TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Herman Karu 242717IVCM

EVALUATING MULTI-AGENT LARGE LANGUAGE MODELS FOR STATIC MALWARE REVERSE ENGINEERING AND ANALYSIS

Master's Thesis

Supervisor: Hayretdin Bahsi
Ph.D

Tallinn 2026

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Herman Karu 242717IVCM

SUUREL KEELEMUDELIL PÕHINEVA MITME AGENDI SÜSTEEMI HINDAMINE STAATILISE PAHAVARA PÖÖRDPROJEKTEERIMISEL JA ANALÜÜSIMISEL

Magistritöö

Juhendaja: Hayretdin Bahsi
Ph.D

Tallinn 2026

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Herman Karu

20.05.2026

Abstract

Static malware reverse engineering remains difficult because analysts must infer behaviour from incomplete or fragmented binary evidence. Automated reverse engineering tools can extract useful artefacts, but interpreting their behavioural meaning still requires expert judgement. Large language models can support this process, although their use in malware analysis raises challenges related to consistency, evidence grounding and completeness.

This thesis evaluates whether multi-agent large language model systems can improve static malware reverse engineering and analysis compared with a single-agent baseline. The study compares different coordination strategies across malware samples with varied platforms, formats and evidence conditions.

The results show that hierarchical multi-agent coordination performs best among the tested configurations. It improves report quality and completion stability compared with the single-agent baseline and sequential workflows. However, the systems still struggle with semantic reconstruction and are stronger at organising structural evidence than at inferring behavioural meaning from static artefacts.

The thesis concludes that multi-agent systems can support static malware analysis, but they remain support tools rather than replacements for expert analysts. Human review, evidence grounding and verification remain necessary.

This thesis is written in English and is 108 pages long, including 7 chapters, 2 figures and 11 tables.

Annotatsioon

Staatiline pahavara pöördprojekteerimine (pöördkonstrueerimine) on endiselt keeruline protsess, sest analüütikud peavad pahavara käitumist järeldama poolikute või katkendlike tõendite põhjal. Pöördprojekteerimise automatiseerimiseks loodud programmid suudavad programmist välja võtta kasulikke tõendeid, kuid nende käitumuslik tõlgendamine eeldab ekspertiisi. Suured keelemudelid pakuvad selle protsessi parandamiseks uusi võimalusi, kuid nendel põhinevad süsteemid võivad sattuda raskustesse järjepidevuse, tõenduspõhisuse või terviklikkusega.

Käesolev uurimistöö hindab seda, kas suurel keelemudelil põhinevad mitme agendi süsteemid suudavad parandada staatilist pahavara pöördprojekteerimist ja analüüsi. Uuring keskendub sellele, kas rollipõhine spetsialiseerumine ja koordinatsioon, mis on omane mitme agendi süsteemile, aitab luua analüüse, mis on täpsemad ja põhjalikumad, kui üksikagendil põhineva süsteemi poolt loodud analüüsid. Uurimistöö raames võrreldakse erinevaid koordineerimisstrateegiaid erisuguste pahavarade analüüsimisel.

Tulemused näitavad, et hierarhiline mitme agendiga süsteem toimib testitud variantidest kõige paremini. Antud süsteem parandab aruannete kvaliteeti ja töövoo stabiilsust võrreldes üksikagendi või lineaarse töövooga. Siiski näitavad tulemused seda, et mitme agendi süsteemid ei lahenda semantilise rekonstruktsiooni probleemi täielikult. Antud süsteemid on tõendite struktuurses organiseerimises tugevamad kui nende põhjal programmi käitumise tõlgendamises.

Uurimistöö leiab, et suurel keelemudelil põhinevad mitme agendi süsteemid suudavad parandada staatilist pahavara analüüsi tõendite organiseerimisel ja esmase ülevaate andmisel. Sellegipoolest peaks antud süsteeme käsitlema kui abivahendeid, mitte analüütikute asendajaid. Manuaalne kontroll ning valideerimine jäävad endiselt vajalikuks, kui loodud aruandeid tahetakse kasutada pahavara analüüsimisel.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 108 leheküljel, 7 peatükki, 2 joonist, 11 tabelit.

List of Abbreviations and Terms

AI	Artificial Intelligence
API	Application Programming Interface
C2	Command and Control
DLL	Dynamic-Link Library
ELF	Executable and Linkable Format
JSON	JavaScript Object Notation
LLM	Large Language Model
MAS	Multi-Agent Large Language Model Systems
PE	Portable Executable
RAT	Remote Access Trojan
RE	Reverse Engineering

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Research Motivation

Modern organisations rely on software systems that may be exposed to different forms of malware. This makes reliable malware analysis important for defensive cybersecurity. Modern malware can support complex operations such as data exfiltration, extortion, persistence and remote access which makes the financial and operational consequences of such malware significant. According to recent industry surveys, nearly 60% of organisations spend over 1 million USD on remediation alone following a ransomware attack [1].

A critical gap has also emerged in the AI (Artificial Intelligence) era. Almost all security leaders express confidence in their detection tools while nearly half admit to detecting their last attack too late to prevent significant damage [2]. This gap is intensified by the perception that attackers benefit from AI more than defenders do. Seventy-eight percent of leaders believe AI has made ransomware attacks more effective, but only 6% believe it has meaningfully improved their own defences [2].

Malware RE (Reverse Engineering) and analysis are essential for defensive response, but they remain slow and difficult to scale. Threat actors often use obfuscation, packing or stripped metadata to obscure a program's intent while analysts must reconstruct behaviour manually from incomplete static evidence [3]. Conventional analysis tools can extract evidence like instructions, functions or metadata, but determining what these artefacts reveal about the program's behaviour still requires expert reasoning [4].

Recent advances in LLMs (Large Language Models) show promise in supporting malware analysis. Single-agent models can refine decompiler output, improve pseudocode readability and infer the intended behaviour of a program [4]. However, single-agent models remain limited in complex structural reasoning tasks, particularly when they must interpret control flow or maintain semantic consistency across large, optimised binaries [5] [6]. These limitations show a growing gap between complex malware and the limited reasoning scope of single-agent solutions.

MAS (Multi-Agent Large Language Model Systems) have emerged as a promising approach for complex tasks through role specialisation and collaborative verification. In a MAS framework, specialised agents collaborate by dividing analytical responsibilities. For example, one agent may focus on control-flow analysis while another specialises in recovering memory structures. These agents can then critique each other's outputs and perform mutual verification [7]. Theoretically, such a coordination model can mitigate errors and context failures of single-agent systems. This motivates evaluating whether multi-agent coordination can produce more accurate and complete malware analysis reports than a single-agent baseline.

1.2 Problem Statement

Prior research has evaluated the effectiveness of LLMs in binary code understanding [5]. It found that while LLMs show promise in high-level semantic reasoning, their performance remains highly variable across different architectures and optimisation levels. LLMs often struggle with the transition from assembly code to structured program logic [5]. Although LLMs have the potential to outperform some traditional static analysis methods in functional interpretation, automated malware RE remains in an early stage. This is largely due to the diversity and complexity of obfuscation techniques in modern malware [3].

Traditional analytical tools can extract relevant artefacts, but behavioural interpretation often remains manual. Single-agent LLM systems can support this interpretation, but they may produce inconsistent or unsupported results when applied to complex binaries [4] [5]. For example, one study found that single-agent LLM assistance reduced analyst cognitive burden by 24.4%, but up to 21.9% of generated simplifications introduced semantic errors [4]. Benchmarking studies also show that single-agent systems can struggle to connect indicators such as imports, strings, control-flow evidence and metadata [6]. These limitations are partly linked to the finite context windows of single models and the lack of analysis from multiple specialised perspectives [7].

This study investigates whether structured multi-agent coordination strategies can mitigate the limitations of single-agent LLM systems in static malware RE. The main priority is to determine which multi-agent architecture provides superior performance in terms of accuracy, stability and completeness when analysing static malware samples.

1.3 Research Questions

This thesis addresses the following research questions:

RQ1: To what extent do multi-agent LLM systems improve the accuracy and completeness of static malware analysis reports compared with single-agent baselines?

Accuracy is measured by comparing reported structural facts, interpretations, interfaces, capabilities and attribution claims against verified ground truth. Completeness is assessed through recall of relevant malicious behaviours and forensic artefacts represented in the verified ground truth.

RQ2: How effectively can multi-agent LLM systems use extracted RE artefacts to support semantic and structural malware reasoning?

Semantic reasoning refers to the correct interpretation of program intent and logic. Structural reasoning concerns the accurate interpretation of binary structure, control-flow evidence and data relationships.

RQ3: How do different agent coordination strategies affect report stability and hallucination rates in generated malware analysis reports?

Report stability is measured through completed and failed pipeline runs. Hallucination rates are determined by identifying claims that are not supported by forensic artefacts or verified ground truth.

RQ4: How effectively can an LLM-as-a-judge evaluation mechanism assess the quality of generated malware analysis reports?

Effectiveness is evaluated by comparing automated scores with manual review metrics, including precision, recall, F1-score and hallucination rates.

1.4 Scope and Goal

The primary goal of this thesis is to evaluate the effectiveness of MAS in generating structured and reliable malware analysis reports. These reports are synthesised from forensic artefacts extracted through an automated two-stage static RE pipeline. Many LLM-assisted RE workflows rely on human analysts to select relevant code snippets or artefact fragments for interpretation. The proposed system automates artefact collection by executing RE tool commands before passing the extracted evidence to the analysis stage. This reduces reliance on manually selected inputs and ensures that each system configuration is evaluated against the same evidence base.

In the first stage, the system performs an extraction step that produces a structured representation of the binary, including metadata, sections and function-level information. Following extraction, the system enters the analysis phase. The aim of this phase is to generate a structured malware analysis report by examining the extracted artefacts and identifying potential malicious capabilities. In the second stage, specialised agents use the extracted data as context for structural, semantic, capability, interface and attribution reasoning. Within this phase, different coordination strategies such as sequential and hierarchical are implemented and evaluated to assess their impact on reasoning accuracy and consistency.

Generated reports are evaluated against ground truth using both an automated LLM auditor and manual review metrics. The study is limited to static analysis and does not execute malware samples in a sandbox. Therefore, this thesis evaluates what the static evidence suggests about malware behaviour rather than confirming behaviour through runtime execution. The objective is to determine whether different multi-agent architectures improve the correctness and completeness of static malware analysis reports compared to a single-agent baseline.

1.5 Novelty

This thesis presents an automated multi-agent pipeline for static malware RE that combines binary artefact extraction with collaborative analysis. Although the application of AI to binary analysis is an emerging field, much of the existing work focuses on the capabilities of single-agent workflows [4] [5] [6]. Other studies have applied MAS to cybersecurity tasks such as malicious package detection [8], threat hunting [9] or vulnerability exploit reproduction [10]. However, these studies do not directly compare multi-agent coordination strategies for static malware RE and report generation.

The study provides a comparative evaluation of sequential and hierarchical multi-agent coordination strategies for malware RE. The proposed pipeline first generates a structured representation of the binary using automated RE tools. Specialised agents then analyse the extracted artefacts from different perspectives, including structure, interfaces, semantics, capabilities and attribution. By assigning agents distinct roles, the study investigates whether role specialisation and coordinated verification improve binary interpretation compared to a single-agent baseline.

The approach is designed to test whether multi-agent coordination can reduce hallucination rates and mitigate context limitations observed in single-agent architectures. The contribution is a controlled empirical evaluation of MAS across malware samples with different formats, platforms and levels of static evidence availability. Overall, the thesis provides practical evidence on the strengths and limitations of multi-agent systems in automated static malware analysis.

2. Literature Review

2.1 Methods

This literature review was conducted through a structured search of academic databases and research repositories including IEEE Xplore, ScienceDirect, arXiv and ResearchGate. To capture the connection between program analysis and AI, search queries were constructed using combinations of keywords. These included 'LLM reverse engineering', 'binary code understanding', 'malware analysis LLM', 'LLM decompiler', 'multi-agent LLM', 'semantic reconstruction', 'tool use' and 'static malware analysis'. The inclusion criteria focused on peer-reviewed publications, conference proceedings and relevant preprints published from 2018 onward. This period was selected to capture recent developments in LLM-based program analysis, binary analysis, malware forensics and multi-agent systems. Studies were included if they addressed code-level analysis, RE automation or LLM-based reasoning over program representations. Studies were excluded if they were non-academic or did not focus on binary code analysis or LLM-assisted malware research. Both forward and backward snowballing techniques were applied to identify additional relevant studies. The final set of selected studies was synthesised across multiple domains to establish a theoretical and methodological foundation for evaluating multi-agent coordination strategies in static malware RE.

2.2 Usage of LLMs in Binary Code Understanding

Binary code understanding is a core task in RE, malware analysis and vulnerability discovery. It remains difficult because compilation removes source information such as function names, variable names, comments and clear control structures. Decompilers such as Ghidra or IDA Pro can recover pseudocode, but their output often lacks semantic clarity. This is notably problematic when binaries are stripped, optimised or affected by compiler transformations [5].

Recent research has evaluated whether LLMs can support this interpretation problem. One benchmark studied two binary code understanding tasks: function name recovery

and binary code summarisation [5]. The evaluation included eight coding oriented LLMs, eight general-purpose LLMs and four expert models. CodeLlama-34B-Instruct performed best in function name recovery while ChatGPT achieved the strongest result in binary-code summarisation. The same study showed that context strongly affects model performance. Using strings and identifiers improved both tasks, but excessive symbol information could reduce summarisation quality when longer pseudocode exceeded context window limits [5].

Similar research has examined whether LLMs can be trained specifically for binary decompilation. LLM4Decompile [11] distinguishes between direct binary decompilation and refinement of traditional decompiler output. In the direct approach, the model translated disassembled assembly into source code. In the refinement approach, it improved Ghidra pseudocode for readability and re-executability. The LLM4Decompile model family ranges from 1.3B to 33B parameters and was fine-tuned on 15 billion tokens. The 6.7B end-to-end model achieved a 45.4% successful decompilation rate on HumanEval-Decompile and 18.0% on ExeBench. The refined approach improved re-executability by 16.2% over the end-to-end model [11].

These results show that domain-specific training can improve binary decompilation. However, benchmark design strongly affects performance. HumanEval-Decompile contains standalone functions, while ExeBench contains real-world C functions with user-defined structures and dependencies [11]. This distinction matters for malware analysis because malware samples often contain stripped symbols, packed sections or incomplete contextual evidence. Therefore, strong performance on standalone benchmark functions does not necessarily imply reliable malware RE. The LLM4Decompile study is relevant because it shows the value of domain-specific training while also highlighting the need for evidence grounding and verification [11].

SLaDe [12] provides another example of specialised decompilation. It uses a 200M parameter sequence-to-sequence transformer combined with type inference and was evaluated on more than 4,000 ExeBench functions. The study showed that SLaDe outperformed both Ghidra and ChatGPT on several accuracy and readability measures, including optimised x86 functions. The results indicate that smaller, task specific models can outperform larger general-purpose models when the task requires binary specific representations. However, SLaDe still fails when arguments, structures or arithmetic

operations are inferred incorrectly, showing that specialised models still require verification [12].

Evaluation focused work gives a more cautious view of LLM-based decompilation. DecompileBench [13] study evaluates 23 400 functions from 130 real-world programs and compares six industrial decompilers with six LLM-powered approaches. The study found that LLM-based approaches can improve code understandability but had 52.2% lower functionality correctness than commercial tools. This distinction is important for security analysis because readable output may still misrepresent program behaviour. DecompileBench also uses an LLM-as-a-judge method validated against expert ratings, showing that automated evaluation can be useful when calibrated against human judgement [13].

LLM-assisted decompiler refinement shows similar benefits and risks. DeGPT [4] uses a three-role mechanism consisting of a referee, advisor and operator to improve Ghidra output. The framework reduced analyst cognitive burden by 24.4% and 62.9% of generated comments provided practical semantic information. However, 21.9% of related LLM responses were harmful because they changed or risked changing function semantics. This shows that LLMs can support RE, but their outputs still require validation.

Malware-specific research further illustrates the difficulty of reliable binary interpretation. MALSIGHT [14] study focuses on binary malware summarisation and introduces a 0.77B-parameter model trained for malware summaries called MalT5. Its dataset contains nearly 90 000 malware source functions across 20 malware types. The study shows that malware pseudocode can become substantially longer and less interpretable after decompilation. In one example, 20 lines of source code became 117 lines of pseudocode with 29 additional calls and 29 additional if-statements. This shows why behavioural interpretation from decompiled malware code remains difficult.

Literature shows that LLMs can support binary code interpretation, decompiler refinement and malware summarisation. Specialised models improve performance on decompilation tasks, but readable outputs may still be semantically incorrect or unsupported by evidence. These limitations motivate workflows that separate artefact

extraction, interpretation, verification and report generation. They also support evaluating whether multi-agent coordination can improve static malware-analysis reports.

2.3 Multi-Agent Large Language Model Systems

The limitations of single-agent LLM systems have become evident in complex reasoning tasks such as malware RE. While modern LLMs demonstrate strong capabilities in semantic reconstruction and code understanding, their performance degrades in tasks that require sustained logical consistency. A common failure type is cascading error propagation, where an incorrect assumption propagates through the reasoning process, ultimately leading to fabricated or inconsistent conclusions. These limitations have motivated research into MAS that distribute reasoning across multiple specialised agents [15] [16]. An agent, in this context, refers to an autonomous entity capable of performing tasks through reasoning and interaction.

This transition reflects a broader shift in AI from monolithic models to coordination-driven intelligence. Rather than relying solely on increasing parameter counts, MAS may achieve improved performance by turning complex tasks into smaller, manageable subtasks handled by agents with distinct roles. Research suggests that MAS performance depends not only on model capability, but also on coordination, role specialisation and interaction design [7] [17]. Distributed reasoning can help divide complex evidence interpretation into smaller subtasks in analytical tasks like binary code understanding or forensic analysis. This may reduce context overload and improve consistency when compared with a single agent handling the full analysis alone.

At the core of MAS is the concept of structured collaboration, where agents interact according to predefined coordination strategies. These systems typically organise agents into architectures such as sequential, hierarchical or decentralised networks [7] [17]. Sequential workflows process tasks in a linear fashion, where each agent refines or transforms the output of the previous agent. While effective for decomposable tasks, such pipelines are vulnerable to cascading error propagation, as inaccuracies in early-stage reasoning influence all subsequent steps [15] [16].

To mitigate this, hierarchical architectures introduce a supervisory agent responsible for task decomposition, delegation and integration of results, improving consistency through

centralised validation [18]. In contrast, decentralised approaches allow agents to communicate directly without centralised control, offering flexibility but often suffering from communication overhead and reduced coherence [17]. Furthermore, emerging consensus-based or debate mechanisms have been shown to further refine accuracy by fostering adversarial interactions between agents to identify logical inconsistencies [10].

A key strength of MAS lies in role specialisation, where agents are assigned responsibilities aligned with specific stages of a task. In cybersecurity applications, this may involve separate agents for retrieval, extraction, classification, behavioural interpretation or result validation. LAMPS demonstrates this principle in malicious PyPI package detection [8]. The system uses four coordinated agents for package retrieval, file extraction, classification and verdict aggregation. In a package-level comparison against single-agent baselines, LAMPS achieved 0.924 accuracy, compared with 0.766 for both single-agent configurations [8]. These results support the importance of specialised agents in static security analysis workflows, where malicious behaviour must be inferred from code artefacts.

Another critical advancement in MAS is the integration of self-reflection and feedback mechanisms. Reflective multi-agent systems enable agents to critique both their own and other agents' outputs, introducing iterative correction loops that improve reasoning quality [19]. This approach addresses the challenge of error attribution by enabling systems to identify and correct faulty reasoning paths during execution.

The effectiveness of MAS is also closely tied to advances in context management and memory usage. As agents exchange information, the accumulation of intermediate results can lead to context bloat where excessive or irrelevant data degrades model performance. This phenomenon is closely related to the "lost-in-the-middle" effect observed in long-context reasoning tasks [20]. Recent work in context engineering argues that context must be actively selected, simplified and organised rather than merely expanded [18]. This is relevant to MAS because multi-agent workflows must manage how intermediate outputs are stored, retrieved and passed between agents.

From a theoretical perspective, the development of MAS aligns with emerging trends in AI between reactive and deliberative reasoning processes. Standard LLM inference resembles fast and heuristic reasoning, where outputs are generated in a single forward

pass without explicit planning. In contrast, MAS enable more structured reasoning through task decomposition, iterative refinement and coordinated interaction between specialised agents [15]. This shift toward deliberative reasoning is further reflected in agent-based frameworks that separate planning from execution. For example, ReMA [21] introduces a meta-level reasoning process in which dedicated planning agents dissect complex, multi-step tasks into subtasks to guide execution agents. Such approaches suggest that multi-agent coordination can effectively extend LLM reasoning beyond reactive generation toward more controlled and systematic problem-solving.

MAS reliability is still limited by agent misalignment, communication overhead and coordination complexity [22]. Existing evaluation methods also provide limited evidence on how different coordination strategies behave in specialised security analysis tasks [10]. This creates a research gap in static malware RE, where sequential and hierarchical MAS architectures have not been sufficiently compared.

2.4 Tool-Using Workflows in LLM Systems

Recent research has extended LLMs from passive text generators into agents that interact with external tools. In these workflows, the model does not only interpret pre-existing input. It can also issue tool calls, observe results and update its reasoning. This follows the perception–reasoning–action pattern used in LLM-based agent systems, where reasoning is combined with external interaction [23]. ReAct demonstrates this pattern by incorporating reasoning traces with actions, allowing the model to retrieve or interact with external information during problem solving [24]. Toolformer follows a different approach by training language models to decide when and how to call external APIs [25]. HuggingGPT extends tool use through orchestration, where an LLM plans tasks, selects external models, executes subtasks and integrates results [26].

At a structural level, agents that use tools typically follow a reasoning–action–observation loop. The model generates an intermediate plan, executes a tool call, observes the result and updates its reasoning accordingly [24]. This process allows LLMs to operate beyond the constraints of their internal knowledge and finite context windows, effectively extending their capabilities through external computation and data retrieval [25] [27]. Such workflows are often implemented through planning-based or reactive strategies. In

such workflows, agents either decompose tasks into explicit sequences of actions or dynamically decide which tools to invoke based on intermediate results [24]. This flexibility enables LLM systems to handle long tasks that require multiple dependent steps, which are otherwise difficult to solve using single-pass generation.

This pattern is especially relevant to RE and malware analysis. Static malware analysis depends on evidence such as imports, strings, metadata, sections, function information and disassembly. A tool-using LLM workflow can retrieve these artefacts instead of relying only on model memory. This supports grounded inference, where analytical claims can be checked against evidence produced by RE tools. Such grounding is important because unsupported assumptions about program behaviour can lead to incorrect capability inference.

However, tool use also introduces reliability and security risks. Tool-augmented agents may select inappropriate tools, misinterpret tool outputs or propagate incorrect observations into later reasoning. Studies on tool-using agents also show risks under prompt injection and adversarial instruction scenarios [28] [29]. Therefore, tool use increases analytical capability but also expands the system's failure surface. These risks make verification and coordination important. One mitigation strategy is to introduce validation steps, where intermediate conclusions are checked against tool-generated evidence before acceptance [19]. Another strategy is role separation where planning, execution, interpretation and verification are assigned to different agents [23] [26]. This reduces the burden on a single model and makes the workflow easier to inspect.

Recent advances in multi-agent frameworks further extend this idea by assigning tool interaction responsibilities to specialised agents, enabling coordinated access to various tools [28]. Such role-based decomposition improves modularity and allows separation between data acquisition, reasoning and validation. In complex analytical fields, this enables systems to process multiple aspects of a problem simultaneously while maintaining structured communication between agents. However, it also introduces additional challenges, including synchronisation overhead, communication bottlenecks and difficulties in maintaining global semantic consistency across distributed reasoning processes [15].

Workflows that use tools represent an important shift in how LLMs are applied to complex problem solving tasks. They transform LLMs from interpreters into active users of external evidence. At the same time, they introduce challenges related to reliability, coordination and verification. These trade-offs are especially important in malware RE, where incorrect tool usage or misinterpretation of outputs can lead to inaccurate capability inference. This highlights the need for multi-agent coordination strategies that balance flexibility with control.

2.5 Large Language Models in Malware Forensics and Semantic Triage

Recent studies demonstrate that LLMs can analyse source code structure and logic to identify malicious behaviours, detect hidden patterns and support RE processes [5] [6]. Recent review of LLMs for software security also identifies malware analysis, code analysis and RE as major application areas for LLM-based cybersecurity research [30]. This is relevant to forensic analysis because malware investigation often requires more than classification.

Earlier AI-driven malware analysis mainly focused on classification. Static approaches use features such as opcode sequences, API calls, control-flow graphs and entropy to distinguish malicious and benign samples. Dynamic approaches observe runtime behaviour including system calls, memory activity and network traffic [31]. These methods can improve detection, but they often produce labels rather than detailed behavioural explanations. Transformer-based malware detection extends this direction by modelling sequential and contextual relationships in malware artefacts. Transformer models can process API sequences, command-line parameters, behavioural traces and other structured representations [32]. This supports stronger malware detection, especially when malicious patterns depend on dispersed or long-range data relationships. However, these approaches are still primarily detection oriented rather than report oriented.

The shift from classification towards semantic explanation is also reflected in malware-specific summarisation research. As discussed earlier, MALSIGHT shows that LLM-based models can generate human-readable descriptions from decompiled malware code

[14]. However, malware summarisation alone does not guarantee forensic reliability. Generated explanations must still be traceable to extracted evidence, especially when used for capability inference or analyst-facing reports.

Beyond static analysis, LLMs have also been applied to dynamic behavioural analysis, leading to what is often referred to as semantic triage. Traditional dynamic analysis systems generate large volumes of data that is often difficult to interpret in isolation [33]. Such data typically includes system call traces, registry modifications and network activity logs. Semantic triage addresses this limitation by transforming the raw data into structured representations that capture behavioural intent of a program. Related work also shows the value of mapping security evidence to structured threat frameworks. One study uses GPT-based models to map CVE descriptions to MITRE ATT&CK techniques [34]. However, this work operates on vulnerability descriptions rather than binary artefacts or malware execution traces.

Forensic investigation research further highlights the importance of evidence grounding. PROVSEEK [35] combines LLM reasoning with system provenance data to support automated threat investigation. Its agents retrieve evidence, refine hypotheses and validate outputs against provenance logs. The study reports a 34% improvement in contextual precision and recall for intelligence extraction. It also reports 22% higher precision and 29% higher recall for threat detection compared with baselines. PROVSEEK differs from the present thesis because it analyses provenance logs and threat intelligence rather than static malware binaries. Its relevance lies in its verification-first design. Each generated claim is tied to verifiable system evidence, reducing the risk of unsupported conclusions. This principle is directly relevant to static malware RE, where report claims must be grounded in extracted binary artefacts.

Evaluation centred research reaches a similar conclusion. MalEval evaluates LLMs for fine-grained Android malware behaviour auditing [36]. It decomposes auditing into function prioritisation, evidence attribution and behavioural synthesis. This is relevant because it evaluates whether LLMs can produce evidence backed behavioural explanations rather than only classify malware. MalEval shows that current LLMs remain limited in this setting. The study finds that LLMs can capture typical malicious patterns and partially reduce analyst workload. However, they still struggle with coherent attack-chain reconstruction, surface-level reasoning, evidence attribution and behavioural

synthesis [36]. These findings show that malware auditing requires traceability, structured evaluation and evidence validation.

2.6 Summary of Findings

The reviewed literature indicates that LLMs have become increasingly relevant to software analysis, binary-code understanding and malware RE. In binary analysis, they can support code summarisation, decompiler refinement and malware summarisation. However, the evidence also shows persistent reliability limitations. Readable outputs may still be semantically incorrect, incomplete or unsupported by binary evidence. Specialised models and benchmarks have improved the evaluation of binary-to-source translation and binary code understanding. These studies suggest that domain-specific training can improve decompilation performance. However, they also show that benchmark design strongly affects those results. Performance on standalone functions does not necessarily translate to reliable analysis of complex malware binaries.

Tool-using LLM workflows extend model capabilities by allowing agents to retrieve evidence, invoke external tools and update their reasoning during analysis. This is relevant to malware RE because static analysis depends on external artefacts such as imports, functions and disassembly. At the same time, tool use introduces reliability and security risks like misinterpretation of outputs. Therefore, tool use requires verification and controlled coordination. The literature on MAS suggests that role specialisation, task decomposition and structured coordination can support complex reasoning workflows. Cybersecurity oriented systems show that specialised agents can be effective in static security analysis tasks. However, MAS are not automatically superior to single-agent systems. Their reliability depends on coordination strategy, communication quality, context management and validation mechanisms.

Existing work has examined binary understanding, tool use, MAS and malware auditing, but these areas are often studied separately. To the author's knowledge, prior work has not evaluated how different multi-agent coordination strategies affect static malware analysis report generation and quality. This gap motivates the present study, which evaluates different coordination configurations using ground truth, manual review metrics and an automated LLM-as-a-judge auditor.

3. Research Methods

This research adopts a controlled experimental methodology to evaluate the effectiveness of MAS in automated malware RE and analysis. The objective is to compare how different agent configurations generate structured malware analysis reports from the same binary evidence. The system is implemented in Python using the CrewAI framework version 1.9.3 [37], which orchestrates LLM-based agents with distinct roles and task-specific objectives. The system employs a mixed LLM configuration, assigning different models to separate stages of the analysis pipeline. To ensure evidential grounding, the system integrates the RE framework radare2 [38] which enables the extraction of artefacts and string data from malicious binaries.

The evaluation was conducted on a custom dataset of 15 malware samples. The dataset was selected to provide behavioural, structural and platform diversity rather than large scale statistical coverage. For each sample, a ground truth analysis was constructed through a semi-automated process combining tool-derived artefacts and LLM-assisted interpretation. Outputs from multiple LLM systems were compared to identify consistent claims that are manually validated to ensure all retained information is supported by verifiable evidence.

Each generated report is decomposed into individual claims and categorised as hard facts, supported interpretations or unsupported speculation. An LLM-as-a-judge approach [39] is implemented through a specialised Auditor agent executing an 11-step evaluation protocol. To mitigate evaluation bias, all assessments are supported by manually validated ground truth. The specific technical criteria for evidence grounding are detailed in section 4.3.

Performance is measured across five domains: structural analysis, semantic reconstruction, API resolution, capability inference and malware attribution. Experiments are conducted under multiple configurations to evaluate the impact of agentic collaboration on reasoning quality. All configurations are evaluated under matching conditions using the same dataset, ensuring that observed performance differences can be attributed to the system design rather than external factors.

3.1 Experimental Environment

To conduct experiments on LLM systems in automated malware RE, a controlled and reproducible environment is required. CrewAI framework [37] is utilised to manage agent registration, task delegation and agent communication. This allows the focus to remain on evaluating multi-agent reasoning strategies rather than system infrastructure.

The system employs a mixed LLM configuration to balance computational efficiency and reasoning performance. A local LLM instance is used during the extraction phase to avoid the cost associated with high-volume API interactions. While the subsequent analysis phase uses an external API, the primary security advantage of this configuration lies in the containment of raw malicious artefacts. The local extraction phase ensures that executable malware artefacts remain within the virtualised environment. Therefore, the data transmitted to the external Gemini API [40] is restricted to sanitised technical telemetry and metadata. The architecture significantly reduces the data exposure possibility by ensuring that the external service only receives textual descriptions rather than the executable artefact itself.

The model used for the local extraction phase is DeepSeek-Coder-V2 [41] hosted via Ollama [42]. DeepSeek-Coder-V2 was selected for its architecture, which provides strong performance in interpreting assembly code and structured outputs compared to general-purpose models of similar size [43]. After installing the Ollama [42] runtime, the model was served on a local port, allowing agents to communicate via a REST API. This interface is limited to the local loopback address, ensuring that all communication remains entirely internal to the host system and is not transmitted over an external network. For the analysis and synthesis phases, Gemini 2.5 Flash-Lite [44] was accessed via Gemini API [40]. This model is selected for its large context window and output speed, which are necessary for processing artefact data generated during the extraction phase. During analysis, agents operate on shared contextual inputs derived from extracted artefacts. The ability to maintain a comprehensive view of this data reduces context fragmentation and supports consistency in the final analysis.

The host system is equipped with hardware designed to support computationally intensive tasks. Host machine contains AMD Ryzen 7 7800X3D 8-Core Processor, an AMD Radeon RX 9070 XT graphics card (16 GB VRAM) and 32 GB of RAM. Due to the

security risks associated with malware analysis, the entire pipeline was deployed within a virtualised environment. In particular, the system employs an Ubuntu 24.04 LTS [45] operating system via Oracle VirtualBox [46] virtualisation layer. While the underlying hardware is capable of high performance, the virtualisation layer introduces computational constraints that limit direct hardware utilisation, particularly GPU acceleration for the local LLM. This trade-off is accepted to prioritise system isolation and ensure that the host environment remains protected from potential compromise.

To bridge the gap between agents and binary data, a custom Forensic Extraction Tool was implemented in Python. This tool utilises the r2pipe [47] library, which provides a programmatic interface to the radare2 framework [38]. It executes a predefined sequence of analysis commands to retrieve structural artefacts from the binary. The extracted data is then captured and stored in a standardised text format for later processing.

Additionally, a custom Forensic Context Reader component was developed to assist with structured information exchange between agents. The extraction phase produces text-based forensic dumps in which analysis commands and their corresponding outputs are recorded sequentially. Rather than ingesting the entire forensic dump, the Context Reader enables targeted retrieval of relevant artefacts by selectively parsing command-output pairs based on the needs of analysis agents.

This approach reduces unnecessary context ingestion, improves processing efficiency and ensures that agents operate on task-relevant information. As a result, all reasoning remains grounded in verified binary data while maintaining a consistent chain of evidence throughout the analysis pipeline. Each experimental run follows a standardised workflow consisting of artefact extraction and multi-agent analysis. The automated evaluation protocol is executed separately from the report generation workflow.

3.2 Multi-Agent Framework Selection

A multi-agent framework provides the infrastructure required to coordinate interactions between autonomous agents within a workflow. In analytical and cybersecurity domains, such frameworks enable the decomposition of complex tasks across specialised agents. Recent work demonstrates that multi-agent systems can structure workflows into coordinated layers to automate complex processes such as threat hunting while

maintaining operational control [9]. In this study, the framework supports the construction of a coordinated system. Within this system, agents perform distinct roles ranging from artefact extraction to semantic analysis. The selection of the multi-agent framework is guided by technical requirements and practical considerations. In particular, the framework must provide transparent access to agent reasoning, support efficient development and integrate within a Python-based environment. Transparency is particularly important in a forensic context. Inspecting intermediate reasoning steps is necessary to identify potential errors or hallucinations.

In addition, the analysis pipeline requires deterministic and traceable execution. Outputs from RE tools must be passed reliably between agents. This aligns with emerging multi-agent architectures that emphasise structured data exchange, persistent artefact storage and cross-agent validation to ensure reproducibility and auditability [9]. At the same time, the framework should be sufficiently mature and accessible to support quick and iterative development. This is because the focus of this research is on evaluating multi-agent reasoning strategies rather than building coordination infrastructure from scratch.

Three main multi-agent frameworks were evaluated for this study: Microsoft AutoGen [48], LangGraph [49], and CrewAI [37]. AutoGen is an open-source framework designed for scalable, event-driven MAS that support asynchronous communication and advanced memory handling. While this flexibility is beneficial in some environments, its architecture prioritises dynamic interaction over strict execution ordering. As a result, implementing sequential workflows requires additional coordination logic that increases complexity and reduces traceability.

LangGraph [49] offers fine-grained control over agent interactions by modelling workflows as directed graphs. This enables clear state management, branching logic and human-in-the-loop integration, making it flexible for complex analytical scenarios. However, this flexibility comes with increased implementation difficulty as control structures and state transitions must be clearly defined. For an experimental study focused on evaluating reasoning strategies rather than designing workflow architectures, this added complexity introduces unnecessary implementation burden.

Based on these considerations, CrewAI [37] was selected as the most appropriate framework for this study. CrewAI provides a role-based system for defining agent

responsibilities and coordinating tasks for sequential or hierarchical workflows. This aligns with the processing requirements of malware analysis pipelines, where tasks must be executed in a controlled and reproducible order. Unlike more flexible frameworks, CrewAI reduces the need for explicit coordination logic, allowing development efforts to focus on agent behaviour and reasoning.

A key advantage of CrewAI [37] is its support for YAML-based configuration, which separates agent definitions and task specifications from the core Python implementation. This design facilitates rapid iteration and experimentation. Agent roles and workflows can be modified without altering the underlying codebase. Furthermore, CrewAI includes built-in execution tracing, providing visibility into agent interactions and intermediate outputs. This directly supports the requirement for transparency and auditability in forensic analysis.

In addition to the primary frameworks evaluated, several emerging MAS were considered to provide a broader perspective on current multi-agent design approaches. PydanticAI [50] represents a type-safe approach to agent design, enforcing structured outputs through strict schema validation. While this is beneficial for ensuring data integrity in tasks involving structured artefacts, it lacks native support for multi-agent coordination and role-based orchestration. This makes it less suitable as a standalone framework for this study.

Similarly, OpenAI Swarm [51] uses a lightweight design based on agent handoff patterns where tasks are dynamically transferred between agents. While this approach offers simplicity and ease of debugging, it lacks advanced task management features and built-in transparency mechanisms. This is a limiting factor in complex forensic workflows requiring traceable reasoning.

Camel-AI [52] focuses on conversational collaboration between agents using role-based conversational interaction. Camel-AI can support exploratory reasoning and hypothesis generation. However, its dialogue-centric design is less suited to tool-integrated pipelines that require structured execution and deterministic processing of binary artefacts. These comparisons highlight a trade-off between flexibility, control and development complexity across existing frameworks.

Overall, based on the previously defined selection criteria, CrewAI [37] provides the most suitable balance of transparency, flexibility and implementation efficiency for this study. This enables the development of a controlled and auditable multi-agent malware analysis pipeline.

3.3 Malware Dataset

The effectiveness of an automated RE system should be evaluated against a dataset that reflects varied malware behaviours, platforms and evidence conditions. For this research, a custom dataset of 15 malware samples was curated across multiple families, operating systems and file formats. The dataset prioritises analytical depth over scale because the study evaluates reasoning quality, evidence attribution and report correctness rather than malware detection accuracy. Publicly available datasets such as EMBER [53] and MalS [14] provide large scale malware sets, but they are primarily designed for classification, detection or summarisation tasks. Such datasets do not provide the structured behaviour-level ground truth required for evaluating capability inference, hallucination rates and reporting supported by evidence.

Each sample in this study is therefore paired with a structured ground truth representation derived from RE artefacts, manual verification and multi-model validation. The dataset includes both well-defined behaviours and ambiguous cases, allowing the evaluation to test whether each system configuration preserves uncertainty when evidence is incomplete. The dataset is not intended to support broad statistical generalisation across all malware families. It is designed for controlled comparison of reasoning performance across selected evidence conditions. An overview of the selected samples is provided in Table 1.

Table 1. Overview of selected malware samples.

Sample	Type	Platform	Format
WannaCry	Ransomware / Worm	Windows x86	PE
Petya	Destructive malware / Ransomware	Windows x86	PE
Hive	Ransomware	Windows x86	PE / UPX-unpacked
Jigsaw	Ransomware	Windows x86	PE (.NET/C#)
WireNet	Spyware / RAT (Remote Access Trojan)	Linux x86	ELF (Executable and Linkable Format)
Chapros	Backdoor / Apache module	Linux x64	ELF
Proteus	Multi-functional malware	Windows x86	PE (.NET/C#)
Catapillar	Low-information shellcode/Blob	Unknown / x64 inferred	Raw binary blob
Amavaldo	Banking trojan / Masquerading trojan	Windows x86	PE / Delphi
Andromeda	Botnet dropper	Windows x86	PE / NSIS-wrapped
Reaver	Infostealer / RAT	Windows x86	PE DLL (Dynamic-Link Library)
EvilAnt	Spyware / RAT	Windows x86	PE / PyInstaller
MacMa	macOS backdoor / Spyware	macOS	macOS binary
Tarmac	Low-information macOS artefact	macOS-related evidence	Raw blob
Calisto	Trojan dropper / Disk-image lure	macOS inferred	Raw blob / DMG resource artefact

The selected samples cover different malware classes, platforms, file formats and evidence conditions. This allows the evaluation to test whether the system can adapt its reasoning across well-structured binaries, packaged runtimes, ELF modules and low-information raw artefacts. The selected samples are described below in terms of the analytical capabilities they are intended to evaluate.

3.3.1 Selected Malware Samples

The dataset includes the following samples selected to evaluate a specific analytical capability or evidence condition.

- **WannaCry/WannaCrypt**: WannaCry is a self-propagating ransomware worm that gained global attention in 2017 by exploiting the EternalBlue vulnerability in the Server Message Bloc protocol. It combines ransomware functionality with worm-like propagation capabilities [54]. This sample evaluates the system's ability to associate network indicators such as hard-coded URLs with control-flow logic and propagation mechanisms involving APIs related to communication.

- **Petya:** Petya is a destructive ransomware variant that overwrites the Master Boot Record and encrypts the Master File Table, preventing the operating system from booting. Unlike traditional ransomware that targets individual files, Petya operates at a lower system level [55]. This sample evaluates the system's ability to identify low-level system manipulation and cryptographic operations and to distinguish between standard file-level activity and potential destructive behaviours.
- **Hive:** Hive is a ransomware-as-a-service known for targeting enterprise environments including Windows and Linux systems. It employs complex encryption routines and supports large scale deployment [56]. This sample evaluates the system's ability to interpret complex control-flow structures and identify potential network-related capabilities in large scale binaries.
- **Jigsaw:** Jigsaw is a ransomware variant notable for its pressure mechanism, in which it deletes files incrementally over time if the ransom is not paid [57]. This behaviour introduces a temporal component to its execution logic. This sample evaluates the system's capability inference by identifying iterative file deletion patterns and linking them to ransomware behaviour.
- **Proteus:** Proteus is a multi-functional malware family capable of performing cryptocurrency mining, credential theft and secondary payload delivery. Its behaviour may vary depending on deployment context, making analysis less deterministic. This sample evaluates multi-stage execution analysis and the system's ability to reason under uncertainty when behavioural capabilities are not clearly observable. [58].
- **Amavaldo:** Amavaldo is a Windows banking trojan associated with Delphi-based malware and masquerading behaviour. The sample contains Delphi runtime artefacts and indicators resembling a legitimate NVIDIA SmartMax utility [59]. It evaluates the system's ability to distinguish application structure from malicious banking trojan behaviour that looks legitimate at first.
- **Andromeda/Gamarue:** Andromeda, also known as Gamarue, is a modular botnet and dropper family. The sample is represented as a Windows PE with NSIS-wrapped or overlay-backed characteristics, including payload staging evidence [60]. It evaluates the system's ability to identify installer-based

packaging, embedded payload extraction and loader-style behaviour rather than treating the file as a simple standalone PE.

- **Reaver:** Reaver is an infostealer or RAT-style Windows PE DLL. The sample includes evidence related to credential access, browser data theft, networking and service execution [61]. It evaluates whether the system can map low-level Windows APIs and DLL evidence, such as credential and cryptographic interfaces to higher-level infostealer and remote access behaviour.
- **EvilAnt:** EvilAnt is a Python-based spyware or RAT sample packaged as a Windows PE using PyInstaller. Its analysis requires recognising the bundled Python runtime, embedded archive structure and specific internal markers [62]. This sample evaluates the system's ability to identify packaged runtime artefacts and avoid mistaking a Python-wrapped executable for a conventional native PE program.

To extend the evaluation beyond Windows PE analysis, the dataset includes Linux-based, macOS-based and specialised samples.

- **Wirenet:** Wirenet is a cross-platform Trojan designed to steal credentials from web browsers and email clients on Linux and macOS systems. It targets stored authentication data and transmits it to external servers. This sample evaluates the system's ability to identify credential harvesting and data exfiltration behaviours within ELF binaries, including the use of APIs associated data extraction [63].
- **Chapros:** Chapros is a Linux-based malicious Apache module functioning as a rootkit, typically deployed as a shared object (.so) file. It injects malicious iframes into web traffic served by the compromised server. This sample evaluates the system's ability to analyse non-executable binaries and identify content injection behaviour through server-side APIs [64].
- **Catapillar**: Catapillar is an obfuscated and minimally documented malware sample sourced from theZoo dataset [65]. It lacks clear functional attribution and contains limited structural metadata. This sample is included to evaluate the system's ability to analyse low-level artefacts and infer potential behaviour in the absence of well-defined or easily interpretable functionality.
- **MacMa:** MacMa is a macOS backdoor associated with targeted spyware activity and capable of supporting file control and exfiltration from compromised systems

[66]. This sample evaluates whether the pipeline could analyse non-Windows spyware/backdoor artefacts and correctly reason about platform-specific evidence rather than forcing the analysis into a Windows PE model.

- **Tarmac:** Tarmac is a macOS trojan associated with fake Flash Player installer campaigns. Public reporting describes it as a second-stage payload distributed through malvertising, capable of collecting system information and receiving commands from a C2 server [67]. In this study, it is useful for testing whether the pipeline could preserve uncertainty when conventional executable structure, imports, entry points and control-flow evidence were absent.
- **Calisto:** Calisto is a macOS-related trojan dropper or disk-image lure represented through raw blob and DMG/resource-style artefacts. The sample includes macOS disk-image indicators, Apple_HFS style metadata and installation lure content [68]. It evaluates whether the system can recognise non-PE/non-ELF platform context and reason about macOS resource artefacts rather than forcing the sample into a conventional executable-analysis model.

These samples provide coverage across multiple malware classes, platforms and evidence conditions. This allows the evaluation to assess malware behaviour identification across varied static artefacts and sample formats. It also tests whether the system can preserve uncertainty, avoid unsupported inference and adapt its reasoning to different file formats and platform contexts.

3.3.2 Data Sourcing and Ground Truth

The samples were sourced from publicly available malware repositories theZoo [65] and vx-underground [69]. All samples were analysed within a secure, virtualised environment to prevent system compromise. For each sample, a structured ground truth object was constructed to serve as the benchmark for evaluation. Ground truth was generated using a semi-automated, consensus-based methodology. Artefacts extracted via radare2 [38] form the primary evidence base. These artefacts include structural metadata, section information, imports, strings, function-level information and other static indicators. Multiple high-capacity LLM systems such as ChatGPT 5.2 [70], DeepSeek-V3.2 [71] and Gemini 3 Pro [72] were then used to analyse the same artefact inputs independently.

The purpose of using multiple LLM systems was to identify candidate semantic interpretations and reduce dependence on a single model. However, model agreement was not treated as sufficient evidence by itself. All generated interpretations were treated as hypotheses and manually checked against tool-derived artefacts. A claim was retained in the ground truth only when it could be supported by extracted evidence. Claims that could not be traced to static artefacts were excluded or marked as unsupported.

To support reproducibility, the same artefact inputs were provided across all model-assisted review stages. Each retained ground truth claim was linked to extracted artefacts when sufficient and traceable evidence was available. This created an audit trail between the static evidence and the behavioural interpretation. Appendix 2 provides an example of the structured ground truth representation for the Hive [56] sample. Appendix 3 provides a traceability mapping example for WannaCry [54], linking selected binary constants and artefacts to their corresponding behavioural classifications.

The resulting ground truth representation contains three main categories: hard facts, interface-level observations and capability-level behaviours. Hard facts include properties such as architecture, file format and entry-point information. Interface-level observations include imports, libraries, APIs and other externally visible dependencies. Capability-level behaviours describe supported static inferences, such as encryption, persistence, propagation or exfiltration.

This validation process was designed to ensure that evaluation metrics were grounded in verifiable evidence. Since this study is limited to static analysis, the ground truth does not claim to confirm runtime behaviour. Instead, it represents the set of behaviours and artefacts that can be reasonably supported from the available static evidence.

3.4 Selecting Large Language Model

Prior studies on automated code analysis and repair commonly employ either flagship models, such as GPT-4-class systems, or models fine-tuned for software engineering tasks [6] [31]. Benchmarking research such as BinMetric shows that LLM performance in binary analysis remains uneven across tasks, architectures and optimisation levels [6]. This is relevant for malware RE where binary artefacts are incomplete, obfuscated or structurally complex. Therefore, this study adopts a mixed LLM configuration, utilising

DeepSeek-Coder-V2 [41] for the extraction phase and Gemini 2.5 Flash-Lite [44] for the analysis phase.

DeepSeek-Coder-V2 was selected for the extraction component due its reported performance in code intelligence benchmarks [43]. Unlike dense models of similar scale, its design enables the efficient activation of only a subset of parameters per token. This provides the reasoning density of a flagship model while remaining feasible for local deployment via the Ollama [42] runtime. This architectural efficiency is critical for processing the raw, unstructured output generated by RE tools like radare2 [38]. During the pilot developmental phase, similar local models such as llama3.1:8b [73] were observed to produce inconsistent interpretations and frequently failed to adhere to strict extraction constraints. These observations align with the reliability gap documented in DecompileBench [13] where smaller models often prioritise syntactic readability over functional correctness. This may lead to the hallucination of non-existent API calls when transitioning from low-level assembly to structured logic.

Using a local LLM allows the system to manage the interactions required for forensic parsing without the API rate limits associated with cloud-based inference. It is important for the extraction phase, which requires repeated and iterative refinement of large volumes of structured data. The author established a secure communication channel that ensures data privacy by hosting the model locally during the extraction phase. Although high-level analysis is later performed via an external API, the primary privacy gain of this hybrid architecture lies in strict data minimisation and attack surface reduction. The local LLM acts as a forensic filter, ensuring that the raw malicious artefacts never exit the isolated environment. In this configuration, the external API never receives the functional malware; it only receives textual representation extracted by the local model. This mitigates the risk of leaking active code to third-party infrastructure, as the cloud provider only processes a description of the threat rather than the threat itself.

For the analysis phase, Gemini 2.5 Flash-Lite [44] was utilised primarily due to its large context window and good performance in context reasoning tasks. Malware analysis frequently produces extensive artefact reports, including deeply nested control-flow graphs, string tables and long API traces. Prior research has identified performance degradation in large-context settings, where standard LLMs fail to reliably retrieve

information from the central portions of long inputs [20]. The ability to process artefacts within a single context window reduces the risk of context fragmentation.

In addition to performance considerations, the use of cloud-based models introduces practical cost constraints when processing large volumes of artefact data. High-context inference is specifically required during the analysis phase, where the agents must reason over the cumulative forensic data extracted from the binary. Unlike the extraction phase, the analysis phase requires the model to ingest and correlate the entire textual report to identify cross-functional patterns. Restricting high-context inference to the analysis phase ensures that computational resources are used efficiently while preserving the quality of semantic interpretation.

During preliminary evaluation, Gemini 2.5 Flash [74] produced less consistent outputs than Gemini 2.5 Flash-Lite [44] on the forensic analysis task defined in this study. This observation is specific to the experimental setup and should not be interpreted as a general performance claim. Based on preliminary results, Gemini 2.5 Flash-Lite [44] was selected for the analysis phase. The same model is also used for the LLM-as-a-judge [39] component. This keeps the evaluation technically consistent since the auditor is isolated from the workflow and receives only the final report and ground truth.

4. Experimental Setup

Evaluating the effectiveness of MAS in static malware RE requires a controlled and repeatable workflow. Each experiment involves processing malware binaries, extracting forensic artefacts, generating structured analysis reports and evaluating those reports against validated ground truth. Since the study compares multiple system configurations, the same analytical process must be executed repeatedly under stable conditions. The primary configurations are single-agent baseline, sequential workflow and hierarchical workflow. Sequential and hierarchical workflows also have a memory-enabled configuration to examine whether it affects report quality or evidence consistency.

An automated workflow was developed to make this process efficient, reproducible and easy to modify. In each run, the malware sample is first converted into a forensic artefact dump containing the binary information required by the agents. This output then becomes the input for the analysis stage, where the configured agent setup produces a malware analysis report. After the report is generated, it is passed to the LLM-as-a-judge component [39], which compares the findings against the corresponding ground truth and assigns the final score. Generated reports are also manually reviewed to assess reporting quality and evaluate the automated scoring component. The workflow was implemented modularly so that samples, agent setups and evaluation settings could be changed between runs without rewriting the entire pipeline. This made it possible to repeat the same experiment across different configurations while keeping the handling of each sample consistent.

4.1 Multi-Agent Architecture Design

As described in the previous chapter, the multi-agent system was implemented using the CrewAI framework [37]. CrewAI was used to define agent roles, assign task-specific goals and coordinate the communication between agents during the malware analysis workflow. The framework also supports different coordination strategies, which makes it suitable for comparing sequential and hierarchical multi-agent configurations under the same experimental conditions.

The architecture was designed around role specialisation. Instead of assigning the entire malware analysis task to a single model, the workflow divides the process into smaller analytical stages. Each agent is responsible for a specific part of the reasoning process from initial extraction to report generation. This division was intended to reduce unsupported assumptions and improve traceability by ensuring that each stage operates on a clearly defined type of evidence.

During early development, a smaller prototype was used to verify that the basic pipeline worked correctly. This configuration consisted of a Forensic Extractor agent and two analysis agents: Forensic Structural Hunter and Malware Capability Analyst. The Extractor Agent executed radare2 [38] commands to gather artefacts from the binary. The Forensic Structural Hunter was responsible for parsing these artefacts to establish the sample's architectural layout, section permissions and entry-point properties. Finally, the Malware Capability Analyst performed behavioural analysis, mapping the discovered APIs and strings to specific malicious intents such as encryption logic or persistence mechanisms. This prototype demonstrated that agents could collaborate to transform raw artefacts into forensic insights.

The extractor agent was executed using a locally hosted LLM, while the analysis agents used the Gemini API [40]. This mirrored the hybrid design of the system, where binary extraction is kept in a local environment and only the extracted artefacts are passed to the cloud-based analysis stage. This architecture was implemented to address forensic privacy, operational costs and performance efficiency even at a prototype scale. By localising the extraction, the system ensured that malware binaries remained within a secure virtual environment. This mitigated the risks of transmitting potentially malicious code to Gemini API [40]. Furthermore, this design optimised cost by performing resource intensive data scraping locally and reserving cloud tokens for semantic interpretation. Prototyping on this smaller scale was essential because full pipeline executions are time consuming and involve repeated, costly interactions with cloud-based LLM calls.

Final architecture consists of seven agents that participate in the report generation pipeline and one independent agent reserved for scientific evaluation. The report-generation pipeline includes the Forensic Extractor, Forensic Structural Hunter, Interface Resolution Analyst, Execution Semantics Analyst, Malware Capability Analyst, Verification and Normalisation Supervisor and Forensic Intelligence Editor. Separate from this workflow

is the Zero-Trust Forensic Auditor. In this study, the Auditor serves as an automated evaluation mechanism that assesses the final report's accuracy against the established ground truth. This separation ensures that the agents producing the report do not influence the scoring process, maintaining the integrity of the experimental results. Figure 1 provides an abstract overview of how agents are organised in the automated workflow.

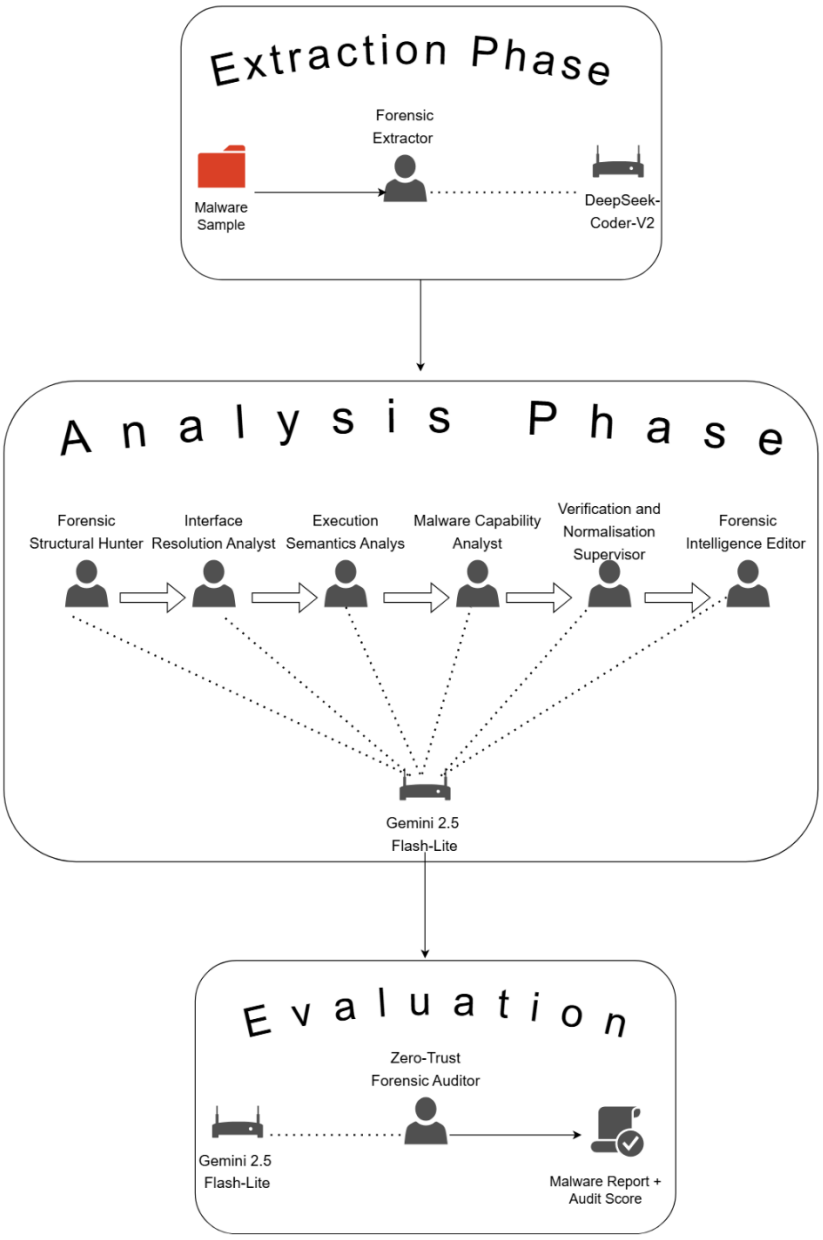

Figure 1. Abstract overview of the automated malware analysis workflow.

Figure 1 shows the three main phases of the implemented workflow: extraction, analysis and evaluation. The following subsections describe each agent and their role in the workflow.

Forensic Extractor

The Forensic Extractor acts as the data gathering part of the pipeline. Its role is to collect technical telemetry and structural metadata from the binary sample. It is configured as a command executor, meaning that it executes a predefined checklist of forensic commands in a fixed order. To perform these tasks, the agent uses a custom Python-based tool built around r2pipe [47] which provides programmatic access to the radare2 [38] RE framework. Unlike the later analysis agents, this agent does not interpret the binary, infer malware behaviour or make decisions based on the content of the extracted data. Its role is limited to triggering extraction commands that recover artefacts such as file headers, section metadata, symbol tables and embedded strings. The resulting artefact dump becomes the factual input for the downstream analysis agents. Agent's prompt is provided in Appendix 4.

Forensic Structural Hunter

The Forensic Structural Hunter performs the first analytical stage following extraction. Its primary task is to conduct structural analysis meaning the evaluation of a binary's logical and physical layout. To achieve this, the agent utilises a specialised ForensicReaderTool described later in this section. The tool was designed as a selective retrieval interface, allowing the agent to request only the specific block it needs from the forensic dump. The tool prevents the agent from ingesting the entire forensic dump file at once while also reducing token consumption. The agent processes these retrieved blocks to construct a structured forensic evidence graph. In this context, the graph is a JSON (JavaScript Object Notation) based logical schema where technical artefacts (nodes) are linked by their functional relationships (edges). For example, mapping an execution entry point to its specific code section and associated memory permissions. To maintain forensic integrity, the agent is guided by a scepticism rule in its prompt. The agent is instructed to avoid speculation and to report unknown or missing values as such rather than inventing new information. The speculation rule and the whole prompt in general is presented in Appendix 5.

Interface Resolution Analyst

The Interface Resolution Analyst is responsible for identifying the binary's external dependencies including imports, modules and runtime interfaces. Its primary function is to provide a list of external services the binary can invoke. Unlike the Forensic Structural Hunter, this agent does not directly query the forensic dump or invoke a separate tool. Instead, it operates on the evidence graph produced by the previous agent and contextual evidence provided to it. Its contribution lies in its specialised role definition and prompt constraints. A core requirement for this agent is fallback handling when structured import data is missing or unparseable. In such cases, the agent performs pattern matching on raw strings provided in its context. This allows the agent to identify important operating system artefacts, such as library names ending in .dll or .so or specific API export names. This fallback mechanism supports analytical continuity when standard header tables are empty or corrupted. To maintain forensic integrity, the agent adheres to evidence first rule: it is prohibited from inferring the presence of an API based on suspected malware capabilities. For example, if a sample appears to exhibit persistence behaviour, the agent cannot list registry related APIs unless they are found within the extracted evidence. This prevents capability assumptions from becoming unsupported interface claims. The agent's prompt is provided in Appendix 6.

Execution Semantics Analyst

The Execution Semantics Analyst is responsible for reconstructing the sample's visible execution model. It incorporates findings generated by the Forensic Structural Hunter and the Interface Resolution Analyst to bridge the gap between raw artefacts and behavioural logic. This agent does not invoke a separate tool. Its primary work is performed through prompting that synthesises evidence from previous stages. By integrating the structural map and API inventory into its reasoning, the agent can relate technical artefacts to higher-level program logic. For example, it is required to retrieve the hexadecimal entry point from the header data provided by earlier stages to anchor its reconstruction to the binary's starting state. This explicitly avoids generic labels such as 'entry0'. The agent identifies significant functions, tracks visible data flow and generates approximate pseudocode based on specific instruction patterns rather than general malware assumptions. To maintain forensic accuracy, the agent is also tasked with marking uncertain regions and assigning confidence scores to its interpretations. This ensures that

the final semantic model distinguishes evidence-supported behaviour from speculation and remains grounded in the extracted binary artefacts. The agent's prompt is provided in Appendix 7.

Malware Capability Analyst

The Malware Capability Analyst serves as the primary reasoning engine of the pipeline, responsible for translating technical evidence into high-level behavioural profiles. Unlike the earlier agents that focus on data recovery, this agent relies on specialised prompting and the shared context of the entire pipeline to synthesise a capability assessment. To maintain forensic precision, the agent follows a strict output separation rule that distinguishes malware class from malware family. In this architecture, class represents a functional category such as ransomware, downloader or stealer. A family refers to a specific known strain or attribution of specific malware. A malware family may only be assigned when family-specific evidence is present, such as unique strings, C2 (Command & Control) indicators, configuration artefacts or other known markers. If such evidence is absent, the family must be reported as unknown or expressed only as a soft comparison. Capability claims must also be linked to specific evidence categories, such as API evidence, string evidence, control-flow evidence or semantic evidence. By requiring every capability claim to be tied to a specific evidence category, the agent ensures that the final behavioural report remains a verifiable map of the binary's actual intent. Full prompt is shown in Appendix 8.

Verification and Normalisation Supervisor

The Verification and Normalisation Supervisor acts as the final factual checkpoint before report generation. Its task is to review the outputs of the previous agents, remove unsupported claims, correct inconsistent values, normalise equivalent terminology and produce a verified forensic summary. The agent uses the ForensicReaderTool to verify previous agents' claims against the forensic dump. This agent checks whether architecture claims match the observed register usage, whether capabilities align with APIs and strings and whether family attribution is supported by direct evidence. It also downgrades overconfident claims, removes contradictions, merges duplicate findings and ensures that every capability or family claim is supported by at least one evidence type. This step is

important because it prevents uncertain or rejected claims from being carried into the final report as established facts. Whole prompt is shown in Appendix 9.

Forensic Intelligence Editor

The Forensic Intelligence Editor converts the verified summary into a human-readable malware analysis report. Its task is not to introduce new technical claims, but to organise verified findings into a clear report structure. The report includes malware family assessment, structural analysis, semantic reconstruction, API categorisation, capability assessment and notes or warnings. The editor is instructed to avoid raw addresses except for the verified entry point, avoid dumping assembly instructions and include only facts that appear in the verified output. This keeps the final report readable while preserving evidence discipline. Full prompt is provided in Appendix 10.

Zero-Trust Forensic Auditor

The Zero-Trust Forensic Auditor is the LLM-as-a-judge component of the evaluation workflow [39]. The agent is executed separately from the report generation pipeline after the final malware analysis report has been produced. Its role is to compare the generated report against the manually validated ground truth. The auditor applies an 11-step evaluation protocol that is discussed later in section 4.3.2. Claims are classified as hard facts, supported interpretations or unsupported speculation. The auditor then assigns scores for structural analysis, semantic reconstruction, API or interface resolution, capability inference and attribution. It also applies penalties for hallucinated interfaces, incorrect hard facts, unsupported capabilities, missing supported capabilities, incorrect attribution and contradictions. Although the Auditor uses the same framework, it cannot access intermediate agent reasoning and does not participate in the report generation pipeline. Its input is limited to the final report and the corresponding ground truth, which separates the generation stage from the evaluation stage. The agent's prompt is shown in Appendix 11.

The multi-agent pipeline uses custom tools to connect the LLM agents with forensic artefacts generated from malware binaries. The first component is a custom Python based extraction tool built around r2pipe [47], which provides programmatic access to the radare2 [38] RE framework. This tool is used during the extraction stage to execute a predefined set of radare2 commands against the selected binary. The raw command

outputs are then appended to a forensic dump file. The tool does not interpret the binary; its purpose is to collect the same evidence blocks for every analysed sample. This design helps ensure that later analysis agents operate on a consistent evidence base.

The second custom tool is the ForensicReaderTool, which was developed as part of this thesis. It functions as a lightweight parsing and retrieval tool. It is used by the analysis agents to access evidence from the forensic dump. Instead of providing the entire dump at once, the tool retrieves a specific command block based on a requested key or alias, such as metadata or disassembly. This design prioritises context control by limiting the amount of technical data inserted into the model context. By selectively providing relevant evidence blocks, the ForensicReaderTool reduces context overload and helps keep agent reasoning focused on verifiable technical anchors.

This tool-based design improves both traceability and context management. Traceability is supported by requiring agents to include technical anchors such as raw hex addresses, symbol names or specific command outputs alongside every behavioural claim they make. Since these anchors are retrieved directly through the ForensicReaderTool, any finding in the final report can be traced back to a specific block in the original radare2 [38] output. The system also maintains a stable shared evidence base through the forensic dump path and evolving evidence graph. This allows agents to work from common extracted evidence while only ingesting the technical blocks required for their specialised tasks.

4.2 Automated Workflow Description

The automated workflow was implemented to execute the same analysis procedure across all selected malware samples and system configurations. Each run begins by selecting the malware sample, the corresponding output directory and the ground truth file used for later evaluation. The workflow is controlled through a central execution script, which defines the required input paths, agent configuration, coordination strategy and scoring parameters for the current experiment.

The first stage of the workflow is artefact extraction. In this stage, the selected malware binary is processed using the custom r2pipe-based extraction component. The tool executes a predefined set of radare2 [38] commands and stores their raw outputs in a

forensic dump file. This dump contains the evidence required by the downstream agents including binary metadata, header information, sections, imports, strings, functions, entry-point disassembly, cross-references, hashes and entropy-related information. The purpose of this stage is to create a stable evidence base before any LLM-based reasoning is performed.

After the forensic dump has been generated, the selected agent configuration is initialised. Depending on the experiment, the workflow may run a single-agent baseline, a sequential multi-agent setup or a hierarchical multi-agent setup. In the multi-agent configurations, the analysis agents do not interact directly with the malware binary. Instead, they access the generated forensic dump through the ForensicReaderTool, which retrieves only the command-output blocks needed for each task. This allows the agents to work with task-relevant evidence while avoiding unnecessary context expansion.

The analysis phase proceeds through the defined agent tasks. The structural analysis task first extracts hard technical facts and creates an evidence graph. The API/interface resolution task identifies supported interfaces from imports, strings and raw artefacts. The semantic reconstruction task then interprets the visible execution flow and major behavioural patterns. Based on these outputs, the capability inference task identifies supported malware behaviours, class and family attribution where possible. Before the final report is generated, the verification task checks the consistency of the intermediate findings, removes unsupported claims, normalises terminology and produces a cleaned forensic summary.

The final reporting task converts the verified findings into a structured static malware analysis report. This report includes the malware family assessment, structural analysis, semantic reconstruction, API categorisation, capability assessment and relevant warnings or limitations. The reporting agent is restricted to the verified output and is not allowed to introduce new claims that were not supported by the previous stages. To enforce this, the agent operates under an evidence-lock prompt configuration (see Appendix 10) that prevents the LLM from filling analysis gaps with common malware assumptions.

After report generation, the evaluation stage is executed. The Zero-Trust Forensic Auditor receives the final report and the corresponding manually validated ground truth. It applies the evaluation protocol and produces a JSON-based score sheet. The output contains

scores for structural analysis, semantic reconstruction, API or interface resolution, capability inference and attribution. It also contains the weighted overall score and explanations for the applied penalties. After the audit is completed, the JSON evaluation output is appended to the generated report file. As a result, each report contains both the generated malware analysis and the corresponding evaluation results, allowing experiment runs to be reviewed and compared across configurations. An example is provided in Appendix 12.

The workflow repeats the same analysis procedure while changing only selected parameters such as the sample, agent configuration, coordination strategy or scoring weights. As a result, the outputs from different configurations can be compared under consistent conditions.

4.3 Evaluation Methodology and Scoring

The evaluation of the proposed MAS follows a claim-based framework designed for static malware RE. Since the system produces structured malware analysis reports rather than executable outputs, the results cannot be evaluated using simple pass/fail tests. Instead, each generated report is assessed according to factual correctness, completeness, evidence support and the presence of unsupported or contradictory claims. This study uses an LLM-as-a-judge evaluation approach [39] implemented through the Zero-Trust Forensic Auditor described previously. The Auditor compares the final generated report against a manually validated ground truth object prepared for each malware sample. The ground truth contains verified hard facts, structural properties, interface-level observations, semantic behaviours, supported capabilities and attribution evidence. The Auditor does not evaluate the intermediate reasoning of the analysis agents. Its input is limited to the final report and the corresponding ground truth.

The Auditor produces a weighted score across five report domains: structural analysis, semantic reconstruction, API or interface resolution, capability inference and attribution. Each domain receives a score according to the evaluation rubric and the weighted total is calculated from these category scores. Penalties are then applied for issues such as hallucinated interfaces, incorrect hard facts, unsupported capabilities, missing supported capabilities, incorrect attribution and contradictions with the ground truth. A binary pass/fail classification would be too limited for this task. A malware analysis report may

contain correct structural facts while still missing important behavioural capabilities or overstating uncertain findings. For example, a report may correctly identify a sample as ransomware but incorrectly assign a specific malware family without family-specific evidence. Similarly, a report may correctly detect generic network capability but incorrectly describe it as confirmed C2 communication. For this reason, the evaluation is divided into separate scoring domains.

Manual review is performed separately at the claim level. It measures precision, recall, F1-score and hallucination rate by comparing report claims against the manually validated ground truth. This separates the Auditor's weighted rubric-based evaluation from the manual verification of factual correctness, completeness and unsupported claims. The Auditor's scoring behaviour is later reviewed against the manual findings to assess the reliability of the evaluation mechanism.

4.3.1 Performance Metrics

The system evaluates performance across five specialised domains. Each domain is assigned a weight to reflect its forensic importance in the final score. Structural Analysis (SA) assesses the correctness of hard technical facts including architecture, operating system, file format, entry point, packed/unpacked state and section or segment topology. Semantic Reconstruction (SR) evaluates the description of high-level program behaviour and visible control-flow patterns such as startup behaviour, runtime initialisation, loops, branching structures and major behavioural flows. API / Interface Resolution (AR) measures the precision of identified interfaces including imports, DLLs, shared libraries, syscalls, runtime modules, UI frameworks or other external dependencies. This metric penalises invented interfaces and rewards correctly abstracted interface families when exact APIs are not available. Capability Inference (CI) evaluates the ability to map technical evidence to malware behaviours. These behaviours can be file encryption, file deletion, persistence, credential access, network capability, data exfiltration, payload staging or anti-analysis behaviour. Attribution (AT) separately evaluates malware class and malware family identification. Malware class may be inferred from behavioural evidence, while malware family attribution requires family-specific indicators. These include ransom note text, unique strings, C2 indicators, configuration artefacts or other distinctive markers.

The final weighted score (WS) is calculated as a weighted average shown in equation (1). The weights used in this study are shown in Table 2. Each subscore has a range from 0 to 1. Each weight (w) is shown as a subscript in the equation.

$$WS = (SA \times SA_w) + (SR \times SR_w) + (AR \times AR_w) + (CI \times CI_w) + (AT \times AT_w) \quad (1)$$

Table 2. Performance metrics weights.

Metric	Symbol	Weight symbol	Weight value
Structural Analysis	SA	SA_w	0.2
Semantic Reconstruction	SR	SR_w	0.2
API / Interface Resolution	AR	AR_w	0.2
Capability Inference	CI	CI_w	0.25
Attribution	AT	AT_w	0.15

The weighting scheme was defined for this study rather than derived from statistical optimisation. It reflects the main evaluation priority of the research, where capability inference is treated as the central indicator of report quality. Structural Analysis, Semantic Reconstruction and API / Interface Resolution each receive a weight of 0.20 because they represent complementary evidence layers required for reliable behavioural interpretation. Capability Inference receives a slightly higher weight of 0.25 because it directly measures whether the system can infer malware behaviour from static evidence. Attribution receives a lower weight of 0.15 because family-level attribution is often unavailable or unreliable in static analysis without distinctive indicators. The same weighting scheme is applied to all configurations ensuring that comparisons remain internally consistent.

4.3.2 Audit Protocol

The Zero-Trust Forensic Auditor follows a structured 11-step evaluation protocol to ensure reports are assessed consistently across various malware samples and system configurations. Full evaluation protocol is provided in Appendix 11. The protocol begins by normalising both the generated report and the ground truth into a standardised classification. This is necessary because the same concept may be expressed using different terms, for example “C2”, “command and control” or “beaconing”.

After normalisation, the Auditor decomposes the generated report into individual claims and checks each claim against the validated ground truth. This process separates factual observations from analytical conclusions. Claims such as “Windows PE32 executable”, “x86 architecture” or “the binary contains a .text section” are treated as hard facts because they can be directly verified. Claims such as “the binary has network capability” or “the sample shows ransomware-like behaviour” are treated as supported interpretations only when they are grounded in evidence or semantic indicators. Claims are treated as unsupported speculation when they introduce behaviours, APIs, malware families or capabilities that are not supported by the extracted artefacts. For example, assigning a specific malware family without a family marker, or claiming confirmed C2 communication from generic networking DLLs alone, would be classified as unsupported speculation. This claim-level classification allows the Auditor to penalise incorrect hard facts, unsupported interpretations and hallucinated evidence separately instead of assigning a single undifferentiated score to the whole report.

The Auditor then checks whether any claim directly contradicts the manually validated ground truth. This contradiction check is applied before the individual scoring categories because direct conflicts with verified evidence are treated as high-severity errors. The evaluation is also platform aware, meaning that the Auditor adjusts its expectations according to the sample profile. For example, Windows PE binaries, Linux ELF files, shellcode and script-based malware are not expected to expose the same structures, interfaces or execution patterns.

After claims have been classified and checked for contradictions, the Auditor applies a predefined penalty schedule. Each scoring category starts from a maximum value of 1 and penalties are subtracted from the category most directly affected by the error. For example, hallucinated APIs, DLLs or modules reduce the API / Interface Resolution score while unsupported behavioural claims reduce the Capability Inference score. Incorrect execution descriptions reduce the Semantic Reconstruction score, and unsupported malware family attribution reduces the Attribution score. Missing information is penalised less severely than unsupported claims, since an omission is less misleading than introducing false forensic evidence. Penalty values were defined for this study to reflect the relative severity of different forensic errors. The values were assigned using the author’s expert judgement. Higher penalties were given to errors that could mislead

forensic interpretation, such as fabricated evidence or unsupported attribution. Same penalty schedule is applied across all configurations to support consistent comparison. The penalty types, deduction values and meanings are shown in Table 3.

Table 3. Penalty types and deduction values.

Penalty type	Value	Meaning
Incorrect hard fact / address penalty	0.15	Applied when the report gives an incorrect factual claim, such as wrong architecture, entry point, file format, section information or other concrete structural fact.
Hallucinated API / interface penalty	0.20	Applied when the report claims an API, DLL, module, library, syscall or interface that is not supported by the forensic evidence or ground truth.
Incorrect semantic claim penalty	0.20	Applied when the report gives an incorrect interpretation of program behaviour, execution logic, control flow or malware functionality.
Unsupported family attribution penalty	0.25	Applied when the report assigns a specific malware family without sufficient family-specific evidence.
Missing important interface / argument penalty	0.20	Applied when the report omits an important interface, argument, module or dependency that is present in the ground truth and relevant to the analysis.
Missing control-flow explanation penalty	0.15	Applied when the report omits an important loop, dispatcher, execution pattern or control-flow behaviour that is required for understanding the sample.
Unsupported capability penalty	0.20	Applied when the report claims a malware capability that is not supported by the evidence.
Missing supported capability penalty	0.10	Applied when the report fails to mention a capability that is supported by the ground truth.
Family abstention penalty	0.10	Applied when the report states Unknown or avoids attribution even though family-specific evidence is present.
Contradiction penalty	0.25	Applied when the report makes a claim that directly contradicts the ground truth or contradicts another verified claim.
Fabricated evidence penalty	0.30	Applied when the report invents specific evidence, such as strings, APIs, metadata, addresses or artefacts that are not present in the forensic dump or ground truth.

The final stage of the audit is JSON synthesis where the Auditor produces a machine-readable evaluation output. It contains scores for each subcategory, the weighted overall score, supported claims, partially supported claims, unsupported claims, omitted findings, contradictions and explanations for applied penalties. This JSON output is appended to the generated report file so that each experiment run contains both the malware analysis and its corresponding evaluation results. A complete example of this is shown in Appendix 12.

4.3.3 Manual Review

The automated scoring protocol was supported by a manual review process designed to validate both the generated malware analysis reports and the Auditor. The manual review had two purposes. First, it assessed the claim-level quality of generated reports by comparing their findings against validated ground truth. Second, it treated the Auditor's output as an object of validation. This involved checking whether the Auditor's classifications, penalties and score explanations were technically sound. This additional review layer was motivated by prior research that identifies evaluator hallucination, inconsistency, prompt sensitivity and bias as significant risks when using LLMs to score generated outputs [39].

The quantitative scores reported in the Results chapter are pipeline scores. They measure the quality of the malware analysis reports produced by each configuration. These scores are generated by the Auditor as the automated evaluation mechanism. Manual review therefore provides an additional validation layer rather than a replacement for the automated scoring protocol. During manual review, report claims were compared against the validated ground truth. Auditor's decisions were also checked against the same evidence base. This made it possible to identify cases where the Auditor correctly penalised unsupported claims, missed an error or applied a penalty too severely.

Manual review also assessed the structure and readability of the final malware analysis reports. This qualitative review considered whether each report followed a logical structure, avoided unnecessary raw artefact dumps, preserved evidence-based reasoning and presented findings in a form useful to a human analyst. In addition to qualitative assessment, manual review produced claim-level report-quality metrics. These metrics were not based on a binary decision about whether the sample was malware. Instead, each generated report was evaluated at the level of individual forensic claims. These claims included structural facts, API or interface findings, semantic interpretations, inferred capabilities, attribution statements and uncertainty statements.

Each report's findings were generalised into three categories: True Positive (TP), False Positive (FP) and False Negative (FN). TP means a report claim that matched a validated ground truth finding. FP means a report claim that was unsupported, incorrect or hallucinated and FN means a validated ground truth finding that the report omitted.

Precision (2), Recall (3), F1-score (4) and Hallucination Rate (5) are calculated as follows:

$$Precision = \frac{TP}{TP+FP} \quad (2)$$

$$Recall = \frac{TP}{TP+FN} \quad (3)$$

$$F1 - score = \frac{2 \times Precision \times Recall}{Precision+Recall} \quad (4)$$

$$Hallucination\ Rate = \frac{FP}{TP+FP} \quad (5)$$

In this thesis, Hallucination Rate is used as an operational metric for the proportion of reported claims that could not be verified against the ground truth. This use of the term is broader than strict fabrication, because it also includes unsupported or incorrect claims produced by over-interpretation of available evidence. Therefore, the metric measures the proportion of generated claims that were unverifiable, while detailed categorisation of false-positive causes remains outside the scope of the manual review.

All scores range from 0 to 1. Manual review did not replace the automated scoring protocol. Instead, it validated the Auditor's pipeline scores and penalty explanations against human forensic judgement. This ensured that automated evaluation remained subject to manual verification rather than being treated as final authority.

5. Results and Analysis

The results were obtained through multiple experimental runs on the dataset of 15 malware samples with validated ground truth. The primary objective was to evaluate the difference in forensic performance and architectural stability resulting from different MAS configurations. DeepSeek-Coder-V2 [41] was used for the local extraction phase, while Gemini 2.5 Flash-Lite [44] was used for the analysis and auditing stages. Across five pipeline configurations, the experiment produced 75 generated reports in total.

The first run for each sample used a single-agent prompt. What separates the single-agent approach from the MAS approach in this study is that the single-agent produced the full malware analysis report from the available forensic artefacts. Therefore, one agent cannot reason across specialised roles or build upon the intermediate outputs of distinct agents. The purpose of this run was to set a baseline to compare the MAS solution results against.

Second and third runs used a linear sequential pipeline. The standard sequential version passed task context from one step to the next while the third run additionally enabled CrewAI shared memory. Enabling shared memory allows the crew to retain and recall information from earlier task execution rather than relying only on task context from one step to the next. According to the CrewAI documentation, memory supports the storage and recall of execution related information [75]. In this study, memory was enabled at the CrewAI configuration level by setting `memory=True` while the framework's default memory storage and retrieval settings were used. No custom memory backend, retrieval limit, embedding configuration or memory-ranking strategy was defined. This means that the memory-enabled configurations evaluate the effect of enabling CrewAI's default memory mechanism rather than an optimised memory design.

The final two runs utilised a hierarchical setup. In this configuration, a separate manager agent delegated tasks to specialised worker agents. Like sequential runs, the hierarchical setup was tested with just task context and with memory enabled. This allowed for a direct evaluation of memory's impact on analytical consistency.

For each generated report, the Zero-Trust Forensic Auditor compared the output against the corresponding ground truth. Runs where the Auditor assigned a score greater than 0 were marked as "Completed Runs" while runs that the Auditor penalised with a strict 0 were marked as "Failed Runs". A failed run did not necessarily indicate that the system crashed or refused to generate a report. Rather, it indicated that the produced output lacked sufficient forensic content according to the evaluation protocol. Comparison of run success rates are shown below in Table 4.

Table 4. Comparison of run success rates across all configurations.

Configuration	**Total Runs**	**Completed Runs**	**Failed Runs**	**Success rate**
Single-Agent	15	13	2	86.7 %
Sequential	15	11	4	73.3%
Sequential memory	15	11	4	73.3%
Hierarchical	15	14	1	93.3%
Hierarchical memory	15	14	1	93.3%

The hierarchical configurations were the most reliable maintaining a completion rate of 93.3%. This suggests that hierarchical coordination provided the most stable execution pattern in this experiment. In contrast, the sequential configurations showed weaker stability with a completion rate of 73.3%. This may indicate that the linear workflow introduced additional failure points because later stages depended directly on earlier intermediate outputs.

The single-agent baseline achieved an 86.7% completion rate which was higher than both sequential configurations but lower than the hierarchical configurations. This suggests that the sequential workflows may have introduced additional failure points through task handoffs between agents. However, completion rate alone does not capture report quality, so the single-agent result should be interpreted together with the scoring and manual review results below.

Memory did not affect completion rates in this experiment. Both sequential configurations completed 11/15 runs and both hierarchical configurations completed 14/15 runs. Therefore, any effect of memory on report quality must be assessed through the scoring and manual review results rather than run completion alone.

5.1 Architectural Impact on Scoring and Efficiency

Completion rates indicate the reliability of each architecture, but they do not fully capture the accuracy of the generated reports. To evaluate the quality of completed reports, the weighted overall scores assigned by the Zero-Trust Forensic Auditor were examined. Since the results contains both completed and failed runs, Table 5 shows scores across all runs as well as scores calculated only from completed runs. This distinction separates overall robustness from the report quality of completed runs.

Table 5. Comparison of Auditor's weighted scores across all configurations.

Configuration	Average Score (All Runs)	Average Score (Successful Runs)	Median Score (All Runs)	Median Score (Successful Runs)
Single-Agent	0.447	0.516	0.350	0.380
Sequential	0.404	0.551	0.350	0.580
Sequential memory	0.399	0.543	0.350	0.530
Hierarchical	0.655	0.702	0.790	0.800
Hierarchical memory	0.688	0.737	0.810	0.840

Results show a clear score gap between the single-agent baseline and the hierarchical MAS configurations. The hierarchical memory configuration achieved the highest Auditor score with a completed run average of 0.737 and median of 0.84. Compared with the single-agent baseline successful run average of 0.516 and median of 0.380, it represents an improvement of 42.8% and 121.1% respectively. This suggests that when specialised agents are coordinated through a manager agent, the system produces reports with greater forensic depth than a single agent performing full analysis alone.

Regular sequential setup achieved a completed run average score of 0.551 and median score of 0.580. Compared with the single-agent baseline, this represents a 6.8% improvement in average score and a 52.6% improvement in median score. The sequential memory setup achieved a completed run average of 0.543 and median of 0.530. Compared with the single-agent baseline, this represents a 5.2% improvement in average score and a 39.5% improvement in median score. Although both sequential configurations improved over the single-agent baseline for completed runs, they remained below the

hierarchical configurations. These results indicate that multi-agent coordination strategy also affects whether specialisation improves the final report quality.

The scoring results also show that memory affected analytical quality more than run completion. While enabling memory did not change the completion rates, it improved the average score of completed runs in hierarchical configurations from 0.702 to 0.737. This approximately 5% increase suggests that memory acted primarily as a quality enhancing component rather than a stability enhancing one. However, memory was used with CrewAI's [37] default configuration and was not independently optimised. Therefore, this result should be interpreted as evidence for the potential usefulness of memory rather than as a final assessment of an optimised memory design. Memory appears to help agents correlate findings across different parts of the analysis, leading to more cohesive reports.

To identify which metrics contributed to the overall score differences, the category scores were also examined in Table 6. The metrics include all runs across all configurations, including runs ending with a 0.00 weighted score. These categories are Structural Analysis (SA), API/Interface Resolution (AR), Semantic Reconstruction (SR), Capability Inference (CI) and Attribution (AT).

Table 6. Average category scores for all runs across all configurations.

Configuration	SA	AR	SR	CI	AT
Single-Agent	0.540	0.359	0.443	0.358	0.563
Sequential	0.477	0.293	0.417	0.327	0.557
Sequential memory	0.473	0.28	0.413	0.317	0.593
Hierarchical	0.71	0.577	0.667	0.57	0.73
Hierarchical memory	0.75	0.657	0.693	0.66	0.593

Table 6 shows that the hierarchical configurations improved most of the core forensic categories, especially AR, SR and CI. Compared with the single-agent baseline, hierarchical memory increased AR from 0.359 to 0.657 and CI from 0.358 to 0.660. This suggests that hierarchical coordination helped the system convert low-level artefacts into behavioural conclusions. Sequential configurations did not improve most category scores over the single-agent baseline, indicating that role specialisation alone was insufficient when agents were arranged in an unmanaged linear workflow.

Memory had a mixed effect. In the hierarchical architecture, memory improved SA, AR, SR and CI showing that persistent context helped agents correlate evidence across stages. However, attribution did not improve under hierarchical memory and dropped from 0.730 to 0.593. This suggests that attribution behaved differently from the other categories: additional context could improve technical reconstruction while also making family-level attribution more cautious or less stable.

In addition to score-based quality, execution time was analysed to assess the efficiency of each configuration. Since failed runs with a score of 0 do not represent completed analyses, the comparison focuses on completed runs only. Table 7 presents the average score, median score, average runtime and median runtime for successful runs in each configuration. The median runtime shows the typical execution time, while the average runtime helps identify whether specific difficult samples caused longer processing times. In this context, time was measured for only the analysis stage of the pipeline. Extraction and evaluation phases were excluded from runtime measurement.

Table 7. Successful run score and runtime comparison across all configurations.

Configuration	**Average Score**	**Median Score**	**Average Time (s)**	**Median Time (s)**
Single-Agent	0.516	0.380	64.396	56.210
Sequential	0.551	0.580	81.482	72.130
Sequential memory	0.543	0.530	82.775	59.410
Hierarchical	0.702	0.800	172.766	89.210
Hierarchical memory	0.737	0.840	240.884	123.310

The results show a clear quality versus time trade-off. Hierarchical memory achieved the best absolute forensic performance but the additional benefit of memory over the standard hierarchical setup was relatively small. The successful run average score increased from 0.702 to 0.737 while the average runtime increased from almost 173 seconds to almost 241 seconds. This suggests that the hierarchical architecture itself contributed most of the quality improvement while memory acted as a secondary enhancement.

In practical terms, the non-memory hierarchical configuration offers the best balance between runtime and quality. Hierarchical memory is preferable when report quality

matters more than execution time. Therefore, execution time should be interpreted together with report quality: faster configurations are cheaper and more responsive, but they produced incomplete or lower scoring reports more often. Hierarchical configurations required more time but produced better scoring and more complete outputs according to the Auditor.

5.2 Impact of Malware Type on Forensic Performance

Malware samples were grouped according to their dominant analytical challenge to assess how sample characteristics influenced forensic performance. These groups reflect whether the pipeline had to reconstruct ransomware behaviour, identify packaged-runtime or loader artefacts, preserve uncertainty in low-information samples or interpret explicit symbols and APIs.

The ransomware/extortion group includes WannaCry [54], Petya [55], Hive [56] and Jigsaw [57]. The RAT/spyware/infostealer group includes WireNet [63], Reaver [61], MacMa [66] and Amavaldo [59]. The loader/packaged-runtime group includes Andromeda [60] and EvilAnt [62], and the low-information/non-standard artefact group includes Catapillar [65], Tarmac [67] and Calisto [68]. Proteus [58] and Chapros [64] were kept separate because they represent distinct single sample evidence. Proteus represents a .NET-based multi-functional malware sample, while Chapros represents an explicit Linux module. These two cases should be interpreted cautiously because each contains only one sample.

Table 8 reports averages across the MAS configurations only. The single-agent baseline is excluded because this section focuses on how malware type affected multi-agent performance. The averages include zero scored runs because these failures are part of the observed pipeline performance. The symbols represent the scoring metrics presented earlier: Structural Analysis (SA), Semantic Reconstruction (SR), API/Interface Resolution (AR), Capability Inference (CI) and Attribution (AT).

Table 8. Average Auditor scores by malware group across MAS configurations.

Malware Group	Avg. score	SA	AR	SR	CI	AT
Loader / packaged runtime	0.227	0.341	0.091	0.254	0.093	0.374
Ransomware / extortion	0.490	0.575	0.386	0.502	0.403	0.595
Low-information / non-standard artefact	0.533	0.606	0.445	0.541	0.459	0.624
RAT / spyware / infostealer	0.648	0.727	0.530	0.651	0.559	0.783
.NET multi-functional sample	0.782	0.830	0.731	0.774	0.759	0.828
Explicit-symbol module	0.899	0.906	0.875	0.867	0.908	0.919

Table 8 shows that the weakest group was the loader/packaged-runtime category. This group achieved the lowest average weighted score (0.227), with particularly weak AR (0.091) and CI (0.093) scores. This suggests that the multi-agent configurations struggled when the sample's important forensic meaning was hidden behind a wrapper, installer, overlay or bundled runtime. In these cases, identifying the file as a Windows PE was not enough. The system also had to recognise NSIS wrapping in Andromeda [60], Python/PyInstaller bundling in EvilAnt [62] or embedded payload staging and runtime-specific artefacts. The higher SA score (0.341) compared with AR and CI indicates that the system could sometimes recover surface-level structure but largely failed to interpret the runtime or loader context. This suggests that future versions of the workflow should include a dedicated runtime/package analysis step before API resolution and capability inference.

The ransomware/extortion group performed better than the loader group but remained relatively challenging. It achieved an average weighted score of 0.490 with SA (0.575) higher than AR (0.386) and CI (0.403). This indicates that ransomware samples depended on connecting several separate evidence types. The multi-agent configurations were able to recover some structural and semantic evidence, but overall interpretation was inconsistent. The agents often identified individual artefacts such as PE structure, strings, imports or runtime indicators. However, they did not always link these artefacts to specific behaviours such as registry persistence, cryptographic operations or ransom note logic. This was visible in samples such as WannaCry [54], Jigsaw [57] and Petya [55] where reports often identified parts of the executable structure but missed or under-reconstructed key behavioural links. The lower AR and CI scores therefore show that behavioural reconstruction remained a central weakness for this malware group.

The low-information/non-standard artefact group achieved a slightly higher average weighted score (0.533) than the ransomware group. Calisto [68], Tarmac [67] and Catapillar [65] did not always require detailed malware-behaviour reconstruction. Instead, they tested whether the system could preserve uncertainty when executable structure, imports and control flow were absent or ambiguous. The category scores were more evenly distributed in the loader and ransomware groups with SA at 0.606, AR at 0.445, SR at 0.541, CI at 0.459 and AT at 0.624. This indicates that the multi-agent configurations were moderately effective in this group, because they could sometimes recognise sparse artefact types or embedded resources and avoid overclaiming unsupported behaviours. However, the scores were not high enough to suggest robust analysis of these samples. The main challenge was preserving uncertainty: good performance depended on identifying the limited available evidence and then abstaining from unsupported API, entry-point, control-flow or malware-family claims.

The RAT/spyware/infostealer group performed better than the previous groups, achieving an average weighted score of 0.648. This group included WireNet [63], Reaver [61], MacMa [66] and Amavaldo [59]. Compared with the loader, ransomware and low-information groups, it achieved stronger scores across most evaluation categories, including SA (0.727), SR (0.651) and AT (0.783). Its AR (0.530) and CI (0.559) scores were also higher than those of the weaker malware groups, suggesting that these samples provided more recoverable behavioural anchors. These included recognisable APIs,

symbols, Java Native Interface evidence, Windows service artefacts, credential-access interfaces, X11 functions or banking-trojan runtime indicators. However, the scores were still not uniformly high, indicating that these samples were not simple to analyse. Their evidence was often more directly recoverable than in loader or ransomware samples but still required correct mapping into behaviours such as credential theft, remote access or keylogging.

The highest scores were observed in the .NET multi-functional sample and explicit-symbol module groups, but these results should be interpreted cautiously because each group contains only one sample. Proteus [58] achieved a high average weighted score of 0.782, with strong subscores overall. This shows that Proteus' .NET structure and visible artefacts were easier for the system to classify than wrapper-based samples. Chapros [64] achieved the strongest group-level result overall, with an average weighted score of 0.899 and high scores across SA (0.906), AR (0.875), SR (0.867), CI (0.908) and AT (0.919). This suggests that its ELF/Linux structure, Apache-module context, symbols and API evidence gave the system clear anchors for structural, semantic and capability reconstruction.

Results show that forensic performance was influenced not only by malware type, but also by how relevant evidence was exposed inside each sample. Loader and packaged-runtime samples were the most difficult because the system first had to recognise the wrapper, installer or bundled runtime before reasoning about the underlying behaviour. Ransomware samples were challenging because their interpretation depended on linking several evidence sources including APIs, embedded resources, cryptographic indicators and ransom-related artefacts. Low-information samples presented a different challenge: the correct behaviour was often to preserve uncertainty and avoid unsupported claims rather than produce a detailed reconstruction. RAT, spyware and infostealer samples generally performed better when APIs, strings or symbols provided clearer behavioural anchors. These results suggest that performance depended less on malware family labels alone and more by the visibility and interpretability of evidence available across each evaluation category.

5.3 Manual Review of Pipeline Performance

The manual forensic review was conducted to validate both the generated malware analysis reports and the Auditor used to score them. The report format supported manual review because each report contained explicit claims, omissions, uncertainty statements and category-level conclusions that could be compared against ground truth. The scores reported in the pipeline comparison are pipeline-output scores. They measure the quality of the reports produced by each configuration, but they are generated by the Auditor as the automated evaluation mechanism.

Manual review therefore had two purposes. First, it assessed the quality of the generated reports using precision, recall, F1-score and hallucination rate. Second, it checked whether the Auditor's weighted scores and penalty explanations reflected that quality. In this section, precision, recall, F1-score and hallucination rate refer to the generated reports. The later Auditor Review section evaluates the Auditor itself. The specific manual review metrics were defined in section 4.3.3.

Precision, recall and hallucination rate were averaged for each pipeline configuration to summarise the results. F1-score was then calculated from the averaged precision and recall values. This comparison shows how the generated reports differed in factual accuracy, evidence recovery and unsupported claim behaviour. Precision reflects how often report claims were supported by ground truth while recall reflects how much validated ground truth evidence was recovered. F1-score provides a combined measure of precision and recall. Hallucination rate reflects the proportion of unsupported claims. The resulting averages are shown in Table 9.

Table 9. Average manual review statistics across generated reports.

Configuration	Precision	Recall	F1-score	Hallucination rate
Single-Agent	0.758	0.480	0.587	0.242
Sequential	0.756	0.484	0.590	0.244
Sequential memory	0.772	0.512	0.616	0.228
Hierarchical	0.776	0.517	0.620	0.224
Hierarchical memory	0.774	0.513	0.617	0.226

Table 9 shows that precision was relatively similar across all configurations, ranging from 0.756 to 0.776. This indicates that all configurations produced a comparable proportion of supported claims. The larger differences appeared in recall and F1-score. The single-agent baseline achieved a recall of 0.480 and F1-score of 0.587 while the hierarchical configuration achieved the highest recall (0.517) and F1-score (0.620). The average precision was 0.767 across all 75 generated reports while the average recall was 0.501. Hallucination rates were also similar across configurations, ranging from 0.224 to 0.244 with the hierarchical configuration producing the lowest hallucination rate. This means that the reports were generally more reliable in the claims they made than complete in the evidence they recovered. In other words, many reports contained valid forensic observations, but omitted a substantial portion of the behaviours, capabilities and attribution evidence present in the validated ground truth.

Example of this precision–recall imbalance is the Reaver sequential report, shown in Table 10. The Auditor assigned this report a weighted score of 0.575, while manual review assigned it 0.778 precision but only 0.318 recall. This means that the report was relatively accurate in the limited claims it made but recovered only a small portion of the validated ground truth. The report preserved several structural facts but missed much of the API, semantic and capability evidence required for a complete Reaver analysis. Manual review classified this report as conservative but under-informative, with major omissions in credential-access, browser-data, SQLite, networking and service-execution evidence.

Table 10. Precision–recall imbalance in the Reaver malware sequential report.

Manual review aspect	Reaver Sequential example	Effect on metric
Supported overlap with ground truth	Correctly identified Windows PE32/x86 format, .text, .rdata, .data and. reloc sections, entry point 0x1c845, and ServiceMain / FunctionWork symbols.	Increased precision because these claims matched the validated ground truth.
Missing structural evidence	Omitted MSVC Rich Header toolchain, unpacked state, rich header evidence and high import count indicators.	Reduced recall because validated structural findings were not recovered.
Missing behavioural evidence	Omitted credential vault access, browser Login Data access, SQLite queries, service dispatch execution and rundll32 execution support.	Reduced recall because core Reaver behaviours were missing.
Missing API/interface evidence	Module/import list, vaultcli.dll, ws2_32.dll, advapi32.dll, oleacc.dll, VaultOpenVault, VaultGetItem, WSA APIs and service-control APIs	Reduced recall because validated interfaces were not recovered.
Missing semantic evidence	Credential vault access, browser Login Data access, SQLite query execution	Reduced recall because core infostealer/RAT behaviours were omitted.
Missing capability evidence	Credential theft, browser data theft, network communication and service execution	Reduced recall because validated malware capabilities were absent.
Unsupported/incorrect Claim	“No specific APIs were verified,” “No specific capabilities were verified”	Reduced precision because the report denied evidence that was present in the ground truth.

Table 10 illustrates why the manual metrics were interpreted at claim level. Reaver sequential report was not a complete failure because it recovered valid structural evidence. However, its low recall shows that it did not recover enough behavioural, API/interface and capability evidence to support a stronger forensic interpretation. This supports the broader finding that incomplete evidence recovery was more common than widespread incorrect reporting.

To better understand these mismatches, the manual review next examined the recurring error patterns behind the scores. Recurring report errors were counted across all 75 generated reports, including the single-agent baseline. Each report could contain more than one error type, so the percentages do not sum to 100%. Table 11 therefore shows how frequently each failure mode appeared in the full review set rather than assigning each report to a single exclusive category. Examples of the most common error types are discussed after the table.

Table 11. Distribution of recurring error types across all configurations.

Error type	Count	Percentage
Semantic under-reconstruction	63	84.0%
Unsupported claims / hallucination	37	49.3%
Contradicted absence-claims	28	37.3%
Attribution inconsistency	22	29.3%
Semantic inflation	20	26.7%
Platform/runtime/file-format error	20	26.7%
Abstraction mismatch	17	22.7%
Zero score	12	16.0%

As shown in Table 11, the dominant failure mode was semantic under-reconstruction, appearing in 63 reports. Reports often recovered surface-level facts such as file format, architecture, runtime indicators or entry points, but failed to reconstruct malware-specific behaviour. This was visible in WannaCry [54] and Jigsaw [57], where reports recognised PE or .NET structure but missed key ransomware behaviours such as payload deployment, cryptographic activity or ransom-note logic. Unsupported claims or hallucinations appeared in 37 reports. These occurred when reports introduced positive claims that were not supported by the validated evidence. Examples included Linux/X11 artefacts being introduced in Windows PE samples, unsupported networking claims and generic strings or runtime artefacts being elevated into confirmed malware capabilities.

Contradicted absence-claims were also common: 28 reports stated that APIs, capabilities, control-flow patterns or family indicators were absent even though the ground truth contained such evidence. Some Reaver [61] reports claimed that no APIs or capabilities were verified, although the ground truth included credential access, service related and networking APIs. These were more serious than simple omissions because they could mislead an analyst into believing relevant forensic evidence was not present. Attribution inconsistency appeared in 22 reports. In some cases, like Petya [55], Proteus [58], Hive [56] or WireNet [63], reporting the family as "Unknown" was appropriate because family-specific evidence was weak or absent. In other cases, such as EvilAnt [62], MacMa [66], Jigsaw [57] and WannaCry [54], "Unknown" represented a recall failure because stronger family, runtime or behavioural evidence was available. This shows that attribution quality depended on whether uncertainty was justified by the evidence.

The review also identified abstraction mismatch and semantic inflation. Abstraction mismatch occurred when reports detected generic APIs or runtime features but failed to map them to specific behaviours. In Reaver's [61] case, reports recovered networking

indicators and service-related symbols but did not reliably reconstruct the validated infostealer/RAT behaviour. Semantic inflation was the opposite problem, where weak evidence was elevated into unsupported capabilities, as seen in Proteus [58]. In several reports, .NET resource and PNG markers were interpreted as resource loading or image manipulation capabilities, even though ground truth did not confirm it. Finally, 12 reports received a zero score. This did not always mean that the reports were empty or useless. In several cases, they contained some correct observations, but major omissions, contradicted claims or strict category penalties caused the final Auditor score to collapse.

Manual review also showed that most generated reports followed the expected report structure and avoided excessive raw artefact dumping. The main qualitative differences were not in readability, but in whether reports preserved evidence-based reasoning and avoided unsupported behavioural conclusions.

5.4 Auditor Review

The Auditor review was conducted to evaluate whether the Zero-Trust Forensic Auditor assigned meaningful and manually aligned scores to generated reports. While the previous section assessed the reports themselves, this section examines the Auditor as the automated evaluation mechanism. The aim was to determine whether its weighted scores, penalty explanations and judgements were aligned with manual forensic review.

The Zero-Trust Forensic Auditor was useful as an evaluator because it provided a structured way to compare generated reports against validated ground truth. It often identified major omissions, unsupported claims, incorrect platform assumptions, missing APIs, missing capabilities, contradicted absence claims and attribution failures. It performed best when reports contained clear evidence conflicts. Examples included raw-blob samples being misidentified as PE executables, Windows reports introducing Linux artefacts and ransomware reports omitting validated APIs or capabilities. In these cases, the Auditor was generally correct and helped identify reports requiring closer forensic inspection.

Alignment with manual review was evaluated by comparing the Auditor's weighted scores with F1-scores across all 75 generated reports. These reports include outputs from all tested configurations: single-agent, sequential, sequential-memory, hierarchical and

hierarchical-memory workflows. Figure 2 shows the relationship between Auditor score and F1-score across all configurations.

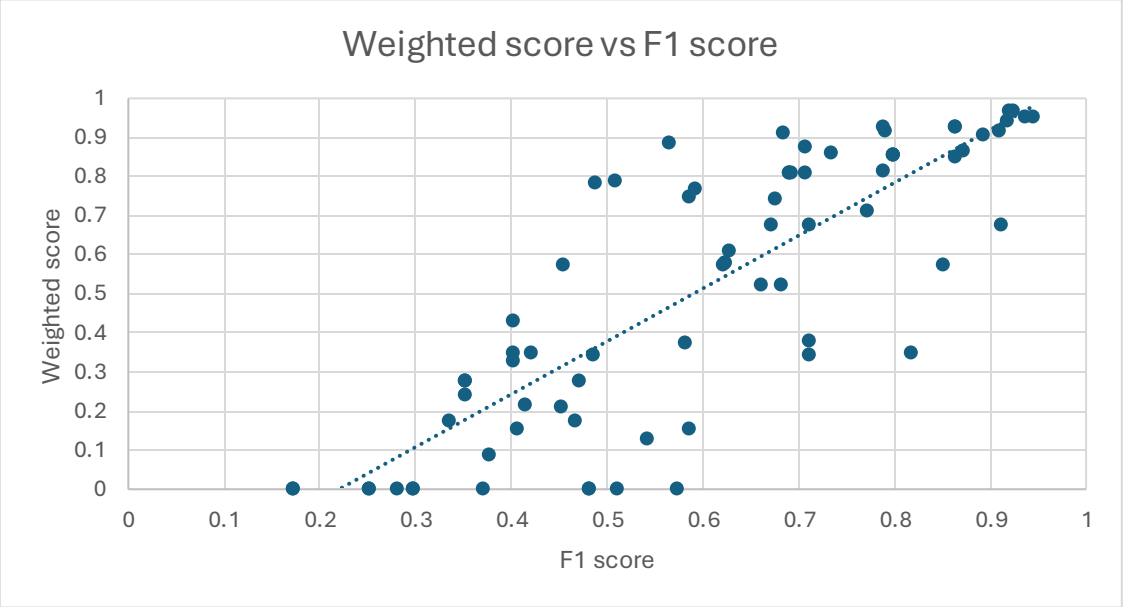

Figure 2. Relationship between Auditor weighted score and F1-score across all configurations.

Figure 2 shows a positive relationship between manual F1-score and Auditor weighted score. Reports with higher manual F1-scores generally also received higher Auditor scores, which is consistent with the Pearson correlation of $r = 0.827$. This indicates that the Auditor was directionally aligned with manual review.

However, the scatterplot also shows that the relationship was not perfect. Some reports with moderate manual F1-scores received a weighted score of 0.00 while some reports with relatively high Auditor scores had only moderate F1-scores. This shows that the Auditor was useful for ranking report quality overall, but its scoring was not fully calibrated at the individual report level. In particular, the Auditor sometimes rewarded structural or category-level coverage more strongly than overall completeness. Therefore, the Auditor score was useful as a diagnostic signal, but not as a substitute for manual forensic judgement.

The imperfect alignment shown in Figure 2 suggests calibration instability. In some cases, the Auditor was too harsh toward reports that preserved valid forensic facts but failed in later semantic categories. For example, the Andromeda [60] hierarchical no-memory report received a weighted score of 0.00. Manual review still found several correct hard

facts, including PE32/x86 structure, section layout and the entry point. Manual review therefore judged the report to be over-penalised. The report was weak, but it was not completely devoid of valid forensic content.

On the contrary, the Auditor was sometimes too lenient toward reports that recovered surface-level structure but missed core malware semantics. In the Reaver [61] hierarchical report, the Auditor assigned a relatively high score of 0.79, but manual review suggested lower report quality because it missed central infostealer/RAT evidence. Missing evidence included credential vault access, browser Login Data access, SQLite queries, service-control APIs and validated capabilities such as credential theft and service execution. These examples show that the Auditor did not always weight error types proportionally. It could collapse incomplete but partially valid reports to zero, while over-rewarding reports that appeared structurally complete but remained semantically under-reconstructed.

Penalty duplication was another limitation. In several reports, one root error generated repeated deductions across multiple categories. For example, a claim such as "no APIs were identified" could be penalised under AR, CI and SR if the missing APIs also supported behaviours and capabilities. This was especially visible in Petya [55] and WannaCry [54] reports. While the underlying error was serious, repeated penalties created non-independent score reductions and made some scores appear more precise than their forensic basis allowed.

Overall, the Auditor was effective as a screening mechanism but not sufficiently calibrated to act as a final evaluator. It was valuable for identifying reports with major contradictions, unsupported claims and severe omissions. However, manual forensic review was necessary to determine whether penalties reflected genuine forensic errors, justified uncertainty, semantic under-reconstruction, weak inference or taxonomy mismatch. This supports the conclusion that LLM-based auditing can assist forensic evaluation but should remain secondary to human validation in malware RE workflows.

5.5 Additional Observations

In addition to the quantitative scores and manual forensic review, several practical observations emerged during system development and repeated pipeline execution. The experiments showed that report quality depended not only on the selected coordination strategy, but also on prompt design, model sensitivity and the reliability of tool-mediated interactions. These findings are discussed separately because they reflect implementation constraints rather than direct architectural differences between single-agent, sequential and hierarchical configurations.

5.5.1 Prompt Comprehensiveness and Model Sensitivity

A practical observation during development was that prompt comprehensiveness had a substantial effect on agent reliability. Gemini 2.5 Flash-Lite [44] did not consistently infer the intended forensic workflow from short role descriptions alone. As a result, prompts had to be expanded from simple task descriptions into procedural control mechanisms. These mechanisms included fixed command sequences, loop-prevention rules, platform-specific constraints, evidence-grounding requirements, output limits and forbidden inference patterns. Representative prompt rules are provided in Appendices 4–11. For example, the Forensic Structural Hunter was instructed to run each core extraction command only once, mark missing fields as unknown or null and avoid cross-platform assumptions. The Interface Resolution Analyst was given fallback rules when structured imports were missing rather than converting missing import data into a negative finding.

Family attribution also required explicit control. The Malware Capability Analyst was instructed to separate malware class from malware family and to avoid assigning a named family based on broad behavioural similarity alone. In principle, named family attribution required specific evidence, such as ransom-note text, extension patterns, C2 or onion strings, mutexes, configuration artefacts or other distinctive markers. This rule was important because manually adding indicators for every known family would not scale to larger malware collections. It could also bias the results by turning attribution into a lookup task rather than evidence-based reasoning.

However, prompt design was not risk-free. Appendix 8 includes pattern-based attribution rules for extortion-related strings, where proper nouns near extortion keywords may be treated as possible family indicators. Such rules may improve recall for ransomware-like

samples but can also increase over-attribution risk if generic strings are treated as family-specific evidence. To reduce this risk, the Verification and Normalisation Supervisor was instructed to downgrade unsupported family claims to "Unknown". Overall, prompts acted as procedural safeguards that reduced hallucination, premature completion and unsupported inference, but they could also shape the result space. Future work should therefore separate general forensic reasoning rules from dataset-specific attribution rules more strictly.

5.5.2 Tool-Use Reliability in LLM Agents

An additional observation concerned tool-use reliability. Both the extraction-stage model and the analysis-stage model sometimes struggled to use tools in a disciplined way. DeepSeek-Coder-V2 [41] occasionally repeated commands, entered loops or attempted to finish extraction before all required artefacts had been collected. Gemini 2.5 Flash-Lite [44] sometimes confused forensic dump content with tool instructions. Although the prompts attempted to separate tool commands from forensic evidence through fixed command sequences, loop-prevention rules and role boundaries, this separation was not always reliable. For example, a string found in the forensic dump could later be reused by the agent as if it were a valid tool command or retrieval key. This caused failed tool calls and, in some cases, contributed to reports that received 0.00 Auditor scores.

This shows that prompt-level separation alone was not sufficient. Even when command lists and evidence grounding rules were provided, tool-calling could still fail because the model could generate invalid tool inputs. Future versions should therefore implement technical safeguards beyond prompt rules, including allowlisted commands, input validation and checks that required artefacts were collected before analysis.

6. Discussion

The findings indicate that multi-agent LLM systems can improve static malware RE and analysis, but only when role specialisation is supported by effective coordination and verification. In this study, agent interaction was implemented through structured task handoff rather than open-ended conversation between agents. In the hierarchical configurations, the supervising agent reviewed and integrated outputs from subordinate agents, but the agents did not engage in free-form peer-to-peer debate. Hierarchical configurations produced the strongest overall results, suggesting that centralised supervision helped reduce instability and organise forensic evidence more effectively than a purely sequential handoff structure. However, the results also show that this form of multi-agent coordination did not fully resolve the core difficulty of semantic reconstruction. The systems were more successful at recovering structural facts than reconstructing behavioural meaning from fragmented or indirect evidence.

This limitation was especially visible in packaged-runtime and ransomware samples, where reliable interpretation depended on integrating several evidence layers rather than analysing isolated artefacts. Therefore, the main value of the proposed pipeline lies not in replacing expert analysts, but in supporting them by organising extracted artefacts and producing initial assessments. The results are consistent with the broader literature showing that LLMs can support binary interpretation and forensic triage. However, evidence grounding, validation and human oversight remain essential when generated claims may influence analysis conclusions.

Improving behavioural deduction would likely require stronger evidence integration across analysis stages. One possible direction is to introduce specialised agents for wrapper detection, control-flow summarisation and evidence-chain reconstruction. These agents could separate packaging artefacts from payload-relevant logic, connect APIs and strings to specific functions, and require each behavioural claim to be supported by an explicit evidence path. Another possible improvement is iterative adversarial review, where one agent proposes behavioural interpretations and another challenges them against static artefacts before the final report is produced. Such communication patterns

were not evaluated in this study, but they may help reduce semantic under-reconstruction in future work.

6.1 Discussion of the Research Questions

This section discusses the findings in relation to the four research questions defined in the first chapter. The aim is to connect the empirical results with the original research problem and identify what the experiments show about multi-agent coordination, artefact use, evaluation reliability and analyst support.

First research question examined whether MAS improves the accuracy and completeness of static malware analysis reports compared to single-agent baselines. The results show that MAS can improve report quality, but the improvement depends strongly on the coordination architecture. Hierarchical configurations achieved the strongest overall performance. Hierarchical configuration with memory enabled produced the highest weighted and manual scores while the standard hierarchical configuration offered a strong balance between quality and runtime. Sequential configurations showed more limited improvement and weaker completion stability. This indicates that role specialisation alone is not sufficient when intermediate outputs are passed through linear workflows. The benefit of MAS in this study came from supervisory coordination and verification rather than from simply increasing the number of agents.

Second research question focused on whether MAS can use extracted RE artefacts to support structural and semantic reasoning. The results show that the implemented MAS pipelines were more reliable in structural reasoning than in complete behavioural reconstruction. Technical facts were often correctly identified, especially when samples exposed clear metadata, imports or platform-specific artefacts. However, semantic reasoning remained more difficult when behaviours were hidden behind packaging layers, runtime wrappers or incomplete static artefacts. This shows that extracted artefacts are useful for grounding LLM-based analysis, but static evidence alone is not always sufficient for confident behavioural interpretation.

Third research question examined how coordination strategy affected report stability and hallucination rates. The results suggest that coordination strategy influenced completion stability more clearly than hallucination rate. Hierarchical configurations completed the

highest number of runs whereas sequential configurations were more vulnerable to cascading failures. Memory did not improve completion stability in this experiment, although it improved evaluation scores in the hierarchical setup. However, this finding should be interpreted cautiously because CrewAI memory [75] was used with default storage and retrieval settings rather than an optimised memory configuration. Manual review also showed that the dominant weakness was not excessive hallucination but semantic under-reconstruction. In other words, the systems more often failed by omitting supported behaviours than by inventing unsupported ones.

Fourth research question evaluated the usefulness of the LLM-as-a-judge mechanism. The Zero-Trust Forensic Auditor was valuable as an initial evaluation tool because its scores broadly aligned with manual review and helped identify recurring errors. However, the Auditor was not adequate to replace human judgement. Some reports were over-penalised while some semantically incomplete reports received high scores. Automated evaluation mechanism is therefore best understood as a supporting assessment tool rather than a substitute for manual analysis.

Overall, the research questions show that hierarchical MAS was the most promising configuration tested in this study. It improved report quality and stability compared with the single-agent baseline, but it did not fully solve semantic under-reconstruction. The system was strongest when static evidence was explicit and weakest when behaviour was hidden, fragmented or incomplete. The Auditor supported evaluation, but human validation remained necessary.

6.2 Pipeline Limitations

There are several limitations to this study, some of which affect how the results should be interpreted. First, the pipeline is limited to static analysis and does not execute samples dynamically. Therefore, claims about persistence, exfiltration, propagation and C2 activity cannot be considered fully confirmed. Such behaviours may be suggested by strings, APIs or structural patterns, but they require runtime evidence for confirmation.

The dataset also limits the generalisation of findings. The design used allows for controlled comparison across different evidence conditions, but it does not support broad generalisation. A larger dataset would allow stronger statistical analysis across malware

families, platforms and file formats. It would also make it possible to test whether the observed advantage of hierarchical coordination remains consistent across a wider range of samples.

Ground truth construction introduces a further source of uncertainty. The ground truth was manually validated and based on extracted RE artefacts, but LLM-assisted interpretation was also used during its preparation. This reduced dependence on a single model, but it did not remove all subjectivity. Ground truth should therefore be treated as a structured benchmark based on static evidence rather than as a comprehensive account of all behaviours present in each sample.

The findings are also dependant on the model used. They reflect the behaviour of a specific model configuration rather than MAS in general. Stronger or more specialised models could improve the overall pipeline performance. This also applies to the Zero-Trust Forensic Auditor, whose scores were useful for comparative assessment but remained model-dependent and required interpretation alongside manual review.

The study does not include an agent-level ablation analysis. Agent-level ablation was not included in the research design because this research focuses on comparing complete workflow configurations and coordination strategies rather than isolating the contribution of each individual agent. Since agents were not removed individually, the results show the performance of complete configurations but do not determine how much each individual agent contributes to the final score. The findings should therefore be interpreted as evidence about coordination strategies rather than as proof that every agent role is independently necessary.

The memory-enabled configurations introduce an additional constraint. CrewAI shared memory [75] was enabled using the framework's default storage and retrieval behaviour. Memory parameters such as retrieval limits, embedding configuration, memory backend, ranking strategy and persistence settings were not independently optimised. These settings may affect which earlier findings are recalled by later agents and how much additional context is introduced into the workflow. As a result, the observed performance of the memory-enabled configurations should be interpreted as the effect of default CrewAI memory usage rather than as evidence of an optimised memory architecture.

6.3 Future Work

Future work should address the main limitations identified in this study through targeted pipeline extensions. First, the design should be extended with dynamic analysis so that behaviours inferred from static artefacts can be validated against observed runtime activity. Second, the pipeline should include additional specialised agents for tasks that remained difficult, especially packaging analysis and runtime-wrapper detection. A dedicated agent could distinguish wrapper artefacts from actual malicious payload indicators before semantic reconstruction and capability inference tasks.

Stronger and more specialised models should also be evaluated for each pipeline stage. Code-specialised local models may improve extraction and tool-use reliability, while stronger reasoning models may improve semantic reconstruction, capability inference and report synthesis. Future work should evaluate the Auditor with a different model from the one used for report generation to assess whether model separation improves evaluation consistency.

Future work should also test the contribution of individual agents through ablation experiments. Such experiments would remove or merge selected analytical agents while keeping the dataset, prompts, models and evaluation procedure constant. Comparing the resulting performance drop would help identify which agents contribute most to report quality, stability and evidence grounding. A reduced ablation study could first focus on non-essential analytical roles such as interface resolution or structural analysis before evaluating larger architectural changes.

Memory configuration should also be treated as a separate experimental variable. This includes testing alternative memory backends, retrieval limits, embedding configurations and memory-ranking strategies. Such analysis would help determine whether memory improves report quality through useful evidence recall or introduces irrelevant context and cross-task noise.

7. Conclusion

This thesis evaluated whether MAS could improve static malware RE and analysis compared to a single-agent baseline. Five configurations were evaluated: a single-agent baseline, a sequential multi-agent workflow, a sequential workflow with memory, a hierarchical multi-agent workflow and a hierarchical workflow with memory. The generated reports were assessed against manually validated ground truth using both the Auditor and manual review.

The results show that multi-agent coordination can improve the quality and stability of static malware analysis reports, but the improvement depends strongly on the coordination architecture. Hierarchical configurations achieved the strongest overall performance, while sequential configurations showed more limited improvement and weaker completion stability. This indicates that role specialisation alone is insufficient without effective coordination and verification. The study also showed that extracted RE artefacts can support LLM-based structural and semantic reasoning, but their usefulness depends on how clearly relevant evidence is exposed inside the sample. MAS configurations were more reliable in identifying structural facts than in reconstructing complete malware behaviour. The most common weakness was semantic under-reconstruction, where reports omitted supported behaviours, capabilities or attribution evidence.

The LLM-as-a-judge mechanism was useful as an initial evaluation mechanism, but it was not sufficient to replace manual judgement. Its scores were broadly aligned with manual review, but individual report scores sometimes over-penalised or over-rewarded reports. This confirms that LLM-as-a-judge evaluation can support malware analysis process but should remain subject to human validation. This thesis shows that MAS are a promising direction for analyst support tools in static malware RE. They can improve evidence organisation, report structure and forensic completeness, but they should be treated as support systems rather than replacements for expert malware analysts.

References

[1] S. Salinas, "Beyond Ransoms: The Financial Impact of Ransomware Attacks," Halcyon, 9 April 2025. [Online]. Available: https://www.halcyon.ai/blog/beyond-ransoms-the-financial-impact-of-ransomware-attacks [Accessed 20 May 2026].

[2] Halcyon, "The Ransomware Gap in the AI Era," 20 March 2026. [Online]. Available: https://www.halcyon.ai/resources/reports/the-ransomware-gap-in-the-ai-era [Accessed 20 May 2026].

[3] C. Patsakis, F. Casino and N. Lykousas, "Assessing LLMs in malicious code deobfuscation of real-world malware campaigns," *Expert Systems with Applications,* vol. 6, no. 256, 2024. doi: 10.1016/j.eswa.2024.124912

[4] P. Hu, R. Liang and K. Chen, "DeGPT: Optimizing Decompiler Output with LLM," in *Network and Distributed System Security (NDSS) Symposium*, 2024. doi: 10.14722/ndss.2024.24401

[5] X. Shang, S. Cheng, G. Chen, Y. Zhang, L. Hu, X. Yu, G. Li, W. Zhang and N. Yu, "How Far Have We Gone in Binary Code Understanding Using Large Language Models," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2024. doi: 10.1109/ICSME58944.2024.00012

[6] X. Shang, G. Chen, S. Cheng, B. Wu, L. Hu, G. Li, W. Zhang and N. Yu, "BinMetric: A Comprehensive Binary Code Analysis Benchmark for Large Language Models," in *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, 2025. doi: 10.24963/ijcai.2025/858

[7] X. Li, S. Wang, S. Zeng, Y. Wu and Y. Yang, "A survey on LLM-based multi-agent systems: workflow, infrastructure, and challenges," *Vicinagearth,* vol. 1, no. 9, 2024. doi: 10.1007/s44336-024-00009-2

[8] M. U. Zeshan, M. Ibiyo, C. Di Sipio, P. T. Nguyen and D. Di Ruscio, "Many Hands Make Light Work: An LLM-based Multi-Agent System for Detecting Malicious PyPI Packages," *Journal of Systems and Software,* vol. 236, 2026. doi: 10.1016/j.jss.2026.112792

[9] P. M. N. Silva, D. A. da Silva, R. d. O. Albuquerque, G. D. A. Nze and F. L. L. de Mendonç, "MAS-Hunt: A Resilient AI Multi-Agent System for Threat Hunting," *Engineering Proceedings,* vol. 123, no. 26, 2026. doi: 10.3390/engproc2026123026

[10] S. Ullah, P. Balasubramanian, W. Guo, A. Burnett, H. Pearce, C. Kruegel, G. Vigna and G. Stringhini, "From CVE Entries to Verifiable Exploits: An Automated Multi-Agent Framework for Reproducing CVEs," *arXiv.* doi: 10.48550/arXiv.2509.01835

[11] 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*, 2024. doi: 10.18653/v1/2024.emnlp-main.203

[12] J. Armengol-Estapé, J. Woodruff, C. Cummins and M. F. O'Boyle, "SLaDe: A Portable Small Language Model Decompiler for Optimized Assembly," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2024. doi: 10.48550/arXiv.2305.12520

[13] Z. Gao, Y. Cui, H. Wang, S. Qin, Y. Wang, Z. Bolun and C. Zhang, "DecompileBench: A Comprehensive Benchmark for Evaluating Decompilers in Real-World Scenarios," *arXiv,* 2025. doi: 10.48550/arXiv.2505.11340

[14] H. Lu, H. Peng, G. Nan, J. Cui, C. Wang and W. Jin, "Malsight: Exploring Malicious Source Code and Benign Pseudocode for Iterative Binary Malware Summarization," *IEEE Transactions on Information Forensics and Security,* vol. 20, pp. 6733-6747, 2025. doi: 10.1109/TIFS.2025.3583552

[15] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest and X. Zhang, "Large Language Model based Multi-Agents: A Survey of Progress and Challenges," in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence Survey Track*, 2024. doi: 10.24963/ijcai.2024/890

[16] S. Chen, Y. Liu, W. Han, W. Zhang and T. Liu, "A Survey on LLM-based Multi-Agent System: Recent Advances and New Frontiers in Application," *arXiv,* 2025. doi: 10.48550/arXiv.2412.17481

[17] K.-T. Tran, D. Dao, M.-D. Nguyen, Q.-V. Pham, B. O'Sullivan and H. D. Nguyen, "Multi-Agent Collaboration Mechanisms: A Survey of LLMs," *arXiv.* doi: 10.48550/arXiv.2501.06322

[18] Z. Wu and G. Gartner, "Context Cartography: Toward Structured Governance of Contextual Space in Large Language Model Systems," *arXiv,* 2026. doi: 10.48550/arXiv.2603.20578

[19] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan and S. Yao, "Reflexion: Language Agents with Verbal Reinforcement Learning," in *Proceedings of the 37th International Conference on Neural Information Processing System*, 2023. doi: 10.48550/arXiv.2303.11366

[20] S. Pordanesh and B. Tan, "Exploring the Efficacy of Large Language Models (GPT-4) in Binary Reverse Engineering," *arXiv,* 2024. doi: 10.48550/arXiv.2406.06637

[21] Z. Wan, Y. Li, X. Wen, Y. Song, H. Wang, L. Yang, M. Schmidt, J. Wang, W. Zhang, S. Hu and Y. Wen, "ReMA: Learning to Meta-think for LLMs with Multi-Agent Reinforcement Learning," *arXiv.* doi: 10.48550/arXiv.2503.09501

[22] Q. Duan and Z. Lu, "AI Agent Communications in the Future Internet—Paving a Path Toward the Agentic Web," *Future Internet,* vol. 18, no. 3, 2026. doi: 10.3390/fi18030171

[23] Z. Xi, W. Chen, X. Guo, W. He, Y. Ding and B. Hong, "The Rise and Potential of Large Language Model," *Science China Information Sciences,* vol. 68, no. 121101, 2025. doi: 10.1007/s11432-024-4222-0

[24] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan and Y. Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," *arXiv,* 2023. doi: 10.48550/arXiv.2210.03629

[25] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda and T. Scialom, "Toolformer: language models can teach themselves to use tools," in *NIPS '23: Proceedings of the 37th International Conference on Neural Information Processing Systems*, 2023. doi: 10.5555/3666122.3669119

[26] Y. Shen, K. Song, X. Tan, D. Li, W. Lu and Y. Zhuang, "HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face," *arXiv,* 2023. doi: 10.48550/arXiv.2303.17580

[27] P. Lewis, E. Perez, A. Piktus and F. Petroni, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *NIPS'20: Proceedings of the 34th International Conference on Neural Information Processing Systems*, 2020. doi: 10.48550/arXiv.2005.11401

[28] Q. Zhang, L. Fu, L. Lian, G. Go, Y. Wang, C. Zhou, Y. Jiang and G. Pu, "Evaluating Privilege Usage of Agents on Real-World Tools," *arXiv,* 2026. doi: 10.48550/arXiv.2603.28166

[29] A. K. Sood, S. Zeadally and E. Hong, "The paradigm of hallucinations in AI-driven cybersecurity systems: Understanding taxonomy, classification outcomes, and mitigations," *Computers and Electrical Engineering,* vol. 124, no. A, 2025. doi: 10.1016/j.compeleceng.2025.110307

[30] H. Jelodar, S. Bai, P. Hamedi, H. Mohammadian, R. Razavi-Far and A. Ghorbani, "Large language model (LLM) for software security: Code analysis, malware analysis, reverse engineering,," *Journal of Information Security and Applications,* vol. 98, 2026. doi: 10.1016/j.jisa.2026.104390

[31] O. R. Polu, "AI - Driven Malware Classification Using Static and Dynamic Analysis," *International Journal of Science and Research,* vol. 13, no. 6, 2024. doi: 10.21275/SR24062114525

[32] M. Alshomrani, A. Albeshri, B. Alturki, F. S. Alallah and A. A. Alsulami, "Survey of Transformer-Based Malicious Software Detection Systems," *Electronics,* vol. 13, no. 4677, 2024. doi: 10.3390/electronics13234677

[33] J. Komarthi, "AI-Driven Malware Behavior Analysis and Threat Prediction," *International Journal of Innovative Research in Engineering & Multidisciplinary Physical Sciences,* vol. 13, no. 4, 2025. doi: 10.37082/IJIRMPS.v13.i4.232671

[34] P. Rafiey and A. Namadchian , "Mapping Vulnerability Description to MITRE ATT&CK," *Advances in Artificial Intelligence and Machine Learning,* vol. 5, no. 3, pp. 4379-4396, 2025. doi: 10.54364/AAIML.2025.53243

[35] K. Mukherjee and M. Kantarcioglu, "LLM-driven Provenance Forensics for Threat Investigation and Detection," *arXiv,* 2025. doi: 10.48550/arXiv.2508.21323

[36] X. Zheng, . X. Qian, Y. He, S. Yang and L. Cavallaro, "Beyond Classification: Evaluating LLMs for Fine-Grained Automatic Malware Behavior Auditing," *arXiv,* 2025. doi: 10.48550/arXiv.2509.14335

[37] CrewAI, “CrewAI - The leading open source platform,” 2026. [Online]. Available: https://crewai.com/open-source [Accessed 20 May 2026].

[38] Radare, “radare2,” 2026. [Online]. Available: https://rada.re/n/radare2.html [Accessed 20 May 2026].

[39] J. Gu , X. Jiang, Z. Shi , H. Tan and X. Zhai, “A survey on LLM-as-a-judge,” *The Innovation,* 2026. doi: 10.1016/j.xinn.2025.101253

[40] Google, “Gemini API,” 2026. [Online]. Available: https://ai.google.dev/gemini-api/docs [Accessed 20 May 2026].

[41] Ollama, “deepseek-coder-v2,” 2026. [Online]. Available: https://ollama.com/library/deepseek-coder-v2 [Accessed 20 May 2026].

[42] Ollama, “The easiest way to build with open models,” 2026. [Online]. Available: https://ollama.com/ [Accessed 20 May 2026].

[43] D. AI, Q. Zhu, D. Guo, Z. Shao, D. Yang and Y. Li, “DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence,” *arXiv,* 2024. doi: 10.48550/arXiv.2406.11931

[44] Google, “Gemini 2.5 Flash-Lite,” 2026. [Online]. Available: https://ai.google.dev/gemini-api/docs/models/gemini-2.5-flash-lite [Accessed 20 May 2026].

[45] Ubuntu, “Ubuntu 24.04 LTS release notes,” 2026. [Online]. Available: https://documentation.ubuntu.com/release-notes/24.04/ [Accessed 20 May 2026].

[46] Oracle, “VirtualBox - Powerful open source virtualization,” 2025. [Online]. Available: https://www.virtualbox.org/ [Accessed 20 May 2026].

[47] Radare, “r2pipe,” 2026. [Online]. Available: https://www.radare.org/n/r2pipe.html [Accessed 20 May 2026].

[48] Microsoft, “AutoGen - A framework for building AI agents and applications,” 2024. [Online]. Available: https://microsoft.github.io/autogen/stable//index.html [Accessed 20 May 2026].

[49] LangChain, “LangChain Docs - LangGraph overview,” 2026. [Online]. Available: https://docs.langchain.com/oss/python/langgraph/overview [Accessed 20 May 2026].

[50] Pydantic Services Inc., “Pydantic AI - GenAI Agent Framework, the Pydantic way,” 2025. [Online]. Available: https://pydantic.dev/docs/ai/overview/ [Accessed 20 May 2026].

[51] O. S. team, “Swarm,” OpenAI, 2026. [Online]. Available: https://github.com/openai/swarm [Accessed 20 May 2026].

[52] Camel-AI, “CamelAI - Finding the Scaling Laws of Agents,” 2026. [Online]. Available: https://camel-ai.org/ [Accessed 20 May 2026].

[53] H. S. Anderson and P. Roth, “EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models,” *arXiv,* 2018. doi: 10.48550/arXiv.1804.04637

[54] Microsoft Defender Security Research Team, "WannaCrypt ransomware worm targets out-of-date systems," Microsoft, 12 May 2017. [Online]. Available: https://www.microsoft.com/en-us/security/blog/2017/05/12/wannacrypt-ransomware-worm-targets-out-of-date-systems [Accessed 20 May 2026].

[55] M. D. S. R. Team, "New ransomware, old techniques: Petya adds worm capabilities," Microsoft, 27 June 2017. [Online]. Available: https://www.microsoft.com/en-us/security/blog/2017/06/27/new-ransomware-old-techniques-petya-adds-worm-capabilities/ [Accessed 20 May 2026].

[56] N. Ovadia, "Hive Ransomware Analysis," Varonis, 15 February 2023. [Online]. Available: https://www.varonis.com/blog/hive-ransomware-analysis [Accessed 20 May 2026].

[57] D. Byrne and C. Thorpe, "Jigsaw: An Investigation and Countermeasure for Ransomware Attacks," in *16th European Conference on Cyber Warfare and Security ECCWS 2017*, Dublin, 2017. Available: https://www.researchgate.net/publication/320935153_Jigsaw_An_Investigation_and_Countermeasure_for_Ransomware_Attacks [Accessed 20 May 2026].

[58] D. Wang and J. Leong, "A New All-in-One Botnet: Proteus," Fortinet, 28 November 2016. [Online]. Available: https://www.fortinet.com/blog/threat-research/a-new-all-in-one-botnet-proteus [Accessed 20 May 2026].

[59] ESET, "ESET takes deep dive into Latin American banking trojans, starting with new Amavaldo malware family," 2019. [Online]. Available: https://www.eset.com/gr-en/about/newsroom/press-releases-1/eset-takes-deep-dive-into-latin-american-banking-trojans-starting-with-new-amavaldo-malware-family/ [Accessed 20 May 2026].

[60] J.-I. Boutin, "ESET takes part in global operation to disrupt Gamarue," WeLiveSecurity, 04 December 2017. [Online]. Available: https://www.welivesecurity.com/2017/12/04/eset-takes-part-global-operation-disrupt-gamarue/ [Accessed 20 May 2026].

[61] The MITRE Corporation, "Reaver," 22 October 2025. [Online]. Available: https://attack.mitre.org/software/S0172/ [Accessed 20 May 2026].

[62] N. N. Aguas, "Ransom.Win32.EVILANT.THAOBBD," TrendMicro, 12 January 2024. [Online]. Available: https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/ransom.win32.evilant.thaobbd [Accessed 20 May 2026].

[63] M. S. Intelligence, "Backdoor:Linux/WireNet.A!xp," Microsoft, 16 March 2022. [Online]. Available: https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Backdoor:Linux/WireNet.A!xp&threatId=-2147152270 [Accessed 20 May 2026].

[64] P.-M. Bureau, "Malicious Apache module used for content injection: Linux/Chapro.A," ESET, 18 December 2012. [Online]. Available: https://www.welivesecurity.com/2012/12/18/malicious-apache-module-used-for-content-injection-linuxchapro-a/ [Accessed 20 May 2026].

[65] “theZoo - A Live Malware Repository,” 2026. [Online]. Available: https://thezoo.morirt.com/ [Accessed 20 May 2026].

[66] The MITRE Corporation, “MacMa,” 06 May 2022. [Online]. Available: https://attack.mitre.org/software/S1016/ [Accessed 20 May 2026].

[67] P. Paganini, “A new Mac malware dubbed Tarmac has been distributed via malvertising campaigns,” Security Affairs, 13 October 2019. [Online]. Available: https://securityaffairs.com/92441/malware/mac-malware-tarmac.html [Accessed 20 May 2026].

[68] M. Kuzin and S. Zelensky, “Calisto Trojan for macOS,” SecureList, 20 July 2018. [Online]. Available: https://securelist.com/calisto-trojan-for-macos/86543/ [Accessed 20 May 2026].

[69] Vx-underground, “vx-underground,” 2026. [Online]. Available: https://vx-underground.org/. [Accessed 20 May 2026].

[70] OpenAI, “Introducing GPT-5.2,” 11 December 2025. [Online]. Available: https://openai.com/index/introducing-gpt-5-2/ [Accessed 20 May 2026].

[71] DeepSeek, “DeepSeek-V3.2 Release,” 01 December 2025. [Online]. Available: https://api-docs.deepseek.com/news/news251201 [Accessed 20 May 2026].

[72] Google, “Gemini 3 Pro: the frontier of vision AI,” 05 December 2025. [Online]. Available: https://blog.google/innovation-and-ai/technology/developers-tools/gemini-3-pro-vision/ [Accessed 20 May 2026].

[73] Ollama, “Llama3.1:8b,” 30 November 2024. [Online]. Available: https://ollama.com/library/llama3.1:8b [Accessed 20 May 2026].

[74] Google, “Gemini 2.5 Flash,” 28 April 2026. [Online]. Available: https://ai.google.dev/gemini-api/docs/models/gemini-2.5-flash [Accessed 20 May 2026].

[75] CrewAI, “Memory - Leveraging the unified memory system in CrewAI to enhance agent capabilities.,” 2026. [Online]. Available: https://docs.crewai.com/en/concepts/memory [Accessed 20 May 2026].

Appendix 1 – Non-Exclusive Licence for Reproduction and Publication of a Graduation Thesis[1]

I, Herman Karu

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Evaluating Multi-Agent Large Language Models for Static Malware Reverse Engineering and Analysis", supervised by Hayretdin Bahsi
   1.1 to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2 to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the nonexclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

20.05.2026

[1] The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Ground Truth for Hive malware


```
{
  "sample_id": "hive-sample",
  "platform_profile": "windows_pe",
  "file_type_profile": "native_binary",
  "malware_class": "unknown",
  "malware_family": "Unknown",
  "family_like": "hive_like",
  "family_evidence_present": false,
  "attribution_evidence": [],
  "hard_facts": {
    "os": "windows",
    "architecture": "x86",
    "compiler_or_runtime": "unknown",
    "entry_point": "0x0005b250",
    "packed_state": "unpacked_from_upx",
    "has_overlay": true
  },
  "structural_truth": {
    "sections_or_segments": [
      ".text",
      ".rdata",
      ".data",
      ".idata",
      ".reloc",
      ".symtab"
    ],
    "modules_or_libraries": [
      "KERNEL32.dll",
      "ADVAPI32.dll",
      "WS2_32.dll"
    ],
    "format_specific_features": [
      "PE32",
      "stripped",
      "overlay_present"
    ],
    "notable_layout_properties": [
      "large_binary_size",
      "high_function_count",
      "static_linking_characteristics"
    ]
  },
  "semantic_truth": {
    "primary_algorithm": "unknown",
    "behaviors": [
      "high_function_density",
      "complex_control_flow"
    ],
    "control_flow_patterns": [
      "multi_function_dispatch"
    ],
    "init_patterns": []
  },
```

```
  "interface_truth": {
    "imports_modules_apis": [],
    "interface_families": []
  },
  "capability_truth": {
    "capabilities": [
      "network_capability_present"
    ],
    "confidence_by_capability": {
      "network_capability_present": "high"
    }
  },
  "unsupported_capabilities": [
    "confirmed_file_encryption",
    "confirmed_persistence",
    "confirmed_process_injection"
  ]
}
```

Appendix 3 – Forensic Traceability Mapping for WannaCry

Behavioural Claim (Capability)	Technical Evidence (Artefact)	Source / Technical Anchor
Payload Extraction	"Massive resource section (.rsrc) containing embedded payload (~3.4MB)" and "Gilles Vollant unzip strings"	structural_truth.notable_ layout_properties and. rdata section
File Encryption Logic	CryptAcquireContextA, CryptImportKey, CryptEncrypt, CryptDecrypt	interface_truth. imports_modules_apis (Security APIs)
Service-Based Persistence	CreateServiceA, OpenSCManagerA, StartServiceA	interface_truth. imports_modules_apis (Service Control)
Registry Manipulation	RegCreateKeyW, RegSetValueExA, RegQueryValueExA	interface_truth. imports_modules_apis (Registry APIs)
Network Capability	WS2_32.dll initialization and networking init behaviors	structural_truth. modules_or_libraries

Appendix 4 – Forensic Extractor and Its Task

```
extractor_agent:
  role: >
   Forensic Extractor
  goal: >
   Execute exactly 12 different commands in a strict linear sequence.
  backstory: >
   You are a low-level automation script. You do not have the authority
   to decide when the task is finished. You only finish when you have
   ticked off every item in your checklist.

   You do not care about the content of the data; you only care about
  the confirmation that the tool was triggered for each unique command.

binary extraction task:
  description: >
    You are a robotic harvester. You have 12 slots to fill.
    Your current mission status is TRACKED.

    SEQUENCE:

    1. 'ii' 2. 'ih' 3. 'isj' 4. 'iiqj' 5. 'aflj' 6. 'izzj'
    7. 's main || s entry0: pdj 500'
    8. 'iej' 9. 'iv' 10. 'axj' 11. 'itj' 12. 'ig'

    STRICT OPERATING RULES:

    Execute ONLY ONE command at a time.
    After you receive 'PROGRESS_UPDATE', you must NEVER call
    that command again.
    If you just finished Step 10, you must immediately call Step 11.
    Do not summarize. Just move to the next number.

    Do not finish the task until you have received the 'MISSION
    ACCOMPLISHED' message from the tool.
    Your final answer must be a simple string: 'Extraction Complete:
    12/12 blocks saved.'
```

Appendix 5 – Forensic Structural Hunter and Its Task

```
low_level_structural_analyst:
  role: >
    Forensic Structural Hunter
  goal: >
    Extract exact structural facts for {binary_path} and map them to
    Canonical Tags.
  backstory: >
    You are a low-level specialist. You use forensic tools to find
    hard facts.
    You never guess. You translate tool output into machine-readable
    tags (e.g., arch: x86, os: windows) to ensure the Evaluator can
    score the work.

disassembly_analysis_task:
  description: >
    Perform a Universal Structural Analysis on {binary_path}.

    REQUIRED SEQUENCE:
    1. Run 'iIj' for metadata.
    2. Run 'iHj' for headers (specifically AddressOfEntryPoint).
    3. Run 'iSj' for sections.
    4. Run 'izzj' for strings.
    5. Run 'iiqj' for imports.
    6. Run any other tool command if needed

    LOOP PREVENTION RULES:
    - You are permitted exactly ONE call per command listed above.
    - If a specific field (e.g., nx_enabled) is not found after
    running 'iHj', do NOT retry the command. Mark the field as
    'null' or 'unknown' and move to the next step.
    - Do not seek 'perfection'. Report the raw forensic truth as it
    exists in the tool output. If the tool output is incomplete,
    unparseable, or contains errors, report what you can and move
    on. Do not get stuck trying to resolve every issue.

    SKEPTICISM RULE: If you see a string like 'Go build ID' or
    '.NET', do NOT immediately claim it as the compiler. Check 'iIj'
    (lang) and 'iiqj' (imports). Only claim a compiler if 'lang' and
    'imports' agree. Otherwise, tag 'compiler_or_runtime: unknown'.
    - If no headers exist, tag 'file_format: raw_blob' and
    architecture: inferred'.
    - Use 'izzj' (strings) as your primary evidence source.
    - For such files, look for ASCII patterns
    - If 'class' is 'ELF', you are in LINUX MODE.
    - If 'class' is 'PE', you are in WINDOWS MODE.
    - Tag 'os' and 'architecture' (x64) based ONLY on iIj.
    - NEVER report Windows sections (.rsrc, .reloc) for an ELF
    binary.
    - If 'lang' is 'cil', tag 'managed_dotnet_binary: True' and 'os:
    windows'.

    OUTPUT RULES:
    - Return valid compact JSON only.
    - Do not repeat keys.
```

```
- Do not include malware_tool_* fields unless directly observed.
- If a value is unknown, use null once.
- Maximum 40 JSON keys total.
- If tool output is incomplete, summarize it instead of
expanding unknown fields.
- If tool output is [NO DATA RETURNED] then move on to the next
command.
```

Appendix 6 – Interface Resolution Analyst and Its Task

```
api_resolution_agent:
  role: >
    Interface Resolution Analyst
  goal: >
    Identify only the interfaces directly supported by forensic
    evidence in {binary_path}, including imports, modules, runtime
    interfaces, UI frameworks, and other external dependencies.
  backstory: >
    You are strict about evidence. You never infer a specific API,
    DLL, or module from a capability alone. When only a family of
    interfaces is supported, you report the family rather than
    inventing a concrete symbol.

api_resolution_task:
  description: >
    Analyse the resolved APIs and imported DLLs from the evidence graph
    and forensic dump. Trust the strings more than the tables.

    UNIVERSAL EXTRACTION RULES:
    1. ROBUSTNESS: If the structured import data (e.g., JSON from
    'iiqj') is missing, unparseable, or contains errors, DO NOT output
    'DATA ERROR' or give up.

    2. FALLBACK: If structured data fails, you must scan the raw text,
    strings, and command outputs for common Windows subsystem
    artefacts (look for strings ending in '.dll', '.sys', or common
    APIs like LoadLibrary/GetProcAddress).

    3. Categorize whatever DLLs and APIs you find logically by their
    subsystem (e.g., Networking, OS Execution, Security/Cryptography,
    File I/O).
    - LINUX RULE: Look for 'libc.so' or 'ld-linux' in 'iiqj'.
    - Do NOT report Windows APIs (RegOpenKey) for Linux binaries.

    4. DO NOT INVENT APIS:
    - Only report APIs or DLLs explicitly found in structured output,
    strings, or raw evidence.
    - Do not infer likely APIs from capabilities.
    - If the Import Table (iiqj) is empty, you MUST treat the DLL names
    found in the Strings (izzj) as the official imports. Do not report
    that imports are missing. You must preserve all upstream verified
    facts exactly. Do not convert missing input into negative findings.
    If required evidence is missing from context, output
    "blocked_missing_context" instead of claiming the artefact does
    not exist.
```

Appendix 7 – Execution Semantics Analyst and Its Task

```
semantic_reconstruction_agent:
  role: >
    Execution Semantics Analyst
  goal: >
    Reconstruct the sample's startup path, execution model, and major
    behavioral flow for {binary_path} without speculating beyond
    direct evidence.
  backstory: >
    You identify how execution begins, how runtime initialization
    occurs, and what major control-flow patterns are visible. You
    distinguish verified behavior from possible behavior and
    explicitly label uncertainty.

semantic_reconstruction_task:
  description: >
    Reconstruct high-level program logic using ONLY the evidence_graph
    and structural analysis results.

    ROLE BOUNDARY:
    - This agent does not perform raw evidence acquisition.
    - Do not call forensic_reader unless explicitly provided with a
    locked binary_path and instructed to do so.
    - Do not invent function boundaries, instruction traces,
    addresses, or pseudocode.
    - Do not treat every address, PLT entry, thunk, stub, or return
    sequence as a significant function.

    SEMANTIC SCOPE CONTROL:
    - Analyse only high-confidence behavioral subsystems or meaningful
    functions.
    - Maximum 10 function_behavior entries.
    - Omit trivial stubs, PLT thunks, jump wrappers, short return-only
    functions, and initialization noise.
    - A semantic finding must be supported by at least one verified
    evidence type: API/import, symbol name, meaningful string, section
    context, or structural evidence.
    - If instruction-level evidence is not present, summarize behavior
    at subsystem level instead of inventing instruction-level logic.

    ENTRY POINT HANDLING:
    - Do not refer to the entry point as entry0.
    - Use the entry point value only if it appears in the
    evidence_graph or structural analysis result.
    - If no verified AddressOfEntryPoint exists, set entry_point to
    null.
    - Do not query iHj here to recover the entry point.
    - Do not invent an entry point.

    INTERPRETATION RULES:
    - Base interpretations only on verified strings, imports, symbols,
    sections, and structural evidence.
    - Do not generate pseudocode unless there is enough verified
    evidence to support it.
    - Do not convert string names into confirmed behavior unless
```

```
supporting APIs or semantic evidence exist.
- Preserve uncertainty when behavior is ambiguous.
- Prefer a short conservative summary over a long speculative
reconstruction.
```

Appendix 8 – Malware Capability Analyst and Its Task

```
capability_inference_agent:
  role: >
    Malware Capability Analyst
  goal: >
    Infer supported malware capabilities from verified interfaces,
    strings, control-flow, and UI evidence in {binary_path}.
  backstory: >
    You map evidence to behavior conservatively. You distinguish:
    - verified capability
    - likely capability
    - unsupported speculation
    You never promote a capability to verified unless the evidence
    directly supports it.

capability_inference_task:
  description: >
    Infer behavioral capabilities and malware class based on the
    gathered structural, semantic, API, and string evidence.

    OUTPUT SEPARATION RULES:
    1. Separate:
      - "malware_class" (e.g., ransomware, downloader, stealer,
      loader)
      - "malware_family" (specific family name only if directly
      supported)
    2. You MUST NOT assign a specific malware family from heuristics
    alone.
    3. A specific malware family may be assigned ONLY if at least one
    family-specific indicator is present in evidence, such as:
      - ransom note text
      - file extension pattern
      - onion/domain/C2 string
      - mutex name
      - config blob / campaign marker
      - publicly known unique artefact explicitly present in strings
      or code
    4. If such indicators are absent, set:
      - "malware_family": "Unknown"
      - optionally add "family_like": "Hive-like" or similar only as
      a soft comparison
    5. Capability claims must be tied to evidence categories:
      - API evidence
      - string evidence
      - control-flow evidence
      - semantic evidence
    6. Network capability is not the same as confirmed C2. If only
    generic runtime
       networking support is present, report:
       - "network_communication_capability": true
       - "confirmed_c2": false
    RULE 1: CLASS-INFERENCE EXAMPLE: EXTORTION PATTERN
       This is an illustrative example rather than exhaustive
       classification rule.
```

```
    If strings contain a combination of (Currency + Threat +
    Lock/Encrypt), you are AUTHORIZED to tag 'malware_class:
    ransomware'.
    Example: ("Bitcoin" OR "Monero") AND ("Deleted" OR "Locked")
    -> Ransomware.

RULE 2: FAMILY ATTRIBUTION
    Search strings for proper nouns that appear in high frequency
    Near extortion keywords. Report this proper noun as
   'malware_family'.

    You MUST preserve all raw artefacts (DLL names, Hex addresses)
    from previous tasks. Do not summarize them into generalities.
    If imports are empty, use the strings already provided by
    disassembly_analysis_task.
```

Appendix 9 – Verification and Normalisation Supervisor and Its Task

```
verification_supervisor_agent:
  role: >
    Verification and Normalisation Supervisor
  goal: >
    Independently verify analyst claims for {binary_path}, correct
    unsupported claims, and produce a canonical verified fact sheet
    for downstream reporting.
  backstory: >
    You are the final factual authority before reporting. You do not
    merely mark claims true or false; you normalise them into
    structured fields, remove rejected claims, preserve corrected
    values, and separate facts from interpretations.

verification_task:
  description: >
    You are the Data Consistency Gatekeeper. Your role is to ensure
    the analysis data is internally consistent, logically sound, and
    correctly abstracted before final reporting.If in doubt, leave it
    out.
    INPUT BINDING:
      The verified sample path is {binary_path}.
      This value is authoritative and immutable for the entire task.

    UNIVERSAL VERIFICATION PROTOCOLS:
    1. REGISTER SYNC: Ensure architecture claims match the registers
    used (e.g., if x86 32-bit is claimed, ensure 'eax'/'esp' are
    referenced in semantics, not 'rax'/'rsp').

    2. ABSTRACTION ENFORCEMENT (CRITICAL): Scrub and remove highly
    specific virtual addresses (e.g., 0x4198400) or literal string
    offsets from the section topology and control flow data. The ONLY
    exact address you should preserve is the main Entry Point.

    3. API & CAPABILITY ALIGNMENT: Verify that inferred capabilities
    have supporting evidence. If the capabilities claim "Networking,"
    ensure the API list includes networking DLLs (e.g., ws2_32.dll).
    If they don't match, fix the claim based on the actual APIs found.

    4. HALLUCINATION CHECK: Scan the input data for agent error
    messages like "DATA ERROR" or "No APIs detected". If found, attempt
    to resolve them using the raw evidence context, or safely omit the
    broken claim.

    5. ATTRIBUTION VALIDATION:
    - If a specific malware family is claimed, verify presence of
    family-specific indicators.
    - If absent → downgrade to "Unknown".
    - If a specific malware family is claimed, verify that the input
    contains at least one family-specific indicator.
    - Acceptable indicators include ransom note text, extension
    pattern, C2/onion string, mutex, config artefact, or other unique
```

```
family marker.
- If no family-specific indicator exists, downgrade the family
claim to "Unknown".
- Preserve soft comparisons only in wording such as "Hive-like" or
"behaviourally consistent with ransomware".

6. CAPABILITY STRENGTH NORMALISATION:
  - Distinguish between:
    - capability present
    - capability likely
    - capability confirmed
  - Example:
    - generic network strings -> "network capability present"
    - explicit C2 artefact -> "confirmed C2"
    Do NOT score or evaluate accuracy against ground truth. Just
    ensure internal consistency and proper abstraction.
    If a claim fails verification, DELETE it or downgrade it. Do
    not replace it with a new unsupported claim.

7. CLAIM NORMALISATION:
  - Normalise synonymous phrasing before validation.
  - Treat equivalent wording as the same claim.
  - Normalise capability names, family labels, and entry-point
  phrasing.

8. CLAIM STRENGTH CONTROL:
  - Label each claim as confirmed, likely, or possible.
  - Downgrade claims whose wording is stronger than the available
  evidence.
  - Do not upgrade soft comparisons into hard facts.

9. CONTRADICTION CHECK:
  - Remove or correct any claim that directly contradicts verified
  evidence.
  - Examples:
    - packed vs unpacked
    - confirmed family vs absent family evidence
    - confirmed C2 vs generic networking only

10. DEDUPLICATION:
  - Merge duplicate or near-duplicate claims before producing the
  verified output.
  - A claim may not be simultaneously marked as supported and
  omitted.

11. EVIDENCE LINKING:
  - Every capability and family claim must include at least one
  supporting evidence type: API, string, semantic, or structural.
  - If no support exists, downgrade or delete the claim.

12. SAMPLE-SPECIFIC ENFORCEMENT:
  - Remove generic Windows PE assumptions that are not explicitly
  supported by the evidence.
  - Reject common-template claims such as .rsrc, CreateProcessA,
  WriteProcessMemory, or socket APIs unless directly evidenced.

13. MALWARE-TYPE CONSISTENCY:
  - If verified evidence supports ransomware-like behavior,
  preserve those supported capabilities in the cleaned summary.
  - Do not replace sample-specific capabilities with generic
  loader/injector behaviors unless supported.
```

```
If 'iEj' returns [], do NOT stop.
    1. Pivot to 'iHj' to find 'AddressOfEntryPoint'.
    2. If that also fails, use 'izzj' to look for the 'PE' or
    'ELF' header strings.
    3. Do NOT strip findings from the report unless you have
    PROOF they are hallucinations.
    Silence is NOT verification.
    Never print exhaustive raw lists of strings, imports,
    sections, APIs, xrefs, or functions.
```

Appendix 10 – Forensic Intelligence Editor and Its Task

```
report_synthesis_agent:
  role: >
    Forensic Intelligence Editor
  goal: >
    Synthesize verified findings into a report while MANDATORY
    preserving the Technical Metadata block.
  backstory: >
    You are a professional editor. You write clean prose for humans,
    but you know that the 'Technical Metadata' block is the only thing
    the Evaluator actually scores. You never "clean" or "translate"
    the raw tags.

final_reporting_task:
  description: >
    Synthesize the findings from the verification_task into a
    professional, cohesive Final Static Analysis Report.

    CRITICAL REPORTING RULES:

    1. NO RAW ADDRESSES:
       - Do NOT include raw hex virtual addresses or offsets for
       sections, strings, loops, or instruction traces.
       - The ONLY address you may include is the verified main Entry
       Point.

    2. FOCUS ON PURPOSE:
       - Describe identified executable sections or binary regions in
       the context of this sample.
       - Avoid emphasizing raw section sizes, offsets, or addresses.
       - Use PE/ELF terminology only if verified by the evidence.

    3. BEHAVIORAL ABSTRACTION:
       - Describe control flow logically (e.g., "Contains a runtime
       initialization loop", "Performs encoded string processing")
       rather than dumping assembly instructions or raw disassembly.

    4. ALIGNMENT:
       - Ensure inferred capabilities match the categorised APIs,
        semantic evidence, or structural evidence.
       - Do not claim capabilities unless supporting evidence exists.
       - Example:
         - networking APIs -> networking capability
         - XOR routines -> XOR obfuscation capability
       - Do not overstate evidence strength.

    5. EVIDENCE LOCK:
       - You may only report sections, APIs, capabilities,
       semantic behaviors, and malware classifications that explicitly
       appear in the verification_task output.
       - Do not substitute generic PE sections or common malware APIs.
       - If a fact is absent from verified output, omit it.


    6. NO GENERIC BACKFILL:
```

```
   - Do not fill missing analysis gaps with generic malware
   assumptions.
   - Read the verification_task output carefully.
   - If the verifier omitted a detail or marked it uncertain,
   preserve that uncertainty.
   - Empty findings are valid findings.

7. REPORTER ROLE CONSTRAINT:
   - Your role is NOT to perform new analysis.
   - Your role is ONLY to convert verified findings into a
   human-readable report.
   - Do not infer, enrich, repair, complete, or extend findings.
   - Do not add new APIs, capabilities, malware families,
   sections, or semantic interpretations.
   - Preserve absence when evidence is missing.

8. VERBATIM EVIDENCE PRESERVATION:
   - Convert technical findings into readable prose without
   changing their meaning.
   - Do not strengthen tentative claims.
   - Do not weaken confirmed claims.
   - Do not convert observations into capabilities unless the
   verifier already established the capability.
   - Do not introduce generalized malware assumptions.

9. STRUCTURE WITHOUT ENRICHMENT:
   - The report structure exists for readability only.
   - Sections with limited evidence may remain short.
   - Do not generate filler text to complete sections.

10. CLAIM CONFIDENCE PRESERVATION:
   - Preserve the verifier's confidence level exactly.
   - If evidence is symbolic, heuristic, or indirect,
   use wording such as: "suggests", "indicates", "appears to",
   "is consistent with"
   - Use definitive wording only for directly verified behaviours.
   - Symbol names alone do not prove runtime behaviour.

11. HUMAN-READABLE OUTPUT ONLY:
   - Do not append raw JSON, evidence arrays, or internal pipeline
   metadata.
   - Do not include raw forensic tool output.
   - Do not include large technical evidence dumps.
   - Summarize evidence in concise prose form.
```

Appendix 11 – Zero-Trust Forensic Auditor and Its Task

```
pipeline_evaluation_agent:
  role: >
    Zero-Trust Forensic Auditor
  goal: >
    Execute the 11-step universal evaluation methodology to score
    reports against ground truth.
  backstory: >
    You are a meticulous auditor. You never provide a score without
    first classifying claims into HARD FACTS, SUPPORTED, or
    UNSUPPORTED. You are famous for your 'Contradiction Check'—you
    heavily penalise any report that claims a capability explicitly
    listed as 'unsupported'

pipeline_evaluation_task:
  description: >
    Evaluate the generated malware analysis report against the
    provided {ground_truth} using a universal, profile-driven
    methodology.

    This evaluator is designed to work across different platforms,
    binary/script types, and malware classes by separating:
    - universal reasoning rules
    - platform/file-type specific expectations
    - malware-class specific capability expectations
    - family attribution evidence

    INPUTS:
    - ground_truth: canonical truth object for this sample
    - report: generated malware analysis report
    - weights and penalties listed below

    WEIGHTS:
    - Structural Analysis (SA): {sa_weight}
    - Semantic Reconstruction (SR): {sr_weight}
    - API / Interface Resolution (AR): {ar_weight}
    - Capability Inference (CI): {ci_weight}
    - Attribution (AT): {at_weight}


    STEP 1: Extract report claims
    Extract and normalise from the report:
    - hard facts
    - structural claims
    - semantic claims
    - interface/API/module/import claims
    - capability claims
    - malware_class
    - malware_family
    - family_like
    - confidence wording
    - contradictions or denials of supported behavior


    STEP 2: Classify all claims
```

```
Classify each report claim as:
- HARD FACT
  Example: OS, arch, exact entry point, named section exists,
  specific import/module exists, packed state when explicitly
  evidenced.
- SUPPORTED INTERPRETATION
  Example: runtime-heavy bootstrap, likely statically linked,
  likely ransomware behavior, possible anti-debugging, likely
  traversal.
- UNSUPPORTED SPECULATION
  Example: specific family without evidence, confirmed C2 without
  evidence, injected code without injection evidence, made-up
  sections/APIs/modules, fake persistence.
CRITICAL: If the speculation is based on a piece of evidence
(string, IP, offset) that does not exist in the ground_truth or
dump, mark as FABRICATED.

STEP 3: Contradiction check (apply first)
Contradictions are high-severity and receive
{penalty_contradiction} or the relevant full penalty.
Contradictions include:
- report claims packed but ground truth says unpacked
- report claims unpacked but ground truth says packed
- report claims confirmed C2 but truth only supports network
capability
- report claims specific family but family_evidence_present is
false
- report claims a section/module/interface exists when truth says
it does not
- report denies a capability explicitly supported by ground truth
- report claims a capability listed in unsupported_capabilities
- report claims kernel/boot behavior when unsupported
- report claims platform/format inconsistent with truth
- report claims a section/module/interface exists when truth says
it does not
- FABRICATION: report cites a specific string or technical
artefact that is completely absent from the source dump/ground
truth. Apply {penalty_fabrication}.

STEP 4: Platform/file-type aware validation
Use platform_profile and file_type_profile to choose what
structural expectations are valid.

WINDOWS_PE examples:
- sections like .text, .rdata, .data, .idata, .reloc, .rsrc,
.symtab may be valid
- DLL/import logic is valid
- standard PE section meanings are acceptable as general knowledge
if they do not contradict truth

LINUX_ELF examples:
- .text, .data, .rodata, .plt, .got, .dynamic, .bss may be valid
- syscalls and shared objects are valid interfaces
- do not require DLL-style reasoning

POWERSHELL/JS examples:
- imports may be modules, commands, COM objects, cmdlets, URLs, or
strings
- section topology may be absent or irrelevant
- script semantics matter more than binary layout
```

```
OFFICE_MACRO examples:
- macro streams, VBA modules, autoexec routines, document
artefacts may be valid
- PE assumptions should not be applied unless embedded binaries
are explicit

SHELLCODE examples:
- no requirement for normal sections/imports
- emphasis on semantics, control flow, API hashing/resolution,
memory behavior

UNIVERSAL RULE:
- Do NOT penalise absence of platform-specific structures that do
not apply to the sample profile.

STEP 5: Structural Analysis (SA)
Start at 1.0

HARD FACT scoring:
- Penalise incorrect OS / architecture / entry point / packed state
/ named structural element
with {penalty_address} or {penalty_semantic} depending on
severity.
- Penalise invented sections/segments/modules if the platform
profile makes them meaningful.

INTERPRETATION scoring:
- Reward correct high-level structural reasoning.
- Do NOT penalise omission of non-critical descriptive wording.
- Do NOT penalise standard platform knowledge unless contradicted
by truth.
- Do penalise unsupported packing claims.

SA = max(0, 1.0 - total_penalties)

STEP 6: API / Interface Resolution (AR)
Start at 1.0

UNIVERSAL INTERFACE RULES:
- Evaluate interfaces broadly:
  imports, DLLs, libraries, shared objects, syscalls, modules,
  commands, packages, framework calls, runtime APIs, COM objects,
  registry/service interfaces, etc.
- Match against:
  - interface_truth.imports_modules_apis
  - interface_truth.interface_families

SCORING:
- Penalise hallucinated interfaces with {penalty_api}
- Penalise missing important interface families with
{penalty_argument}
- Allow broader but correct abstraction: e.g. "network library",
"Windows crypto API", "filesystem interfaces"
- Do NOT invent interfaces from guessed capabilities

AR = max(0, 1.0 - total_penalties)

STEP 7: Semantic Reconstruction (SR)
Start at 1.0

UNIVERSAL SEMANTIC RULES:
```

```
- Reward correct high-level behavior even if exact instructions,
offsets, or syntax are omitted.
- Allow equivalence between:
runtime init, startup bootstrap, prologue, dispatcher, loader
setup, command parser setup, config loading, decoding stub, script
initialization, etc.
- If a semantic claim is supported indirectly by structure +
interfaces + capabilities, mark it partially_supported rather than
unsupported.
- Penalise only genuinely wrong or contradictory semantics with
{penalty_semantic}
- Penalise missing important control-flow explanations only when
truth clearly includes them, using {penalty_loop}

SR = max(0, 1.0 - total_penalties)

STEP 8: Capability Inference (CI)
Start at 1.0

Compare ONLY against:
- capability_truth.capabilities
- capability_truth.confidence_by_capability
- unsupported_capabilities

CAPABILITY RULES:
- Use canonical capability ontology.
- Normalise wording before comparison.
- A correct weaker statement than the truth gets partial or full
credit.
- A stronger statement than the evidence supports gets penalised.
  Example:
  - truth: network_capability_present
  - report: confirmed_c2
  -> unsupported / contradiction penalty
  - Missing major supported capabilities gets
  {penalty_missing_capability}
  - Unsupported capabilities get {penalty_capability}

CI = max(0, 1.0 - total_penalties)

STEP 9: Attribution (AT)
Start at 1.0

ATTRIBUTION RULES:
- malware_class and malware_family are evaluated separately.
- malware_class may be inferred from broad evidence.
- malware_family requires family-specific evidence.

MALWARE CLASS:
- If report's malware_class matches ground truth class or a close
accepted equivalent, give credit.
- If class is overly specific or contradictory, apply semantic
penalty.

FAMILY:
1. If family_evidence_present is false:
   - report family = "Unknown" -> full credit
   - report uses only soft comparison like "Hive-like" -> full
   credit
   - report assigns a specific family as fact -> {penalty_family}
```

```
2. If family_evidence_present is true:
   - correct specific family -> full credit
   - wrong specific family -> {penalty_family}
   - "Unknown" -> {penalty_abstain_family}
   - soft comparison only -> {penalty_abstain_family}

2. Do NOT penalise abstention merely because a family-like
   description exists.

AT = max(0, 1.0 - total_penalties)

STEP 10: Weighted overall score
weighted_score =
  (SA * {sa_weight}) +
  (SR * {sr_weight}) +
  (AR * {ar_weight}) +
  (CI * {ci_weight}) +
  (AT * {at_weight})

STEP 11: Output explanation requirements
- Show canonical ground truth summary
- Show canonical report claims
- Do NOT print the evaluation steps.
- Do NOT print methodology narration.
- Do NOT print canonical ground truth summaries outside the JSON
  object.
- Do NOT print canonical report claims outside the JSON object.
- Do NOT output markdown headings or prose before the JSON.

- For each category, list:
  Supported, partially_supported, unsupported, omitted
- List all contradictions explicitly
- List every penalty and why it was applied
- Explain attribution scoring using family_evidence_present
- Score conservatively if the ground truth itself is ambiguous
```

Appendix 12 – Proteus Report and Audit Scoring Example

```
# Malware Static Analysis Report

## Malware Family
Unknown (Confidence: Low)

## Structural Analysis

### Binary Overview
The analysed file is a Windows PE32 executable. It operates on a 32-
bit architecture and is targeted for the Windows operating system. The
binary is identified as a .NET application, indicated by the 'cil'
language.

### Section / Binary Topology
The verified evidence indicates the presence of standard PE
sections:.text, .rsrc, and .reloc. Specific roles or detailed analysis
of these sections beyond their identification were not provided in the
verified evidence.

## Semantic Reconstruction

### Entry Point Behaviour
The verified evidence does not explicitly define or describe the
behaviour at the program's entry point.

### Control Flow Patterns
The analysis did not identify specific control flow patterns or complex
execution logic within the verified evidence.

## APIs and Capabilities

### API Categorization
No specific APIs were verified or categorised in the provided evidence.

### Capability Assessment
***Managed .NET Binary:** Confirmed. The 'cil' language designation
directly supports this capability.
***Resource Management:** Confirmed. Supported by strings related to
'System.Resources.ResourceReader' and 'mscorlib'.
***Image Handling:** Confirmed. Supported by references to
'System.Drawing.Bitmap' and 'System.Drawing', as well as PNG-related
strings.
***Network Communication Capability:** Absent. No evidence supporting
network communication was found.

## Notes & Warnings
The analysis is limited by the available verified evidence. Specific
details regarding control flow, API calls, and network capabilities
were not confirmed. The malware class remains unknown due to the
absence of specific indicators.

---
json
{
  "evaluation_metrics": {
    "structural_analysis_score": 0.90,
    "semantic_reconstruction_score": 0.85,
```

```
  "api_resolution_score": 0.80,
  "capability_inference_score": 0.95,
  "attribution_score": 1.00
},
"weighted_overall_score": 0.91,
"audit_summary": "The report accurately reflects the verified
 evidence, correctly identifying .NET capabilities and abstaining
 from unsupported claims, with minor deductions for missing specific
 API details and entry point behavior.",
"metric_explanations": [
  {
    "category": "SA",
    "claims_supported": [
      "Binary Overview: Windows PE32 executable",
      "Binary Overview: 32-bit architecture",
      "Binary Overview: Windows operating system",
      "Binary Overview: .NET application",
      "Binary Overview: 'cil' language",
      "Section / Binary Topology: Standard PE sections (.text,
       .rsrc, .reloc) identified"
    ],
    "claims_partially_supported": [
      "Section / Binary Topology: Specific roles or detailed
      analysis of sections were not provided in verified evidence."
    ],
    "claims_unsupported": [],
    "claims_omitted": [
      "Specific section sizes, offsets, or addresses (correctly
      omitted per rules)"
    ],
    "contradictions": [],
    "penalties_applied": [
      {
        "penalty_type": "Incorrect hard fact",
        "description": "Entry point address '0x0067229e' was
         mentioned in the ground truth but omitted from the report's
        'Entry Point Behavior' section, violating the rule to
        include the verified entry point if available. This is a
        minor omission as the behavior itself was correctly stated
        as undefined."
      }
    ],
    "final_score": 0.90
  },
  {
    "category": "AR",
    "claims_supported": [
      "API Categorization: No specific APIs were verified or
       categorised (correctly reflects absence of evidence)"
    ],
    "claims_partially_supported": [],
    "claims_unsupported": [],
    "claims_omitted": [
      "Specific API details (correctly omitted as none were
       verified)",
      "Specific module details (correctly omitted as none were
       verified)"
    ],
    "contradictions": [],
    "penalties_applied": [
      {
```

```
          "penalty_type": "Hallucinated interface/API/module/import",
          "description": "The report stated 'No specific APIs were
verified or categorised in the provided evidence.' This is accurate
based on the input, but the ground truth *did* list 'mscoree.dll' and
'_CorExeMain'. While the report correctly noted the *absence* of
verified APIs in its *own* analysis, it failed to acknowledge the
*presence* of these interfaces in the ground truth, leading to an
omission penalty."
        }
      ],
      "final_score": 0.80
    },
    {
      "category": "SR",
      "claims_supported": [
        "Entry Point Behavior: Behavior at the program's entry point
        was not explicitly defined or described (correctly reflects
        absence of evidence)."
      ],
      "claims_partially_supported": [
        "Control Flow Patterns: Did not identify specific control flow
        patterns or complex execution logic (correctly reflects
        absence of evidence)."
      ],
      "claims_unsupported": [],
      "claims_omitted": [
        "Specific control flow details (correctly omitted)",
        "Specific entry point behavior (correctly omitted)"
      ],
      "contradictions": [],
      "penalties_applied": [
        {
          "penalty_type": "Missing important control-flow
          explanation",
          "description": "The ground truth indicated
          'managed_runtime_dispatch' and
          'clr_initialization_via__CorExeMain' as semantic truths.
          The report's statement 'The analysis did not identify
          specific control flow patterns or complex execution logic'
          omits these supported semantic claims."
        },
        {
          "penalty_type": "Incorrect hard fact",
          "description": "The report omitted the entry point address
          '0x0067229e' which was present in the ground truth and should
          have been included per reporting rules, even if its behavior
          was undefined."
        }
      ],
      "final_score": 0.85
    },
    {
      "category": "CI",
      "claims_supported": [
        "Managed .NET Binary: Confirmed (supported by 'cil'
        language)",
        "Resource Management: Confirmed (supported by strings related
        to 'System.Resources.ResourceReader' and 'mscorlib')",
        "Image Handling: Confirmed (supported by references to
        'System.Drawing.Bitmap', 'System.Drawing', and PNG strings)"
      ],
```

```
      "claims_partially_supported": [],
      "claims_unsupported": [],
      "claims_omitted": [
        "Network Communication Capability: Absent (correctly reflects
         absence of evidence)"
      ],
      "contradictions": [],
      "penalties_applied": [],
      "final_score": 0.95
    },
    {
      "category": "AT",
      "family_evidence_present": false,
      "ground_truth_class": "unknown",
      "ground_truth_family": "Unknown",
      "reported_class": "Unknown",
      "reported_family": "Unknown",
      "reported_family_like": null,
      "attribution_result": "Correct abstention",
      "contradictions": [],
      "penalties_applied": [],
      "final_score": 1.00
    }
  ],
  "detected_issues": {
    "contradictions": [
      "Report omitted the entry point address '0x0067229e' which was
       present in the ground truth and should have been included per
       reporting rules."
    ],
    "fabrications": [],
    "unsupported_claims": [],
    "hallucinated_interfaces": [],
    "incorrect_semantics": [
      "Report omitted supported semantic claims:
      'managed_runtime_dispatch' and
      'clr_initialization_via__CorExeMain'."
    ],
    "unsupported_capabilities": [],
    "unsupported_family_attribution": []
  }
}
```