context, as fine-tuning the LLM on forensic-specific datasets (Q&A samples from curated digital artifacts and digital forensic research articles) enhances the performance in tasks such as evidence analysis and report generation.

In practice, an LLM-driven volatile memory forensics system could only work in tandem with traditional tools. The memory image could be pre-processed by frameworks such as *Volatility3* to extract human-readable artifacts, which are then fed to the LLM for interpretation, summary, and cross-correlation with threat intelligence. Early conceptual studies indicate that such a system could rapidly highlight anomalous processes, suspicious loaded modules, or rare API call sequences and explain why they warrant attention [94].

At this point, it is important to recognize that fully realizing this vision requires overcoming many challenges, and the current implementations remain experimental. The complex and low-level nature of volatile memory and the need for precise accuracy mean that LLMs in volatile memory forensics must be still carefully validated by an analyst. As research progresses, we expect to see LLMs increasingly assist in memory forensic investigations. They could automatically generate a timeline of an attack from memory snapshots or offer advice to triage the memory and thus become an intelligent co-analyst working alongside human analysts.

3.4 Challenges and Limitations of Large Language Models

While LLMs offer powerful new capabilities, there are also significant challenges and limitations to their use in general, including in domain-specific tasks such as cybersecurity and digital forensics. Primary concerns relate to the accuracy and reliability of the LLM

output. Hallucinations are the main challenge for LLMs given that they could produce false or misleading information [107]. A benign process could be flagged as malware, or a summary could include misleading information and incorrect IoCs. Verifiability and accuracy are key for digital forensics. Ottmann et al. [65] show that even traditional memory forensic methods suffer from significant inconsistencies, indicating that LLMs must be evaluated not only on accuracy but also on their robustness against unreliable input data. Michelet and Breitinger [60] also see hallucinations as a significant drawback in AI-written reports. They emphasize the importance of human review and validation processes to ensure hallucination-free data. In the case of criminal investigations, this is also due to the chain of custody. Mitigating this would require the model to be grounded in domain-specific knowledge and factual data. RAG is one idea to address this issue. Fayyazi et al. [23] significantly improved an LLM's explanations of attack techniques by providing relevant context, thus reducing mistaken or context-free responses. However, domain adaptation remains a challenge as general-purpose LLMs are usually trained on natural language and code from public sources, whereby they typically lack an understanding of specialized forensic artifacts (e.g. *windows.pslist*, *windows.malfind*). A fine-tuned LLM could interpret or identify critical details in a memory image. Du et al. [22] demonstrate that models fine-tuned on domain-specific tasks (in their case, software vulnerabilities) perform much better than untuned models. In the niche of memory forensic scenarios, creating large datasets for training could enhance LLM performance. However, creating such datasets is difficult due to the sensitive and varied nature of real incident data. The research community has only begun to develop benchmarks for security-related LLM evaluation. Bhatt et al. [9] introduce a secure coding benchmark and Tihanyi et al. [92] present a cybersecurity knowledge evaluation set. Similar benchmarks for forensic analysis are missing. The lack of standardized evaluation makes it difficult to quantify an LLM's effectiveness or compare it against traditional forensic tools, which makes it challenging to trust and widely adopt these models. One practical limitation is the input context window of current LLMs, as analyzing an entire memory dump might exceed it and force the analysts to slice the data and potentially lose global context. Lin and Mohaisen [51] emphasize that context window length significantly affects LLM performance in vulnerability detection, highlighting the importance of prompt design and input size constraints in memory analysis scenario.

Security and ethical issues further complicate the deployment of LLMs in volatile memory forensics. Law enforcement environments have strict restrictions of the use of AI from the recently enacted EU AI Act. Evidence in criminal investigations relies on the dependability of the tools and techniques used . Furthermore, the interpretability of LLMs is actually restricted. Although traditional security tools offer a high degree of interpretability for analysts, LLM decision-making processes often lack transparency. Finally, there are also adversarial risks that manipulate the LLM, such as specially crafted text in log files or alerts to fool or break the LLM.

From a security standpoint, LLMs can become targets of adversarial manipulation. Attackers might exploit an LLM-based analysis system through prompt injections or crafted inputs in the raw memory file. Shi et al. [84] described that a backdoor attack with hidden triggers in the input can cause an LLM to produce incorrect or attacker-controlled

outputs. In a memory forensic scenario, this could mean that a embedded sequence of bytes or text in a memory sample causes the LLM to ignore certain malicious data or misclassify it as benign. Xu et al. [101] also show that carefully crafted adversarial prompts can trick LLMs into violating the safeguard mechanisms (jailbreaking). This raises concerns that an LLM integrated into a forensic pipeline could be coerced into overlooking evidence or revealing sensitive analysis steps unless it is properly secured. Chernyshev et al. [12] propose leveraging LLM invocation logs as a novel digital forensic evidence source in LLM-integrated applications, suggesting that forensic readiness must also extend to the LLMs themselves.

The lack of explainability in many LLM decisions is another limitation. When an LLM flags a process from a memory dump malicious, it might be difficult to extract the idea behind that decision. Palacio et al. [66] tackled this issue in the context of code analysis, aligning the ASTxplainer model to predict with interpretable structures (AST nodes). Explainable AI for forensic reasoning is still emerging. For digital forensics, where expert witnesses must often explain their methods in court, a "black box" AI that cannot justify its conclusions is problematic. Accordingly, transparency and the development of tools that can translate an LLM's internal reasoning into auditable explanations are necessary.

Data protection and legal concerns are also important. Memory dumps frequently contain personal data, credentials, and other confidential information. Using online LLM APIs in investigations could violate data-handling policies or laws if sensitive content is uploaded to third-party servers. To avoid exposing case data, Michelet and Breitinger [60] highlight using local models in their forensic report experiments. Retaining analysis in-house would force organizations to rely on less powerful open-source models or invest in expensive infrastructure to host large models securely. In summary, LLMs open exciting avenues in cybersecurity and especially volatile memory forensics, although current issues exist concerning reliability, security, interpretability, and data governance. Ongoing research topics towards building robust, explainable and domain-trained LLMs will be critical in overcoming these limitations and enabling safe integration into forensic workflows.

3.5 Study Contributions

In contrast to existing studies that focus on other cybersecurity topics or have a narrow set of Volatility plugins, a specific malware type (e.g., ransomware), or single LLMs, memory forensics in practice deals with heterogeneous data (process listings, DLL data, loaded modules, memory mappings, network information, thread activity, etc.) and various malware techniques. Our research aims to expand the breadth of analysis by incorporating additional plugins such as psscan, netscan, hollowprocesses, processghosting, suspicious_threads, malfind, ldrmodules, svcdiff, cmdline, dlllist, and getsids. We also evaluate different LLMs from different vendors such as OpenAI, Gemini, and xAi, and different modes (normal vs. 'thinking'), providing insights into their relative strengths and weaknesses. Finally, we test memory captures from a variety of malware families

(e.g., remote access trojan, credential-stealing, process injection malware, and PowerShell attacks) to gauge the generalization of LLM-driven detection.

4 Design and Implementation of MemoryInvestigator

4.1 Requirements and Goals of the System

MemoryInvestigator was conceived as a companion to volatile memory forensics. The goal of the prototype is to enhance the efficiency and accessibility of memory forensics for an IT expert and reduce labor-intensive tasks. In order to reach this goal, we set up functional and non-functional requirements, which are listed in Table 4.1.

Category	Requirement
Functional Requirements	Cross-Platform Compatibility: Running on Windows and Linux.
	Volatility3 Wrapper: Automate execution of *Volatility3* modules.
	Front-End: Provide an analyst-friendly front-end.
	Interactive Process Exploration: Interactive exploration of processes as a table and graph.
	AI-Assisted Analysis: AI-assisted analysis via LLM chat interface.
	Output Condensation: Condense *Volatility3* output to fit into the context window of an LLM.
	Data Cleaning: Remove empty entries, redundant fields, and error messages.
	Environment Setup: Possibility to upload memory images.
Non-Functional Requirements	Portability: Single-directory install (no need for an external database).
	Reproducibility: All intermediate artifacts must persist.
	Extensibility: Allow users to select different LLMs.
	UX Considerations: Hide AI features until basic analysis is completed.

Table 4.1: *Functional and Non-Functional Requirements*

4.2 Architecture Overview

We built MemoryInvestigator based on the specified goal and requirements, using a layered architecture [42]. Table 4.2 summarizes the responsibilities and key modules. We extended the typical layers (user interface front-end, business logic back-end, data access layer, database) to our specific needs. In the following sections, the software is described based on the architecture overview provided in Table 4.2.

Layer	Responsibility	Modules
User Interface Front-End	Streamlit UI, navigation, user feedback	`main.py`, `pages/*`
Orchestration	Workspace preparation, activate temporary folder, *Volatility3* download and installation, global file upload handler	`utils/folder_setup.py`, `utils/volatility_downloader.py`, `utils/file_handler.py`
Forensic Engine	Batch execution of *Volatility3* plug-ins	`utils/volatility_analysis.py`
Data Structuring Tree-of-Table	Build Tree-of-Table JSON	`utils/tree_builder.py`
Data Structuring RAG	Build different RAGs	`utils/build_rag_from_books.py`, `utils/build_rag_from_books_and_volatility3_data.py`, `utils/get_malpedia_references.py`
AI Services	LLM Integration: prompt templates, data selection and chunking, model switch, result merging	`utils/chat_handler.py`, `utils/select_tree.py`, `utils/initialize_rag_chat.py`, `utils/gemini_thinking.py`, `utils/json_divider.py`
Persistence	Numbered working folders (01_memory ...)	Auto-created at start-up with: `utils/folder_setup.py`
Database	Internal Streamlit Database	

Table 4.2: *Architecture Layers of MemoryInvestigator*

4.3 Technology Choices

The prototype is implemented entirely in Python 3.11, motivated by different factors. Python 3 aligns natively with the *Volatility3* framework, which itself is written in Python and officially supported on all major desktop operating systems, meaning that no foreign function wrappers or IPC layers are required for memory forensics. The most time-consuming stages of the workflow are the *Volatility3* plug-ins and the LLM inference, both of which are external to MemoryInvestigator; therefore, the modest runtime overhead of Python is not a bottleneck. To shorten development time, Python 3 offers a rich framework ecosystem (e.g. Flask, Pandas, PyVis, Streamlit) and compatible AI libraries (e.g. LangChain, Tensorflow).

For the user interface, we selected Streamlit. In a small test project, we compared Streamlit with Flask and Dash, finding that it consistently required the least code and delivered native widgets (file uploader, data grid) out-of-the-box. The framework is expressly recognized for rapid prototyping [88]. For our different business layers, we used different third-party dependencies. Among others, we used LangChain for RAG and vector store embedding, streamlit-agraph for graphical process representation, and jq for JSON querying and transformation. A detailed list is available in Appendix A.1.

Due to compatibility problems, *Volatility3* Version 2.8 is downloaded and installed on the local machine and not implemented as a dependency. The auto-downloaded Volatility binaries are signed releases from the official GitHub repository, and the tool runs strictly on local hardware. Therefore, no memory images leave the analyst's machine unless the user explicitly prompts to an LLM.

At start-up, the application creates a self-contained workspace in Windows in O: or in Linux in the /tmp/MemoryInvestigator folder. Because every intermediate artifact (Volatility JSON, Trees-of-Tables, RAG indices) is written to numbered sub-folders, experiments are fully reproducible and auditable even when the Streamlit session is closed. The following folder structure is automatically created, either under O: (Windows) or /tmp/MemoryInvestigator (Linux):

```
00_tools              # Volatility3 and dependencies
01_memory             # Uploaded memory dumps
02_volatility_output  # Volatility3 output files
03_trees              # Process trees
04_data_extraction    # Extracted artifacts
05_standard_rag       # Standard RAG (PDFs, Malpedia)
06_experimental_rag   # Exp. RAG (Volatility3 Output, PDFs)
07_help               # Documentation
```

Listing 4.1: *Self-Contained Workspace*

We published MemoryInvestigator on GitHub (`https://github.com/jan-hendrik-lang/MemoryInvestigator`) and the project structure is shown in Appendix A.2. The setup process is kept simple and presented in detail in Appendix A.3. Put simply, the GitHub repository must be cloned, the required dependencies installed, and the app started. In a Linux environment (Appendix A.4), additional steps must been taken.

4.4 User Interface Frontend

The MemoryInvestigator front-end is implemented as a Streamlit application. While main.py handles app setup and dynamic navigation, the visual components (i.e., user interfaces for displaying and interacting with data) are implemented in the files under pages/. Every user interface page has its own Python file in pages/. Navigation items are injected at runtime by importing the subordinate scripts from pages/, meaning that no build step is required when a page is added or removed. Streamlit's reactive execution model refreshes only those widgets whose underlying variables have changed, keeping the code base concise while still allowing interactivity on commodity hardware. The MemoryInvestigator user interface is documented in Appendix B. The pages in Table 4.3 are available in MemoryInvestigator.

Page	Description
Data Input	Handles file upload, and starts the forensic engine.
Display Data as Table	Presents Volatility output in searchable table.
Display Data as Graph	Process tree also rendered interactively as a graph.
Extract Data	Extract specific files from the Memory Dump
Analysis with Tree-of-Table	AI Integration to chat with a LLM. The user can choose between different LLMs, or specify the forensic data that will be sent to the LLM.
Analysis with Standard RAG	AI Integration to chat with an LLM with an additional RAG built out of PDFs or Malpedia content. Here, the user can choose between different LLMs, embeddings or specify the forensic data that will be sent to the LLM.
Analysis with Experimental Forensic RAG	AI Integration to chat with a LLM, but now the RAG is built out of the forensic artifacts from *Volatility3* and PDFs or Malpedia content. Again, the user can choose between different LLMs, embeddings or specify the forensic data that will be sent to the LLM.
Help	Embeds Markdown documentation.

Table 4.3: *Overview of the User Interface Frontend Pages*

The interface deliberately reveals functionality in the order that an analyst would need it. After start-up, the navigation bar only shows the data input and help page (Figure B.1. On the data input page, the analyst can enter information about the case and upload a memory dump, which can also be manually transferred to the folder '01_memory' to save upload time. Once a memory dump is present in the folder, the analyst is able to click 'Analyze Data and Build a Basic Tree', which starts that the forensic analysis and builds the basic data structure, mainly explained in Section 4.6.

If the mentioned process is finished, the user gains access to every page of Table 4.3 with the opportunity to analyze the memory dump in further depth. All fields and labels are in plain language to ensure easy understanding. All API keys are entered on the specific page and stored only in memory, whereby the values are cleared automatically when the session terminates.

With the 'Renew Environment' button, all files and data of the session ar deleted and a new analysis can be stated. The 'Delete Session' button also deletes all files and additionally closes the application.

In sum, the user interface layer translates complex forensic workflows into an incremental, guided experience that minimizes cognitive load while still exposing expert-level controls when necessary.

4.5 Orchestration

The orchestration layer glues the user interface to the forensic tooling by preparing an isolated workspace, installing third-party dependencies (e.g. *Volatility3*) on first launch, and providing a file upload handler.

At start-up, the main.py calls the function create_drive() from utils/folder_setup.py. The helper mounts a virtual drive O: on Windows or a temporary folder under /tmp/MemoryInvestigator on Linux, then creates seven numbered sub-directories, listed in Listing 4.1 and an empty .json settings. The same module offers remove_drive() to tear everything down when the analyst resets the environment.

Immediately afterwards, main.py invokes utils/volatility_downloader.py. The script fetches the signed *Volatility3* release, extracts it into 00_tools, purges any obsolete YARA package that might clash with the new binaries, and installs Volatility's Python dependencies from the bundled requirements.txt. All steps run silently.

When the analyst drops a memory dump onto the web page, handle_memory_upload() in utils/file_handler.py validates the extension (.vmem, .raw, .mem), saves the file to 01_memory, and returns the canonical path for downstream processing.

The build_pages() function within main.py inspects the workspace: if neither a basic tree-of-table nor a `windows.pslist.json` file exists, only the *environment* and *help* pages are registered with Streamlit. Once a first analysis run finishes, every page is visible to the user. Two sidebar buttons allow the user to 'Renew Environment,' which recreates the virtual drive and re-installs Volatility, or 'Delete Session,' which calls remove_drive() and then terminates the Streamlit process.

4.6 Forensic Engine

We automated the execution of multiple *Volatility3* modules and structured the output to be processed efficiently to facilitate analysis with the help of an LLM within the forensic workflow.

Therefore, we selected a comprehensive set of Volatility3 modules to capture key forensic artifacts that are relevant to memory investigations: *windows.cmdline*, *windows.psscan*,

windows.netscan, *windows.netstat*, *windows.dlllist*, *windows.getsids*, *windows.hollowprocesses*, *windows.ldrmodules*, *windows.malfind*, *windows.processghosting*, *windows.svcdiff*, and *windows.suspicious_threads*. Appendix C.1 lists the extracted fields. We selected these modules not only for their forensic relevance but also because other modules often generate excessive output that would hinder efficient automation.

Providing large-scale tabular data to LLMs is a challenge. LLMs were mainly designed for prose, whereby even providing the *windows.pslist* output exceeds typical context windows. Therefore, the extracted data must be structured and cleaned. During the investigation, multiple methods for integrating *Volatility3* output with LLM processing were explored:

1. Creation of an SQL database: Storing *Volatility3* output in an SQL database and allowing the LLM to query it. This proved error-prone and inefficient due to unpredictable query results and database complexity.
2. Direct use of *Volatility3* by LLM: Allowing LLM to execute Volatility3 commands dynamically. This approach was found to be unstable and difficult to control, leading to inconsistent results.
3. RAG based on *Volatility3* output: RAG sends only a small subset of the data to the LLM. It struggled to provide comprehensive solutions that required insight from multiple forensic artifacts.
4. Tree-of-tables method (implemented): This approach was chosen because it provides a balance between flexibility and efficiency. This algorithm organizes structured data into a hierarchical tree-of-table format, optimizing it for AI analysis and improved interpretability. This approach aligns with recent findings by Zhang et al. [106], who emphasized the importance of structured input representation when applying LLMs to tabular data interpretation.

4.7 Data Structuring: Tree-of-Table Algorithm

Ji et al. [46] introduced the tree-of-table algorithm, which employs a hierarchical tree structure. The algorithm follows multiple steps and is implemented in the following manner:

1. Table Condensation and Decomposition: First, the large table must be divided into task-relevant parts to focus on pertinent information. For this purpose, we describe relevant fields (Appendix C.1) in the output produced from *Volatility3*.
2. Table-Tree Construction: Ji et al. employed a "breadth-first" approach, breaking down the issue into multiple general, independent yet interconnected subprocesses and subsequently refining each subprocess into more detailed solutions through iteration. In our approach, this step is easier because every process has a unique ID (PID) by default and can be connected via the parent process ID (PPID). Therefore, even each *Volatility3* module output has a containing PID and can therefore be added to this particular process. The possibility remains that a process could be

exited but, e.g. the network traffic remains without a PID in the memory dump. If something like this happens, these artifacts will be added to the end of the tree.

3. Table-Tree Execution: Finally, Ji et al. utilize the table-tree execution as a "depth-first" search technique to methodically explore and calculate intermediate results within the tree framework, improving reasoning efficiency by handling and retaining subtrees separately. If the tree-of-table is too large for the context window, it can be divided into multiple chunks manually by the user. In our software, the user is also able to build specific trees and therefore provide the LLM with increasingly detailed information in a continuous process. To conduct this study in detail, we only used the data and modules presented.

The pseudocode for our approach is available in Appendix C.2 and a graphical representation of the algorithm is available in Appendix C.3. An example output is available in Appendix D. Broken down to the bare minimum, for our approach this means:

1. Parse JSON output from *Volatility3* modules.
2. Construct a parent-child relationship for processes using PID and PPID values.
3. Attach additional metadata (e.g., network activity, loaded DLLs, suspicious threads) to each process node.
4. Store the structured output in a format optimized for LLM parsing.

To minimize noise in the LLM input, the extracted *Volatility3* data was pre-processed with some regex commands shown in Listing 4.2. Empty entries will be removed, e.g. redundant fields such as empty children nodes. Moreover, error messages such as "Required memory at 0x20 is not valid (process exited?)" or "Required memory at 0x9ABCD123 is inaccessible (swapped)" were deleted.

```
# Read the Tree-of-Table file and pass it to the variable chunk
chunk = file.read()

# Replace empty children with a space
chunk = re.sub(r'"children": \[\]', ' ', chunk)

# Remove error message
chunk = re.sub(r'Required memory at 0x[0-9a-fA-F]+ is not valid \(process exited\?\)', '', chunk)

# Remove error message
chunk = re.sub(r'Required memory at 0x[0-9a-fA-F]+ is inaccessible \(swapped\)', '', chunk)

# Replace all whitespace (newlines, tabs, spaces) with a single space
cleaned_chunk = re.sub(r'\s+', ' ', chunk).strip()
```

Listing 4.2: *Regex Commands for Data Cleaning*

4.8 Data Structuring: Retrieval-Augmented Generation

RAG improves LLM by giving it a knowledge base outside of the trained data, thus enhancing the capabilities of an LLM. The prototype offers two self-contained pipelines, namely standard RAG and experimental forensic RAG, both of which are implemented with LangChain and a disk-backed Chroma vector store so that every embedding persists in the numbered workspace and can be re-used across sessions.

With the standard RAG, the user can give the LLM a factual grounding and domain-specific knowledge. All PDF files that the analyst uploads or places in the (05_standard_rag) folder and—optionally—external threat-intel web pages retrieved from Malpedia ingest from build_standard_rag(). If the analyst activates the use of Malpedia by clicking the checkbox and supplies a malware family name (e.g. win.emotet), the get_references_from_malpedia() helper contacts the Malpedia API and extracts the reference URLs. The helper first sanitizes each HTML page with BeautifulSoup, strips cookie banners and other boilerplate, detects language via langid (documents not in English or German are skipped), and attaches metadata such as the title, author and publication date. Every document, PDF or cleaned web page is then split into 1,000 token chunks with a 200-token overlap, embedded by Google Gemini or OpenAI (user choice), and stored in a local Chroma collection. The function returns a LangChain retriever object, which downstream prompts can query with semantic similarity search. Beside the system message, the prompt also contains the user message with the tree-of-table data and data from the RAG. This functionality keeps the knowledge base current without fine-tuning an LLM.

The experimental forensic RAG extends the corpus with *Volatility3* JSON output from 02_volatility_output. PDFs and JSON artifacts are embedded together in a Chroma store, which allows the LLM to answer questions that require both narrative context from manuals or reports and raw artifacts from the live dump, such as 'Does the suspicious thread align with known Emotet injection patterns?' Therefore, the tree-of-table data is not included in the prompt because it is sent to the LLM with the RAG function.

Due to the operational flow, both builders first check whether a Chroma store already exists, and if so, it is re-opened in read-only mode, saving the analyst from re-embedding unchanged material. The resulting retriever is passed to the AI services layer, which injects it into the selected LLM and merges the top-k passages into the prompt. Because vector stores reside in the workspace hierarchy, every retrieval decision is auditable and can be replayed on a different machine simply by copying the numbered folders.

In sum, the RAG layer bridges unstructured reference material, structured *Volatility3* output, and the tokenbudget limits of modern LLMs. After a few tests, it was clear that RAG does not extend the use of MemoryInvestigator. Therefore, the evaluation in Section 6 is based on the tree-of-table algorithm without RAG.

4.9 Integration with Large Language Model Processing

MemoryInvestigator treats every LLM as a plugable back-end. A central dispatcher in utils/chat_handler.py receives three inputs—(a) the system message with a selected slice

of structured data, (b) the user message, and (c) an optional RAG retriever—and emits a single streaming response to the UI. The dispatcher instantiates either an OpenAI or GoogleGenerativeAI chat object according to the analyst's choice. The messages are:

1. System Message: You are a forensic RAM analyst assistant specializing in Windows memory analysis. Analyze the JSON tree of Windows memory artifacts to detect intrusions or malicious activities. Cross-check your findings with known threats and provide clear, specific reasons for flagging any anomalies (e.g., unusual parent-child relationships, code injection, execution from nonstandard locations). If you are unsure, ask clarifying questions, and if you don't know, say so. Generate a structured forensic report highlighting confirmed threats while minimizing noise. Data: "Tree-of-Table Data Placeholder"
2. User Message: Analyze the data and determine whether there is an anomaly.

4.10 Large Language Models Used

For this research, we selected the most prominent LLMs currently available from major AI providers. They represent the most advanced AI systems from the largest AI companies, ensuring that our results reflect the current state of the art in combining LLMs with forensics. We selected OpenAI GPT4o, OpenAI o1, Gemini 2.0 Flash, Gemini 2.0 Flash-Thinking, Grok 3, and Grok 3 with thinking mode enabled.

All models except Grok 3 and Grok 3 ('thinking mode') were tested using their API as automation was a key factor in evaluating forensic workflows. Since Grok 3 did not provide API access at the time of this research, its evaluation was limited to manual testing in the web-based window. Therefore, Grok 3 cannot be selected in MemoryInvestigator.

4.11 System Overview

Figure 4.1 presents a container-level architectural overview of the MemoryInvestigator prototype, following the principles of the C4 model. The system is centered around a modular pipeline that supports memory dump upload, automated forensic extraction using *Volatility3*, structured artifact organization, and AI-assisted analysis via RAG and LLMs. The user interacts with the platform through a Streamlit-based web UI, which communicates with a Python-based orchestration layer that manages tool setup and workflow coordination. Back-end services handle forensic processing, tree-based data structuring, RAG embedding, and LLM querying. Intermediate artifacts and embeddings are persisted in a self-contained workspace, ensuring reproducibility. External APIs such as LLMs (both standard and reasoning-based models) and Malpedia are seamlessly integrated.

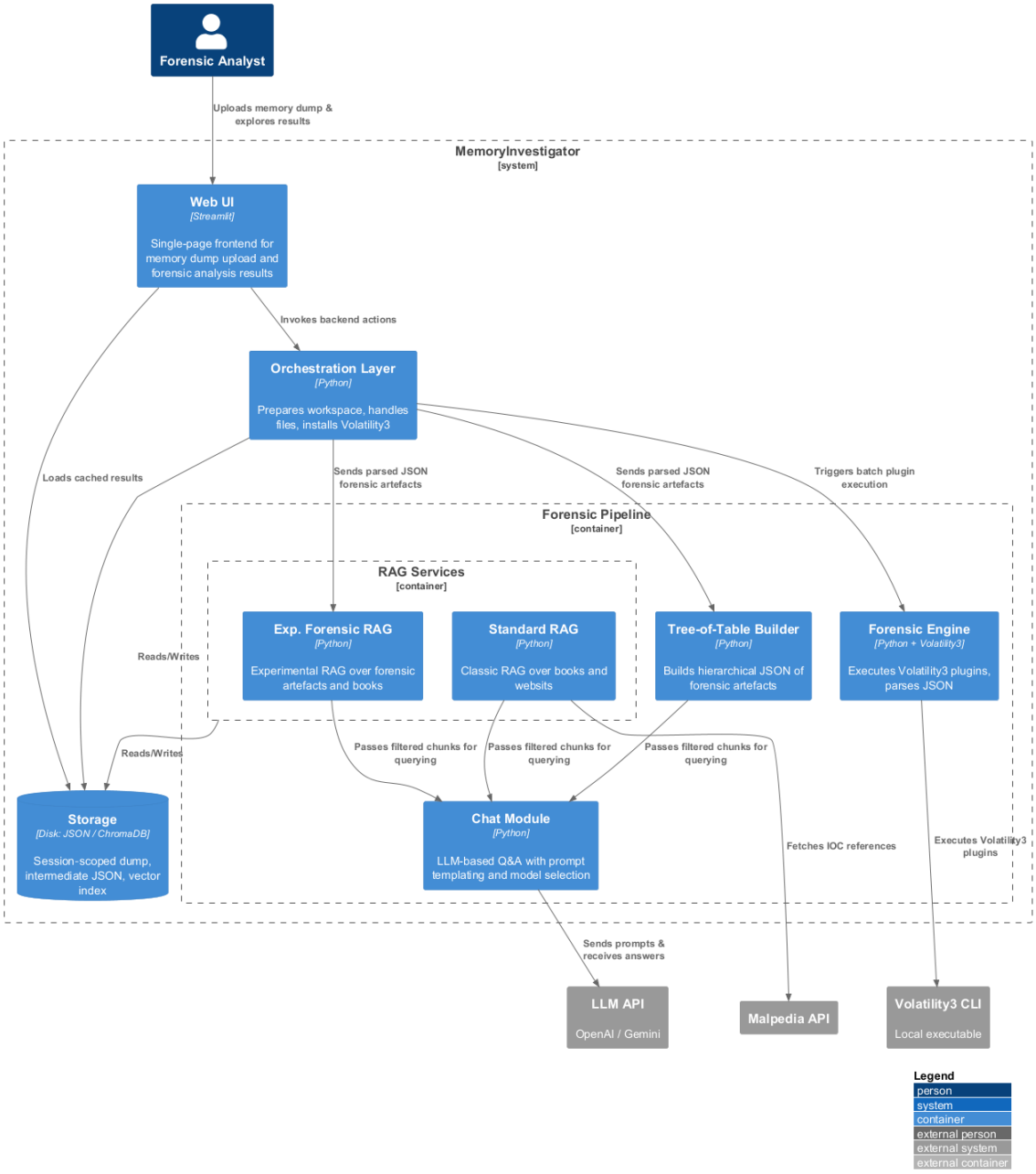


Figure 4.1: *MemoryInvestigator C4 Diagram*

5 Experimental Setup

We built an experimental pipeline to explore how LLMs can assist forensic analysis. Using Volatility3, we automatically harvested forensic artifacts from RAM images then cleaned and normalized the plugin output into structured prompts tailored for LLMs. These prompts were sent to multiple standard and reasoning-enabled LLM models to measure each model with standard classification metrics.

5.1 Environment Configuration and Tools

A dedicated test environment was set up to perform a controlled analysis of malware detection using LLMs and *Volatility3*. The configuration ensured reproducibility and provided a stable framework for forensic evaluations. This also ensured that legitimate processes were present alongside potential malware samples to evaluate FP rates effectively.

1. Virtualization Setup: VMware Workstation 17, Windows 10 Version 22H2 (4GB RAM, 1 vCPUs)
2. Network Settings: Windows 10 VM was connected to a Kali Linux VM via VMware Workstation 17's internal (host-only) network to conduct the self-produced attacks.
3. Simulated User Activity: To replicate a real-world environment and create background noise in memory, Firefox (web browsing), Foxit PDF Reader (document handling), LibreOffice (word processing), TeamViewer (remote access simulation), WinRAR (file archiving), and ProcMon (process monitoring) were run concurrently.
4. Memory Capture Tool: BelkaSoft's RamCapture64.exe

5.2 Malware Sample Selection

We selected a range of self-produced and real-world malware samples. The self-produced malware samples were generated using msfvenom and PowerShell Empire, while the real-world malware samples were downloaded from MalwareBazaar. MalwareBazaar [6] is a community-driven free malware repository maintained by Abuse.ch that allows security researchers to share malware samples. In addition to MalwareBazaar, we used Malpedia [69] and joesandbox [53] to gather detailed information on the specific malware. Therefore, we established the following primary criteria for selecting malware samples:

1. Diversity of Attack Techniques: Each malware employs different evasion and persistence mechanisms, including process injection, registry modifications, and event log manipulation.
2. Real-World Threat Landscape: These malware samples are actively used in cybercrime, making their detection valuable for practical forensic investigations.
3. Challenges for LLM-Based Analysis: Some malware families store IoCs outside traditional *Volatility3* plugins, posing challenges for LLM-based detection.

5.3 Test Cases Overview

Overall, we created eight distinct scenarios. Table 5.1 provides a detailed overview of the malware used. There was also a clean image (not listed in the table), taken from a Windows 10 machine without malware. This should identify potential FP behavior on clean and malware-free images. In addition, the image can be used to compare with infected memory images to identify anomalous behaviors.

Malware Name	Type	Volatility3 Modules	LLM Detection	Detection Issues	SHA256 Hash
TCP Reverse Shell and Process Injection build with msfvenom	Process Injection	malfind	Detected	None	without
PowerShell Empire	PowerShell Script	cmdline	Detected	None	without
Quasar RAT	Remote Access Trojan	.netscan	Detected	None	be0d150d8ba2b3d607c23fac6aff6caf97525565f392e9daf3dd1baaabfcf447
MassLogger	Keylogger	psscan, netscan	Detected	None	fbaec035008b4d3722c9b832c534d85660e7c80027a29d1d8310b77b2ad54fc7
DarkCloud	Banking Trojan	psscan, malfind	Detected	None	79f7e76ba3a8780a2dd46d78aac07f2235f457ee2771ef4556b85c12f2be0061
LockiBot	Credential Stealer	psscan, malfind	Detected	None	1eb1ecc1ff01e9f4934a79daa57db69c96f975b90d649429417c560c1d36bab1
LockBit (aka Worm.m0yv)	Ransomware	registry.hivelist, registry.prinkey	Not Detected	IoCs in strings, scheduled tasks, and registry. IoCs not found with the selected Volatility modules.	9733092223c428fc0e44a90b01c7f77a97bb1205def8be1224ac68969182638e

Table 5.1: *Overview of Malware Samples Used*

5.4 Test Case: Process Injection With msfvenom

Metasploit's Framework msfvenom utility—which unifies the earlier msfpayload and msfencode programs—has become a cornerstone in contemporary offensive security experimentation and pedagogy. By enabling researchers to generate architecture-specific binaries or raw shellcode and chain these payloads through configurable encoders, msfvenom offers a reproducible method for examining both exploit reliability and the efficacy of modern intrusion detection countermeasures. The interface facilitates large-scale, automated studies of antivirus evasion and intrusion-set behavior, while its tight integration with Metasploit's handler modules allows repeatable end-to-end attack simulations in controlled laboratory environments. [63] First, we used msfvenom to generate a reverse TCP shell payload (listing 5.1).

```
1 sudo msfvenom --payload windows/meterpreter/reverse_tcp
    LHOST=192.168.241.138 LPORT=8080 --format exe --out
    putty
2 sudo msfdb run
3 use exploit/multi/handler
4 set PayLOAD windows/meterpreter/reverse_tcp
5 set LHOST 192.168.241.138
6 set LPORT 8080
```

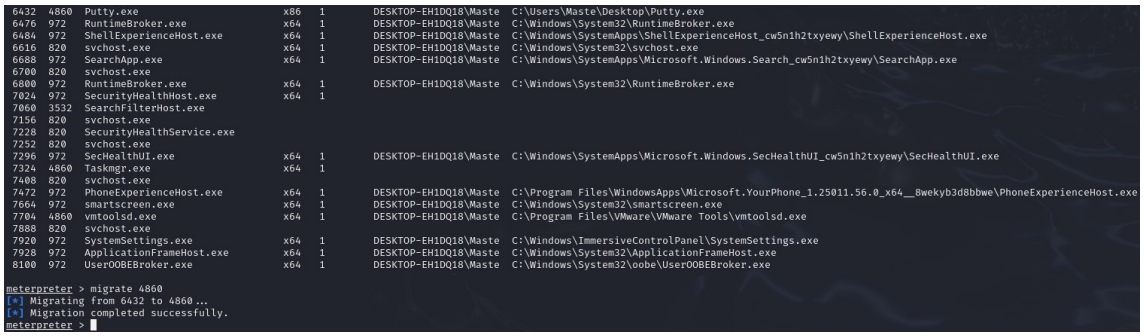
Listing 5.1: *Building Reverse Shell with msfvenom*

After executing the putty.exe file on the victim computer, the payload was injected into a remote process to simulate an adversary executing code on a compromised system. For this purpose, we used the migration operation shown in Listing 5.2.

```
ps
migrate <PID>
```

Listing 5.2: *Injection in a Process*

Meterpreter's ps command revealed a range of resident processes that were suitable for masquerade, whereby we selected one that was already trusted by the operating system. With migrate <PID>, we invoked the classic *VirtualAllocEx -> WriteProcessMemory -> CreateRemoteThread* chain, relocating the Meterpreter thread from putty.exe into the long-lived target (in our lab run, Taskmgr.exe with PID 4860) so that the original dropper could terminate without closing the session. The migration is displayed in Figure 5.1. The privileges and network profile of the adopted process are inherited while simultaneously blending into normal system activity. This single step has three practical consequences: (a) it stabilizes the shell because the new host process is unlikely to be closed by the user, (b) it can implicitly elevate privileges if the operator chooses a service or system process that already holds a higher-integrity token, and (c) it confounds endpoint controls that focus on unsigned binaries or parent–child launch chains rather that memory behavior. From a forensic perspective, the move leaves several tell-tale residues: (a) an orphaned executable region in memory that begins with the MZ header of the reflective DLL but is not backed by any file on disk, (b) a parent-process anomaly where the adopted process appears to descend from a transient and now-exited putty.exe instance, (c) a fresh thread whose start address sits inside read–write–execute memory rather than within a loaded image section, and (d) an open TCP socket that remains tied to the new PID while terminating in the analyst's listener. All of these artifacts could be exposed by *Volatility3* malfind, pslist, ldrmodules and netscan. This lab thus replicates a realistic attacker workflow in which a benign-looking utility becomes both the delivery vehicle for a code injection.



```
 6432  4860  Putty.exe                    x86   1        DESKTOP-EH1DQ18\Maste  C:\Users\Maste\Desktop\Putty.exe
 6476  972   RuntimeBroker.exe            x64   1        DESKTOP-EH1DQ18\Maste  C:\Windows\System32\RuntimeBroker.exe
 6484  972   ShellExperienceHost.exe      x64   1        DESKTOP-EH1DQ18\Maste  C:\Windows\SystemApps\ShellExperienceHost_cw5n1h2txyewy\ShellExperienceHost.exe
 6616  820   svchost.exe                  x64   1        DESKTOP-EH1DQ18\Maste  C:\Windows\System32\svchost.exe
 6688  972   SearchApp.exe                x64   1        DESKTOP-EH1DQ18\Maste  C:\Windows\SystemApps\Microsoft.Windows.Search_cw5n1h2txyewy\SearchApp.exe
 6700  820   svchost.exe
 6800  972   RuntimeBroker.exe            x64   1        DESKTOP-EH1DQ18\Maste  C:\Windows\System32\RuntimeBroker.exe
 7024  972   SecurityHealthHost.exe       x64   1
 7060  3532  SearchFilterHost.exe
 7156  820   svchost.exe
 7228  820   SecurityHealthService.exe
 7252  820   svchost.exe
 7296  972   SecHealthUI.exe              x64   1        DESKTOP-EH1DQ18\Maste  C:\Windows\SystemApps\Microsoft.Windows.SecHealthUI_cw5n1h2txyewy\SecHealthUI.exe
 7324  4860  Taskmgr.exe                  x64   1
 7408  820   svchost.exe
 7472  972   PhoneExperienceHost.exe      x64   1        DESKTOP-EH1DQ18\Maste  C:\Program Files\WindowsApps\Microsoft.YourPhone_1.25011.56.0_x64__8wekyb3d8bbwe\PhoneExperienceHost.exe
 7664  972   smartscreen.exe              x64   1        DESKTOP-EH1DQ18\Maste  C:\Windows\System32\smartscreen.exe
 7704  4860  vmtoolsd.exe                 x64   1        DESKTOP-EH1DQ18\Maste  C:\Program Files\VMware\VMware Tools\vmtoolsd.exe
 7888  820   svchost.exe
 7920  972   SystemSettings.exe           x64   1        DESKTOP-EH1DQ18\Maste  C:\Windows\ImmersiveControlPanel\SystemSettings.exe
 7928  972   ApplicationFrameHost.exe     x64   1        DESKTOP-EH1DQ18\Maste  C:\Windows\System32\ApplicationFrameHost.exe
 8100  972   UserOOBEBroker.exe           x64   1        DESKTOP-EH1DQ18\Maste  C:\Windows\System32\oobe\UserOOBEBroker.exe

meterpreter > migrate 4860
[*] Migrating from 6432 to 4860...
[*] Migration completed successfully.
meterpreter >
```

Figure 5.1: *Migration to Another Process*

5.5 Test Case: PowerShell Empire Attack

PowerShell Empire excels at camouflaging itself among routine Windows activity. A quick survey of running processes or network connections on a compromised host usually reveals nothing more suspicious than an ordinary powershell.exe, leaving defenders with few obvious cues to separate malicious behavior from day-to-day operations. However, the framework is not completely invisible given that durable artifacts linger in certain forensic hotspots, most notably Windows event logs and volatile memory of the system, which can provide conclusive proof of compromise even when an adversary rotates or obfuscates their C2 profile [55]. The procedure began with the initialization of the PowerShell Empire server, shown in Listing 5.3.

```
./ps-empire server
```

Listing 5.3: *Start the C2 Empire Server*

Displaying a running server in Listing 5.4, the PowerShell Empire Client is used to establish a communication channel for post-exploitation activities. An HTTP-based listener was selected—a stager configured for Windows PowerShell—linked to the specified HTTP-based listener to set up a remote control.

```
./ps-empire client
uselistener http
set Name httpListener
usestager windows_powershell
set Listener httpListener
execute
```

Listing 5.4: *Run the PowerShell Empire Client*

To establish a connection to the command and control (C2) server, the script in Listing 5.5 was executed on the victim's client. The script is de-obfuscated in Listing 5.6.

```
@echo off
start /B powershell.exe -nop -ep bypass -w 1 -enc
    KABOAGUAdwAtAE8AYgBqAGUAYwBOACAATgBlAHQALgBXAGUAYg
    BDAGwAaQBlAG4AdAApAC4AUAByAG8AeAB5AC4AQwByAGUAZABl
    AG4AdABpAGEAbABzAD0AWwBOAGUAdAAuAEMAcgBlAGQAZQBuAH
    QAaQBhAGwAQwBhAGMAaABlAF0AOgA6AEQAZQBmAGEAdQBsAHQA
    TgBlAHQAdwBvAHIAawBDAHIAZQBkAGUAbgBOAGkAYQBsAHMAOw
    BpAHcAcgAoACcAaAB0AHQAcAA6AC8ALwAxADkAMgAuADEANgA4
    AC4AMgA0ADEALgAxADMAOAA6ADgAMAAvAGQAbwB3AG4AbABvAG
    EAZAAvAHAAbwB3AGUAcgBzAGgAZQBsAGwALwAnACkALQBVAHMA
    ZQBCAGEAcwBpAGMAUABhAHIAcwBpAG4AZwB8AGkAZQB4AA==
```

Listing 5.5: *Base64 obfuscated Script execution on the victims client*

```
(New-Object Net.WebClient).Proxy.Credentials=[Net.
    CredentialCache]::DefaultNetworkCredentials;iwr('http
    ://192.168.241.138:80/download/powershell/')-
    UseBasicParsing|iex
```

Listing 5.6: *De-Obfuscated Base64 Script*

Finally, the victim's client was selected to interact with it in Listing 5.7.

```
agents
interact LXD9H2S8
```

Listing 5.7: *Interaction with the Victims Client*

Event logs are invaluable but they can be purged, lost, or simply never collected. When this happens, memory forensics becomes the most reliable fallback. Analyzing a RAM image exposes the same Empire stager that is visible in the logs, complete with the characteristic launcher string *powershell.exe -noP -sta -w 1 -enc*. This flag combination ships with Empire's default HTTP listener and it often remains unchanged despite the fact that attackers can customize it, making it a dependable detection signature. The accompanying base64-obfuscated payload 5.5—preserved in memory—supplies a second, equally distinctive indicator, particularly in environments where administrators rarely encode their own scripts. [55]

After the established interaction with the client, Starkiller was started for a easier interaction. Starkiller is the graphical front-end for PowerShell Empire, displayed in Figure 5.2. It serves over PowerShell Empire REST API. The web-based interface supports multiple operators, allowing them to spin up listeners, generate stagers, task agents, and review results through an intuitive dashboard. Starkiller carry out the same action as CLI, only faster and with richer context. It can generate PowerShell, Python, DLL, EXE and macro stagers in one click but also use post-exploitation modules and—among other things—share harvested credentials, and manage exfiltrated files. [19]

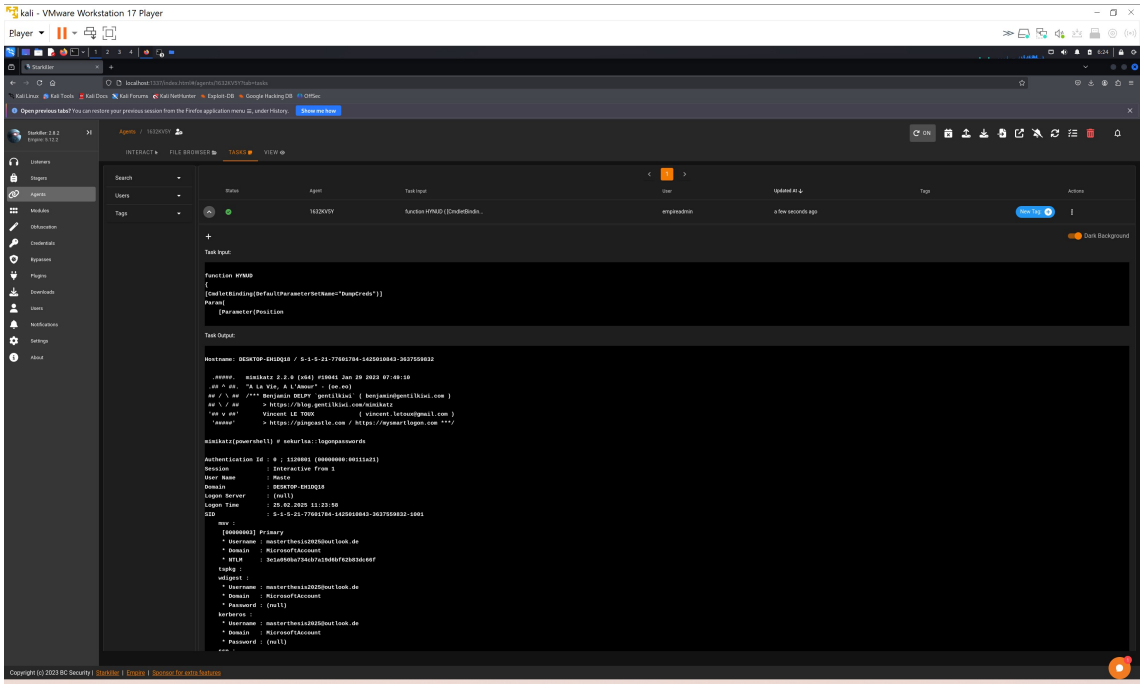


Figure 5.2: *PowerShell Empire Graphical Front-End: Starkiller*

5.6 Test Cases: Real-World Malware Samples

The real-world malware samples used in our analysis included a selection of known threats chosen to represent a variety of attack techniques and persistence strategies.

This variety of malware enabled testing the LLM detection using different techniques, evaluating the limitations of the standard Volatility3 modules, and analyzing the effectiveness of LLMs, ultimately determining whether they can improve the forensic analysis. First, in Table 5.2 we show a memory-forensics–oriented comparison of the real-world malware samples extracted from different CTI artifacts. The CTI artifacts were gathered from publicly available "gray literature" sources—security vendor blogs, incident response reports, and threat hunt reports—rather than peer-reviewed publications. Such sources are standard in operational CTI because they offer more time-efficient and detailed indicators than the formal academic literature. Second, we used joesandbox for behavior analysis to gain a deeper understanding of the specific malware samples. Joesandbox [53] is a automated online malware analysis tool. In addition to online sandbox environments, tools such as FLARE-VM from Mandiant are common in malware analysis workflows. [34]

Due to time constraints and the broad scope of this thesis, only one real-world malware sample—*Quasar RAT*—is explained in detail in the following section. *Quasar RAT* was selected as a representative case study due to its forensic relevance, network activity, and frequent appearance in real-world attacks. The remaining malware samples are explained in a more concise manner, enabling a focused yet illustrative examination of the capabilities and limitations of MemoryInvestigator and the integrated LLMs, without compromising the overall breadth of the evaluation.

Across all five families—Quasar RAT, MassLogger, DarkCloud, LokiBot, and LockBit—a consistent trend towards stealthy operation, multi-layered persistence, and runtime obfuscation is observed. These traits are precisely what challenge traditional memory analysis techniques.

The Quasar RAT sample was analyzed by Joe Sandbox [78] and is shown in Figure 5.3 as a behavior graph. The behavior graph shows its multi-stage infection process and remote access functionalities on a Windows 10 system. The sample was identified through machine learning and multiple antivirus engines. Upon execution, the malware decrypts embedded strings to conceal its true behavior and drops additional malicious files. The malware hides the origin of the file by removing the zone.identifier stream and thus masking that it was downloaded from the internet. The malware also modifies Windows Defender settings by adding directory exclusions. This attempts to evade the local protection and bypasses PowerShell execution policies to enable further script-based activity. To access or manipulating encrypted data, the malware also loads the BitLocker PowerShell module. Persistence is gained via scheduled tasks, created using schtasks.exe and multiple processes, including client.exe, which should ensure continuous operation. Identified by Sigma and Suricata rules, some of the processes initiate C2 communication with hardcoded IP addresses and domains. QuasarRAT installs a global keyboard hook to execute its keylogging activity.

MassLogger continues this theme with a heavily obfuscated .NET loader that modifies MSIL code at runtime. It uses advanced anti-analysis features such as PowerShell-based

Defender exclusions and IP geolocation to avoid sandbox execution. It logs keystrokes, captures screenshots, and harvests sensitive browser and email credentials. Its behavior in memory is non-injective yet dynamic,making detection difficult without in-depth inspection and full memory dumps.

DarkCloud—a modular stealer with worm-like propagation—combines injection tactics with extensive registry manipulation. It leverages process memory allocation and WMI access to remain persistent while exfiltrating sensitive data over multiple channels. While not as deeply obfuscated as MassLogger, it requires correlation of memory artifacts with external indicators such as network traffic and registry writes for successful detection.

LokiBot is more traditional in design but still problematic. It avoids injection and instead resides in standalone processes, dynamically resolving APIs and maintaining mutex-based persistence. It steals a wide range of credentials and uses misleading filenames and encrypted strings to resist static analysis. Interestingly, it showed behavior such as Defender exclusion, schtasks.exe usage, and loading suspicious PowerShell modules, adding layers to its evasion strategy.

LockBit (worm.m0yv) represents a more aggressive, time-sensitive threat. Designed for rapid encryption and propagation via GPO and PsExec, its memory behavior is marked by high CPU usage, embedded cryptographic material, and transient execution. If not captured mid-execution, forensic traces are minimal. Behavior data confirms attempts to spread over SMB, the use of Tor addresses, and evasion via syscall obfuscation and alternative data streams.

We include the malware behavior analysis of all other malware in the Appendix E, serving as a foundation for understanding the specific behavior.

Malware	Type & Capabilities	Persistence Mechanisms	Memory Behavior	Detection Challenges	Source
Quasar RAT	C#-based Remote access trojan with full control (keylogging, file ops, shell); often delivered via in-memory loaders such as SharpLoader.	Copies to `%APPDATA%`, sets `Run` key (e.g., "Quasar Client Startup"); may install as a service.	Injects .NET payload via process hollowing into trusted hosts (e.g., `RegAsm.exe`); strings decrypted only at runtime.	Payload often never written to disk; masquerades as benign process with RWX regions. strong obfuscation.	[99] [4] [79] [58] [44] [21] [45] [25] [91] [75] [59] [8] [78]
MassLogger	.NET infostealer & keylogger; multistage, runtime MSIL rewriting, heavy obfuscation.	Copies to `%APPDATA%`; establishes Run key; optional "melt"; can add Defender exclusions; batch/VBS relaunch.	Single .NET process whose IL is modified on the fly; decrypted config and logs live in RAM; no external injection.	Obfuscated IL fools static plugins; AES-encrypted strings; anti-VM may skip execution. Config extraction requires full dump and decryption.	[35] [31] [15] [16] [28] [11] [41] [77] [5]
DarkCloud	Modular stealer with worm-like propagation (SMTP, Telegram, FTP); broad data theft.	Dropper copies itself, sets Scheduled Task or Startup VBS; deletes originals.	Final VB stealer injected into system process; many network connections; clears logs/shadows.	Injected into legit process; multi-host propagation obscures origin; multiple exfil channels; requires correlating unusual network + injected code.	[1] [76] [18] [40] [24] [7]
LokiBot	Native password stealer & keylogger; exfiltration via HTTP POST.	Creates hidden folder with copy + Run key; GUID-based mutex; maintains `.lck`/`.hdb`/`.kdb`.	Runs as standalone process; dynamically resolves APIs; stolen data and keylog buffers in RAM; plain-HTTP network buffers visible.	Blends with user processes if name looks legit; must spot path/mutex; strings/config encrypted; no injections to flag with `malfind`.	[3] [48] [30] [57] [54] [26] [70] [13]
LockBit (worm.m0yv)	Ransomware; ultra-fast encryption, auto-propagation via GPO/PsExec.	Usually run directly (no long-term persistence); GPO deployment acts as temporary persistence on many hosts.	High-CPU native process; embedded AES/RSA keys in memory; opens many SMB sessions when propagating; deletes logs.	May terminate/self-delete post-encryption; if not dumped in time, only residual artifacts remain; log wiping removes timeline; single-host image may miss wider spread.	[2] [47] [20] [72] [87] [10] [14] [86] [36]

Table 5.2: *Memory-Forensics-Oriented Comparison of the Real-World Malware Samples*

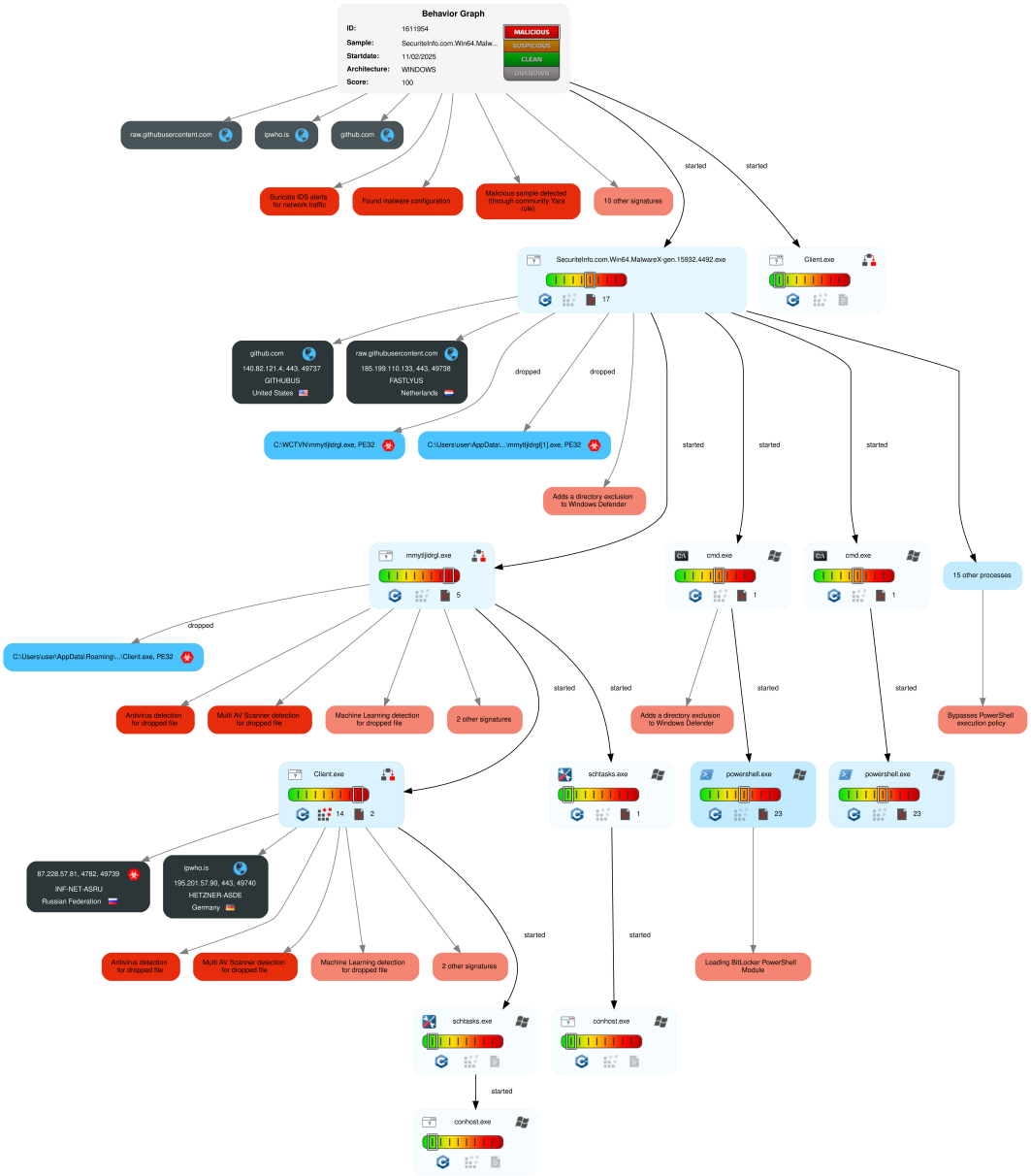


Figure 5.3: *QuasarRAT Behavior Analysis [78]*

6 Evaluation

6.1 Evaluation Metrics: Accuracy, Precision, Recall, and F1 Score

Our evaluation followed a two-step process. First, we extracted all processes from each memory image using the *Volatility3* psscan plugin, which serves as our ground-truth list of running processes. Second, each memory image was analyzed five times by the LLM. For each run, we compared the LLM's classification of processes to the ground truth and recorded the following outcomes for each process. A true positive (TP) occurs when the LLM correctly identifies a malicious process, whereas a true negative (TN) refers to a benign process correctly identified as nonmalicious, which also includes benign processes that are not shown in the LLM's output. A false positive (FP) occurs when a benign process is incorrectly flagged as malicious. Finally, a false negative (FN) describes a malicious process that is not flagged by the LLM. Using these values, we compute the following standard evaluation metrics:

1. Accuracy: Inspired by Oh et al. [64], accuracy is the fraction of processes correctly classified by the model, providing an overall success rate.

$$Accuracy = \frac{(TP + TN)}{(TP + TN + FP + FN)} \tag{6.1}$$

2. Precision: How many of the processes that the model flagged as malicious were truly malicious? Low precision means that the model has many false alarms.

$$\text{Precision} = \begin{cases} \frac{\text{TP}}{\text{TP+FP}}, & \text{if TP} + \text{FP} > 0 \\ \text{NaN}, & \text{otherwise} \end{cases} \tag{6.2}$$

3. Recall (sensitivity): How many of the truly infected processes did the model successfully catch? Low recall means that the model missed some malware cases (FNs).

$$\text{Recall} = \begin{cases} \frac{\text{TP}}{\text{TP+FN}}, & \text{if TP} + \text{FN} > 0 \\ \text{NaN}, & \text{otherwise} \end{cases} \tag{6.3}$$

4. F1 score: The harmonic mean of precision and recall provides a balanced measure that penalizes extreme trade-offs. An F1 of 1.0 reflects a perfect model and a lower score indicates an increasing number of false positives or false negatives.

$$F1 - Score = \begin{cases} \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision+Recall}}, & \text{if Precision, Recall} \neq \text{NaN and Precision} + \text{Recall} > 0 \\ \text{NaN}, & \text{otherwise} \end{cases} \tag{6.4}$$

For classification purposes, each process was considered a separate instance. If the LLM's output contained any indication that a process might be malicious—whether

explicitly naming malware or referencing suspicious artifacts—it was counted as a positive classification. This binary mapping allowed us to calculate metrics at the process level. We also analyzed the causes of misclassification to better understand the behavior of the model. For false positives, we examined which benign indicators might have been misinterpreted as malicious. For false negatives, we investigated whether the model missed relevant indicators or failed to connect them to a malicious pattern.

6.2 Evaluation Metrics: Statistical Analysis

We used statistical tests to compare the performance across the different LLMs. We used a one-way ANOVA to test for any significant differences in detection metrics between the models [62]. The ANOVA can tell us whether the performance of at least one model is significantly different from the others when considering all samples. Furthermore, we conducted paired t-tests for specific planned comparisons (such as 'thinking mode' vs. normal for the same base model). For instance, we directly compared Gemini Flash vs. Gemini Flash-Thinking on the same set of images to ascertain whether the reasoning mode provided a statistically significant improvement in detection success. To further analyze pairwise differences between models following a significant ANOVA result, we also performed post-hoc comparisons using the Tukey HSD test for F1 scores across the LLMs. We also explored the variance across different malware families to determine which model performs best for which malware. All statistical tests were performed using R and we considered p-values < 0.05 as significant. The results of these tests are reported in the next section, along with descriptive performance differences.

6.3 Overall Results

Across the 240 memory analysis trials (8 scenarios x 1 memory sample per scenario × 6 LLMs × 5 runs each), all LLMs achieved high overall accuracy, albeit with varying balances of precision and recall. The following tables summarize the key performance metrics across all LLMs (Table 6.1) and Viruses (Table 6.2). These quantitative overviews provide a foundation for the more in-depth performance analysis presented in the following sections. A full overview of the data is provided in Appendix F.1. Our statistical results are shown in Appendix F.2, and the Tukey HSD test is summarized in Appendix F.3.

LLM	Attempts	Mean Process Count	Mean TP	Mean TN	Mean FN	Mean FP	Mean Accuracy	Mean Precision	Mean Recall	Mean F1
Gemini 2.0 Flash	40	179.12	0.92	172.68	0.20	5.33	96.91	0.13	0.85	0.32
Gemini 2.0 Flash-Thinking	40	179.12	1.12	172.78	0.00	5.22	97.07	0.17	1.00	0.37
Grok 3	40	179.12	0.87	174.33	0.25	3.67	97.82	0.18	0.83	0.43
Grok 3 ThinkingMode	40	179.12	0.88	174.60	0.30	3.35	97.97	0.19	0.80	0.45
OpenAI GPT4o	40	179.12	1.07	173.18	0.05	4.83	97.26	0.17	0.97	0.36
OpenAI o1	40	179.12	1.12	174.33	0.00	3.67	97.94	0.22	1.00	0.44

Table 6.1: *Overview of Results Illustrated with LLMs*

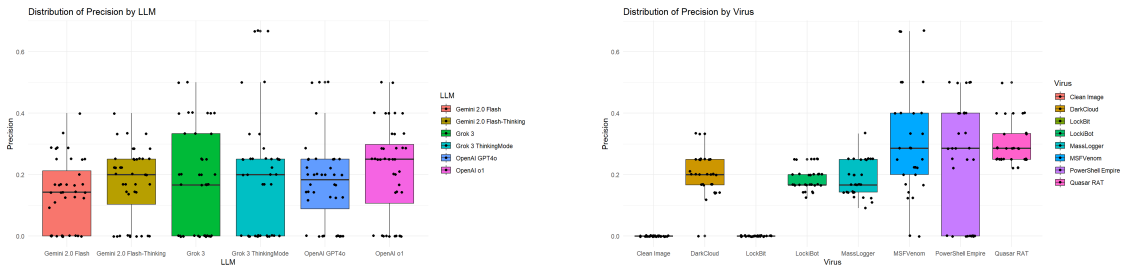
Virus	Attempts	Mean Process Count	Mean TP	Mean TN	Mean FN	Mean FP	Mean Accuracy	Mean Precision	Mean Recall	Mean F1
Clean Image	30	175.00	0.00	171.47	0.00	3.53	97.98	0.00	NaN	NaN
DarkCloud	30	177.00	0.94	171.99	0.07	4.00	97.70	0.20	0.95	0.34
LockBit	30	190.00	0.00	185.10	0.00	4.90	97.42	0.00	NaN	NaN
LockiBot	30	199.00	1.00	193.43	0.03	4.53	97.71	0.19	0.98	0.31
MassLogger	30	172.00	1.00	166.13	0.00	4.87	97.17	0.19	1.00	0.31
MSFVenom Code Injection	30	170.00	1.70	163.43	0.30	4.57	97.14	0.31	0.85	0.47
PowerShell Empire	30	178.00	1.33	172.67	0.67	3.33	97.75	0.24	0.67	0.53
Quasar RAT	30	172.00	2.00	164.97	0.00	5.03	97.07	0.30	1.00	0.45

Table 6.2: *Overview of Results Illustrated with Viruses*

6.4 Overall Detection Performance

The accuracy alone is not very informative because the data is highly imbalanced. The number of benign (TN) processes massively outweighs the number of malicious (TP) ones. All models had accuracy in the range of ~0.97–0.98 (97%+) across the full dataset.

The more revealing metrics are precision and recall. Recall reflects how sensitive the LLMs were to the presence of malware, namely how often they correctly identified infected processes. Two of the six LLM configurations (OpenAI o1 and Gemini-Thinking) achieved a recall of 100%, which means that they did not miss any of the malware-infected processes in our test. These high recall values indicate that LLMs—especially when guided by reason—are very effective at not overlooking suspicious evidence. In practical terms, high recall is important in forensics based on the aim of catching everything malicious, and most LLMs met this criterion in our experiments.



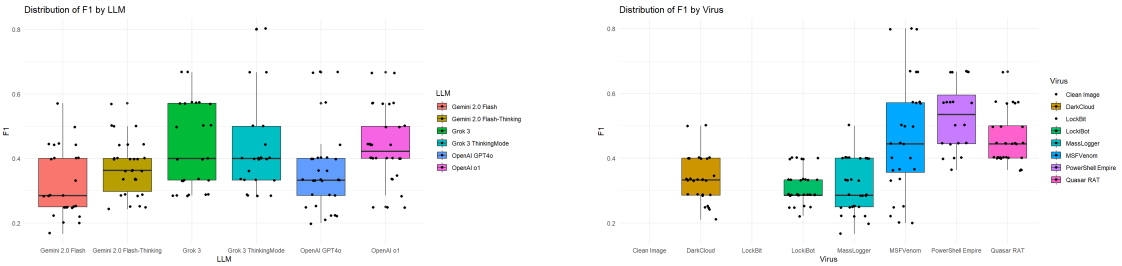
(a) *Precision per LLM configuration*

(b) *Precision per virus scenario*

Figure 6.1: *Average Precision Across Experiments: (a) Precision Per LLM Configuration; (b) Precision Per Virus Scenario*

The downside was the extremely low precision for most models, with precision (Fig. 6.1) values ranging for approximately 13% at worst to about 22% at best among the LLMs. In other words, most of the LLM alerts were FPs. For example, a model that achieved 100% recall only had ~21% precision, meaning that it was a false alarm four out of five times when it said 'malware.' Even the best precision (around 21–22%) is very low, indicating that if an analyst trusted the model output blindly, they would spend a lot of time investigating nonexistent issues. This aligns with our observation that the models tended to over-predict malware: when confronted with ambiguous or incomplete evidence, the LLMs often erred on the side of assuming something was malicious. This behavior

likely comes from the precautionary principle embedded in many AI safety alignments and the model's tendency to hallucinate plausible threats when uncertain.



(a) *F1 score per LLM configuration* (b) *F1 score per virus scenario*

Figure 6.2: *Average F1 Score Across Experiments: (a) F1 Score Per LLM Configuration; (b) F1 Score Per Virus Scenario*

Given the combination of high recall and low precision, the F1 scores were moderate (Fig. 6.2). The highest F1 score achieved by any LLM was about 0.45 (out of 1.0), and the lowest was around 0.32. The models that performed better on F1 managed to maintain high recall while slightly improving precision. In particular, OpenAI o1 and xAI's Grok 3 ('thinking mode') had the top F1 scores (~0.44 and ~0.45). OpenAI GPT4o had one of the lower F1 scores (~0.36) due to a combination of near-perfect recall but only mediocre precision. Google's Gemini Flash in normal mode had the lowest F1 (~0.32) as it missed some cases (recall ~85%) and had very low precision (~13%), indicating difficulties in both directions when not using the thinking mode. Gemini in thinking mode improved to an F1 score of ~0.37 by achieving perfect recall (100%), albeit still with low precision (~17%).

In general, these results show that all LLMs were effective in finding malware when it was present (a few missed detections), whereby especially Grok 3 and Grok 3 with thinking mode did not find the PowerShell Empire base64 encoded string. No LLMs were reliable in ignoring benign data (many false detections). In a practical sense, the LLMs functioned as very liberal detectors as they would rather flag ten items—including the real malware—than miss the malware entirely. To some extent, this mirrors the behavior of an inexperienced analyst who overinterprets every anomaly as evil. It also reflects the inherent challenge that the memory data often contained some anomalies or unusual patterns (even in clean scenarios) for which the LLMs did not have the contextual knowledge to dismiss as benign. For instance, benign administrative tools or system processes sometimes leave traces (in malfind or netscan) that appear suspicious out of context, and the LLM would treat them as malware evidence.

6.5 Model-Wise Performance Differences

We observed notable differences between LLMs, particularly between those run in 'thinking mode' vs. normal mode.

6.5.1 Reasoning ('Thinking') vs Standard Modes

The variants of the models that were prompted to perform a reasoning analysis (Gemini-Thinking, Grok 3 in thinking mode and OpenAI o1) generally performed better than their straightforward counterparts. Gemini 2.0 Flash in normal mode had the lowest recall (missed infections) and lowest precision (also many false alarms), while Gemini 2.0 Flash-Thinking achieved perfect recall and slightly higher precision. This suggests that the chain-of-thought prompt helped the model to catch all malware cases and perhaps avoid a few false positives by deliberating more. Similarly, xAI's Grok 3 in thinking mode had a marginal improvement in precision over Grok 3 normal, as well as roughly comparable recall (both very high). The Grok difference was smaller, indicating that Grok 3 is robust even without special prompting, although the reasoning mode still gave it a minor boost in F1 score. OpenAI o1 achieved 100% recall and the highest precision (around 21.6%) among all, giving it one of the highest F1 scores. By contrast, GPT-4o had 96.7% recall (it missed one case, which happened to be a tricky one) and about 17% precision.

Statistical analysis confirms this, with a paired t-test between the normal vs. thinking mode for these models showing a significant improvement in F1 score ($p < 0.05$) when reasoning was enabled, especially for Gemini. This finding confirms that reasoning-based LLM models outperform standard models in this task, aligning with our expectation that memory forensic interpretation is a complex reasoning problem for which models that emulate an analytical approach perform better.

6.5.2 OpenAI Models

One interesting outcome was that the OpenAI o1 slightly outperformed the OpenAI GPT-4o model on our metrics. Upon examining the outputs, we found that OpenAI o1 was more focused as it often identified the malicious pattern (e.g. the base64 encoded PowerShell command) and stuck to it, while GPT-4 sometimes "overexplained" and brought up multiple possibilities (including some incorrect ones, damaging precision). It is possible that GPT4's broader knowledge led it to imagining additional threats, while o1 was slightly more constrained. However, both OpenAI models performed well and the difference might not be considerable in practice.

6.5.3 Notable Strengths

Each model had specific strengths. For instance, we found that OpenAI o1 and Gemini-Thinking particularly excelled at recognizing obfuscated payloads. In the PowerShell Empire scenario, the memory contained a base64-encoded PowerShell command, which is how the Empire stager transmits its payload. The OpenAI o1 model was very effective at spotting this base64 string in the cmdline output, decoding it, and recognizing it as malicious. Gemini's thinking mode also managed to flag the presence of a base64 string and correctly interpret it as likely malicious PowerShell code, shown in Figure 6.3. By contrast, some other models might have overlooked it or did not understand its significance, leading to misses in the Empire cases (which explains why not all had 100% recall; the Empire scenario was one where the Grok 3 models of xAI failed).

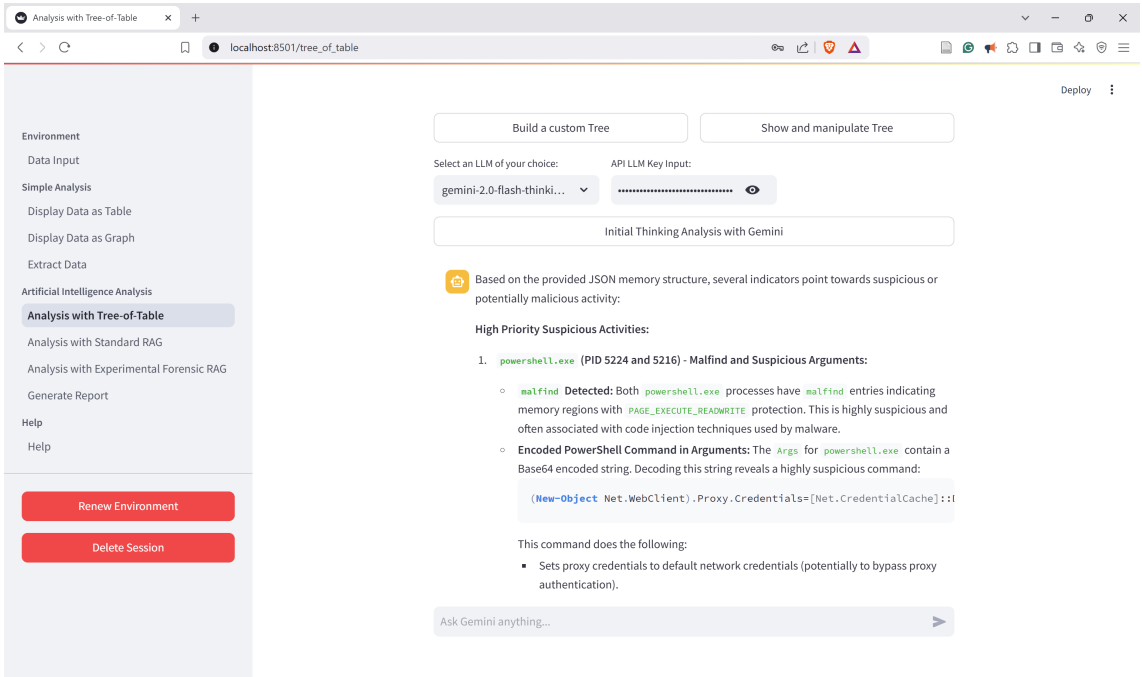


Figure 6.3: *Base64 Encoding from Gemini 2.0 Flash-Thinking*

Meanwhile, xAI's Grok 3 showed strength in detecting network-related anomalies (Fig. 6.4). For malware such as Quasar RAT—which exhibits network connections to unusual external addresses—Grok was adept at focusing on the netscan output and identifying suspicious connections or ports. It often highlighted the presence of an uncommon port connection or a remote IP that appeared to be anomalous for the process in question, which gave Grok an edge in those scenarios, sometimes even naming the RAT based on known indicators.

6.5.4 Efficiency

Although not quantified by a single number, we also considered how concise and clear the model's outputs were, since part of our interest is efficiency, namely how quickly an analyst could reach a conclusion with the help of the model. In this respect, OpenAI GPT-4 gave the most detailed answers (sometimes too detailed, listing many possibilities). OpenAI o1 tended to give a more direct "yes, there is malware because I see X and Y." Gemini in normal mode was sometimes uncertain ("there might be something suspicious but not sure"), whereas in thinking mode it was more decisive after reasoning through. One qualitative observation is that the reasoning modes made the AI explanation more structured, which could help an analyst to follow the logic. This clarity is valuable in a forensic report or summary of the investigation.

Grok

JL

2. Network Activity (netscan)

The `netscan` data reveals active and listening network connections. Let's examine key entries:

- **System (PID 4)**:
  - UDPv4, `192.168.241.139:138` → `*`, `LISTENING`. This is typical for NetBIOS name service on a local network.
- **firefox.exe (PID 5552)**:
  - TCPv4, `192.168.241.139:49841` → `34.107.152.202:443`, `ESTABLISHED`. Likely a legitimate HTTPS connection (Google Cloud IP range).
- **taskhostw.exe (PID 4736)**:
  - TCPv4, `192.168.241.139:49876` → `13.93.211.12:443`, `ESTABLISHED`. Microsoft Azure IP, possibly telemetry or updates.
- **FoxitPDFReader (PID 7180)**:
  - TCPv4, `192.168.241.139:49879` → `3.160.150.109:443`, `ESTABLISHED`. Amazon AWS IP, possibly for cloud services or updates.
- **explorer.exe (PID 5264)**:
  - TCPv4, `192.168.241.139:49880` → `2.23.209.3:443`, `ESTABLISHED`. Akamai CDN,

Wie kann Grok helfen?

DeepSearch

Think

Grok 3

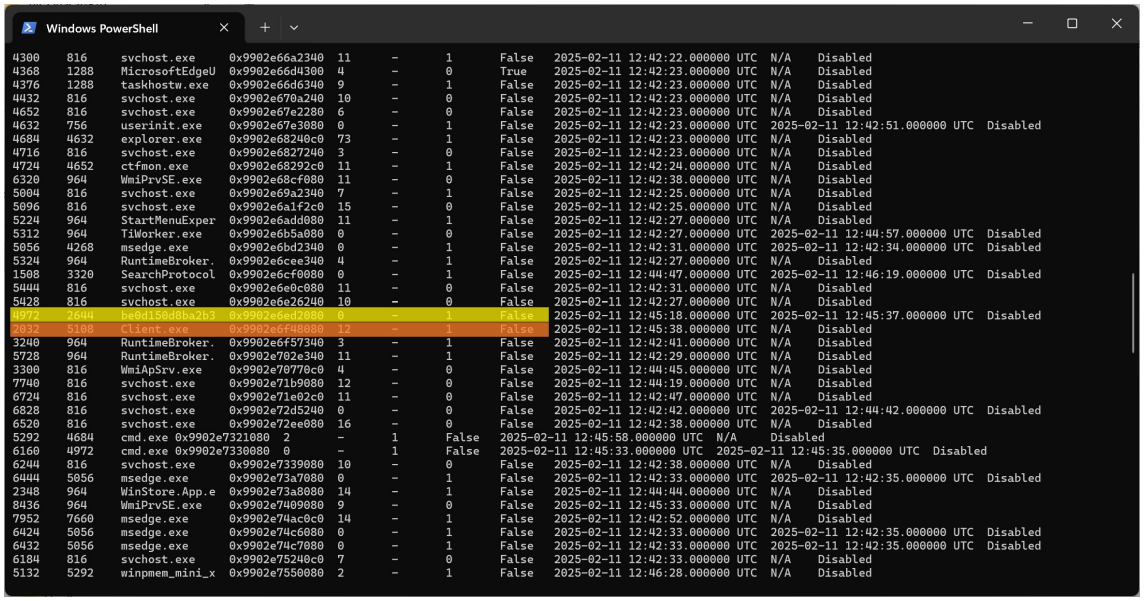
Figure 6.4: *Exhibition of Network Connections from Grok3*

6.6 Malware Family-Specific Results

Each malware scenario provides individually insights into which ones were easy or difficult to detect. First, we will take a closer look at the QuasarRAT scenario and why it was detected by the LLMs.

6.6.1 Quasar RAT

Quasar RAT was a special test case, which we want to describe in further detail. We want to explain what a human analyst would find in a simple first view and why the LLM was able to catch it. In the infected memory image of Quasar RAT, the malware behavior left multiple suspicious footprints. From Section 5.6 combined with the behavior graph from Figure 5.3 and the knowledge from the CTI data from Table 5.2, we know that Quasar RAT will drop additional malicious files, spawn multiple processes (such as Client.exe), and start PowerShell activities and C2 communication. We analyze the memory dump Using the SANS Memory Forensic CheatSheet [93]. First, we look at the windows.psscan output and identify rogue processes.



4300	816	svchost.exe	0x9902e66a2340	11	-	1	False	2025-02-11 12:42:22.000000 UTC	N/A	Disabled	
4368	1288	MicrosoftEdgeU	0x9902e66d4300	4	-	0	True	2025-02-11 12:42:23.000000 UTC	N/A	Disabled	
4376	1288	taskhostw.exe	0x9902e66d6340	9	-	1	False	2025-02-11 12:42:23.000000 UTC	N/A	Disabled	
4432	816	svchost.exe	0x9902e670a240	10	-	0	False	2025-02-11 12:42:23.000000 UTC	N/A	Disabled	
4652	816	svchost.exe	0x9902e67e2280	6	-	0	False	2025-02-11 12:42:23.000000 UTC	N/A	Disabled	
4632	756	userinit.exe	0x9902e67e3080	0	-	1	False	2025-02-11 12:42:23.000000 UTC	2025-02-11 12:42:51.000000 UTC	Disabled	
4684	4632	explorer.exe	0x9902e68240c0	73	-	1	False	2025-02-11 12:42:23.000000 UTC	N/A	Disabled	
4716	816	svchost.exe	0x9902e6827240	3	-	0	False	2025-02-11 12:42:23.000000 UTC	N/A	Disabled	
4724	4652	ctfmon.exe	0x9902e68292c0	11	-	1	False	2025-02-11 12:42:24.000000 UTC	N/A	Disabled	
6320	964	WmiPrvSE.exe	0x9902e68cf080	11	-	0	False	2025-02-11 12:42:38.000000 UTC	N/A	Disabled	
5004	816	svchost.exe	0x9902e69a2340	7	-	1	False	2025-02-11 12:42:25.000000 UTC	N/A	Disabled	
5096	816	svchost.exe	0x9902e6a1f2c0	15	-	0	False	2025-02-11 12:42:25.000000 UTC	N/A	Disabled	
5224	964	StartMenuExper	0x9902e6add080	11	-	1	False	2025-02-11 12:42:27.000000 UTC	N/A	Disabled	
5312	964	TiWorker.exe	0x9902e6b5a080	0	-	0	False	2025-02-11 12:42:27.000000 UTC	2025-02-11 12:44:57.000000 UTC	Disabled	
5056	4268	msedge.exe	0x9902e6bd2340	0	-	1	False	2025-02-11 12:42:31.000000 UTC	2025-02-11 12:42:34.000000 UTC	Disabled	
5324	964	RuntimeBroker.	0x9902e6cee340	4	-	1	False	2025-02-11 12:42:27.000000 UTC	N/A	Disabled	
1508	3320	SearchProtocol	0x9902e6cf0080	0	-	1	False	2025-02-11 12:44:47.000000 UTC	2025-02-11 12:46:19.000000 UTC	Disabled	
5444	816	svchost.exe	0x9902e6e0c080	11	-	0	False	2025-02-11 12:42:31.000000 UTC	N/A	Disabled	
5428	816	svchost.exe	0x9902e6e26240	10	-	0	False	2025-02-11 12:42:27.000000 UTC	N/A	Disabled	
4972	2644	Be9dA50d9ba2b3	0x9902e6e4f080	0	-	1	False	2025-02-11 12:45:18.000000 UTC	2025-02-11 12:45:37.000000 UTC	Disabled	
2032	5108	Client.exe	0x9902e6f48080	12	-	1	False	2025-02-11 12:45:38.000000 UTC	N/A	Disabled	
3240	964	RuntimeBroker.	0x9902e6f57340	3	-	1	False	2025-02-11 12:42:41.000000 UTC	N/A	Disabled	
5728	964	RuntimeBroker.	0x9902e702e340	11	-	1	False	2025-02-11 12:42:29.000000 UTC	N/A	Disabled	
3300	816	WmiApSrv.exe	0x9902e70770c0	4	-	0	False	2025-02-11 12:44:45.000000 UTC	N/A	Disabled	
7740	816	svchost.exe	0x9902e71b9080	12	-	0	False	2025-02-11 12:44:19.000000 UTC	N/A	Disabled	
6724	816	svchost.exe	0x9902e71e02c0	11	-	0	False	2025-02-11 12:42:47.000000 UTC	N/A	Disabled	
6828	816	svchost.exe	0x9902e72d5240	0	-	0	False	2025-02-11 12:42:42.000000 UTC	2025-02-11 12:44:42.000000 UTC	Disabled	
6520	816	svchost.exe	0x9902e72ee080	16	-	0	False	2025-02-11 12:42:38.000000 UTC	N/A	Disabled	
5292	4684	cmd.exe 0x9902e7321080	2	-	1	False	2025-02-11 12:45:58.000000 UTC	N/A	Disabled		
6160	4972	cmd.exe 0x9902e7330080	0	-	1	False	2025-02-11 12:45:33.000000 UTC	2025-02-11 12:45:35.000000 UTC	Disabled		
6244	816	svchost.exe	0x9902e7339080	10	-	0	False	2025-02-11 12:42:38.000000 UTC	N/A	Disabled	
6444	5056	msedge.exe	0x9902e73a7080	0	-	1	False	2025-02-11 12:42:33.000000 UTC	2025-02-11 12:42:35.000000 UTC	Disabled	
2348	964	WinStore.App.e	0x9902e73a8080	14	-	1	False	2025-02-11 12:44:44.000000 UTC	N/A	Disabled	
8436	964	WmiPrvSE.exe	0x9902e7409080	9	-	0	False	2025-02-11 12:45:33.000000 UTC	N/A	Disabled	
7952	7660	msedge.exe	0x9902e74ac0c0	14	-	1	False	2025-02-11 12:42:52.000000 UTC	N/A	Disabled	
6424	5056	msedge.exe	0x9902e74c6080	0	-	1	False	2025-02-11 12:42:33.000000 UTC	2025-02-11 12:42:35.000000 UTC	Disabled	
6432	5056	msedge.exe	0x9902e74c7080	0	-	1	False	2025-02-11 12:42:33.000000 UTC	2025-02-11 12:42:35.000000 UTC	Disabled	
6184	816	svchost.exe	0x9902e75240c0	7	-	0	False	2025-02-11 12:42:33.000000 UTC	N/A	Disabled	
5132	5292	winpmem_mini_x	0x9902e7550080	2	-	1	False	2025-02-11 12:46:28.000000 UTC	N/A	Disabled	

Figure 6.5: *QuasarRAT windows.psscan Output with the Suspicious Indicators Marked in Yellow and Red*

The process marked in yellow was the Quasar RAT malware sample executed by ourselves. This process contains zero threads and is therefore an exited process. We can see this process because we look into the memory dump with windows.psscan. The windows.pslist module walks the the PsActiveProcessHead, which is a doubly linked list of _EPROCESS structures representing all currently active processes in memory. Instead of walking through the list, windows.psscan uses pool scanning, namely searching the kernel memory pool for specific patterns or structures (_EPROCESS BLOCK, DRIVER_OBJECT, etc.) based on known characteristics, rather than relying on a list. The process marked in red is the *client.exe*. From CTI information [59], we know that Quasar RAT hides itself as *client.exe*. Interestingly, the PID or PPID is not connected to the executed Quasar RAT malware sample. The PPID of the *client.exe* with the number 5108 is not existing as a process. This could be normal behavior and not suspicious due to its an orphaned process, although it could also be an attempt to hide a process, e.g. with DCOM and indicate malware behavior.

At the next step of the SANS Memory Forensic CheatSheet [93], it is necessary to explore the process objects with windows.dlllist, windows.vadinfo, or windows.cmdline. In the windows.dlllist, we found that *client.exe* resides in *AppData* path, which is often exploited by malware authors to evade detection. Furthermore the process exhibits a full .NET runtime environment (clr.dll, MSCOREE.DLL, and multiple native images (.ni) DLLs). This indicates the use of obfuscation, reflective loading, or runtime assembly manipulation techniques. Moreover, the webengine4.dll implies the use of an embedded web server or browser functionality, which is atypical for standard user applications. In summary, this indicates the hypothesis that *client.exe* could be malware. The windows.dlllist output of the PID 2032 can be found in Appendix G.1. The analysis of the windows.netscan modul shows a SYN_SENT to the IP 87.228.57.81:4782. The windows.netscan output can be found in Appendix G.2. The IP also indicates an IoC due to the CTI reports.

The next stage is to look for evidence of code injection. This can be achieved with the windows.malfind module and the output is shown in Figure 6.6. The windows.malfind module finds various results.

```
Windows PowerShell
2032    Client.exe      0xfe0000        0xfeffff        VadS    PAGE_EXECUTE_READWRITE  2       1       Disabled        N/A
00 00 00 00 00 00 00 00 4c 1a f8 32 cf ab 00 01 ........L..2....
ee ff ee ff 02 00 00 00 20 01 fe 00 00 00 00 00 ........ .......
20 01 fe 00 00 00 00 00 00 00 fe 00 00 00 00 00  ...............
00 00 fe 00 00 00 00 00 0f 00 00 00 00 00 00 00 ................
0xfe0000:       add     byte ptr [rax], al
0xfe0002:       add     byte ptr [rax], al
0xfe0004:       add     byte ptr [rax], al
0xfe0006:       add     byte ptr [rax], al
0xfe0008:       sbb     r15b, al
0xfe000b:       xor     cl, bh
0xfe000d:       stosd   dword ptr [rdi], eax
0xfe000e:       add     byte ptr [rcx], al
0xfe0010:       out     dx, al
2032    Client.exe      0x1b560000      0x1b56ffff      VadS    PAGE_EXECUTE_READWRITE  2       1       Disabled        N/A
00 00 00 00 00 00 00 00 69 6a 33 da 54 0c 00 01 ........ij3.T...
ee ff ee ff 02 00 00 00 20 01 56 1b 00 00 00 00 ........ .V.....
20 01 56 1b 00 00 00 00 00 00 56 1b 00 00 00 00  .V.......V.....
00 00 56 1b 00 00 00 00 0f 00 00 00 00 00 00 00 ..V............
0x1b560000:     add     byte ptr [rax], al
0x1b560002:     add     byte ptr [rax], al
0x1b560004:     add     byte ptr [rax], al
0x1b560006:     add     byte ptr [rax], al
0x1b560008:     imul    ebp, dword ptr [rdx + 0x33], 0xc54da
0x1b56000f:     add     esi, ebp
2032    Client.exe      0x7ff40de70000  0x7ff40de7ffff  VadS    PAGE_EXECUTE_READWRITE  1       1       Disabled        N/A
00 00 00 00 00 00 00 00 78 0d 00 00 00 00 00 00 ........x.......
0c 00 00 00 49 c7 c2 00 00 00 00 48 b8 a0 49 7d ....I......H..I}
94 f9 7f 00 00 ff e0 49 c7 c2 01 00 00 00 48 b8 .......I......H.
a0 49 7d 94 f9 7f 00 00 ff e0 49 c7 c2 02 00 00 .I}.......I.....
0x7ff40de70000: add     byte ptr [rax], al
0x7ff40de70002: add     byte ptr [rax], al
0x7ff40de70004: add     byte ptr [rax], al
0x7ff40de70006: add     byte ptr [rax], al
0x7ff40de70008: js      0x7ff40de70017
0x7ff40de7000a: add     byte ptr [rax], al
0x7ff40de7000c: add     byte ptr [rax], al
0x7ff40de7000e: add     byte ptr [rax], al
0x7ff40de70010: or      al, 0
0x7ff40de70012: add     byte ptr [rax], al
0x7ff40de70014: mov     r10, 0
0x7ff40de7001b: movabs  rax, 0x7ff9947d49a0
0x7ff40de70025: jmp     rax
0x7ff40de70027: mov     r10, 1
0x7ff40de7002e: movabs  rax, 0x7ff9947d49a0
0x7ff40de70038: jmp     rax
2032    Client.exe      0x7ff40de80000  0x7ff40df1ffff  VadS    PAGE_EXECUTE_READWRITE  2       1       Disabled        N/A
d8 ff ff ff ff ff ff ff 08 00 00 00 00 00 00 00 ................
01 00 00 00 00 00 00 00 00 02 0e 03 38 00 00 00 ............8...
68 01 02 08 0c 00 00 00 d8 5d 1f 7d f9 7f 00 00 h........].}....
00 10 1d 7d f9 7f 00 00 18 85 36 7d f9 7f 00 00 ...}......6}....
```

Figure 6.6: *QuasarRAT windows.psscan Output with Different Obvious Parts*

```
add     byte ptr [rax], al
sbb     r15b, al
xor     cl, bh
stosd   dword ptr [rdi], eax
out     dx, al
```

Listing 6.1: *Disassembly of Region 1 (0xfe0000)*

The first interesting result is the instructions in Listing 6.1, which appear to be either NOP-equivalent padding or obfuscated shellcode. The lack of meaningful operational flow suggests this memory area could serve as a staging ground for shellcode loaders or a placeholder for future payloads.

```
imul    ebp, dword ptr [rdx + 0x33], 0xc54da
```

Listing 6.2: *Disassembly of Region 2 (0x1b560000)*

The instruction in Listing 6.2 indicates a more advanced computation. The use of indexed memory access and immediate multiplication hints at unpacking logic or custom loader routines, consistent with behaviors observed in malware attempting to obfuscate or reconstruct its payload in memory.

```
mov     r10, 0
movabs  rax, 0x7ff9947d49a0
jmp     rax
```

Listing 6.3: *Disassembly of Region 3 (0x7ff40de70000)*

Listing 6.3 contains a typical shellcode pattern where control flow is redirected via a hardcoded jump to a memory address. Such sequences are typical in shellcode stubs and serve to transfer execution to the main payload, likely the core Quasar RAT component in this case.

```
fdivr   st(7)
```

Listing 6.4: *Disassembly of Region 4 (0x7ff40de80000)*

Listing 6.3 is a floating-point reverse division instruction that is anomalous and might represent either a malformed opcode, decoy instruction, or misinterpreted data. Nevertheless, the RWX permissions and context strongly suggest that this region might still contain injected or encoded data.

Finally, *client.exe* was extracted from the memory dump and sent to VirusTotal (Figure 6.7). In real-life scenarios, analysts must be careful sending viruses or hashes to VirusTotal because it could alarm the intruders that the malware was found and lead to the intruders executing their action on the target. This led to the conclusion that the *client.exe* process is the malicious Quasar RAT.

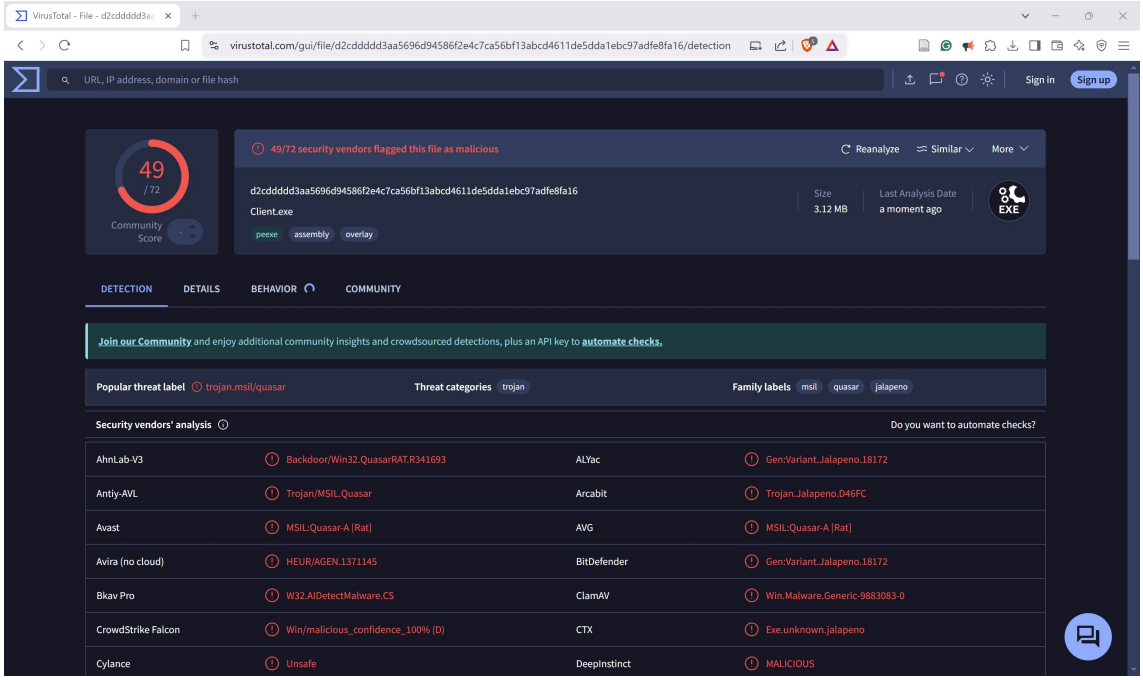


Figure 6.7: *Client.exe VirusTotal Check [95]*

The detection rate of Quasar RAT was effectively 100% across models, which means that each LLM flagged every infected Quasar RAT memory as malicious. Differences between the different models can also be observed. GPT4o catches the windows.malfind and the windows.netscan IoC (Appendix G.3), while Gemini 1.5 only catches the windows.malfind IoC (Appendix G.4). The main concern among the LLMs was the high FP rate due to the overinterpretation of Volatility's malfind output.

6.6.2 MassLogger

Across all LLMs evaluated, MassLogger was identified and demonstrated a recall of about 100%. The LLM recognized every infection in the test case. MassLogger is a credential stealer that left behind distinctive forensic traces. A running process with unusual behavior is mostly suspicious. Due to this behavior, it relatively easy for the LLMs to flag the presence of malware. Precision remained low in some instances as several models also misclassified unrelated benign processes, mostly incorrectly flagged from the windows.malfind output.

6.6.3 Process Injection (msfvenom)

This scenario had a moderately high detection rate, with a total recall of around 85% (Gemini 2.0 Flash and OpenAI GPT4o missed a couple of injection cases). The injection attack was straightforward, involving a userland process with injected msfvenom code. *Volatility3* malfind shows the process injection in the injected process with an additional MZ header. The precision for process injection was relatively higher (~30%) compared to

other malware. We interpret this as the models being slightly more confident and specific when they see a clear code injection, whereby they flag it and usually do not hallucinate other problems. Often the output would be something like "Malfind plugin finds unusual MZ header in the process." Process injection attacks were one scenario where LLMs were both effective and slightly more precise. Nonetheless, some false positives occurred in clean or other scenarios due to malfind (which we discuss later in the false positives section).

6.6.4 PowerShell Empire

Recall for the Empire agent scenario was about 66% (two-thirds of the models' attempts caught it). Those that succeeded were typically those adept at decoding the base64 payload. The others missed it, likely because the memory artifacts can appear relatively normal given that a PowerShell process in memory by itself is not malicious unless you notice the encoded command or unusual network connection. The precision among those that detected Empire was around 24%, slightly higher than average, suggesting that when they recognized something as an Empire agent, they did not over-flag too much else, possibly because the Empire artifacts were distinct. However, some models that failed to detect PowerShell Empire did not necessarily flag the image as clean either, whereby a couple gave uncertain answers, which we treated as a miss. The Empire scenario underscores that an LLM can miss a fileless attack without specific pattern recognition (like decoding base64). The better-performing models (OpenAI o1, Gemini-Thinking) explicitly called out the base64 PowerShell command in the cmdline output and stated that this is a known malware technique, thus catching Empire. This mirrors what a skilled analyst would do, namely decode suspicious commands and identify them as malicious.

6.6.5 DarkCloud

The DarkCloud banking trojan was generally caught (95% recall). Most LLMs flagged DarkCloud as malicious, whereas one model missed one DarkCloud instance. The precision of DarkCloud was low (~20%), meaning that the models flagged other things in those cases.

6.6.6 LockiBot

LockiBot had very high recall (~98%), with essentially all but one detection on all tries. Therefore, LLMs can also catch LockiBot as the models consistently flagged it. Precision was among the lowest (~18%), meaning that whenever LokiBot was present, the models also raised other suspicions. They often labeled it generically as "some infection present."

6.6.7 LockBit Ransomware

This was the more difficult scenario. Across all models and samples, effectively none of the LLMs were able to correctly identify the presence of LockBit in the memory image. The recall for LockBit was 0% in our LLM tests. As mentioned earlier, the Volatility plugins

did not surface obvious indicators for LockBit as it was not running at the snapshot as a separate process (it might have been terminated). There were no network connections (ransomware often does not need C2), and no base64 commands or anything. Essentially, the memory dumps looked clean in terms of the output of our chosen plugins. The LLMs accordingly saw nothing strongly suspicious and thus typically output that the system looked clean or only minor benign issues. This highlights a fundamental limitation that if the relevant IoCs are absent from the data provided (in this case, our Volatility plugin set did not include, e.g., memory strings or data from the registry), LLMs will be blind to the attack. The LockBit scenario dropped the recall of each model by a chunk. However, again, the LLMs overinterpreted benign processes in this scenario.

Statistically, if we exclude LockBit and the clean image from the calculation, the average recall of LLMs significantly increases (~90%). When including LockBit, it declines for all. This scenario underscores the importance of data coverage: the LLM cannot perform magic but rather can only draw conclusions from the evidence that it sees. If key evidence is missing (or outside its context window), the LLM approach fails. We will revisit this point in the discussion.

6.7 False Positives

A large portion of FPs for the LLMs were due to overinterpreting Volatility's malfind output. The malfind plugin often showed memory regions that it considered suspicious. For a clean system, these could be artifacts of legitimate programs, such as anti-malware software injecting code for hooking shown in Figure 6.8 or normal .NET dynamic code generation).

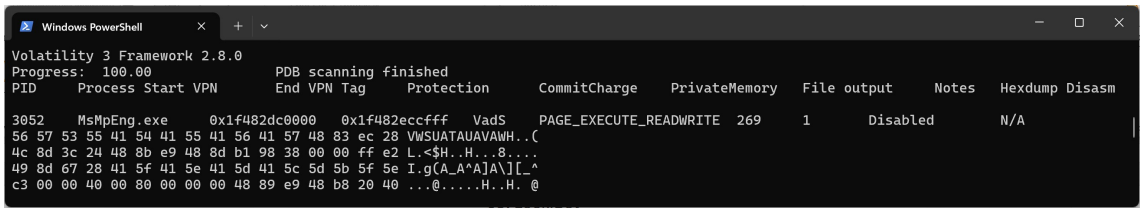


Figure 6.8: *MsMpEng in windows.malfind Output*

LLMs almost universally see that "malfind found suspicious memory regions in process X" and conclude that process X is likely infected. For example, on the clean image, malfind showed a couple of code segments in svchost.exe, or MsMpEng.exe (which is actually normal behavior). The LLMs said: "svchost.exe looks like it has injected code, this is likely malicious." In reality, an expert would recognize those memory sections as normal or corroborate with baseline knowledge to see that they match the known good patterns. This issue of malfind causing FPs was anticipated as a known noisy indicator. Without a reliable baseline or whitelist, the LLM erred on the side of assuming the worst case. Even if a process in psscan looked incorrect due to an error (Figure G.3), it was overinterpreted and reported as infected. Essentially, anything that the model did not recognize as known

good became 'suspicious.' This is where a baseline in memory forensics could sort out well-known processes and only provide the LLM the unknown data beyond the baseline.

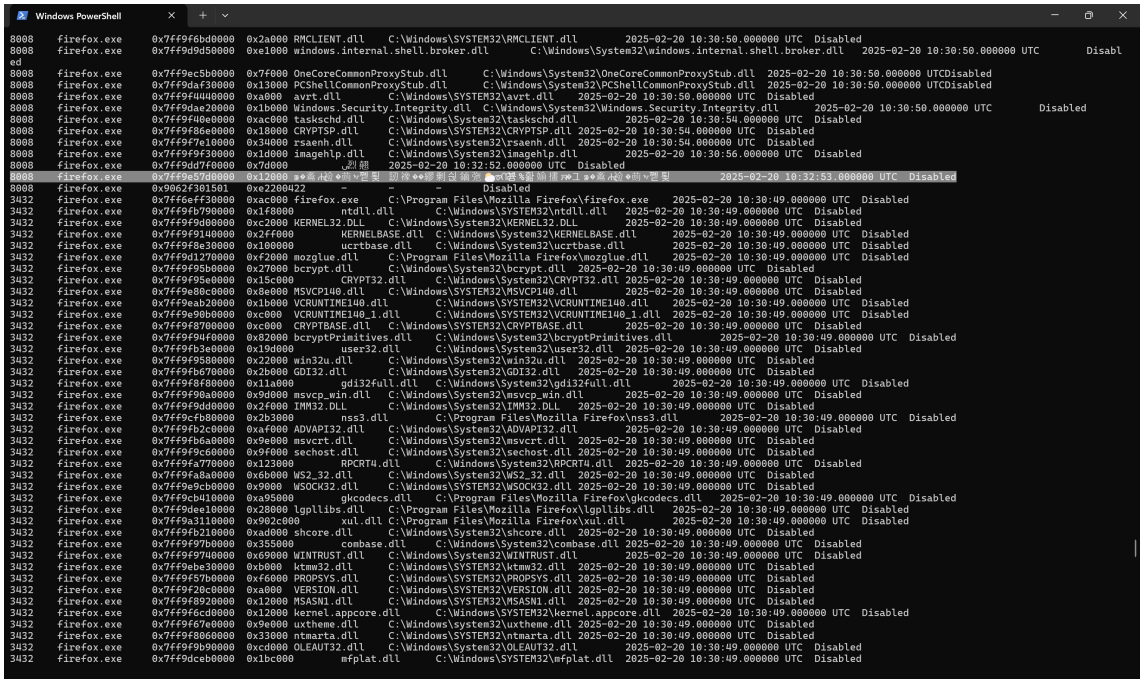


Figure 6.9: *Unknown Characters in windows.psscan Output*

6.8 False Negatives

Aside from the LockBit case—which was uniformly missed due to a lack of evidence—the other FNs were mostly in the PowerShell Empire scenario. Some models did not detect the Empire agent, and they probably did not decode base64 or realize what it was. They might have lacked knowledge about how base64 or PowerShell Empire works, and perhaps they simply did not 'connect the dots' that a long base64 string on a command line is a telltale sign, indicating a gap for these models. It is interesting because models like OpenAI o1 clearly had either training data or the reasoning ability to know this pattern, whereas others did not. In our study, only one configuration behaved that way (Gemini normal mode seemed slightly conservative and thus missed a couple). Most models were aggressive and hence had few FNs aside from LockBit.

6.9 Statistical Significance

The one-way ANOVA concerning the performance of the six LLMs (using the F1 score as the dependent variable) produced a statistically significant result ($p < 0.01$), indicating that not all models performed equally, with some systematic differences. Post-hoc comparisons (Tukey HSD) showed that the Gemini Flash (normal) was significantly worse than OpenAI o1 and Grok with thinking mode enabled in F1, which matches our observations

of it being the lowest performer vs. top performers. The differences between the four main models (OpenAI o1, Grok with thinking mode enabled, Grok normal, and OpenAI GPT4) were not statistically significant as they were in a similar band. Gemini-Thinking was slightly lower and Gemini normal significantly lower. In summary, the results demonstrate that while all tested LLMs can effectively process memory forensic data to find evidence of malware, there are quantitative differences in how well they perform and how much noise they introduce. Reasoning-enhanced models tend to produce better-balanced results (high recall with slightly less terrible precision), and different models have speciality areas, i.e., some are better at decoding obfuscation, and others at spotting network anomalies. No model was a clear winner in all aspects, although OpenAI o1 and Grok 3 with thinking mode enabled consistently performed near the top.

7 Discussion

We employed the Design Science Research Method from Peffers et. al [68] to build MemoryInvestigator to complement volatile memory forensics, exploring its effectiveness in detecting different malware scenarios based on the overall research question: *"Can the automation of Volatility3 in combination with an LLM be used as a proof of concept to make queries in memory forensics more efficient and user-friendly?"* This chapter explores the key findings, their implications for integrating LLMs in digital forensics, outlines the limitations of this work, and provides recommendations for forensic practitioners.

7.1 Key Findings

Evaluating MemoryInvestigator derived several relevant findings. In addition to having a user-friendly front-end to upload memory dumps and the automation of Volatility3 to simplify an analyst's work, we have demonstrated that LLMs can effectively detect malware. We used *OpenAI GPT4o*, *OpenAI o1*, *Gemini 2.0 Flash*, *Gemini 2.0 Flash-Thinking*, *Grok 3*, *and Grok 3 with thinking mode enabled* on various attack scenarios, namely *process injection (via msfvenom)*, a *PowerShell Empire-based attack*, and real-world malware including *Quasar RAT*, *MassLogger*, *DarkCloud*, *LockBit*, *and LockiBot*). All tested LLMs found evidence of malware in the memory dumps, with the exception of *LockBit* due its lack of evidence in the data provided. Accordingly, it can be concluded that LLM-driven analysis can successfully support but not replace an forensic analyst. The FN scenario of *LockBit* was driven by a lack of context, rooted by the LLMs' limited context windows and its lack of evidence in the Volatility3 output, emphasizing the importance of the coverage of forensic data. Given that the LLM can only process about the data it sees, the AI analysis will encounter blind spots if important evidence is missing. Therefore, it is necessary to expand data sources (beyond our current set of plugins) to handle more sophisticated or subtle attacks.

Higher recall (often 100%) and F1 scores emphasize that reasoning-based LLMs ("thinking" mode) improve detection performance by helping models to correlate subtle clues across different artifacts. This underlines that memory forensic analysis benefits from an AI approach that emulates human analytical reasoning. However, high recall comes at the expense of low precision. In this study, all LLMs recorded low precision (with many false alarms), frequently below 20%. They often flagged benign artifacts because they did not have environment-specific knowledge to recognize them as harmless, highlighting that a human analyst remains necessary. This finding also indicates the need of a baseline of a clean system. Indeed, this trade-off between recall and precision emerged as a core characteristic of the system. Each LLM has its own complementary strengths, whereby Grok 3 was effective in identifying network anomalies (suspicious connections, ports), while OpenAI o1 and Gemini (thinking) excelled at detecting obfuscated code payloads such as base64-encoded scripts. This finding indicates the importance of careful model selection or using multiple AI agents in tandem.

Our findings confirm that using LLMs in memory forensics is promising yet challenging. While we confirmed that LLMs can act as capable "virtual analysts" that find malware traces and even offer hypotheses about them, how you prompt the LLM and what data you feed it critically affect its performance. The results provide a foundation for improving the system and hint at the need to focus on improving precision, providing context, and enlarging data scope without sacrificing the LLMs' impressive recall and analytical depth. In the following sections, we discuss the implications of these findings and propose ways to address the limitations identified.

7.2 Implications for Forensics and LLM Integration

The successful implementation of MemoryInvestigator—and thus the successful integration of LLMs into memory forensics—holds several key implications for the field. Given that MemoryInvestigator automates several labor-intensive steps and LLMs offer a form of intelligent automation in memory analysis, this augments an analyst's capabilities. An LLM can rapidly sift through voluminous and complex forensic outputs (process lists, dll lists, network connections) and highlight what looks suspicious, offering the potential to significantly speed up investigations. Rather than manually cross-referencing plugin outputs, an analyst can leverage the LLM to obtain a distilled report of likely issues. The high recall observed means that an investigator is unlikely to miss major malware activity when using the tool as an assistant. In practice, this could transform the triage phase of incident response, whereby an LLM could quickly screen memory dumps from many machines to flag those that warrant closer attention.

Due to the lower entry level to memory forensics, the stated implications can be understand as its democratization. Whereas in-depth memory analysis traditionally requires substantial expertise, our LLM-based approach lowers the barrier to entry for less-experienced practitioners by providing a natural language interface and explanation of findings. An analyst who is not a Volatility expert can still interpret the LLMs report, which might say (for example) "Process X is suspicious because it injected code into Process Y and opened a connection to Z." This makes memory forensics more accessible and could enable the broader use of these techniques in IT departments or security teams that lack a dedicated forensics specialist. In an educational context, such a tool can serve as a training aid as students can learn from the AI's reasoning, effectively observing how an expert might walk through the analysis.

Besides the traditional tools in cyber security—which mostly remains a "black box"—we have highlighted the importance of explainable cybersecurity tool. In a field where trust is crucial (analysts need to trust the tool but also need to verify evidence), a tool such as a LLM that provides more information for each flag (e.g., "I flag process A because it loaded module B which is known malicious") is much more useful than a black-box alert (e.g. a simple Volatility3 output). This offers promise that LLMs could be integrated into existing forensic workflows as a co-pilot. For example, a forensic platform might have an "Analyze with AI" feature that runs something similar to MemoryInvestigator on collected artifacts. For vendors and tool developers, incorporating AI can enhance their products, although

this should be undertaken in a way that complements the analyst. Based on our study, AI is best used as an assistant that provides hypotheses and highlights, rather than an oracle. It could also enable new workflows such as iterative investigation, whereby the analyst asks follow-up questions to the LLM ("why do you think this process is malicious?" or "show me more about that network connection"), leveraging the interactive nature of language models.

Due to its low precision, it is essential that the human analyst remains in the loop. While the LLM can dramatically enhance speed and coverage, given its propensity for false positives and occasional false negatives, a human must validate and make the final judgments. In a legal context (e.g., forensic reports for court), the LLM findings would need expert confirmation. Therefore, it is unlikely that LLM will replace the forensic analyst but rather amplify their capabilities. It can handle the tedious parts and surface leads, allowing the human to apply intuition, contextual awareness, and decision-making to those leads. This synergy can lead to more effective and efficient investigations, provided that the limitations are understood.

An LLM-driven analysis is only as good as the data and context that it possess. For full integration into forensic practice, it is necessary to provide the AI with richer context. This could mean using "baseline" information about the system and thus sorting out most well-known processes to reduce false positives. This also means expanding the data sources beyond what we used, whereby registry hives, event logs, memory strings, kernel structures could contain crucial evidence that the current pipeline misses. Forensic practitioners looking to use AI should not neglect traditional data collection but rather engage in more comprehensive data collection to reach its full potential.

In summary, integrating LLMs into memory forensics stands holds the potential to make memory analysis faster, more accessible, and potentially more thorough, changing how we approach volatile memory in incident response. However, it also requires providing sufficient context to the AI and ensuring expert oversight. This work lays the groundwork for a new class of forensic tools that blend AI-driven analysis with traditional methods, pointing to a future in which memory forensics is not a niche expert task but rather a routine element of security operations augmented by intelligent assistants.

7.3 Limitations of the Study

Despite our encouraging results, it is essential to critically assess the limitations of this study and the MemoryInvestigator approach. The most immediate limitation is the prevalence of FPs. As detailed in Section 6.7, the LLM models frequently flagged benign artifacts as malicious, leading to low precision. In practical terms, this could overwhelm analysts by necessitating investigating non-issues. The FPs stem from the LLMs' lack of context about what is normal for a given system and their inclination to err on the side of caution. This issue means that the current system in its raw form is not ready for unsupervised use as it requires either significant post-processing (to filter obvious benign findings) or an analyst prepared to spend time vetting each alert. Reducing FPs is imperative before deployment in a production forensic workflow.

Our study was limited to analyzing outputs from select Volatility3 plugins (process lists, malfind, network connections, DLLs, handles, etc.), meaning that it did not include critical artifacts such as registry entries, event logs, kernel callbacks, or raw memory strings. As the LockBit case showed, this can lead to blind spots where the LLM has no data to identify a threat. Thus, one limitation of our evaluation is that it does not guarantee LLM performance on malware that hides outside the examined artifacts. The findings are most directly applicable to malware that manifests in user-space process and memory artifacts, whereas attacks focusing on persistence mechanisms or purely in-kernel hooks might evade detection in our current setup. This underscores the fact that our results cannot prompt a blanket statement that "LLMs catch all malware in memory" but rather they catch what the chosen plugins catch. Expanding artifact coverage is a necessary next step (as addressed in Future Research, Section 8.2).

While we evaluated a total of seven malware scenarios (plus one clean scenario) including a mix of RATs, stealers, trojans, injections, and ransomware, the sample size is still relatively small. The malware chosen included known or synthesized examples, which might not cover the full diversity of real-world attacks. For instance, we did not test advanced kernel rootkits or a wide array of fileless techniques beyond what was included. Moreover, each scenario was represented by one memory image (or a small number of runs), which could bias results. For instance, perhaps our particular Quasar RAT sample was "easy" to detect, or potentially a different ransomware might have shown some IoCs that LockBit did not. Therefore, one limitation concerns generalizability as the results might differ with other malware or on systems with different configurations. While we somewhat mitigated this by using multiple runs and LLMs, a broader evaluation would strengthen confidence in our findings.

The LLM models themselves are a moving target, and we used specific versions available up to early 2025 (e.g., OpenAI GPT-4o, xAI's Grok 3, etc.). These models will likely improve, and new ones will emerge. Similarly, prompting strategies can evolve. Conversely, some LLM changes (like stricter safety filters) could potentially suppress valid analysis. The dependency on third-party LLM APIs (for OpenAI and possibly Gemini) is a limitation as those models are not fully under our control and might change. If an API changes output format or a model is updated, our system might require adjustments. In summary, our results are a snapshot in time with particular LLM instances, whereby any future adoption should be re-validated with the then-current models.

Although the LLMs often obtained sensible data, uploading sensitive memory data (which may contain personal or confidential information) to a third-party AI service can be problematic or against policy. While we did not explore this in depth in this study, it reflects a limitation for real-world use as organizations might require on-premise or open-source models for analysis to avoid data exfiltration concerns. The deployment might be limited by infrastructure or cost until smaller or more efficient models are viable.

While standard, our use of accuracy, precision, recall, and F1 scores might fail to capture the full picture of an AI assistant's value. For instance, all FPs were counted equally, although in practice some are trivial to dismiss while others could be confusing. Likewise, our ground truth labeling of what constitutes a correct vs. incorrect flag was binary, meaning that we did not give partial credit for an LLM that identified the correct

malicious process but called it by an incorrect malware name. One could argue that identifying the malicious process is the crucial part in an investigation context, whereas the misnaming might be a lesser issue. Our quantitative approach might thus underrate models that gave essentially correct outcomes with minor mistakes, reflecting a limitation of our evaluation method. Future research could seek to employ more nuanced scoring, perhaps considering the relevance of each output line.

In summary, the study is constrained by FPs, the data scope, testbed size, evolving AI technology, and practical cost and privacy considerations. These limitations restrict the findings and emphasize that MemoryInvestigator is a prototype rather than a plug-and-play solution. We have flagged many of these issues throughout the thesis (and here) to ensure a clear-eyed view of what the results mean. Importantly, none of these limitations are insurmountable as each provides direction for improvements, which we will outline in the next section and Chapter 8. Recognizing these shortcomings is crucial for anyone attempting to build upon or deploy this approach in the field.

7.4 Recommendations for Practitioners

Due to our findings and the limitations discussed, we propose several practical recommendations for digital forensic practitioners and incident responders who wish to leverage LLMs for memory analysis.

First, the LLM-generated report should be treated as an initial hypothesis or triage result, rather than acting on LLM findings without verification. If the AI flags a process as malicious, it should be cross-checked using traditional means (e.g. examine the process memory manually, check it against threat intelligence, etc.). While LLMs can dramatically narrow down where to look, a human should confirm before drawing conclusions in an investigation. A baseline system should be used to reduce the number of FPs, like [29]. This can help the system to distinguish between normal anomalies and truly suspicious ones as having more relevant data improves the LLM's chance of catching sophisticated threats. Moreover, multiple data sources should be incorporated to avoid blind spots, as seen in the *LockBit* case. It is recommended to include registry hives, event logs, and other memory-derived artifacts (such as atoms, driverscan, etc.) as needed. If a critical artifact (e.g., signs of ransomware activity in memory) is available via another forensic tool, that output should be included accordingly.

Second, LLMs that encourage reasoning ("thinking" mode) should be preferred as our findings suggest that this improves the results. Moreover, it is recommended to use a predefined system message that instructs the LLM to explain why it flags something and be methodical to improve detection and obtain a clear rationale for each alert, which is useful for documentation and confidence. If the LLM identifies a specific malware family or describes how an attack happened, those details should be verified in the artifacts. For instance, the AI might name a malware based on similarity when in fact it is a different strain, or it might invent a plausible story. It is important to always corroborate AI-generated intel with external sources (check if the evidence truly matches known indicators). Accordingly, the AI's narrative should be used as a lead rather than proof.

Third, it is recommended to consider the sensitivity of memory dumps given that they might contain usernames, messages, or other confidential data from the system. If using cloud-based LLM APIs, it is essential to ensure compliance with data-handling policies. Where possible, non-essential sensitive information should be scrubbed or redacted from the data before sending it to an API. For highly sensitive investigations (e.g., classified or proprietary environments), it is preferred to use an on-premises LLM solution to avoid transmitting data externally. This might involve running an open-source model locally or within a secure enclave. Given that large memory dumps can exceed an LLM's context window size, the data given to the model should be selected and pre-filtered to include only relevant sections (for instance, processes and modules of interest, rather than every single process). If necessary, the analysis should be broken into chunks (e.g., analyze processes in batches) to fit the context window. Practitioners should be deliberate in what data they feed into the model to achieve optimal results.

For an effective defense-in-depth, LLM analysis should be used alongside conventional scanning (antivirus, YARA rules, IoC searches). If the LLM flags something, it is recommended to run a hash check or virus scan on that process. Conversely, if a traditional tool finds malware, it is suggested to observe whether the LLM provides additional context or missed related activity. The approaches can complement each other as the LLM might catch odd behaviors that signatures miss, while signatures can confirm whether a flagged file is a known malware. Practitioners should not hesitate to interact with the LLM beyond the initial query. Many LLM interfaces allow follow-up questions, whereby a practitioner can ask, "Why did you flag process X specifically?" or "What would make you confident that process Y is benign?". This can extract more insight and potentially clarify which indicators the model based its decision on. The LLM should be treated as an analyst assistant with whom you can have a conversation about the evidence. For accountability purposes, it is suggested to maintain a record of what the LLM reports. If the investigation becomes part of a legal case, having a log of the AI's suggestions and the analyst's verification steps will be important. Such a report should note that an AI tool was used for initial analysis (if relevant to procedure) and record any confirmed findings that originated from it. This not only provides transparency but also helps improve the tool over time by assessing which of the AI's FPs commonly occur and feeding that back into refining prompts or filters.

By following these recommendations, practitioners can maximize the benefits of LLM-based memory analysis while mitigating the risks and limitations. Table 7.1 summarizes our best-practice suggestions for using LLMs in memory forensics.

Practice Area	Recommendation
Role of LLM	Use LLMs as a triage assistant, not a final decision-maker. Always verify flagged items manually.
Baseline	Build a baseline out known benign behavoir (e.g., known clean processes) to reduce false positives.
Artifact Scope	Include diverse memory artifacts (registry, event logs, strings, etc.) to improve detection coverage.
Model Selection	Use reasoning-based LLMs to improve detection accuracy and interpretability.
Verification	Validate all AI-generated claims, especially specific malicious processes before acting.
Data Privacy	Avoid uploading sensitive memory data to external APIs. Use local models for high-security cases.
Input Size Management	Pre-filter and batch input data to respect LLM context window limits while retaining relevance.
Tool Integration	Combine LLM analysis with traditional forensic tools (YARA, antivirus, hashes) for confirmation.
Iterative Use	Use interactive prompts to follow up on LLM hypotheses (e.g., "Why is this process suspicious?").
Documentation	Log and archive LLM outputs and your verification steps for transparency and auditability.

Table 7.1: *Best-Practice Tips for Using LLMs in Memory Forensics*

8 Conclusion and Future Work

8.1 Summary of Contributions

By exploring the integration of LLMs into memory forensics, this work makes several theoretical and practical contributions.

Our MemoryInvestigator automates memory forensic queries and leverages LLMs to analyze the results, providing a novel pipeline that feeds structured Volatility3 output to an AI model with a prompt. This system demonstrates a new way of conducting memory analysis through interactive, natural language summaries and explanations, rather than manually parsing tool output. Our tool thus reflects a concrete contribution that others can build upon for AI-assisted forensic analysis.

To our knowledge, this is the first systematic comparative analysis of multiple state-of-the-art LLMs (*OpenAI GPT4o*, *OpenAI o1*, *Gemini 2.0 Flash*, *Gemini 2.0 Flash-Thinking*, *Grok 3*, *and Grok 3 with thinking mode enabled*) on the problem of malware detection in memory images. Using a test dataset spanning seven distinct attack scenarios (including own created attacks and real-world malware samples) and measuring detection performance using accuracy, precision, recall, and F1 scores, this study provides quantitative evidence that reasoning-enabled LLMs can achieve high recall in detecting in-memory threats, outperforming standard LLMs. Moreover, the issue of high FPs rates emerged as a common finding among all tested models. These findings make a valuable contribution to the cybersecurity and AI research communities, confirming some intuitions (e.g., that reasoning improves analytic tasks) with real experimental data.

Our detailed analysis confirmed that different AI models have different strengths (e.g., one is better at decoding obfuscation, another at catching network indicators). We also identified phenomena such as the models' tendency to hallucinate additional threats and the limitations when evidence is missing. In summarizing these findings, we extracted practical guidelines for using LLMs in forensic analysis, such as providing context, verifying outputs, and using AI as a triage assistant rather than a final arbiter. These insights fill a knowledge gap on how advanced language models actually perform on low-level security tasks, as well as the limitations that practitioners must consider.

We have shown that integrating LLMs enables obtaining an intelligible forensic report in plain language from raw memory data. This conceptual contribution illustrates a path towards making complex memory analysis results readable and interpretable by those without deep expertise. The system effectively translates base64 encoding and other specific plugin outputs into sentences and hypotheses, which can lower the barrier for entry into memory forensics, suggesting a broader impact where AI tools enable junior analysts or IT generalists to benefit from deep forensic techniques.

We have highlighted specific areas for improvement by pinpointing that high FP rates stem from a lack of contextual knowledge and that certain malware (such as LockBit) eluded detection due to being out of scope. This clarity on limitations provides a roadmap for future researchers (e.g., incorporate system baseline), laying the foundation for contin-

ued research at the intersection of AI and digital forensics. Accordingly, this ensures that our work can be a stepping stone rather than a standalone curiosity.

Overall, this thesis contributes a proof-of-concept system and an evaluation study that together advance the idea of AI-assisted memory forensics. By showing both what works (LLMs can find malware in memory, and reasoning prompts help) and what remains difficult (the need to manage FPs and data scope), our findings and prototype developed collectively demonstrate the practical viability of integrating LLM intelligence into forensic workflows and provide a basis for both practitioners and researchers to build on.

8.2 Further Research

Our work opens several avenues for future research and development. Future studies should focus on expanding the range of forensic artifacts supplied to the LLM, extending beyond Volatility3's core plugins to include data such as Windows registry hives, event logs (EVTX), raw memory strings, kernel driver lists, and others. Feeding these into the analysis (in a structured way) will enable capturing IoCs that our current system misses. For example, incorporating registry and file system artifacts might enable detecting persistence mechanisms or ransomware encryption keys that were invisible to our plugin-only approach. The challenge here is handling the volume and relevance of data, necessitating research on how to smartly pre-filter or prioritize artifacts so that important signals (e.g., suspicious Run key entries or anomalous kernel hooks) are brought to the LLM's attention without overflowing it with extraneous details.

Our findings on complementary model strengths hint at combining models. Future work can explore ensemble techniques, where multiple LLMs (or multiple prompts to the same LLM) analyze the memory data and their results are aggregated. For example, one model might focus on networking artifacts while another focuses on process injection, with a meta-algorithm reconciling their findings. Similarly, combining the LLM approach with traditional rule-based detection could be fruitful. Research could investigate frameworks that take LLM output and run it through a post-processing filter of known effective indicators (or a whitelist of processes) to automatically discard likely FPs. This hybrid approach could leverage AI's broad pattern recognition with the reliability of deterministic rules or signatures. Designing an effective orchestration of these components is a research challenge in itself, possibly requiring new algorithms to weigh the confidence of AI vs. rule-based outputs.

Another suggested future direction involves making the AI analysis process iterative. In our current design, the LLM sees the data once and outputs a report. We envision a system where the LLM can ask questions or request more data if it is uncertain, effectively representing a dialogue between the AI agent and the forensic tool. For instance, the AI might respond, "I see signs of malware X, but I'm not fully sure; can I see handles for process Y or a dump of that process's memory?" The system could then fetch that additional data (via Volatility or another tool) and feed it back in for a refined analysis. Implementing this requires exploring how to detect the AI's uncertainties and map them

to actionable data queries. It touches on active learning and could strongly enhance the detection of more subtle threats by not limiting the AI to a one-shot analysis.

While we focused on one memory image at a time, future studies might explore scenarios with multiple memory snapshots (e.g., from multiple machines in an enterprise or from the same machine over time). An AI could correlate information across these: for example, noticing that the same strange process appears on ten machines, which strengthens confidence that it is malicious, or observing that a process started injecting code only after a certain time stamp. Research here would involve feeding the LLM aggregated or comparative data and devising prompts to let it perform a differential analysis. This could extend MemoryInvestigator into an enterprise threat-hunting tool that finds patterns among many endpoints rather than only one.

While we applied LLMs to memory forensics, a natural progression is to explore similar integrations in disk forensics or network forensics. Future research could test whether an LLM can analyze file system metadata or timeline artifacts to identify traces of malware on disk, or read packet captures to flag malicious traffic. The structure and approach might be analogous, namely converting data to text and prompting the model accordingly. However, each domain has its nuances, and research would need to tailor the prompts and evaluate performance in those contexts. Given that success in memory forensics suggests LLMs might be broadly useful across digital forensics (e.g., helping parse log files, identifying suspicious user activity sequences, etc.), a logical next step would involve expanding the investigation to those areas.

Another forward-looking research topic is the robustness of LLM analysis against adversarial manipulation. If attackers are aware that an AI is analyzing memory, could they craft memory artifacts to fool the AI (like adversarial examples)? For instance, they might deliberately include misleading strings or patterns to trigger FPs or conceal real indicators. Future work should test LLMs against such scenarios and develop defenses, perhaps secondary validation checks or adversarial training of the model to resist being misled. If this approach is ever widely deployed, it will be crucial to ensure that the AI's analysis is reliable in the face of clever malware attempts to evade or poison it.

In summary, the outlook is rich with possibilities and this work's encouraging results can act as a springboard, outlining a clear trajectory including addressing precision issues, broadening scope, and deepening the intelligence of the system. Pursuing these will not only enhance MemoryInvestigator but also contribute to the emerging field of AI-driven digital forensics, potentially leading to more autonomous and effective investigative tools.

8.3 Final Thoughts

In conclusion, this work represents a step towards a new paradigm in digital forensics where AI assists and amplifies human investigators. We demonstrated have that LLMs—when properly guided—can sift through low-level memory data and produce meaningful, actionable insights, which have traditionally required a skilled human analyst. The promising results with high detection rates and rich explanatory outputs achieved with a tool that can converse in natural language indicate that AI has a genuine role to play

in augmenting forensic analysis with currently available technology. However, our work also highlight the challenges associated with this integration. For instance, the AI must therefore be used thoughtfully given that memory forensics is unforgiving in terms of accuracy, as an investigation cannot tolerate many mistakes or uncertainties. FPs and incomplete coverage remind us that these models—powerful as they are—do not possess true understanding or judgment, thus representing assistants that need supervision. The takeaway idea is one of human-AI collaboration, combining the speed and scale of machine analysis with the intuition and validation of human experts. Striking this balance will enable more efficient investigations that do not sacrifice reliability. The implications of this work extend beyond simply catching malware, highlighting how to make complex cybersecurity tasks more accessible. One could imagine future incident response platforms where an analyst asks, "What happened on this machine?" and an AI agent like MemoryInvestigator narrates the story, including which processes did what, how an intrusion unfolded, and which evidence is most critical, all backed by raw data that the analyst can verify. This kind of narrative, reasoned reporting could transform cybersecurity operations, training, and perhaps even how reports are presented in court, with AI providing a first draft of the explanation). On a broader scale, the successful blending of AI with forensics is a microcosm of AI's potential in various fields that require specialized expertise. It hints that with careful alignment, AI can serve as a force multiplier for experts rather than a replacement. In an industry facing a talent shortage (cybersecurity), such tools could help to bridge the gap by enabling each analyst to handle more cases or helping new analysts to climb the learning curve faster. There are also ethical and professional considerations as we move forward, whereby ensuring the transparency of AI decisions, maintaining the chain of custody and evidence integrity, and avoiding over-reliance on AI in critical judgments will all be important. The forensic community might need to develop guidelines or standards for using AI in investigations, such as how to document AI assistance in reports, or how to validate AI findings independently. Our project contributes to starting this conversation by providing an early case study of what AI can and cannot do in this context. In closing, optimizing memory forensics through automation and AI is part of the continual evolution of digital investigations. Each innovation—from early memory dump analyzers to modern frameworks such as Volatility—has expanded what investigators can achieve. LLMs represent the next innovation, opening the door to intelligent automation that learns from vast knowledge and can reason about evidence. While the work here is a proof of concept, it has shown sufficient potential to warrant optimism. With further refinement, it can become a practical tool and ultimately a standard component in every incident responder's toolkit. The intersection of AI and cybersecurity remains in its early days, and we have much to learn. This thesis has hopefully contributed a piece to that puzzle, demonstrating successes, identifying pitfalls, and inspiring future exploration. By embracing these advanced tools while also respecting the fundamentals of forensic science, we can move towards a future where investigations are faster, more accessible, and perhaps even more insightful than ever before. While the ultimate goal of uncovering the truth hidden in our digital systems remains, intelligent machines have emerged as new allies to help us in this quest.

List of Figures

1.1 Design Science Research Process [68] 4

4.1 MemoryInvestigator C4 Diagram 24

5.1 Migration to Another Process 27
5.2 PowerShell Empire Graphical Front-End: Starkiller 29
5.3 QuasarRAT Behavior Analysis [78] 33

6.1 Average Precision Across Experiments: (a) Precision Per LLM Configuration; (b) Precision Per Virus Scenario 36
6.2 Average F1 Score Across Experiments: (a) F1 Score Per LLM Configuration; (b) F1 Score Per Virus Scenario 37
6.3 Base64 Encoding from Gemini 2.0 Flash-Thinking 39
6.4 Exhibition of Network Connections from Grok3 40
6.5 QuasarRAT windows.psscan Output with the Suspicious Indicators Marked in Yellow and Red 41
6.6 QuasarRAT windows.psscan Output with Different Obvious Parts 42
6.7 Client.exe VirusTotal Check [95] 44
6.8 MsMpEng in windows.malfind Output 46
6.9 Unknown Characters in windows.psscan Output 47

B.1 MemoryInvestigator Screenshot: Data Input 77
B.2 MemoryInvestigator Screenshot: Display Data as a Table 77
B.3 MemoryInvestigator Screenshot: Display Data as a Graph 78
B.4 MemoryInvestigator Screenshot: Search Data in the Graph 78
B.5 MemoryInvestigator Screenshot: Extract Data 79
B.6 MemoryInvestigator Screenshot: Tree-of-Table Analysis 79
B.7 MemoryInvestigator Screenshot: Tree-of-Table Analysis 80
B.8 MemoryInvestigator Screenshot: Experimental Forensic RAG 80
B.9 MemoryInvestigator Screenshot: Temporary Folder Structure 81

C.1 Tree-of-Table Algorithm Graphic 84

E.1 Malware Behavior: MassLogger [77] 87
E.2 Malware Behavior: DarkCloud [76] 88
E.3 Malware Behavior: LockiBot [48] 88
E.4 Malware Behavior: LockBit [47] 89

G.1 MemoryInvestigator Screenshot: QuasarRAT GPT4o 101
G.2 MemoryInvestigator Screenshot: QuasarRAT Gemini 1.5 Pro 102

List of Listings

4.1 Self-Contained Workspace 17
4.2 Regex Commands for Data Cleaning 21

5.1 Building Reverse Shell with msfvenom 26
5.2 Injection in a Process 27
5.3 Start the C2 Empire Server 28
5.4 Run the PowerShell Empire Client 28
5.5 Base64 obfuscated Script execution on the victims client 28
5.6 De-Obfuscated Base64 Script 28
5.7 Interaction with the Victims Client 29

6.1 Disassembly of Region 1 (0xfe0000) 43
6.2 Disassembly of Region 2 (0x1b560000) 43
6.3 Disassembly of Region 3 (0x7ff40de70000) 43
6.4 Disassembly of Region 4 (0x7ff40de80000) 43

A.1 MemoryInvestigator Project Structure 75
A.2 MemoryInvestigator Installation Guide 76
A.3 Additional Linux Steps 76

C.1 Mapped Fields to Volatility3 Output 82
C.2 Pseudocode Tree-of-Table Builder 82

D.1 Basic-build Tree 85
D.2 Custom-build Tree 86

G.1 Volatility3 windows.dlllist Quasar RAT Output 92
G.2 Volatility3 windows.netscan Quasar RAT Output 96

List of Tables

4.1 Functional and Non-Functional Requirements 15
4.2 Architecture Layers of MemoryInvestigator 16
4.3 Overview of the User Interface Frontend Pages 18

5.1 Overview of Malware Samples Used 26
5.2 Memory-Forensics-Oriented Comparison of the Real-World Malware Samples 32

6.1 Overview of Results Illustrated with LLMs 35
6.2 Overview of Results Illustrated with Viruses 36

7.1 Best-Practice Tips for Using LLMs in Memory Forensics 55

A.1 Third-party Dependencies 74

F.1 Results overview illustrated using the LLM and Virus 90

F.2 Summary of statistical tests comparing the type of thinking and viruses in performance metrics. Significant p-values ($p < 0.05$) are marked with an asterisk (*) 91
F.3 Tukey HSD test results for LLM (F1 score comparisons) Significant p-values ($p < 0.05$) are marked with an asterisk (*) 91

Bibliography

[1] abuse.ch. *DarkCloud Sample - abuse.ch MalwareBazaar*. URL: https://bazaar.abuse.ch/sample/79f7e76ba3a8780a2dd46d78aac07f2235f457ee2771ef4556b85c12f2be0061/ (visited on 05/14/2025).

[2] abuse.ch. *LockBit 3.0 Sample*. Feb. 14, 2025. URL: https://bazaar.abuse.ch/sample/9733092223c428fc0e44a90b01c7f77a97bb1205def8be1224ac68969182638e/ (visited on 05/14/2025).

[3] abuse.ch. *LokiBot Sample Analysis*. Sept. 16, 2024. URL: https://bazaar.abuse.ch/sample/1eb1ecc1ff01e9f4934a79daa57db69c96f975b90d649429417c560c1d36bab1/ (visited on 05/11/2025).

[4] abuse.ch. *MalwareBazaar Sample QuasarRAT*. Feb. 19, 2025. URL: https://bazaar.abuse.ch/sample/edcd9db80fa6bb9d601b9827fd5e745b8b0c6057a34b02584ded808f2ede0a48/ (visited on 05/14/2025).

[5] abuse.ch. *MassLogger Sample*. URL: https://bazaar.abuse.ch/sample/fbaec035008b4d3722c9b832c534d85660e7c80027a29d1d8310b77b2ad54fc7/ (visited on 05/14/2025).

[6] abuse.ch and Spamhaus. *MalwareBazaar: Malware Sample Exchange Platform*. URL: https://bazaar.abuse.ch/ (visited on 05/14/2025).

[7] AhnLab Security Emergency-response Center (ASEC). *DarkCloud Malware Information - AhnLab ASEC Blog*. May 17, 2023. URL: https://asec.ahnlab.com/en/53128/ (visited on 05/14/2025).

[8] M. ATT&CK. *QuasarRAT, Software S0262*. MITRE. May 7, 2024. URL: https://attack.mitre.org/software/S0262/ (visited on 05/14/2025).

[9] M. Bhatt, S. Chennabasappa, C. Nikolaidis, S. Wan, I. Evtimov, et al. *Purple Llama CyberSecEval: A Secure Coding Benchmark for Language Models*. Dec. 2023. DOI: 10.48550/arXiv.2312.04724. URL: http://arxiv.org/abs/2312.04724 (visited on 05/14/2025).

[10] BleepingComputer. *LockBit Ransomware Self-Spreads to Quickly Encrypt 225 Systems*. May 4, 2020. URL: https://www.bleepingcomputer.com/news/security/lockbit-ransomware-self-spreads-to-quickly-encrypt-225-systems/ (visited on 05/14/2025).

[11] CERT-AGID. *MassLogger: An analysis by CERT-AGID*. June 9, 2020. URL: https://cert-agid.gov.it/wp-content/uploads/2020/06/CERT-AGID_MassLogger-20200609.pdf (visited on 05/14/2025).

[12] M. Chernyshev, Z. Baig, and R. R. M. Doss. "Towards Large Language Model (LLM) Forensics Using LLM-based Invocation Log Analysis". en. In: *Proceedings of the 1st ACM Workshop on Large AI Systems and Models with Privacy and Safety Analysis*. Salt Lake City UT USA: ACM, Nov. 2023. ISBN: 9798400712098. DOI: 10.1145/3689217.3690616. URL: https://dl.acm.org/doi/10.1145/3689217.3690616 (visited on 05/14/2025).

[13] CISA. *AA20-266A: LokiBot Malware Advisory*. Sept. 24, 2020. URL: https://www.cisa.gov/news-events/cybersecurity-advisories/aa20-266a (visited on 05/14/2025).

[14] CISA. *Stop Ransomware: LockBit*. Mar. 16, 2023. URL: https://www.cisa.gov/sites/default/files/2023-03/aa23-075a-stop-ransomware-lockbit.pdf (visited on 05/14/2025).

[15] Cisco Talos. *A year of FAJAN: Evolution and Bloomberg*. Apr. 21, 2021. URL: https://blog.talosintelligence.com/a-year-of-fajan-evolution-and-bloomberg/ (visited on 05/14/2025).

[16] Cisco Talos. *Masslogger Credential Exfiltration Campaigns*. Feb. 17, 2021. URL: https://blog.talosintelligence.com/masslogger-cred-exfil/ (visited on 05/14/2025).

[17] V. Clairoux-Trepanier, I.-M. Beauchamp, E. Ruellan, M. Paquet-Clouston, S.-O. Paquette, et al. *The Use of Large Language Models (LLM) for Cyber Threat Intelligence (CTI) in Cybercrime Forums*. en. Oct. 2024. DOI: 10.48550/arXiv.2408.03354. URL: http://arxiv.org/abs/2408.03354 (visited on 05/14/2025).

[18] Criticalstart. *DarkCloud Malware - Critical Start*. URL: https://www.criticalstart.com/darkcloud-malware/#:~:text=Attack%20Pattern%3A%20Phishing%20Emails (visited on 05/14/2025).

[19] Cx01N. *An introduction to starkiller*. BC Security. URL: https://bc-security.org/an-introduction-to-starkiller/ (visited on 05/14/2025).

[20] Cybereason. *Threat Analysis: Assemble - LockBit 3.0*. URL: https://www.cybereason.com/hubfs/dam/collateral/reports/Threat-Analysis-Assemble-LockBit-3.pdf (visited on 05/14/2025).

[21] Cybersecurity and I. S. Agency. *Quasar Open-Source Remote Administration Tool*. Feb. 14, 2019. URL: https://www.cisa.gov/news-events/analysis-reports/ar18-352a (visited on 05/14/2025).

[22] X. Du, M. Wen, J. Zhu, Z. Xie, B. Ji, et al. "Generalization-Enhanced Code Vulnerability Detection via Multi-Task Instruction Fine-Tuning". In: *Findings of the Association for Computational Linguistics: ACL 2024*. Ed. by L.-W. Ku, A. Martins, and V. Srikumar. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024. DOI: 10.18653/v1/2024.findings-acl.625. URL: https://aclanthology.org/2024.findings-acl.625/ (visited on 05/14/2025).

[23] R. Fayyazi, R. Taghdimi, and S. J. Yang. *Advancing TTP Analysis: Harnessing the Power of Large Language Models with Retrieval Augmented Generation*. July 2024. DOI: 10.48550/arXiv.2401.00280. URL: http://arxiv.org/abs/2401.00280 (visited on 05/14/2025).

[24] F. FKIE. *DarkCloud (Malware Family)*. URL: https://malpedia.caad.fkie.fraunhofer.de/details/win.darkcloud (visited on 05/14/2025).

[25] F. FKIE. *Quasar RAT (Malware Family)*. URL: https://malpedia.caad.fkie.fraunhofer.de/details/win.quasar_rat (visited on 05/14/2025).

[26] Fortinet. *LokiBot Targets Microsoft Office Document Using Vulnerabilities and Macros*. July 12, 2023. URL: https://www.fortinet.com/blog/threat-research/lokibot-targets-microsoft-office-document-using-vulnerabilities-and-macros (visited on 05/14/2025).

[27] V. Foundation. *Volatility 3: The Volatile Memory Extraction Framework*. URL: https://github.com/volatilityfoundation/volatility3 (visited on 05/14/2025).

[28] Fr3d.hk. *MassLogger: Frankenstein's Creation*. June 10, 2020. URL: https://fr3d.hk/blog/masslogger-frankenstein-s-creation (visited on 05/14/2025).

[29] G. Fraser. "Creating a Baseline of Process Activity for Memory Forensics". en. In: *SANS* (2021).

[30] Fraunhofer FKIE. *LokiBot (Malware Family)*. URL: https://malpedia.caad.fkie.fraunhofer.de/details/win.lokipws (visited on 05/14/2025).

[31] Fraunhofer FKIE. *MassLogger (Malware Family)*. URL: https://malpedia.caad.fkie.fraunhofer.de/details/win.masslogger (visited on 05/14/2025).

[32] U. Frisk, B. Foster, K. .-. S. 4N6, MattCore, skelsec, et al. *ufrisk/MemProcFS*. URL: https://github.com/ufrisk/MemProcFS (visited on 05/14/2025).

[33] Z. Gao, H. Wang, Y. Zhou, W. Zhu, and C. Zhang. *How Far Have We Gone in Vulnerability Detection Using Large Language Models*. Dec. 2023. DOI: 10.48550/arXiv.2311.12420. URL: http://arxiv.org/abs/2311.12420 (visited on 05/14/2025).

[34] A. M. M. Gómez, N. Huynh, MalwareMechanic, P. Kacherginsky, E. Chernofsky, et al. *Flare-vm*. URL: https://github.com/mandiant/flare-vm (visited on 05/14/2025).

[35] Google Cloud Threat Intelligence. *Bypassing Masslogger Anti-Analysis: Man-in-the-Middle Approach*. Aug. 6, 2020. URL: https://cloud.google.com/blog/topics/threat-intelligence/bypassing-masslogger-anti-analysis-man-in-the-middle-approach?hl=en (visited on 05/14/2025).

[36] Halcyon. *RaaS MQ Power Rankings: Ransomware Malicious Quartile Q2 2024*. Aug. 1, 2024. URL: https://www.halcyon.ai/raas-mq/power-rankings-ransomware-malicious-quartile-q2-2024 (visited on 05/14/2025).

[37] I. Hamid and M. M. H. Rahman. "A Comprehensive Literature Review on Volatile Memory Forensics". en. In: *Electronics* 15 (July 2024). ISSN: 2079-9292. DOI: 10.3390/electronics13153026. (Visited on 05/14/2025).

[38] X. Han, S. Yuan, and M. Trabelsi. *LogGPT: Log Anomaly Detection via GPT*. en. Dec. 2023. DOI: 10.48550/arXiv.2309.14482. URL: http://arxiv.org/abs/2309.14482 (visited on 05/14/2025).

[39] R. Hazra, G. Venturato, P. Z. D. Martires, and L. D. Raedt. *Can Large Language Models Reason? A Characterization via 3-SAT*. en. Oct. 2024. DOI: 10.48550/arXiv.2408.07215. URL: http://arxiv.org/abs/2408.07215 (visited on 04/08/2025).

[40] Hive Pro. *DarkCloud Stealer: A Multi-Stage Malware That Pilfers Sensitive Data*. Feb. 22, 2023. URL: https://www.hivepro.com/wp-content/uploads/2023/02/DarkCloud-Stealer-A-Multi-Stage-Malware-That-Pilfers-Sensitive-data_TA2023095.pdf (visited on 05/14/2025).

[41] A. Ho. *MassLogger v3: a .NET stealer with serious obfuscation*. Feb. 22, 2021. URL: https://decoded.avast.io/anhho/masslogger-v3-a-net-stealer-with-serious-obfuscation/ (visited on 05/14/2025).

[42] C. Hu. *An Introduction to Software Design: Concepts, Principles, Methodologies, and Techniques*. en. Cham: Springer International Publishing, 2023. ISBN: 978-3-031-28310-9 978-3-031-28311-6. DOI: 10.1007/978-3-031-28311-6. (Visited on 05/14/2025).

[43] Y. Hu, F. Zou, J. Han, X. Sun, and Y. Wang. "LLM-TIKG: Threat intelligence knowledge graph construction utilizing large language model". In: *Computers & Security* (Oct. 2024). ISSN: 0167-4048. DOI: 10.1016/j.cose.2024.103999. (Visited on 05/14/2025).

[44] F. iSIGHT Intelligence. *APT10 (MenuPass Group): New Tools, Global Campaign Latest Manifestation of Longstanding Threat*. Google Cloud Blog. Apr. 6, 2017. URL: https://cloud.google.com/blog/topics/threat-intelligence/apt10-menupass-group (visited on 05/14/2025).

[45] D. Iuzvyk and T. Peck. *Analyzing OBSCURE#BAT: Threat Actors Lure Victims into Executing Malicious Batch Scripts to Deploy Stealthy Rootkits*. Securonix Threat Research. Mar. 13, 2025. URL: https://www.securonix.com/blog/analyzing-obscurebat-threat-actors-lure-victims-into-executing-malicious-batch-scripts-to-deploy-stealthy-rootkits (visited on 05/14/2025).

[46] D. Ji, L. Zhu, S. Gao, P. Xu, H. Lu, et al. *Tree-of-Table: Unleashing the Power of LLMs for Enhanced Large-Scale Table Understanding*. en. Nov. 2024. DOI: 10.48550/arXiv.2411.08516. URL: http://arxiv.org/abs/2411.08516 (visited on 05/14/2025).

[47] Joe Sandbox. *Joe Sandbox Analysis Report: LockBit 3.0 Sample*. URL: https://www.joesandbox.com/analysis/1614920 (visited on 05/14/2025).

[48] Joe Sandbox. *LokiBot Analysis Report - Analysis ID 1512075*. Jan. 1, 2024. URL: https://www.joesandbox.com/analysis/1512075 (visited on 05/11/2025).

[49] E. Karlsen, X. Luo, N. Zincir-Heywood, and M. Heywood. *Benchmarking Large Language Models for Log Analysis, Security, and Interpretation*. en. Nov. 2023. DOI: 10.48550/arXiv.2311.14519. URL: http://arxiv.org/abs/2311.14519 (visited on 04/11/2025).

[50] M. H. Ligh, A. Case, J. Levy, and A. Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. en. Cham: Wiley, 2014. ISBN: 978-1-118-82509-9.

[51] J. Lin and D. Mohaisen. *Evaluating Large Language Models in Vulnerability Detection Under Variable Context Windows*. en. Jan. 2025. DOI: 10.48550/arXiv.2502.00064. URL: http://arxiv.org/abs/2502.00064 (visited on 05/14/2025).

[52] Y. Liu, S. Tao, W. Meng, J. Wang, W. Ma, et al. "Interpretable Online Log Analysis Using Large Language Models with Prompt Strategies". en. In: *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. Lisbon Portugal: ACM, Apr. 2024. ISBN: 9798400705861. DOI: 10.1145/3643916.3644408. URL: https://dl.acm.org/doi/10.1145/3643916.3644408 (visited on 05/14/2025).

[53] J. S. LLC. *Deep Malware Analysis*. joesecurity. URL: https://www.joesecurity.org/ (visited on 05/14/2025).

[54] Logpoint. *Hiding in Plain Sight: The Subtle Art of Loki Malware's Obfuscation*. Nov. 7, 2024. URL: https://www.logpoint.com/en/blog/emerging-threats/hiding-in-plain-sight-the-subtle-art-of-loki-malwares-obfuscation/ (visited on 05/14/2025).

[55] M. C. Long II. *Disrupting the Empire: Identifying PowerShell Empire Command and Control Activity*. 2018. URL: https://www.sans.org/white-papers/38315/ (visited on 05/14/2025).

[56] H. Lu, H. Peng, G. Nan, J. Cui, C. Wang, et al. *MALSIGHT: Exploring Malicious Source Code and Benign Pseudocode for Iterative Binary Malware Summarization*. Nov. 2024. DOI: 10.48550/arXiv.2406.18379. URL: http://arxiv.org/abs/2406.18379 (visited on 05/14/2025).

[57] M. Magnezi. *LokiBot Malware Analysis*. Dec. 1, 2024. URL: https://0xmrmagnezi.github.io/malware%20analysis/LokiBot/ (visited on 05/14/2025).

[58] MaxXor. *Quasar: Remote Administration Tool for Windows*. GitHub. URL: https://github.com/quasar/Quasar (visited on 05/14/2025).

[59] X. Mertens. *Quasar RAT Delivered Through Updated SharpLoader*. SANS Internet Storm Center. Nov. 18, 2023. URL: https://isc.sans.edu/diary/30414 (visited on 05/14/2025).

[60] G. Michelet and F. Breitinger. "ChatGPT, Llama, can you write my report? An experiment on assisted digital forensics reports written using (local) large language models". In: *Forensic Science International: Digital Investigation*. DFRWS EU 2024 - Selected Papers from the 11th Annual Digital Forensics Research Conference Europe (Mar. 2024). ISSN: 2666-2817. DOI: 10.1016/j.fsidi.2023.301683. (Visited on 05/14/2025).

[61] S. Mitra, S. Neupane, T. Chakraborty, S. Mittal, A. Piplai, et al. *LOCALINTEL: Generating Organizational Threat Intelligence from Global and Local Cyber Knowledge*. en. Feb. 2025. DOI: 10.48550/arXiv.2401.10036. URL: http://arxiv.org/abs/2401.10036 (visited on 05/14/2025).

[62] D. C. Montgomery. *Design and Analysis of Experiments*. Hoboken, NJ: John Wiley & Sons, Inc, 2017.

[63] Offsec. *MSFvenom*. URL: https://www.offsec.com/metasploit-unleashed/msfvenom/ (visited on 05/14/2025).

[64] D. B. Oh, D. Kim, D. Kim, and H. K. Kim. "volGPT: Evaluation on triaging ransomware process in memory forensics with Large Language Model". en. In: *Forensic Science International: Digital Investigation* (July 2024). ISSN: 26662817. DOI: 10.1016/j.fsidi.2024.301756. (Visited on 05/14/2025).

[65] J. Ottmann, F. Breitinger, and F. Freiling. "An Experimental Assessment of Inconsistencies in Memory Forensics". en. In: *ACM Transactions on Privacy and Security* 1 (Feb. 2024). ISSN: 2471-2566, 2471-2574. DOI: 10.1145/3628600. (Visited on 05/14/2025).

[66] D. N. Palacio, A. Velasco, D. Rodriguez-Cardenas, K. Moran, and D. Poshyvanyk. *Evaluating and Explaining Large Language Models for Code Using Syntactic Structures*. Aug. 2023. DOI: 10.48550/arXiv.2308.03873. URL: http://arxiv.org/abs/2308.03873 (visited on 05/14/2025).

[67] H. Pearce, B. Tan, P. Krishnamurthy, F. Khorrami, R. Karri, et al. *Pop Quiz! Can a Large Language Model Help With Reverse Engineering?* Feb. 2022. DOI: 10.48550/arXiv.2202.01142. URL: http://arxiv.org/abs/2202.01142 (visited on 05/14/2025).

[68] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee. "A Design Science Research Methodology for Information Systems Research". en. In: *Journal of Management Information Systems* 3 (Dec. 2007). ISSN: 0742-1222, 1557-928X. DOI: 10.2753/MIS0742-1222240302. (Visited on 05/14/2025).

[69] D. Plohmann, M. Clauss, S. Enders, and E. Padilla. "Malpedia: A Collaborative Effort to Inventorize the Malware Landscape". In: *The Journal on Cybercrime & Digital Investigations* 1 (2018).

[70] R3MRUM. *Loki Bot Inside Out*. July 13, 2017. URL: https://r3mrum.wordpress.com/2017/07/13/loki-bot-inside-out/ (visited on 05/14/2025).

[71] S. Rajapaksha, R. Rani, and E. Karafili. *A RAG-Based Question-Answering Solution for Cyber-Attack Investigation and Attribution*. en. Aug. 2024. DOI: 10.48550/arXiv.2408.06272. URL: http://arxiv.org/abs/2408.06272 (visited on 05/14/2025).

[72] D. Reading. *LockBit 3.0 Variant Generates Custom Self-Propagating Malware*. Apr. 16, 2024. URL: https://www.darkreading.com/endpoint-security/lockbit-3-0-variant-generates-custom-self-propagating-malware (visited on 05/14/2025).

[73] S. Sai, U. Yashvardhan, V. Chamola, and B. Sikdar. "Generative AI for Cyber Security: Analyzing the Potential of ChatGPT, DALL-E, and Other Models for Enhancing the Security Space". en. In: *IEEE Access* (2024). ISSN: 2169-3536. DOI: 10.1109/ACCESS.2024.3385107. (Visited on 05/14/2025).

[74] S. Sai, U. Yashvardhan, V. Chamola, and B. Sikdar. "Generative AI for Cyber Security: Analyzing the Potential of ChatGPT, DALL-E, and Other Models for Enhancing the Security Space". In: *IEEE Access* (2024). DOI: 10.1109/access.2024.3385107.

[75] T. Sandapolla and K. Kathiresan. *Double Trouble: QuasarRAT's Dual DLL Sideloading in Focus*. Uptycs Threat Research. Oct. 20, 2023. URL: https://www.uptycs.com/blog/threat-research-report-team/quasar-rat (visited on 05/14/2025).

[76] J. Sandbox. *Joe Sandbox Analysis Report: DarkCloud Sample*. Feb. 26, 2025. URL: https://www.joesandbox.com/analysis/1624417 (visited on 05/14/2025).

[77] J. Sandbox. *Joe Sandbox Analysis Report: MassLogger Sample (Analysis ID 1624564)*. URL: https://www.joesandbox.com/analysis/1624564 (visited on 05/14/2025).

[78] J. Sandbox. *Joe Sandbox Analysis Report: QuasarRAT Sample*. URL: https://www.joesandbox.com/analysis/1611954 (visited on 05/14/2025).

[79] M. Sapir, T. Bar, N. Rimer, T. Malivanchuk, Y. Samuel, et al. *Downeks and Quasar RAT Used in Recent Targeted Attacks Against Governments*. Unit 42, Palo Alto Networks. Jan. 30, 2017. URL: https://unit42.paloaltonetworks.com/unit42-downeks-and-quasar-rat-used-in-recent-targeted-attacks-against-governments/ (visited on 05/14/2025).

[80] M. Scanlon. *ChatGPT for Digital Forensics*. URL: https://github.com/markscanlonucd/ChatGPT-for-Digital-Forensics (visited on 05/14/2025).

[81] M. Scanlon, F. Breitinger, C. Hargreaves, J.-N. Hilgert, and J. Sheppard. "ChatGPT for digital forensic investigation: The good, the bad, and the unknown". In: *Forensic Science International: Digital Investigation* (Oct. 2023). ISSN: 2666-2817. DOI: 10.1016/j.fsidi.2023.301609. (Visited on 05/14/2025).

[82] S. Shafee, A. Bessani, and P. M. Ferreira. "Evaluation of LLM-based chatbots for OSINT-based Cyber Threat Awareness". In: *Expert Systems with Applications* (Feb. 2025). ISSN: 0957-4174. DOI: 10.1016/j.eswa.2024.125509. (Visited on 05/14/2025).

[83] B. Sharma, J. Ghawaly, K. McCleary, A. M. Webb, and I. Baggili. "ForensicLLM: A local large language model for digital forensics". en. In: *Forensic Science International: Digital Investigation* (Mar. 2025). ISSN: 26662817. DOI: 10.1016/j.fsidi.2025.301872. (Visited on 05/14/2025).

[84] J. Shi, Y. Liu, P. Zhou, and L. Sun. *BadGPT: Exploring Security Vulnerabilities of ChatGPT via Backdoor Attacks to InstructGPT*. Feb. 2023. DOI: 10.48550/arXiv.2304.12298. URL: http://arxiv.org/abs/2304.12298 (visited on 05/14/2025).

[85] G. Siracusano, D. Sanvito, R. Gonzalez, M. Srinivasan, S. Kamatchi, et al. *Time for aCTIon: Automated Analysis of Cyber Threat Intelligence in the Wild*. July 2023. DOI: 10.48550/arXiv.2307.10214. URL: http://arxiv.org/abs/2307.10214 (visited on 05/14/2025).

[86] SOCRadar. *Dark Web Profile: LockBit 3.0 Ransomware*. Apr. 27, 2023. URL: https://socradar.io/dark-web-profile-lockbit-3-0-ransomware/ (visited on 05/14/2025).

[87] Sophos. *LockBit 3.0: Black Attacks and Leaks Reveal Wormable Capabilities and Tooling*. Nov. 30, 2022. URL: https://news.sophos.com/en-us/2022/11/30/lockbit-3-0-black-attacks-and-leaks-reveal-wormable-capabilities-and-tooling/ (visited on 05/14/2025).

[88] streamlit. *Streamlit: A faster way to build and share data apps*. streamlit. URL: https://streamlit.io (visited on 05/14/2025).

[89] G. S. Suma, S. Dija, and K. L. Thomas. "A novel methodology for windows 7 x 64 memory forensics". en. In: *2014 IEEE International Conference on Computational Intelligence and Computing Research*. Coimbatore, India: IEEE, Dec. 2014. ISBN: 978-1-4799-3974-9 978-1-4799-3975-6. DOI: 10.1109/ICCIC.2014.7238400. URL: http://ieeexplore.ieee.org/document/7238400/ (visited on 04/09/2025).

[90] H. Tan, Q. Luo, J. Li, and Y. Zhang. "LLM4Decompile: Decompiling Binary Code with Large Language Models". In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Ed. by Y. Al-Onaizan, M. Bansal, and Y.-N. Chen. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024. DOI: 10.18653/v1/2024.emnlp-main.203. URL: https://aclanthology.org/2024.emnlp-main.203/ (visited on 05/14/2025).

[91] B. C. T. R. Team. *Threat Spotlight: MenuPass/QuasarRAT Backdoor*. BlackBerry Blog. June 10, 2019. URL: https://blogs.blackberry.com/en/2019/06/threat-spotlight-menupass-quasarrat-backdoor (visited on 05/14/2025).

[92] N. Tihanyi, M. A. Ferrag, R. Jain, T. Bisztray, and M. Debbah. *CyberMetric: A Benchmark Dataset based on Retrieval-Augmented Generation for Evaluating LLMs in Cybersecurity Knowledge*. June 2024. DOI: 10.48550/arXiv.2402.07688. URL: http://arxiv.org/abs/2402.07688 (visited on 05/14/2025).

[93] C. Tilbury. *SANS MEMORY FORENSICS CHEATSHEET 3.0*. 2024. URL: https://sansorg.egnyte.com/dl/4QwNVeoj02 (visited on 05/14/2025).

[94] P. Tseng, Z. Yeh, X. Dai, and P. Liu. *Using LLMs to Automate Threat Intelligence Analysis Workflows in Security Operation Centers*. en. July 2024. DOI: 10.48550/arXiv.2407.13093. URL: http://arxiv.org/abs/2407.13093 (visited on 05/14/2025).

[95] VirusTotal. *VirusTotal Report Client.exe*. URL: https://www.virustotal.com/gui/file/d2cddddd3aa5696d94586f2e4c7ca56bf13abcd4611de5dda1ebc97adfe8fa16 (visited on 05/14/2025).

[96] A. Wickramasekara, F. Breitinger, and M. Scanlon. "Exploring the Potential of Large Language Models for Improving Digital Forensic Investigation Efficiency". en. In: *Forensic Science International: Digital Investigation* (Mar. 2025). ISSN: 26662817. DOI: 10.1016/j.fsidi.2024.301859. (Visited on 05/14/2025).

[97] A. Wickramasekara and M. Scanlon. "A Framework for Integrated Digital Forensic Investigation Employing AutoGen AI Agents". en. In: *2024 12th International Symposium on Digital Forensics and Security (ISDFS)*. San Antonio, TX, USA: IEEE, Apr. 2024. ISBN: 9798350330366. DOI: 10.1109/ISDFS60797.2024.10527235. URL: https://ieeexplore.ieee.org/document/10527235/ (visited on 05/14/2025).

[98] M. Willing. *LETHAL-FORENSICS/MemProcFS-Analyzer*. URL: https://github.com/LETHAL-FORENSICS/MemProcFS-Analyzer (visited on 05/14/2025).

[99] N. Wong. *Quasar Remote Access Tool and Its Security Risks*. Darktrace. Feb. 23, 2024. URL: https://www.darktrace.com/blog/quasar-remote-access-tool-when-a-legitimate-admin-tool-falls-into-the-wrong-hands (visited on 05/14/2025).

[100] P. N. Wudali, M. Kravchik, E. Malul, P. A. Gandhi, Y. Elovici, et al. *Rule-ATTCK Mapper (RAM): Mapping SIEM Rules to TTPs Using LLMs*. 2025. arXiv: 2502.02337 [cs.CR]. URL: https://arxiv.org/abs/2502.02337.

[101] Z. Xu, Y. Liu, G. Deng, Y. Li, and S. Picek. *A Comprehensive Study of Jailbreak Attack versus Defense for Large Language Models*. May 2024. DOI: 10.48550/arXiv.2402.13457. URL: http://arxiv.org/abs/2402.13457 (visited on 05/14/2025).

[102] P. Yan, S. Tan, M. Wang, and J. Huang. *Prompt Engineering-assisted Malware Dynamic Analysis Using GPT-4*. Dec. 2023. DOI: 10.48550/arXiv.2312.08317. URL: http://arxiv.org/abs/2312.08317 (visited on 05/14/2025).

[103] Z. Yin, Z. Wang, W. Xu, J. Zhuang, P. Mozumder, et al. *Digital Forensics in the Age of Large Language Models*. en. Apr. 2025. DOI: 10.48550/arXiv.2504.02963. URL: http://arxiv.org/abs/2504.02963 (visited on 05/14/2025).

[104] P. Yosifovich, A. Ionescu, M. Russinovich, and D. Solomon. *Windows Internals, Part 1*. en. Harlow: Pearson Education, 2017. ISBN: 978-0-7356-8418-8 978-0-13-398644-0.

[105] N. Zahan, P. Burckhardt, M. Lysenko, F. Aboukhadijeh, and L. Williams. *Leveraging Large Language Models to Detect npm Malicious Packages*. en. Jan. 2025. DOI: 10.48550/arXiv.2403.12196. URL: http://arxiv.org/abs/2403.12196 (visited on 05/14/2025).

[106] H. Zhang, P. Chang, and Z. Ji. *Bridging the Gap: Deciphering Tabular Data Using Large Language Model*. en. Aug. 2023. DOI: 10.48550/arXiv.2308.11891. URL: http://arxiv.org/abs/2308.11891 (visited on 05/14/2025).

[107] J. Zhang, H. Bu, H. Wen, Y. Liu, H. Fei, et al. "When LLMs meet cybersecurity: a systematic literature review". en. In: *Cybersecurity* 1 (Feb. 2025). ISSN: 2523-3246. DOI: 10.1186/s42400-025-00361-w. (Visited on 05/14/2025).

[108] J. Zhao, Y. Rong, Y. Guo, Y. He, and H. Chen. "Understanding Programs by Exploiting (Fuzzing) Test Cases". In: *Findings of the Association for Computational Linguistics: ACL 2023*. Ed. by A. Rogers, J. Boyd-Graber, and N. Okazaki. Toronto, Canada: Association for Computational Linguistics, July 2023. DOI: 10.18653/v1/2023.findings-acl.678. URL: https://aclanthology.org/2023.findings-acl.678/ (visited on 05/14/2025).

A Design and Implementation of MemoryInvestigator

A.1 Third-Party Dependencies Overview

Dependency	Why it is included
streamlit (1.41.0)	Lightweight web interface for rapid prototyping and sharing of the application.
streamlit-agraph (0.0.45)	Interactive, client-side graph visualization inside the Streamlit UI.
langchain, langchain-core, langchain-community, langchain-openai, langchain-chroma, langchain-text-splitters (0.3.*)	Orchestration layer for large-language-model (LLM) calls, retrieval, vector-store integration, and document chunking.
langchain-google-genai (2.0.7)	Adapter for Google's Gemini family models, giving parity with OpenAI endpoints.
openai (1.59.3)	Direct client for calling OpenAI models used in the pipeline.
pandas (2.2.3)	Tabular data manipulation and in-memory analytics.
protobuf (5.28.3)	Fast, schema-based serialization between components.
requests (2.32.3)	Reliable HTTP client for RESTful calls (e.g. to APIs or vector stores).
beautifulsoup4 (4.12.3)	Robust HTML parsing for web-scraped content.
jq (1.8.0)	Lightweight JSON querying and transformation in preprocessing stages.
langid (1.1.6)	Automatic language detection for multilingual corpora.
keyboard (0.13.5)	Local keyboard-event hooks enabling interactive shortcuts during demos.
psutil (6.1.1)	Cross-platform system monitoring to surface resource-usage stats in the UI.

Table A.1: *Third-party Dependencies*

A.2 MemoryInvestigator Project Structure

```
MemoryInvestigator/
|
+-- .streamlit/
|   +-- config.toml
|
+-- pages/
|   +-- data_input.py
|   +-- display_data.py
|   +-- experimental_forensic_rag.py
|   +-- extract_data.py
|   +-- graph.py
|   +-- help.py
|   +-- report.py
|   +-- standard_rag.py
|   +-- tree_of_table.py
|
+-- screenshots/
|
+-- utils/
|   +-- __pycache__/
|   +-- build_rag_from_books.py
|   +-- build_rag_from_books_and_volatility3_data.py
|   +-- chat_handler.py
|   +-- file_handler.py
|   +-- folder_setup.py
|   +-- gemini_thinking.py
|   +-- get_malpedia_references.py
|   +-- initialize_rag_chat.py
|   +-- json_divider.py
|   +-- select_tree.py
|   +-- tree_builder.py
|   +-- volatility_analysis.py
|   +-- volatility_downloader.py
|
+-- .gitignore
+-- config.py
+-- LICENSE
+-- main.py
+-- README.md
+-- requirements.txt
```

Listing A.1: *MemoryInvestigator Project Structure*

A.3 Standard Installation

```
# Clone the repository
git clone https://github.com/jan-hendrik-lang/
    MemoryInvestigator.git
cd MemoryInvestigator

# Install required dependencies
pip install -r requirements.txt

# Start the Streamlit application
streamlit run main.py
```

Listing A.2: *MemoryInvestigator Installation Guide*

A.4 Additional Linux Steps

```
# Install Python 3.11 and the virtual environment packages
sudo apt-get install python3.11-dev python3.11-venv

# Verify the Python 3.11 installation path
which python3.11

# Typically, the path is /usr/bin/python3.11. Adjust the
    subsequent commands if necessary.
# Create a virtual environment in the Home directory:
cd
mkdir virtual_env
/usr/bin/python3.11 -m venv ~/virtual_env/
    venv_with_python3.11
source ~/virtual_env/venv_with_python3.11/bin/activate
```

Listing A.3: *Additional Linux Steps*

B MemoryInvestigator Screenshots

B.1 Data Input

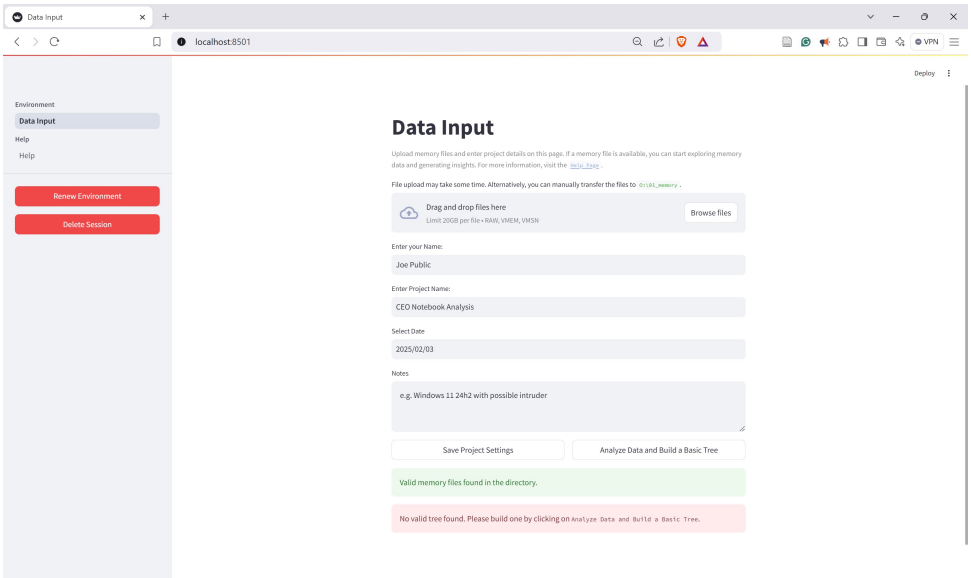


Figure B.1: *MemoryInvestigator Screenshot: Data Input*

B.2 Display Data as a Table

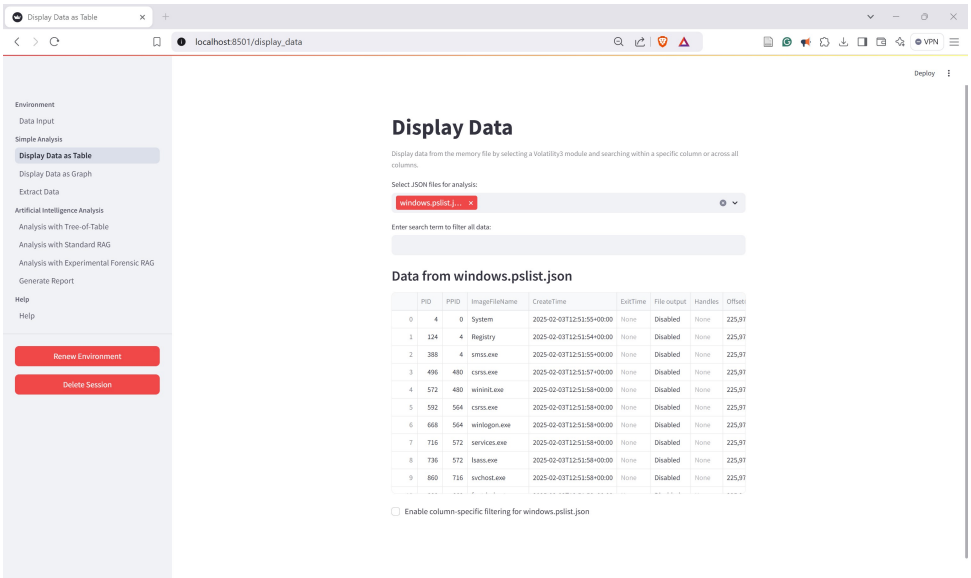


Figure B.2: *MemoryInvestigator Screenshot: Display Data as a Table*

B.3 Display Data as a Graph

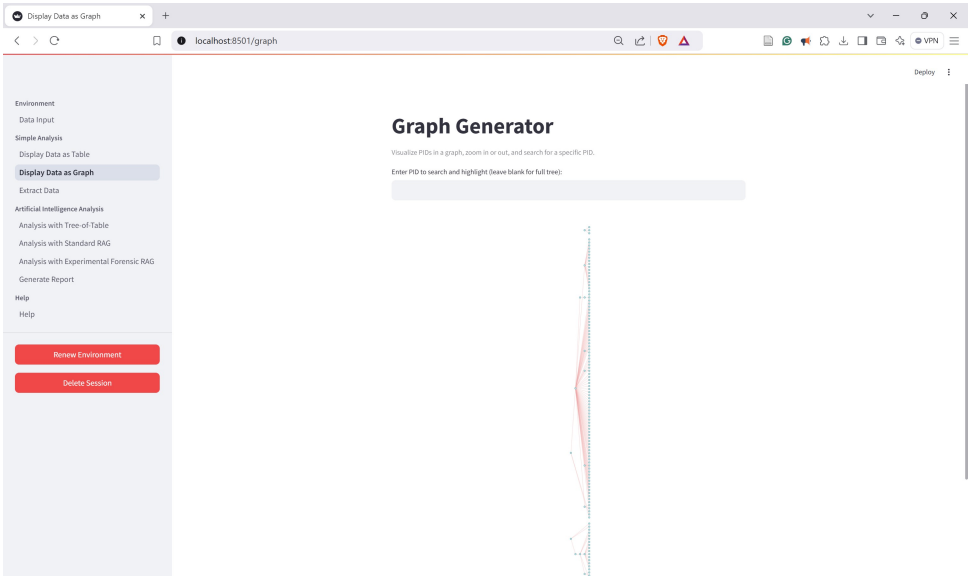


Figure B.3: *MemoryInvestigator Screenshot: Display Data as a Graph*

B.4 Search Data in the Graph

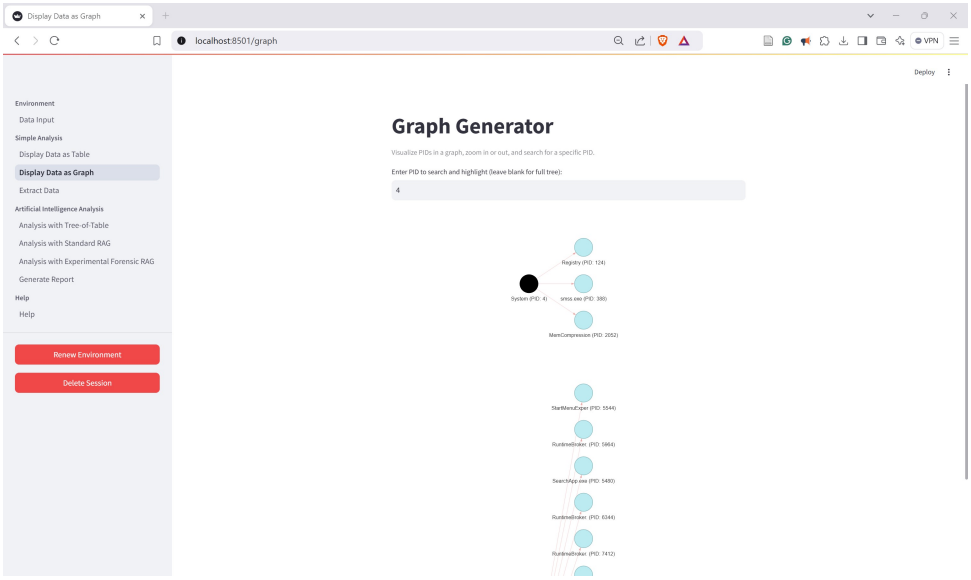


Figure B.4: *MemoryInvestigator Screenshot: Search Data in the Graph*

B.5 Extract Data

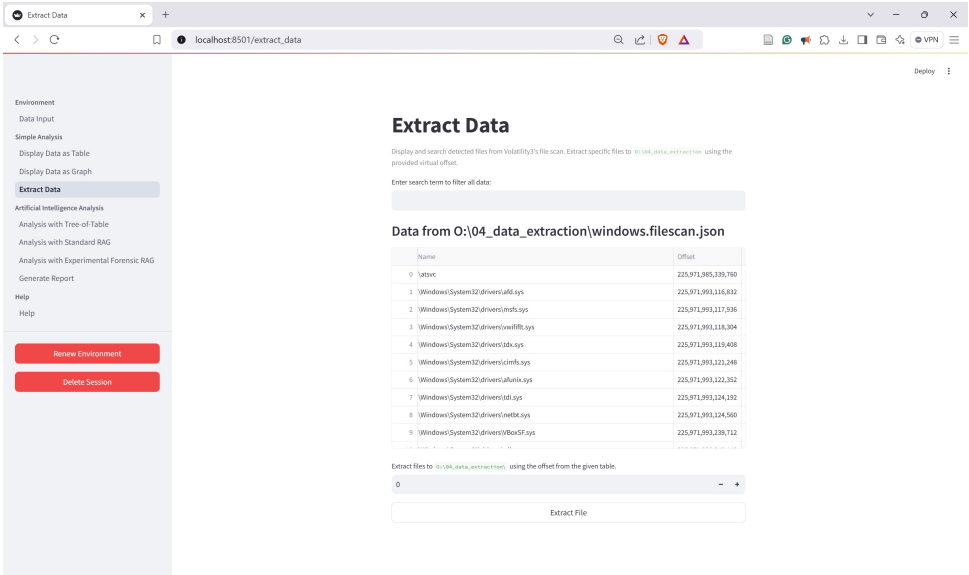


Figure B.5: *MemoryInvestigator Screenshot: Extract Data*

B.6 Tree-of-Table Analysis

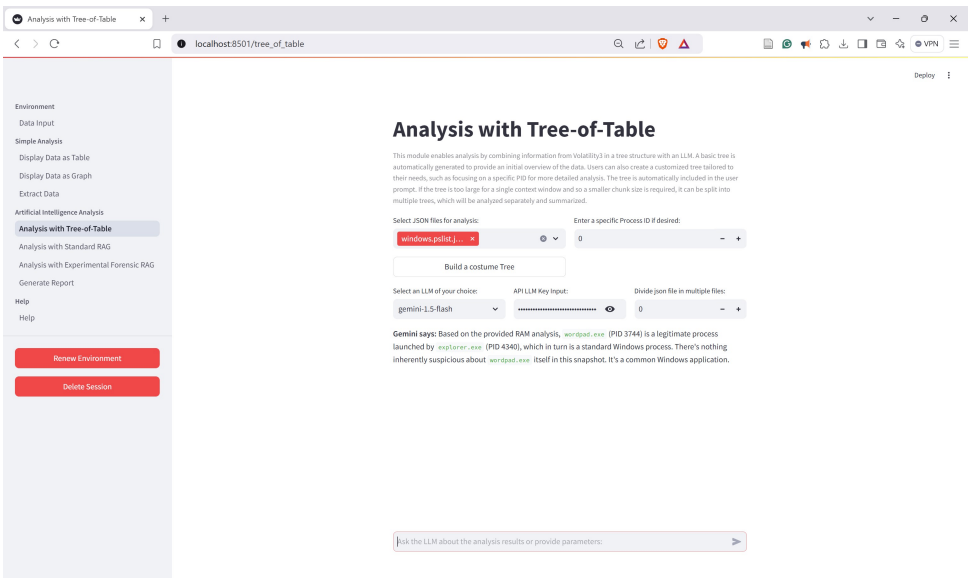


Figure B.6: *MemoryInvestigator Screenshot: Tree-of-Table Analysis*

B.7 Standard RAG

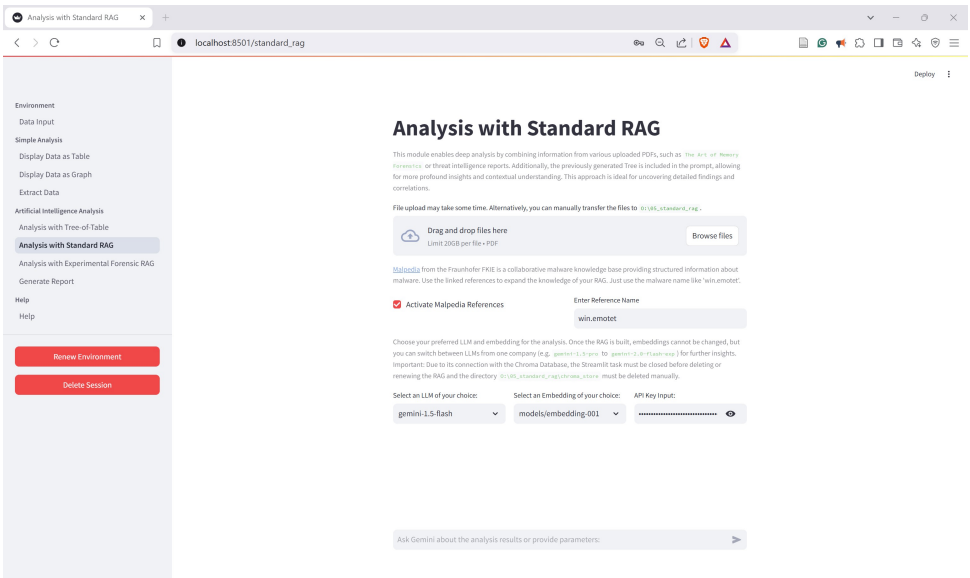


Figure B.7: *MemoryInvestigator Screenshot: Tree-of-Table Analysis*

B.8 Experimental Forensic RAG

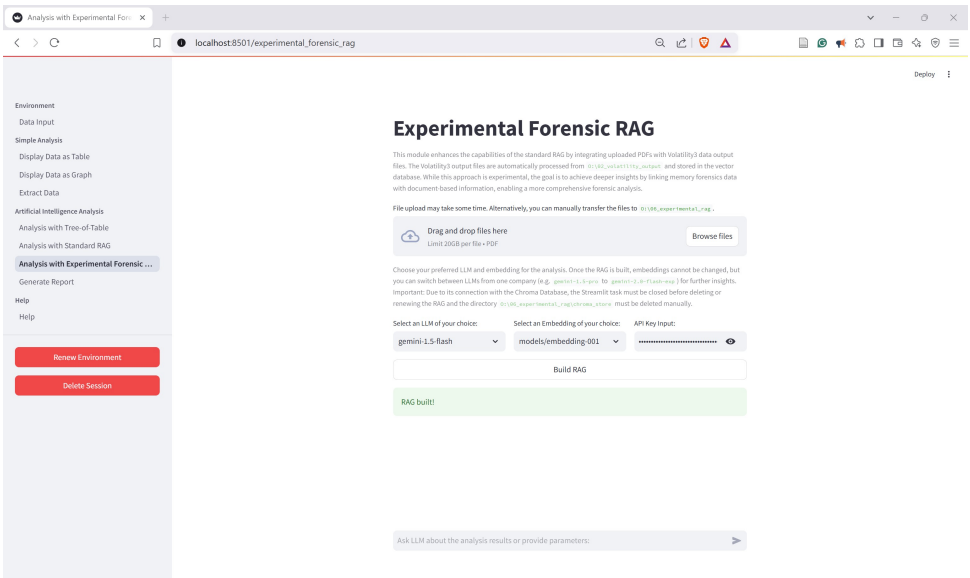


Figure B.8: *MemoryInvestigator Screenshot: Experimental Forensic RAG*

B.9 Temporary Folder Structure

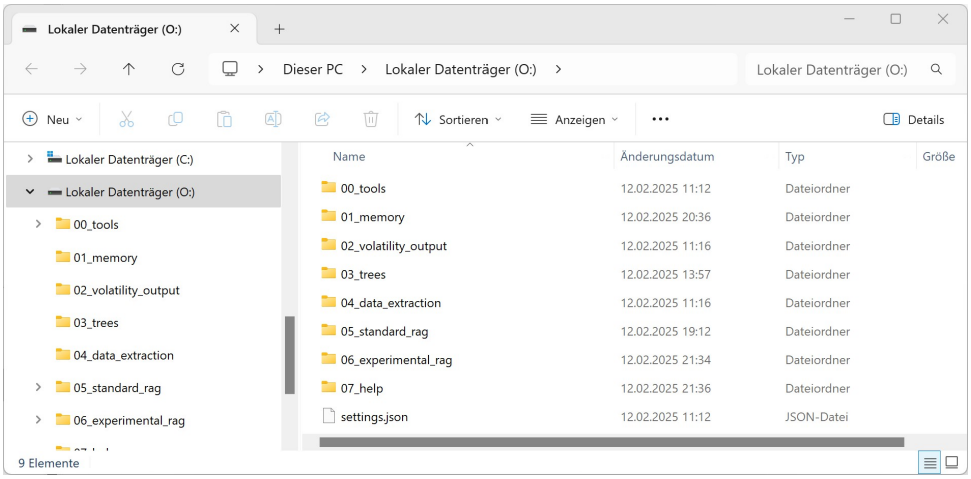


Figure B.9: *MemoryInvestigator Screenshot: Temporary Folder Structure*

C Tree-of-Table Algorithm

C.1 Mapped Volatility3 Modules to Their Specific Fields

```
field_mapping = {
    "windows.psscan.json": ["PID", "PPID", "ImageFileName", "CreateTime", "ExitTime"],
    "windows.cmdline.json": ["Args"],
    "windows.netscan.json": ["Created", "Owner", "Proto", "LocalAddr", "LocalPort", "ForeignAddr", "ForeignPort", "State"],
    "windows.netstat.json": ["Created", "Owner", "Proto", "LocalAddr", "LocalPort", "ForeignAddr", "ForeignPort", "State"],
    "windows.dlllist.json": ["LoadTime", "Path"],
    "windows.getsids.json": ["SID"],
    "windows.hollowprocesses.json": ["Process", "Notes"],
    "windows.ldrmodules.json": ["InInit", "InLoad", "InMem", "MappedPath"],
    "windows.malfind.json": ["Process", "Protection", "Disasm", "Hexdump", "Notes"],
    "windows.processghosting.json": ["Process", "FILE_OBJECT", "DeletePending", "Path"],
    "windows.svcdiff.json": ["Binary", "Binary (Registry)", "Order", "Start", "State", "Dll"],
    "windows.suspicious_threads": ["TID", "Process", "Context", "Address", "Note"],
}
```

Listing C.1: *Mapped Fields to Volatility3 Output*

C.2 Pseudocode Tree-of-Table Builder

```
// Build hierarchical tree
FUNCTION build_tree(selected_files, pid=None):
    CREATE root node "System Analysis"
    CREATE empty process_nodes dictionary

    // Build process tree
    IF windows.psscan.json exists in selected_files THEN
        CREATE "Processes" node
        FOR EACH process in selected_files[process_list_file]:
            GET PID, PPID, and name
            IF filtering by PID AND it does not match THEN CONTINUE

            CREATE process node with PID, PPID, and children list
```

```
            STORE node in process_nodes dictionary

            IF PPID exists in process_nodes THEN
                ATTACH node to parent (PPID)
            ELSE
                ADD node to "Processes" list
            END IF
        END FOR
        ADD "Processes" to root
    END IF

    // Attach additional data to process nodes
    FOR EACH file in selected_files:
        IF file is "windows.psscan.json" THEN CONTINUE

        CREATE file_node
        GET relevant fields for file

        FOR EACH record in file:
            GET PID
            IF filtering by PID AND it does not match THEN
                CONTINUE

            STORE relevant field data in specific_data

            IF PID exists in process_nodes THEN
                MERGE specific_data into process_nodes[PID
                    ]
            ELSE
                CREATE new node with filename, PID, and
                    specific_data
                ADD to file_node children
            END IF
        END FOR

        IF file_node has children THEN
            ADD file_node to root
        END IF
    END FOR
```

Listing C.2: *Pseudocode Tree-of-Table Builder*

C.3 Tree-of-Table Algorithm Graphic

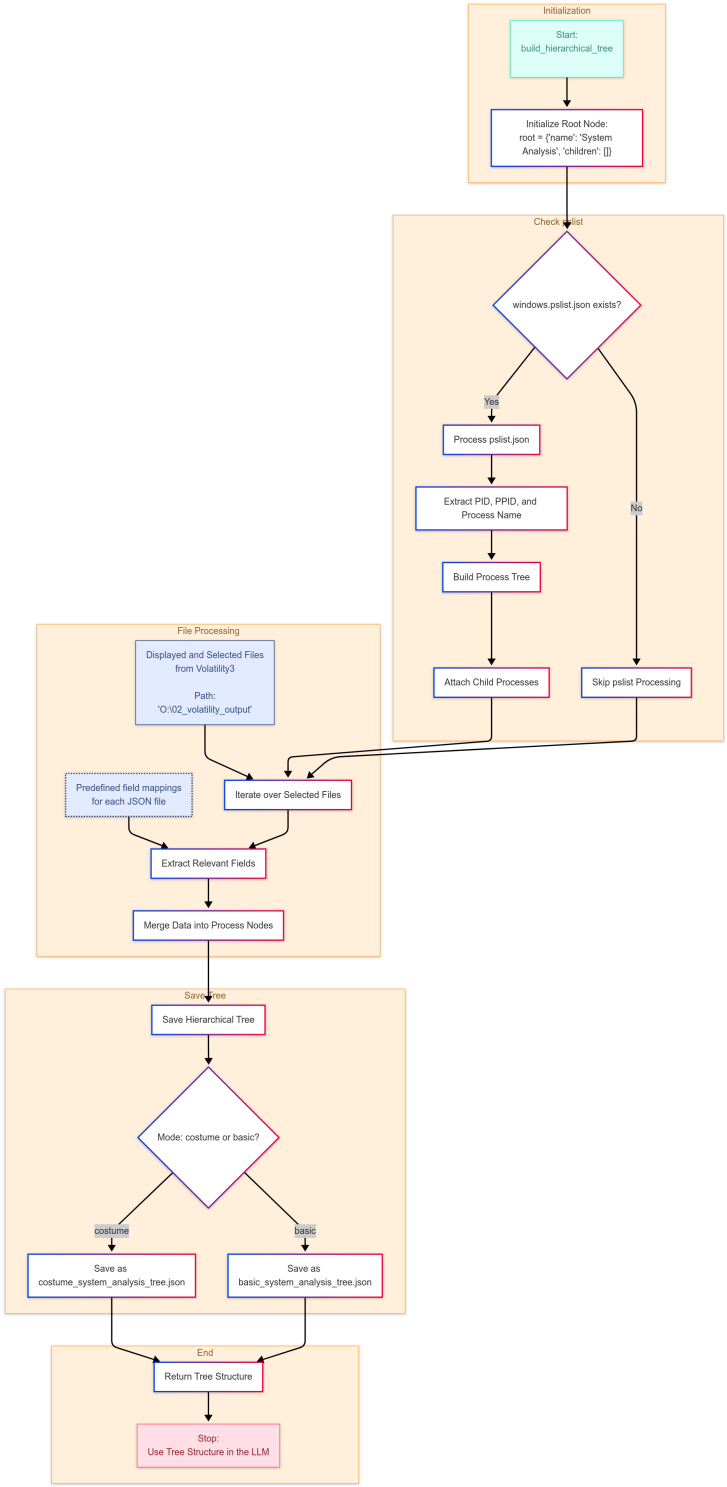


Figure C.1: *Tree-of-Table Algorithm Graphic*

D Excerpt of Trees

D.1 Basic-build Tree


```
{
    "name": "System Analysis",
    "children": [
        {
            "name": "Processes",
            "children": [
                {
                    "name": "System",
                    "pid": 4,
                    "ppid": 0,
                    "children": [
                        {
                            "name": "Registry",
                            "pid": 92,
                            "ppid": 4,
                            "children": []
                        },
                        {
                            "name": "smss.exe",
                            "pid": 328,
                            "ppid": 4,
                            "children": []
                        },
                        {
                            "name": "MemCompression",
                            "pid": 1848,
                            "ppid": 4,
                            "children": []
                        }
                    ]
                }
            ]
        }
    ]
}
```


Listing D.1: *Basic-build Tree*

D.2 Custom-build Tree


```
{
    "name": "System Analysis",
    "children": [
        {
            "name": "Processes",
            "children": [
                {
                    "name": "System",
                    "pid": 4,
                    "ppid": 0,
                    "SID": [
                        "S-1-5-18",
                        "S-1-5-32-544",
                        "S-1-1-0",
                        "S-1-5-11"
                    ],
                    "ldrmodules": {
                        "InInit": false,
                        "InLoad": false,
                        "InMem": false,
                        "MappedPath": "\\Windows\\System32
                           \\ntdll.dll"
                    },
                    "netscan": {
                        "Created": "2023-06-02T17
                           :45:43+00:00",
                        "Owner": "System",
                        "Proto": "UDPv4",
                        "LocalAddr": "192.168.10.210",
                        "LocalPort": 137,
                        "ForeignAddr": "*",
                        "ForeignPort": 0,
                        "State": "LISTENING"
                    },
                    "netstat": {
                        "Created": "2023-06-02T17
                           :45:43+00:00",
                        "Owner": "System",
                        "Proto": "UDPv4",
                        "LocalAddr": "192.168.10.210",
                        "LocalPort": 138,
                        "ForeignAddr": "*",
                        "ForeignPort": 0,
                        "State": "LISTENING"
                    }
                }
            ]
        }
    ]
}
```


Listing D.2: *Custom-build Tree*

E Malware Behavior

E.1 MassLogger

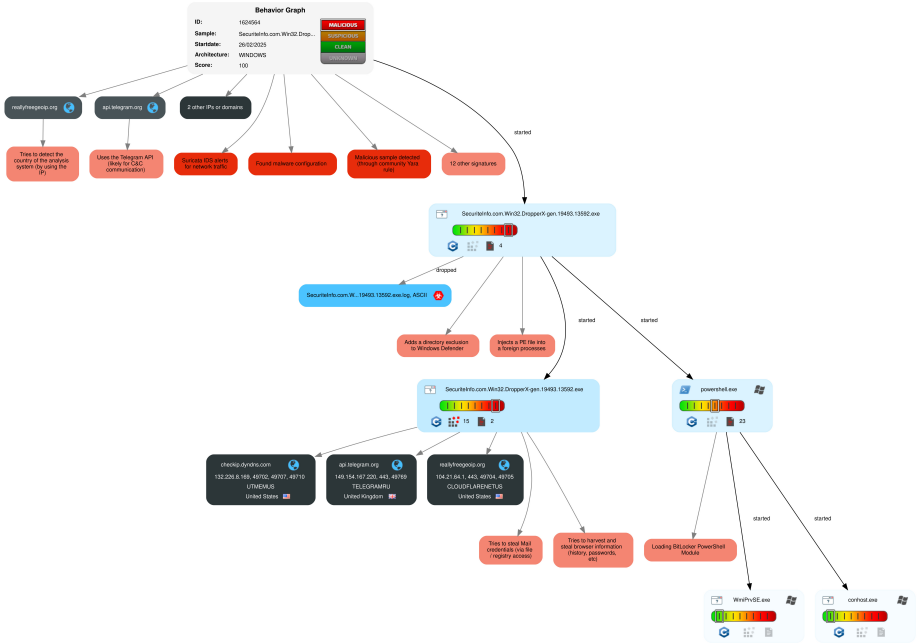


Figure E.1: *Malware Behavior: MassLogger [77]*

E.2 DarkCloud

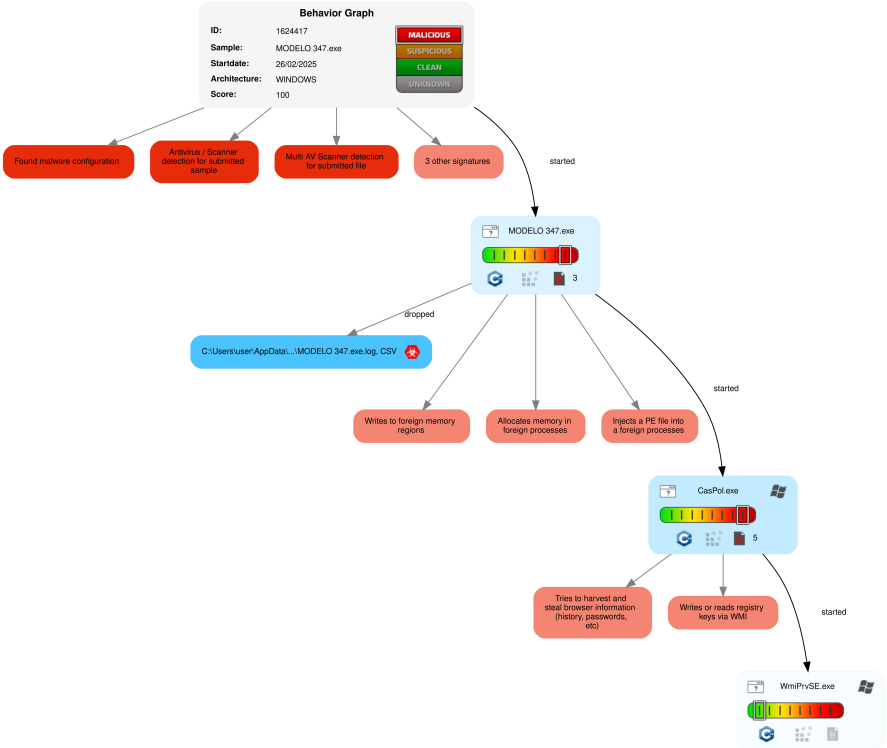


Figure E.2: *Malware Behavior: DarkCloud [76]*

E.3 LockiBot

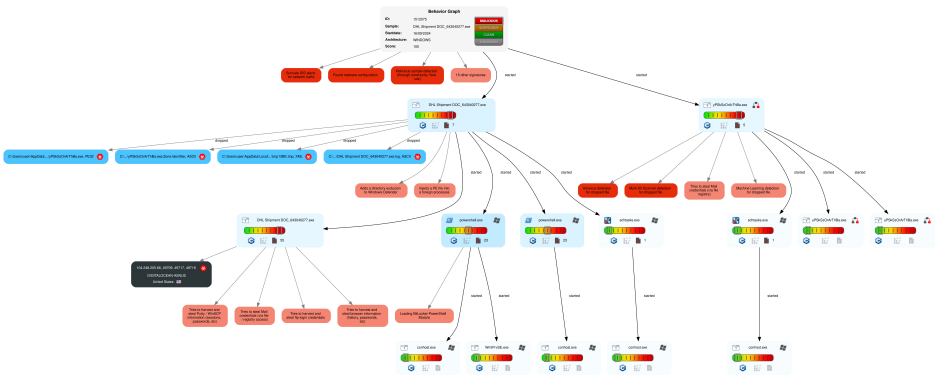


Figure E.3: *Malware Behavior: LockiBot [48]*

E.4 LockBit (aka Worm.m0yv)

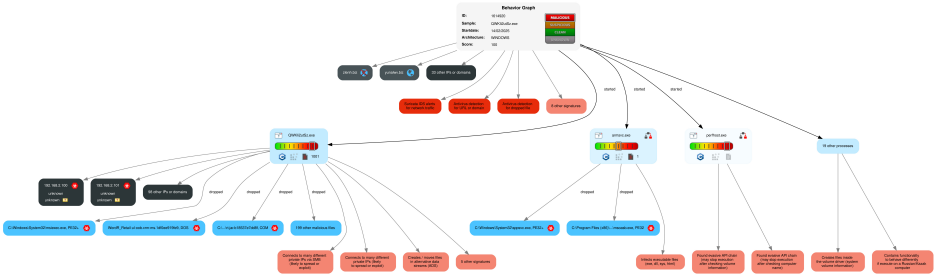


Figure E.4: *Malware Behavior: LockBit [47]*

F Evaluation Summary of Experimental Results

F.1 Tabular Summary of Experimental Results

LLM	Virus	Attempts	Mean Process Count	Mean TP	Mean TN	Mean FN	Mean FP	Mean Accuracy	Mean Precision	Mean Recall	Mean F1
Gemini 2.0 Flash	Clean Image	5	175.00	0.00	172.20	0.00	2.80	98.40	0.00	NaN	NaN
Gemini 2.0 Flash	DarkCloud	5	177.00	0.80	171.00	0.20	5.00	97.06	0.14	0.80	0.29
Gemini 2.0 Flash	LockBit	5	190.00	0.00	184.00	0.00	6.00	96.84	0.00	NaN	NaN
Gemini 2.0 Flash	LockiBot	5	199.00	1.00	192.20	0.00	5.80	97.09	0.15	1.00	0.26
Gemini 2.0 Flash	MassLogger	5	172.00	1.00	163.60	0.00	7.40	95.70	0.12	1.00	0.22
Gemini 2.0 Flash	MSFVenom Code Injection	5	170.00	0.60	162.60	1.40	5.40	96.00	0.09	0.30	0.23
Gemini 2.0 Flash	PowerShell Empire	5	178.00	2.00	171.40	0.00	4.60	97.42	0.31	1.00	0.47
Gemini 2.0 Flash	Quasar RAT	5	172.00	2.00	164.40	0.00	5.60	96.74	0.26	1.00	0.42
Gemini 2.0 Flash-Thinking	Clean Image	5	175.00	0.00	168.40	0.00	6.60	96.23	0.00	NaN	NaN
Gemini 2.0 Flash-Thinking	DarkCloud	5	177.00	0.96	171.04	0.00	5.00	97.18	0.16	1.00	0.28
Gemini 2.0 Flash-Thinking	LockBit	5	190.00	0.00	185.00	0.00	5.00	97.37	0.00	NaN	NaN
Gemini 2.0 Flash-Thinking	LockiBot	5	199.00	1.00	193.00	0.00	5.00	97.49	0.17	1.00	0.29
Gemini 2.0 Flash-Thinking	MassLogger	5	172.00	1.00	167.60	0.00	3.40	98.02	0.24	1.00	0.38
Gemini 2.0 Flash-Thinking	MSFVenom Code Injection	5	170.00	2.00	162.20	0.00	5.80	96.59	0.26	1.00	0.41
Gemini 2.0 Flash-Thinking	PowerShell Empire	5	178.00	2.00	171.00	0.00	5.00	97.19	0.30	1.00	0.46
Gemini 2.0 Flash-Thinking	Quasar RAT	5	172.00	2.00	164.00	0.00	6.00	96.51	0.25	1.00	0.40
Grok 3	Clean Image	5	175.00	0.00	171.40	0.00	3.60	97.94	0.00	NaN	NaN
Grok 3	DarkCloud	5	177.00	0.96	172.44	0.00	3.60	97.97	0.22	1.00	0.36
Grok 3	LockBit	5	190.00	0.00	184.80	0.00	5.20	97.26	0.00	NaN	NaN
Grok 3	LockiBot	5	199.00	1.00	193.60	0.00	4.40	97.79	0.19	1.00	0.31
Grok 3	MassLogger	5	172.00	1.00	167.00	0.00	4.00	97.67	0.21	1.00	0.34
Grok 3	MSFVenom Code Injection	5	170.00	2.00	164.80	0.00	3.20	98.12	0.39	1.00	0.56
Grok 3	PowerShell Empire	5	178.00	0.00	173.40	2.00	2.60	97.42	0.00	0.00	NaN
Grok 3	Quasar RAT	5	172.00	2.00	167.20	0.00	2.80	98.37	0.42	1.00	0.59
Grok 3 ThinkingMode	Clean Image	5	175.00	0.00	172.40	0.00	2.60	98.51	0.00	NaN	NaN
Grok 3 ThinkingMode	DarkCloud	5	177.00	1.00	172.80	0.20	3.00	98.19	0.26	0.90	0.39
Grok 3 ThinkingMode	LockBit	5	190.00	0.00	185.40	0.00	4.60	97.58	0.00	NaN	NaN
Grok 3 ThinkingMode	LockiBot	5	199.00	1.00	194.00	0.20	3.80	97.99	0.21	0.90	0.34
Grok 3 ThinkingMode	MassLogger	5	172.00	1.00	167.00	0.00	4.00	97.67	0.21	1.00	0.34
Grok 3 ThinkingMode	MSFVenom Code Injection	5	170.00	2.00	166.60	0.00	1.40	99.18	0.60	1.00	0.75
Grok 3 ThinkingMode	PowerShell Empire	5	178.00	0.00	174.20	2.00	1.80	97.87	0.00	0.00	NaN
Grok 3 ThinkingMode	Quasar RAT	5	172.00	2.00	164.40	0.00	5.60	96.74	0.27	1.00	0.42
OpenAI GPT4o	Clean Image	5	175.00	0.00	171.60	0.00	3.40	98.06	0.00	NaN	NaN
OpenAI GPT4o	DarkCloud	5	177.00	0.96	171.64	0.00	4.40	97.51	0.19	1.00	0.31
OpenAI GPT4o	LockBit	5	190.00	0.00	185.60	0.00	4.40	97.68	0.00	NaN	NaN
OpenAI GPT4o	LockiBot	5	199.00	1.00	193.40	0.00	4.60	97.69	0.19	1.00	0.31
OpenAI GPT4o	MassLogger	5	172.00	1.00	165.40	0.00	5.60	96.74	0.16	1.00	0.27
OpenAI GPT4o	MSFVenom Code Injection	5	170.00	1.60	160.80	0.40	7.20	95.53	0.18	0.80	0.29
OpenAI GPT4o	PowerShell Empire	5	178.00	2.00	173.00	0.00	3.00	98.31	0.43	1.00	0.59
OpenAI GPT4o	Quasar RAT	5	172.00	2.00	164.00	0.00	6.00	96.51	0.25	1.00	0.40
OpenAI o1	Clean Image	5	175.00	0.00	172.80	0.00	2.20	98.74	0.00	NaN	NaN
OpenAI o1	DarkCloud	5	177.00	0.96	173.04	0.00	3.00	98.31	0.25	1.00	0.40
OpenAI o1	LockBit	5	190.00	0.00	185.80	0.00	4.20	97.79	0.00	NaN	NaN
OpenAI o1	LockiBot	5	199.00	1.00	194.40	0.00	3.60	98.19	0.22	1.00	0.36
OpenAI o1	MassLogger	5	172.00	1.00	166.20	0.00	4.80	97.21	0.19	1.00	0.31
OpenAI o1	MSFVenom Code Injection	5	170.00	2.00	163.60	0.00	4.40	97.41	0.32	1.00	0.49
OpenAI o1	PowerShell Empire	5	178.00	2.00	173.00	0.00	3.00	98.31	0.42	1.00	0.58
OpenAI o1	Quasar RAT	5	172.00	2.00	165.80	0.00	4.20	97.56	0.33	1.00	0.49

Table F.1: *Results overview illustrated using the LLM and Virus*

F.2 Summary of Statistical Tests

Metric	Test	Test Stat.	p-value	Effect Size	Interpretation
Accuracy	ANOVA (ThinkingType)	$F(1,224) = 8.22$	< 0.01 *	—	Significant main effect of ThinkingType
	t-test	$t = -2.68$	0.0078 *	$\Delta M = 0.333$	Thinking > Normal
	Pearson	$r = 0.171$	0.0078 *	—	Weak positive correlation
	Spearman	$\rho = 0.172$	0.0074 *	—	Weak positive correlation
Recall	ANOVA (ThinkingType)	$F(1,168) = 1.90$	0.17–0.22	—	No significant effect
	t-test	$t = -1.24$	0.2185	$\Delta M = 0.05$	No significant difference
	Pearson	$r = 0.092$	0.2184	—	No significant correlation
	Spearman	$\rho = 0.104$	0.1642	—	Weak, non-significant correlation
Precision	ANOVA (ThinkingType)	$F(1,224) = 6.42$	≈ 0.01 *	—	Significant main effect
	t-test	$t = -1.65$	0.1004	$\Delta M = 0.032$	Not statistically significant
	Pearson	$r = 0.106$	0.1003	—	Weak correlation
	Spearman	$\rho = 0.125$	0.0538	—	Borderline significance
F1 Score	ANOVA (ThinkingType)	$F(1,155) = 9.27$	< 0.01 *	—	Significant effect of ThinkingType
	t-test	$t = -2.34$	0.0204 *	$\Delta M = 0.047$	Thinking > Normal
	Pearson	$r = 0.179$	0.0204 *	—	Weak positive correlation
	Spearman	$\rho = 0.207$	0.0071 *	—	Significant positive correlation

Table F.2: *Summary of statistical tests comparing the type of thinking and viruses in performance metrics. Significant p-values ($p < 0.05$) are marked with an asterisk (*)*

F.3 Tukey HSD Test Results for LLM (F1 Score Comparisons)

Compared Model A	Compared Model B	Difference	Lower CI	Upper CI	p-value
Gemini 2.0 Flash-Thinking	Gemini 2.0 Flash	0.0492	-0.0451	0.1436	0.6620
Grok 3	Gemini 2.0 Flash	0.1120	0.0133	0.2107	0.0162 *
Grok 3 Thinking Mode	Gemini 2.0 Flash	0.1270	0.0283	0.2257	0.0038 *
OpenAI GPT4o	Gemini 2.0 Flash	0.0420	-0.0524	0.1363	0.7939
OpenAI o1	Gemini 2.0 Flash	0.1170	0.0227	0.2114	0.0060 *
Grok 3	Gemini 2.0 Flash-Thinking	0.0628	-0.0335	0.1591	0.4182
Grok 3 Thinking Mode	Gemini 2.0 Flash-Thinking	0.0778	-0.0185	0.1741	0.1886
OpenAI GPT4o	Gemini 2.0 Flash-Thinking	-0.0072	-0.0991	0.0846	0.9999
OpenAI o1	Gemini 2.0 Flash-Thinking	0.0678	-0.0240	0.1597	0.2770
Grok 3 Thinking Mode	Grok 3	0.0150	-0.0856	0.1156	0.9981
OpenAI GPT4o	Grok 3	-0.0700	-0.1663	0.0263	0.2942
OpenAI o1	Grok 3	0.0050	-0.0913	0.1014	0.9999
OpenAI GPT4o	Grok 3 Thinking Mode	-0.0850	-0.1813	0.0113	0.1172
OpenAI o1	Grok 3 Thinking Mode	-0.0099	-0.1063	0.0864	0.9997
OpenAI o1	OpenAI GPT4o	0.0751	-0.0168	0.1669	0.1777

Table F.3: *Tukey HSD test results for LLM (F1 score comparisons) Significant p-values ($p < 0.05$) are marked with an asterisk (*)*

G Examples MemoryInvestigator Results

G.1 Volatility3 windows.dlllist Quasar RAT Output of PID 2032

```
Volatility 3 Framework 2.8.0

PID     Process Base    Size    Name    Path    LoadTime    File output

2032    Client.exe      0x610000        0x324000        Client.exe      C:\Users\Maste\AppData\Roaming\SubDir\Client.exe        2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9ca8d0000  0x1f8000        ntdll.dll       C:\Windows\SYSTEM32\ntdll.dll   2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9bd390000  0x65000 MSCOREE.DLL     C:\Windows\SYSTEM32\MSCOREE.DLL 2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c8e20000  0xbd000 KERNEL32.dll    C:\Windows\System32\KERNEL32.dll        2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c8130000  0x2f6000        KERNELBASE.dll  C:\Windows\System32\KERNELBASE.dll      2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c5910000  0x90000 apphelp.dll     C:\Windows\SYSTEM32\apphelp.dll 2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9ca620000  0xaf000 ADVAPI32.dll    C:\Windows\System32\ADVAPI32.dll        2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c89e0000  0x9e000 msvcrt.dll      C:\Windows\System32\msvcrt.dll  2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9ca6e0000  0x9c000 sechost.dll     C:\Windows\System32\sechost.dll 2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c9510000  0x126000        RPCRT4.dll      C:\Windows\System32\RPCRT4.dll  2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9bd2e0000  0xaa000 mscoreei.dll    C:\Windows\Microsoft.NET\Framework64\v4.0.30319\mscoreei.dll    2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c8d00000  0x55000 SHLWAPI.dll     C:\Windows\System32\SHLWAPI.dll 2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c5e90000  0x12000 kernel.appcore.dll      C:\Windows\SYSTEM32\kernel.appcore.dll  2025-02-11 12:45:38.000000 UTC  Disabled
```

```
2032    Client.exe      0x7ff9c15e0000  0xa000  VERSION.dll     C:\Windows\SYSTEM32\VERSION.dll 2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9947d0000  0xb34000 clr.dll C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll 2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c9010000  0x19e000 USER32.dll      C:\Windows\System32\USER32.dll 2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c8590000  0x22000 win32u.dll          C:\Windows\System32\win32u.dll  2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c9c70000  0x2c000 GDI32.dll            C:\Windows\System32\GDI32.dll   2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c85c0000  0x11a000 gdi32full.dll   C:\Windows\System32\gdi32full.dll            2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c8090000  0x9d000 msvcp_win.dll   C:\Windows\System32\msvcp_win.dll 2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c8830000  0x100000 ucrtbase.dll    C:\Windows\System32\ucrtbase.dll            2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9bd1f0000  0x16000 VCRUNTIME140_CLR0400.dll        C:\Windows\SYSTEM32\VCRUNTIME140_CLR0400.dll    2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9b5f90000  0xbd000 ucrtbase_clr0400.dll    C:\Windows\SYSTEM32\ucrtbase_clr0400.dll        2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9ca860000  0x30000 IMM32.DLL            C:\Windows\System32\IMM32.DLL   2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9ca780000  0x8000  psapi.dll            C:\Windows\System32\psapi.dll   2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff97d1d0000  0x165e000 mscorlib.ni.dll C:\Windows\assembly\NativeImages_v4.0.30319_64\mscorlib\4bc5e5252873c08797895d5b6fe6ddfd\mscorlib.ni.dll        2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c8ee0000  0x12b000 ole32.dll       C:\Windows\System32\ole32.dll 2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c91b0000  0x354000 combase.dll     C:\Windows\System32\combase.dll 2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c7fb0000  0x82000 bcryptPrimitives.dll    C:\Windows\System32\bcryptPrimitives.dll        2025-02-11 12:45:38.000000 UTC  Disabled
```

```
34 2032    Client.exe      0x7ff9c59e0000  0x9e000 uxtheme.
      dll     C:\Windows\system32\uxtheme.dll 2025-02-11
      12:45:38.000000 UTC  Disabled
35 2032    Client.exe      0x7ff9b37b0000  0x14f000
      clrjit.dll      C:\Windows\Microsoft.NET\Framework64\v4
      .0.30319\clrjit.dll      2025-02-11 12:45:38.000000 UTC
        Disabled
36 2032    Client.exe      0x7ff97c550000  0xc73000
      System.ni.dll   C:\Windows\assembly\NativeImages_v4
      .0.30319_64\System\7497aebfb6ef3f9995d6359fb147361f\
      System.ni.dll    2025-02-11 12:45:38.000000 UTC
      Disabled
37 2032    Client.exe      0x7ff9acce0000  0x1f5000
      System.Drawing.ni.dll   C:\Windows\assembly\
      NativeImages_v4.0.30319_64\System.Drawing\
      bf4a13d46bfcad25312a09a9a8bbf70b\System.Drawing.ni.dll
          2025-02-11 12:45:38.000000 UTC  Disabled
38 2032    Client.exe      0x7ff97b4a0000  0x10ad000
      System.Windows.Forms.ni.dll     C:\Windows\assembly\
      NativeImages_v4.0.30319_64\System.Windows.Forms\90
      e07608923a83d3685b7c14c74a78f9\System.Windows.Forms.ni.
      dll        2025-02-11 12:45:38.000000 UTC  Disabled
39 2032    Client.exe      0x7ff996130000  0xa75000
      System.Core.ni.dll      C:\Windows\assembly\
      NativeImages_v4.0.30319_64\System.Core\2
      d55b15f490a5f1b0b99cadb62487838\System.Core.ni.dll
      2025-02-11 12:45:38.000000 UTC  Disabled
40 2032    Client.exe      0x7ff9acba0000  0x133000
      System.Configuration.ni.dll     C:\Windows\assembly\
      NativeImages_v4.0.30319_64\System.Configuration\88
      f1276d9dcd33bfacc14b88f2e5aba4\System.Configuration.ni.
      dll        2025-02-11 12:45:38.000000 UTC  Disabled
41 2032    Client.exe      0x7ff97abf0000  0x8ac000
      System.Xml.ni.dll       C:\Windows\assembly\
      NativeImages_v4.0.30319_64\System.Xml\
      e2418680770c37983aeea3960873cf4d\System.Xml.ni.dll
      2025-02-11 12:45:38.000000 UTC  Disabled
42 2032    Client.exe      0x7ff9c9ed0000  0x745000
      shell32.dll     C:\Windows\System32\shell32.dll
      2025-02-11 12:45:38.000000 UTC  Disabled
43 2032    Client.exe      0x7ff9c6090000  0x79b000
      windows.storage.dll     C:\Windows\SYSTEM32\windows.
      storage.dll 2025-02-11 12:45:38.000000 UTC  Disabled
44 2032    Client.exe      0x7ff9c7920000  0x2d000 Wldp.dll
            C:\Windows\SYSTEM32\Wldp.dll    2025-02-11
      12:45:38.000000 UTC  Disabled
45 2032    Client.exe      0x7ff9c8930000  0xad000 SHCORE.dll
           C:\Windows\System32\SHCORE.dll  2025-02-11
      12:45:38.000000 UTC  Disabled
46 2032    Client.exe      0x7ff9c7ee0000  0x25000 profapi.
      dll     C:\Windows\SYSTEM32\profapi.dll 2025-02-11
      12:45:38.000000 UTC  Disabled
```

```
2032    Client.exe      0x7ff9c8800000  0x27000 bcrypt.dll
          C:\Windows\System32\bcrypt.dll  2025-02-11
    12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c7880000  0x18000 CRYPTSP.
    dll     C:\Windows\SYSTEM32\CRYPTSP.dll 2025-02-11
    12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c6fb0000  0x34000 rsaenh.dll
          C:\Windows\system32\rsaenh.dll  2025-02-11
    12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c7870000  0xc000  CRYPTBASE.
    dll   C:\Windows\SYSTEM32\CRYPTBASE.dll
    2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9b1540000  0x29a000
    comctl32.dll    C:\Windows\WinSxS\amd64_microsoft.
    windows.common-controls_6595b64144ccf1df_6.0.19041.3636
    _none_60b6a03d71f818d5\comctl32.dll    2025-02-11
    12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c9ca0000  0x114000
    MSCTF.dll       C:\Windows\System32\MSCTF.dll
    2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c8a80000  0xcd000 OLEAUT32.
    dll    C:\Windows\System32\OLEAUT32.dll
    2025-02-11 12:45:38.000000 UTC  Disabled
2032    Client.exe      0x7ff9c8430000  0x15d000
    crypt32.dll     C:\Windows\System32\crypt32.dll
    2025-02-11 12:45:39.000000 UTC  Disabled
2032    Client.exe      0x7ff9c7ab0000  0x12000 MSASN1.dll
          C:\Windows\SYSTEM32\MSASN1.dll  2025-02-11
    12:45:39.000000 UTC  Disabled
2032    Client.exe      0x7ff9c7e40000  0x32000 SspiCli.
    dll     C:\Windows\SYSTEM32\SspiCli.dll 2025-02-11
    12:45:39.000000 UTC  Disabled
2032    Client.exe      0x7ff979380000  0x186d000
    System.ServiceModel.ni.dll      C:\Windows\assembly\
    NativeImages_v4.0.30319_64\System.ServiceModel\
    a4415b4508d54a3de077f5b98905dc81\System.ServiceModel.ni
    .dll  2025-02-11 12:45:40.000000 UTC  Disabled
2032    Client.exe      0x7ff9ac850000  0x344000
    System.Runtime.Serialization.ni.dll     C:\Windows\
    assembly\NativeImages_v4.0.30319_64\System.Runteb92aa12
    #\6dbac3c863eeb593ce4b92fcf3eb6a74\System.Runtime.
    Serialization.ni.dll        2025-02-11 12:45:40.000000
    UTC  Disabled
2032    Client.exe      0x7ff9b6980000  0x12000
    Accessibility.ni.dll    C:\Windows\assembly\
    NativeImages_v4.0.30319_64\Accessibility\1
    a1e3e9c9fd1c316613e984718ddb700\Accessibility.ni.dll
          2025-02-11 12:45:40.000000 UTC  Disabled
2032    Client.exe      0x7ff9c96a0000  0x6b000 ws2_32.dll
          C:\Windows\System32\ws2_32.dll  2025-02-11
    12:45:40.000000 UTC  Disabled
2032    Client.exe      0x7ff9c7680000  0x6a000 mswsock.
    dll     C:\Windows\system32\mswsock.dll 2025-02-11
    12:45:40.000000 UTC  Disabled
```

```
2032    Client.exe      0x7ff9ac7a0000  0xa8000 webengine4.dll  C:\Windows\Microsoft.NET\Framework64\v4.0.30319\webengine4.dll  2025-02-11 12:45:54.000000 UTC  Disabled
2032    Client.exe      0x7ff9c7e10000  0x2e000 USERENV.dll     C:\Windows\SYSTEM32\USERENV.dll 2025-02-11 12:45:54.000000 UTC  Disabled
```

Listing G.1: *Volatility3 windows.dlllist Quasar RAT Output*

G.2 Volatility3 windows.netscan Quasar RAT Output

```
Volatility 3 Framework 2.8.0

Offset  Proto   LocalAddr       LocalPort       ForeignAddr     ForeignPort     State   PID     Owner   Created

0x9902e12ac730  TCPv4   192.168.241.129 139     0.0.0.0 0       LISTENING       4       System  2025-02-11 12:42:16.000000 UTC
0x9902e12ac890  TCPv4   0.0.0.0 445     0.0.0.0 0       LISTENING       4       System  2025-02-11 12:42:17.000000 UTC
0x9902e12ac890  TCPv6   ::      445     ::      0       LISTENING       4       System  2025-02-11 12:42:17.000000 UTC
0x9902e12ac9f0  TCPv4   0.0.0.0 49668   0.0.0.0 0       LISTENING       2436    spoolsv.exe     2025-02-11 12:42:17.000000 UTC
0x9902e12ac9f0  TCPv6   ::      49668   ::      0       LISTENING       2436    spoolsv.exe     2025-02-11 12:42:17.000000 UTC
0x9902e1867a70  TCPv4   0.0.0.0 49667   0.0.0.0 0       LISTENING       1288    svchost.exe     2025-02-11 12:42:15.000000 UTC
0x9902e443e470  TCPv4   0.0.0.0 49666   0.0.0.0 0       LISTENING       1340    svchost.exe     2025-02-11 12:42:15.000000 UTC
0x9902e443e470  TCPv6   ::      49666   ::      0       LISTENING       1340    svchost.exe     2025-02-11 12:42:15.000000 UTC
0x9902e443e5d0  TCPv4   0.0.0.0 49664   0.0.0.0 0       LISTENING       824     lsass.exe       2025-02-11 12:42:14.000000 UTC
0x9902e443e5d0  TCPv6   ::      49664   ::      0       LISTENING       824     lsass.exe       2025-02-11 12:42:14.000000 UTC
0x9902e443e730  TCPv4   0.0.0.0 49667   0.0.0.0 0       LISTENING       1288    svchost.exe     2025-02-11 12:42:15.000000 UTC
```

```
16 0x9902e443e730  TCPv6   ::      49667   ::      0
      LISTENING       1288    svchost.exe     2025-02-11
      12:42:15.000000 UTC
17 0x9902e443e890  TCPv4   0.0.0.0 49668   0.0.0.0 0
      LISTENING       2436    spoolsv.exe     2025-02-11
      12:42:17.000000 UTC
18 0x9902e443e9f0  TCPv4   0.0.0.0 49664   0.0.0.0 0
      LISTENING       824     lsass.exe       2025-02-11
      12:42:14.000000 UTC
19 0x9902e443ecb0  TCPv4   0.0.0.0 49665   0.0.0.0 0
      LISTENING       696     wininit.exe     2025-02-11
      12:42:14.000000 UTC
20 0x9902e443f910  TCPv4   0.0.0.0 135     0.0.0.0 0
      LISTENING       500     svchost.exe     2025-02-11
      12:42:14.000000 UTC
21 0x9902e443fa70  TCPv4   0.0.0.0 135     0.0.0.0 0
      LISTENING       500     svchost.exe     2025-02-11
      12:42:14.000000 UTC
22 0x9902e443fa70  TCPv6   ::      135     ::      0
      LISTENING       500     svchost.exe     2025-02-11
      12:42:14.000000 UTC
23 0x9902e443fbd0  TCPv4   0.0.0.0 49666   0.0.0.0 0
      LISTENING       1340    svchost.exe     2025-02-11
      12:42:14.000000 UTC
24 0x9902e443fe90  TCPv4   0.0.0.0 49665   0.0.0.0 0
      LISTENING       696     wininit.exe     2025-02-11
      12:42:14.000000 UTC
25 0x9902e443fe90  TCPv6   ::      49665   ::      0
      LISTENING       696     wininit.exe     2025-02-11
      12:42:14.000000 UTC
26 0x9902e59b3690  UDPv6   fe80::34fe:b94f:6ac2:95ff
      546     *       0               1512    svchost.exe
           2025-02-11 12:42:15.000000 UTC
27 0x9902e5d23090  UDPv4   0.0.0.0 0       *       0
             2092    svchost.exe     2025-02-11
      12:42:16.000000 UTC
28 0x9902e5d23090  UDPv6   ::      0       *       0
             2092    svchost.exe     2025-02-11
      12:42:16.000000 UTC
29 0x9902e5d24b20  UDPv4   0.0.0.0 5353    *       0
             2092    svchost.exe     2025-02-11
      12:42:16.000000 UTC
30 0x9902e5d24b20  UDPv6   ::      5353    *       0
             2092    svchost.exe     2025-02-11
      12:42:16.000000 UTC
31 0x9902e5d252f0  UDPv4   0.0.0.0 5353    *       0
             2092    svchost.exe     2025-02-11
      12:42:16.000000 UTC
32 0x9902e5d25930  UDPv4   0.0.0.0 5355    *       0
             2092    svchost.exe     2025-02-11
      12:42:17.000000 UTC
33 0x9902e5d25930  UDPv6   ::      5355    *       0
             2092    svchost.exe     2025-02-11
      12:42:17.000000 UTC
```

```
0x9902e5d25c50  UDPv4   0.0.0.0 5355    *       0
            2092    svchost.exe     2025-02-11
    12:42:17.000000 UTC
0x9902e5d284f0  UDPv4   192.168.241.129 138     *       0
                    4       System  2025-02-11
    12:42:16.000000 UTC
0x9902e5d28b30  UDPv4   192.168.241.129 137     *       0
                    4       System  2025-02-11
    12:42:16.000000 UTC
0x9902e5e87cd0  UDPv4   0.0.0.0 500     *       0
            2772    svchost.exe     2025-02-11
    12:42:17.000000 UTC
0x9902e5e87cd0  UDPv6   ::      500     *       0
            2772    svchost.exe     2025-02-11
    12:42:17.000000 UTC
0x9902e5e87e60  UDPv4   0.0.0.0 4500    *       0
            2772    svchost.exe     2025-02-11
    12:42:17.000000 UTC
0x9902e5e887c0  UDPv4   0.0.0.0 500     *       0
            2772    svchost.exe     2025-02-11
    12:42:17.000000 UTC
0x9902e5e88950  UDPv4   0.0.0.0 4500    *       0
            2772    svchost.exe     2025-02-11
    12:42:17.000000 UTC
0x9902e5e88950  UDPv6   ::      4500    *       0
            2772    svchost.exe     2025-02-11
    12:42:17.000000 UTC
0x9902e5e8a3e0  UDPv4   0.0.0.0 0       *       0
            2772    svchost.exe     2025-02-11
    12:42:17.000000 UTC
0x9902e5ffc230  TCPv4   0.0.0.0 49670   0.0.0.0 0
    LISTENING       2780    svchost.exe     2025-02-11
    12:42:19.000000 UTC
0x9902e5ffc4f0  TCPv4   0.0.0.0 49670   0.0.0.0 0
    LISTENING       2780    svchost.exe     2025-02-11
    12:42:19.000000 UTC
0x9902e5ffc4f0  TCPv6   ::      49670   ::      0
    LISTENING       2780    svchost.exe     2025-02-11
    12:42:19.000000 UTC
0x9902e5ffc910  TCPv4   0.0.0.0 49669   0.0.0.0 0
    LISTENING       816     services.exe    2025-02-11
    12:42:18.000000 UTC
0x9902e5ffc910  TCPv6   ::      49669   ::      0
    LISTENING       816     services.exe    2025-02-11
    12:42:18.000000 UTC
0x9902e5ffcd30  TCPv4   0.0.0.0 49669   0.0.0.0 0
    LISTENING       816     services.exe    2025-02-11
    12:42:18.000000 UTC
0x9902e6003530  UDPv4   0.0.0.0 0       *       0
            2772    svchost.exe     2025-02-11
    12:42:17.000000 UTC
0x9902e6003530  UDPv6   ::      0       *       0
            2772    svchost.exe     2025-02-11
    12:42:17.000000 UTC
```

```
0x9902e6005c40  UDPv4   0.0.0.0 0       *       0
            2780    svchost.exe     2025-02-11
    12:42:17.000000 UTC
0x9902e6005c40  UDPv6   ::      0       *       0
            2780    svchost.exe     2025-02-11
    12:42:17.000000 UTC
0x9902e6008030  UDPv4   0.0.0.0 0       *       0
            2780    svchost.exe     2025-02-11
    12:42:17.000000 UTC
0x9902e6021030  UDPv4   127.0.0.1       63878   *       0
                    3076    svchost.exe     2025-02-11
    12:42:18.000000 UTC
0x9902e61ff910  TCPv4   0.0.0.0 5040    0.0.0.0 0
    LISTENING       5096    svchost.exe     2025-02-11
    12:42:26.000000 UTC
0x9902e620b730  TCPv4   192.168.241.129 49833
    204.79.197.222  443     ESTABLISHED     5592
    SearchApp.exe   2025-02-11 12:45:56.000000 UTC
0x9902e6963010  TCPv4   192.168.241.129 49683
    204.79.197.203  443     CLOSED  4684    explorer.exe
        2025-02-11 12:42:26.000000 UTC
0x9902e69644a0  TCPv4   192.168.241.129 49806
    13.107.21.239   443     ESTABLISHED     7944    msedge.
    exe     2025-02-11 12:43:52.000000 UTC
0x9902e6964930  TCPv4   192.168.241.129 49802
    20.189.173.1    443     CLOSED  6516    SkypeApp.exe
        2025-02-11 12:43:29.000000 UTC
0x9902e6987050  TCPv4   192.168.241.129 49684
    40.113.110.67   443     ESTABLISHED     3044    svchost
    .exe    2025-02-11 12:42:27.000000 UTC
0x9902e6a28a80  UDPv4   0.0.0.0 5050    *       0
            5096    svchost.exe     2025-02-11
    12:42:26.000000 UTC
0x9902e6eb08a0  TCPv4   192.168.241.129 49831
    20.189.173.7    443     ESTABLISHED     5592
    SearchApp.exe   2025-02-11 12:45:51.000000 UTC
0x9902e6ed8010  TCPv4   192.168.241.129 49815
    20.189.173.17   443     ESTABLISHED     7648
    OneDrive.exe    2025-02-11 12:44:41.000000 UTC
0x9902e6ee5010  TCPv4   192.168.241.129 49713
    20.44.229.112   443     CLOSED  4284    svchost.exe
        2025-02-11 12:42:40.000000 UTC
0x9902e6eeb010  TCPv4   192.168.241.129 49836
    87.228.57.81    4782    SYN_SENT        2032    Client.
    exe     2025-02-11 12:46:29.000000 UTC
0x9902e6f80a20  TCPv4   192.168.241.129 49828
    88.221.197.83   443     CLOSE_WAIT      5592
    SearchApp.exe   2025-02-11 12:45:50.000000 UTC
0x9902e70b4010  TCPv4   192.168.241.129 49832
    204.79.197.239  443     ESTABLISHED     7944    msedge.
    exe     2025-02-11 12:45:52.000000 UTC
0x9902e70d5010  TCPv4   192.168.241.129 49743
    23.212.88.118   443     CLOSED  6724    svchost.exe
        2025-02-11 12:42:51.000000 UTC
```

```
0x9902e70f8890  TCPv4   0.0.0.0 7680    0.0.0.0 0
    LISTENING       6724    svchost.exe     2025-02-11
    12:42:48.000000 UTC
0x9902e70f8890  TCPv6   ::      7680    ::      0
    LISTENING       6724    svchost.exe     2025-02-11
    12:42:48.000000 UTC
0x9902e73858a0  TCPv4   192.168.241.129 49718
    20.44.229.112   443     CLOSED  4284    svchost.exe
         2025-02-11 12:42:42.000000 UTC
0x9902e7659100  UDPv6   fe80::34fe:b94f:6ac2:95ff
    57866   *       0               6244    svchost.exe
         2025-02-11 12:42:39.000000 UTC
0x9902e7659740  UDPv6   ::1     1900    *       0
            6244    svchost.exe     2025-02-11
    12:42:39.000000 UTC
0x9902e7659d80  UDPv4   192.168.241.129 57868   *       0
                    6244    svchost.exe     2025-02-11
    12:42:39.000000 UTC
0x9902e765a0a0  UDPv6   ::1     57867   *       0
            6244    svchost.exe     2025-02-11
    12:42:39.000000 UTC
0x9902e765a6e0  UDPv6   fe80::34fe:b94f:6ac2:95ff
    1900    *       0               6244    svchost.exe
         2025-02-11 12:42:39.000000 UTC
0x9902e765b810  UDPv4   127.0.0.1       57869   *       0
                    6244    svchost.exe     2025-02-11
    12:42:39.000000 UTC
0x9902e765bb30  UDPv4   192.168.241.129 1900    *       0
                    6244    svchost.exe     2025-02-11
    12:42:39.000000 UTC
0x9902e765bcc0  UDPv4   127.0.0.1       1900    *       0
                    6244    svchost.exe     2025-02-11
    12:42:39.000000 UTC
0x9902e79daa20  TCPv4   192.168.241.129 49799
    184.25.50.104   80      ESTABLISHED     6516
    SkypeApp.exe    2025-02-11 12:43:16.000000 UTC
0x9902e7a82a20  TCPv4   192.168.241.129 49830
    88.221.197.83   443     CLOSE_WAIT      5592
    SearchApp.exe   2025-02-11 12:45:50.000000 UTC
0x9902e7a94a20  TCPv4   192.168.241.129 49808
    40.126.31.1     443     CLOSED  6184    svchost.exe
         2025-02-11 12:43:53.000000 UTC
0x9902e7b7ba20  TCPv4   192.168.241.129 49779
    40.113.110.67   443     ESTABLISHED     7648
    OneDrive.exe    2025-02-11 12:43:05.000000 UTC
0x9902e7b83a20  TCPv4   192.168.241.129 49809
    146.75.122.172  80      ESTABLISHED     6520    svchost
    .exe     2025-02-11 12:43:53.000000 UTC
0x9902e7b85b20  TCPv4   192.168.241.129 49829
    88.221.197.83   443     ESTABLISHED     5592
    SearchApp.exe   2025-02-11 12:45:50.000000 UTC
0x9902e7bb9a20  TCPv4   192.168.241.129 49816
    40.69.83.104    443     CLOSED  5096    svchost.exe
         2025-02-11 12:44:47.000000 UTC
```

```
0x9902e7d623c0  UDPv4   0.0.0.0 59025   *       0
            6516    SkypeApp.exe    2025-02-11
    12:43:15.000000 UTC
0x9902e7d623c0  UDPv6   ::      59025   *       0
            6516    SkypeApp.exe    2025-02-11
    12:43:15.000000 UTC
0xd40000099a80  UDPv4   0.0.0.0 5050    *       0
            5096    svchost.exe     2025-02-11
    12:42:26.000000 UTC
```

Listing G.2: *Volatility3 windows.netscan Quasar RAT Output*

G.3 QuasarRAT GPT4o

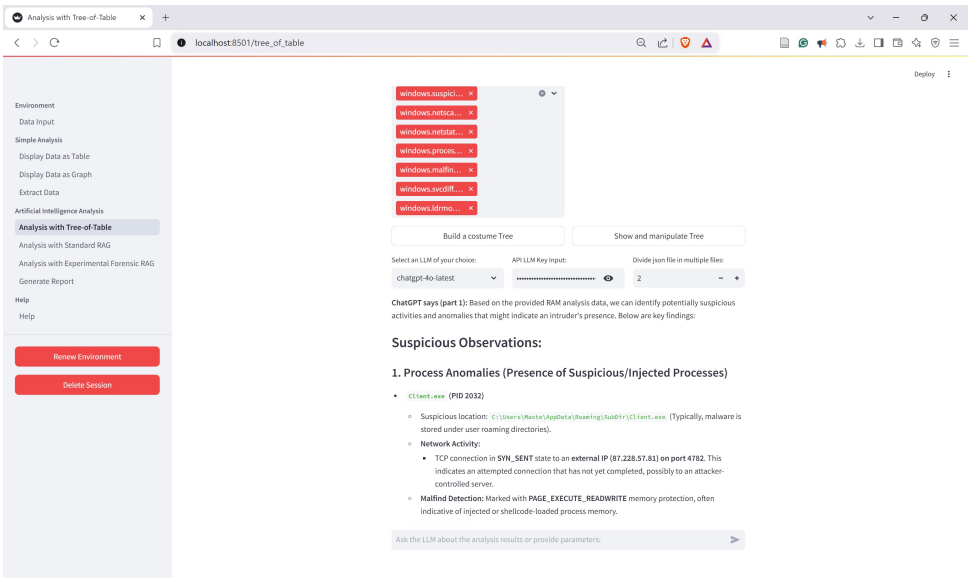


Figure G.1: *MemoryInvestigator Screenshot: QuasarRAT GPT4o*

G.4 QuasarRAT Gemini 1.5 Pro

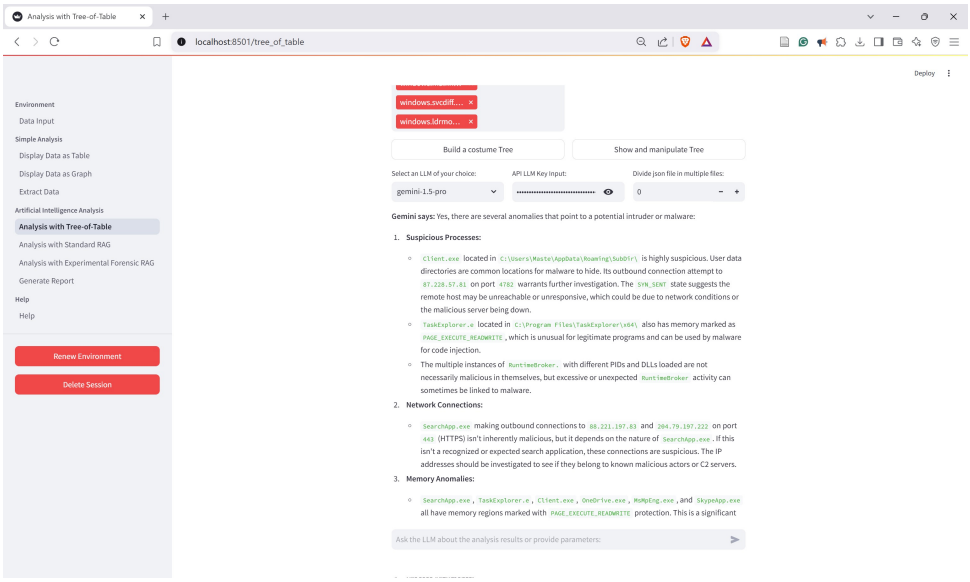


Figure G.2: *MemoryInvestigator Screenshot: QuasarRAT Gemini 1.5 Pro*