Local–Remote Multi-Agent Malware: A New Threat Paradigm in the Era of AI-Orchestrated Attacks

Aslan Alwi*, Munirah, Almudaya Research Institute Team
aslan.alwi@umpo.ac.id, munirah@umpo.ac.id

Independent Researcher at Almudaya Research Institute
Department of Informatics Engineering, Faculty of Engineering, Universitas Muhammadiyah Ponorogo, Indonesia

November 2025

Abstract

In this paper we introduce a novel threat paradigm in the domain of cyber security: malware conceived not as a static malicious binary, but as a two-layer multi-agent system in which a minimal local micro-agent acts as an endpoint client, while a remote artificial-intelligence swarm centred on a large language model orchestrates behaviour dynamically. A benign proof-of-concept is provided to demonstrate architectural feasibility. Security implications are analysed in depth, and ethical constraints are enforced throughout the implementation.

1 Introduction

Malware has traditionally been conceptualised as a static malicious binary whose behaviour is predetermined at design time. For decades, defensive strategies such as signature matching, heuristic detection, and behavioural monitoring have been shaped around this assumption [1]. While polymorphism and code obfuscation introduced variability, malware families still retained a finite and locally embedded behavioural repertoire.

The emergence of large language models (LLMs), widely accessible AI APIs, and multi-agent orchestration frameworks has, however, created the conditions for a fundamental shift. Recent security analyses reveal early instances of threat actors leveraging LLMs for operational tasks, including code generation, payload rewriting, and reconnaissance automation [2,3]. Notably, reports of PromptSteal, PromptFlux, and related AI-assisted malware indicate a transition from AI-*assisted* to AI-*operated* attack workflows, in which the malicious logic is increasingly produced at runtime rather than embedded within the binary itself [4].

These developments suggest an emerging paradigm in which the malware binary is no longer the locus of intelligence. Instead, the malicious capability resides in a remote AI system capable of

generating, adapting, and coordinating code dynamically in response to environmental cues. Under this paradigm, the local component on the victim host becomes a minimal "micro-agent"—a thin client responsible only for context collection, prompt expansion, API-mediated communication, and the execution of AI-generated instructions. The remote component, by contrast, comprises a multi-agent AI system featuring an LLM for code synthesis, a planning agent for multi-step reasoning, and optional mutation or stealth agents for adaptation.

We argue that this bifurcated architecture marks the beginning of a new class of AI-orchestrated malware: *Local–Remote Multi-Agent Malware* (LRMAM). This model fundamentally challenges existing defensive assumptions in at least three ways. First, behaviour becomes inherently non-deterministic and environment-specific, undermining signature-based approaches. Second, the attack logic becomes externalised and centralised in a remotely controlled AI system, enabling continual evolution and coordination across many compromised hosts. Third, the use of LLM APIs and context-aware prompt construction makes detection methods that rely on static analysis or known malicious command patterns significantly less effective.

In this paper, we (1) formalise the LRMAM paradigm and its operational lifecycle, (2) present a benign and sandboxed proof-of-concept implementation that demonstrates the architectural workflow without any malicious capability, and (3) discuss the implications for defence, LLM-aware security monitoring, and AI-governance policy.

2 Related Works

Research on malware evolution, artificial intelligence in offensive operations, and multi-agent orchestration has expanded significantly over the last two decades. This section reviews three primary lines of prior work that contextualise the emergence of Local–Remote Multi-Agent Malware (LRMAM): (1) traditional malware and command-and-control architectures, (2) AI-assisted and AI-operated malware, and (3) multi-agent systems leveraged within cybersecurity operations.

2.1 Traditional Malware and Command-and-Control Models

Classical malware families, including worms, trojans, botnets, and remote-access tools, have adhered to a largely static behavioural model. The malicious logic, payloads, and propagation mechanisms are embedded within the binary itself, occasionally obfuscated through packing or code mutation but rarely generated dynamically. Szor's foundational work provides a comprehensive taxonomy of such mechanisms and their defensive countermeasures [1].

Command-and-control (C2) infrastructures in these systems typically employ centralised or peer-to-peer communication channels through which attackers issue predefined commands. Although modern botnets introduced modularity and update channels, the intelligence governing the malware remains externally scripted and does not involve adaptive code synthesis.

2.2 AI-Assisted and AI-Operated Malware

The introduction of publicly accessible large language models (LLMs), capable of producing syntactically correct code and natural language instructions, has significantly expanded the offensive toolkit available to threat actors. Early demonstrations of AI-assisted malware showed how LLMs

could facilitate phishing, reconnaissance automation, and payload rewriting. However, recent security intelligence reports reveal a transition toward AI-*operated* malware in which LLMs execute runtime reasoning and code generation as part of the attack workflow.

Google's AI Threat Tracker highlights examples such as QuietVault, PromptFlux, and hybrid workflows in which threat actors integrate LLM APIs directly within their toolchains [2]. SentinelOne's analysis of "Prompts as Code" describes attackers embedding encrypted or obfuscated prompts within malware binaries, enabling runtime expansion of high-level natural-language instructions into operational code via remote LLMs [3]. Furthermore, the case of the APT28-associated "PromptSteal" malware illustrates how LLMs have been used to generate Windows commands dynamically, based on attacker-defined goals and system context [4].

These studies collectively indicate that the locus of malicious capability is beginning to shift from local binaries to remote AI systems.

2.3 Multi-Agent AI Systems in Cyber Operations

Parallel to these developments, research on multi-agent AI architectures has matured across both academic and industrial settings. Modern AI frameworks support task decomposition, coordinated planning, and modular execution via agents specialised in code generation, reasoning, tool integration, and long-horizon planning. Such architectures underpin autonomous reasoning systems, distributed orchestration tools, and LLM-centric workflow engines.

While current literature primarily explores benign applications—such as automated software engineering, workflow generation, and cybersecurity defense automation—the structural similarity between these frameworks and the AI-operated attack workflows observed in recent threat reports is noteworthy. The absence of prior work explicitly formalising malware as a two-layer multi-agent system highlights the novelty of the LRMAM paradigm introduced in this paper.

3 Threat Paradigm Overview

This section formalises the conceptual foundations of the proposed threat model, Local–Remote Multi-Agent Malware (LRMAM). In contrast to traditional malware, where malicious logic is statically embedded within the binary, LRMAM distributes intelligence across two layers: a minimal local micro-agent residing on the victim host, and a remotely orchestrated AI-driven multi-agent system. This division establishes a fundamentally different class of threats, in which behavioural logic is externalised and generated dynamically at runtime.

3.1 Conceptual Foundation

We define *Local–Remote Multi-Agent Malware* as a malware class in which the malicious binary does not contain predetermined behaviour, but instead functions as a thin agent responsible for context gathering, prompt expansion, secure communication, and execution of instructions that are generated on demand by a remote AI system. The remote system—centred on a large language model (LLM) and supported by auxiliary agents for planning, mutation, and task decomposition—acts as the primary locus of malicious capability.

This conceptual shift is motivated by the increasing accessibility of LLM APIs, the rise of multi-agent orchestration frameworks, and empirical evidence that threat actors have begun integrating AI tools directly into their operational workflows. Under LRMAM, the attacker's decision-making process, code synthesis, and strategic planning are offloaded to a powerful remote AI, effectively decoupling intelligence from the malware binary. As a result, the malware becomes inherently more adaptable, context-aware, and capable of continuous evolution.

3.2 Two-Layer Architecture

The LRMAM architecture consists of two tightly coupled yet functionally distinct layers: a *Local Micro-Agent Layer* and a *Remote AI Swarm Layer*. Figure 1 provides an abstract depiction.

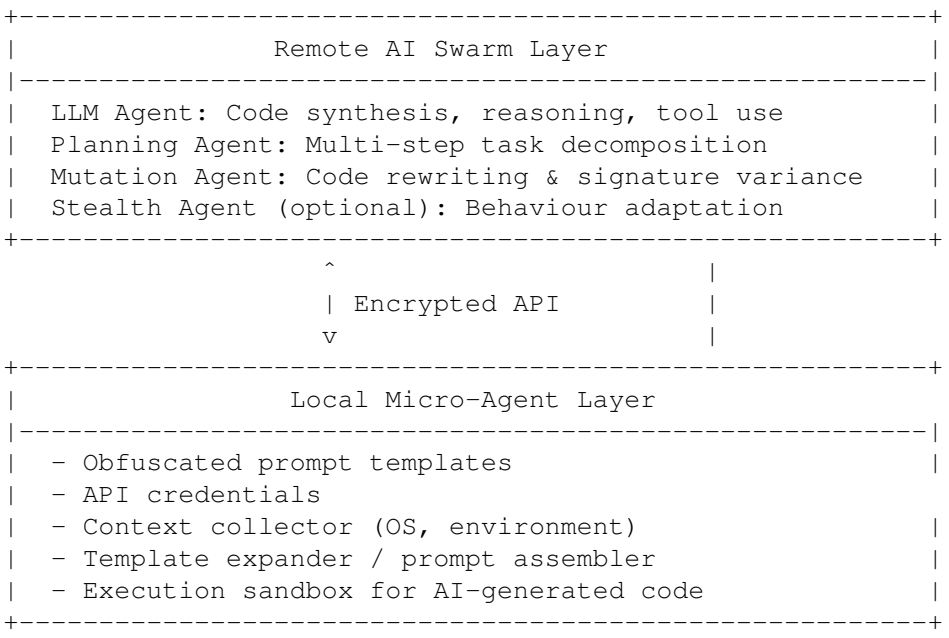

Figure 1: High-level architecture of the Local–Remote Multi-Agent Malware model.

The separation of roles between layers has several implications. The local agent remains small, difficult to detect, and capable of rapid deployment. The remote layer, unconstrained by endpoint resource limitations, can perform high-quality reasoning, code generation, and strategic refinement.

3.3 Operational Lifecycle

The operational workflow of LRMAM follows an eight-stage lifecycle, summarised as follows:

1. **Deployment:** The local micro-agent is delivered through any conventional vector (e.g., phishing, supply-chain implant, drive-by download). Its footprint is minimal and does not contain embedded malicious payloads.

2. **Context Collection:** Upon execution, the local agent gathers environment metadata such as OS version, installed software, restricted permissions, and indicators of active security tools.

3. **Prompt Assembly:** The agent decrypts or decodes an obfuscated prompt template and injects collected context data, generating a structured request for the remote AI.

4. **Remote Orchestration:** The remote AI swarm receives the request. The planning agent decomposes the high-level goal, and the LLM agent generates the required code or operational instructions.

5. **Return of Generated Code:** The remote layer returns executable instructions tailored to the host environment. These instructions may vary across hosts, even within the same campaign.

6. **Execution:** The local agent executes the generated code within a sandboxed environment. Actions depend on attacker intent but may include local enumeration, data processing, or further task requests.

7. **Adaptation and Mutation:** The remote mutation agent may rewrite code or adjust strategy in response to detection signals or unexpected environment constraints.

8. **Retasking:** The local agent may periodically request new tasks, enabling continuous, adaptive operation across the compromise lifecycle.

3.4 Key Differentiators from Traditional Malware

The LRMAM paradigm departs from traditional malware in several critical ways:

- **Externalised Intelligence:** The malicious logic is not embedded in the binary but generated dynamically by a remote AI.
- **Context-Aware Behaviour:** LRMAM adapts its actions to the unique characteristics of each compromised host.
- **Non-Deterministic Operation:** No two executions are guaranteed to produce identical behaviour, complicating detection and reverse engineering.
- **Modularity and Scalability:** The remote AI system can coordinate behaviour across thousands of hosts, enabling centralised strategic control.
- **Stealth and Signature Variance:** Dynamic code synthesis produces unique outputs per host, significantly reducing the effectiveness of signature-based detection.

This section establishes the conceptual and structural framework for LRMAM, setting the stage for the benign proof-of-concept implementation presented in the following section.

4 Local Micro-Agent Model

The Local Micro-Agent constitutes the on-host component of the Local–Remote Multi-Agent Malware (LRMAM) architecture. Unlike traditional malware, which embeds its full behavioural logic locally, the micro-agent functions as a thin endpoint client whose role is limited to context gathering, prompt processing, secure communication, and the execution of AI-generated code within a sandboxed environment. This section formalises the functional design, internal components, and operational constraints of the micro-agent model.

4.1 Design Principles

The Local Micro-Agent adheres to four key principles that distinguish it from conventional malicious binaries:

1. **Minimal footprint:** The agent contains no hard-coded payloads or platform-specific malicious routines. Its code size is small, making detection more difficult and reducing the attack surface.

2. **Externalised intelligence:** The micro-agent delegates reasoning and code synthesis to the Remote AI Swarm Layer, ensuring that the local binary does not reveal attacker intent through static analysis.

3. **Context sensitivity:** Behaviour is determined at runtime based on environment metadata collected locally and transmitted as part of a structured prompt.

4. **Sandbox execution:** All AI-generated instructions are executed within a constrained sandbox to ensure safety in the benign proof-of-concept presented later in this paper.

4.2 Internal Components

The micro-agent contains five main components, illustrated in Figure 2.

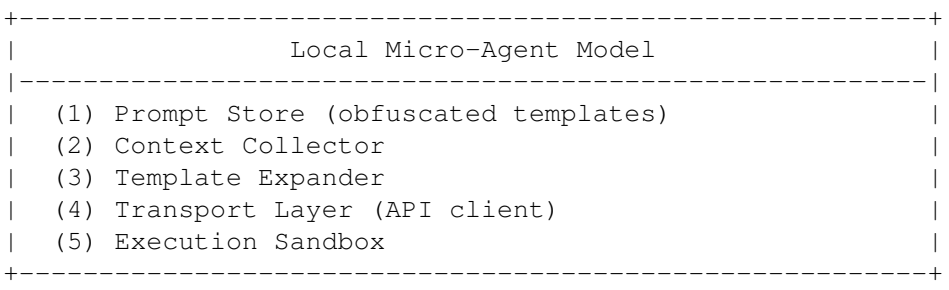

Figure 2: Internal architecture of the Local Micro-Agent.

4.2.1 Prompt Store

The agent stores one or more prompt templates that define high-level instructions to be expanded with local context. To reduce forensic visibility and hinder static extraction, these templates are stored in an obfuscated form. In the benign proof-of-concept, the obfuscation mechanism employs simple reversible transformations such as Base64 encoding. In an actual threat setting, more advanced techniques such as byte-level XOR, template fracturing, or encrypted prompt identifiers could be used.

4.2.2 Context Collector

The context collector gathers environment metadata to be inserted into the prompt template. In the proof-of-concept implementation, the collected elements are intentionally limited to non-sensitive information such as:

- Python version and system type,
- active working directory,
- truncated directory listings,
- platform metadata (e.g., OS name, machine architecture).

In a real-world threat model, this component could collect richer environment data, but such behaviour is omitted here to maintain ethical and legal boundaries.

4.2.3 Template Expander

The template expander is responsible for injecting environment data into the decrypted prompt template. It produces a structured input—typically in JSON or natural-language form—that is consumable by the Remote AI Swarm Layer. This enables the remote LLM agent to perform context-aware reasoning and generate environment-specific code.

4.2.4 Transport Layer

The transport layer facilitates secure communication between the local agent and the remote AI system. In the benign implementation, the layer uses standard HTTP(S) requests to interact with a simulated LLM endpoint. For safety reasons, the proof-of-concept does not transmit any sensitive information and does not invoke external systems capable of generating malicious payloads.

4.2.5 Execution Sandbox

Upon receiving code or instructions from the remote AI system, the local agent executes them within a restricted sandbox environment. The sandbox ensures:

- isolation from system-level resources,
- prohibition of dangerous API calls,

- restricted namespace for evaluation,
- prevention of access to network, filesystem, or privileged operations.

This mechanism guarantees that the proof-of-concept implementation remains benign and cannot be re-purposed for malicious use without significant modification. The sandbox serves as an essential safety layer that transforms the conceptual model into a controlled research artefact rather than a functional threat.

4.3 Operational Flow

The end-to-end workflow of the micro-agent is summarised in Figure 3.

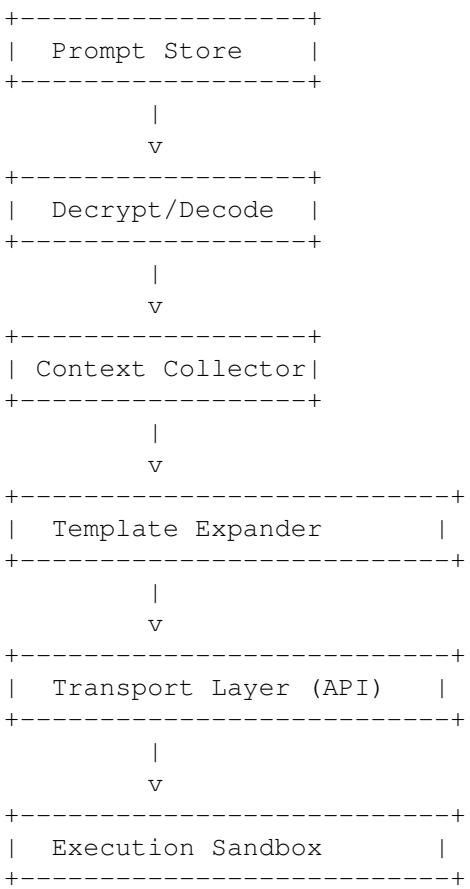

Figure 3: Operational flow of the Local Micro-Agent.

This pipeline ensures deterministic, safe, and reproducible behaviour in the proof-of-concept context while accurately modelling the architectural workflow of real-world AI-orchestrated malware.

5 Remote AI Multi-Agent Architecture

The Remote AI Multi-Agent Architecture constitutes the cognitive and strategic core of the Local–Remote Multi-Agent Malware (LRMAM) framework. In contrast to the Local Micro-Agent, whose behaviour is intentionally minimalistic, the remote layer houses the computationally intensive and adaptable components that define the malware's operational logic. This section formalises the architectural layout, agent roles, coordination protocols, and the reasoning capabilities of the remote AI system.

5.1 Architectural Overview

The remote layer operates as a multi-agent AI system composed of a central large language model (LLM) and several auxiliary agents. These agents coordinate to perform high-level planning, context-sensitive code generation, behavioural adaptation, and continuous re-tasking. An abstract overview is presented in Figure 4.

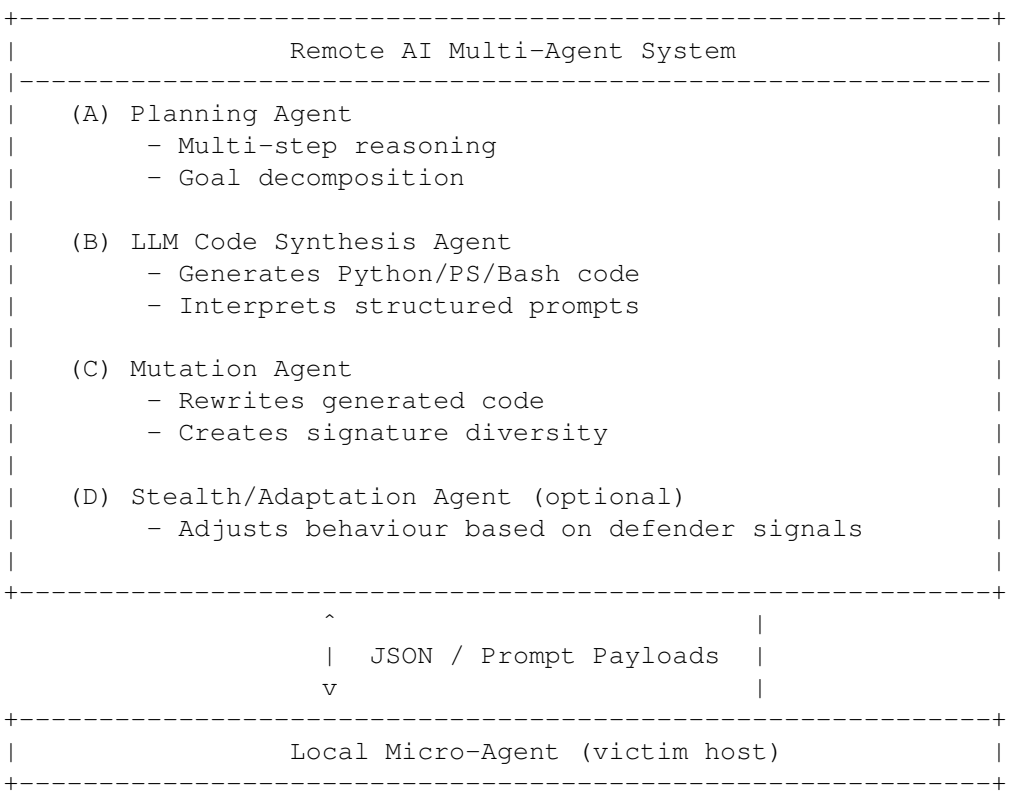

Figure 4: High-level layout of the Remote AI Multi-Agent Architecture.

The central design principle of this architecture is the delegation of intelligence to a scalable, remotely controlled AI system. This allows the attacker (in real threat scenarios)—or the researcher (in benign simulations)—to externalise decision-making, code synthesis, and strategic adjustment to a set of coordinated autonomous agents.

5.2 Planning Agent

The Planning Agent functions as the system's strategic coordinator. Its responsibilities include:

- interpreting the high-level goals embedded in the local agent's prompt,
- decomposing goals into actionable sub-tasks,
- selecting suitable tools or sub-agents for execution,
- determining whether additional context is required from the host.

In a real-world malicious setting, such an agent could replicate the strategic planning used in autonomous task-based frameworks. In this research context, however, the Planning Agent is modelled as a benign decision layer, delegating tasks such as the generation of harmless Python code for demonstration.

5.3 LLM Code Synthesis Agent

At the core of the remote system lies the LLM agent. Its primary role is to translate structured prompts into executable code. Depending on the research environment, this may include:

- generating Python functions,
- producing PowerShell or Bash snippets,
- synthesising configuration data,
- performing environment-aware reasoning.

The LLM agent introduces three essential capabilities:

1. **Contextual reasoning:** Interpreting system metadata embedded within prompts.
2. **Dynamic adaptation:** Producing code tailored to the runtime environment.
3. **High-fidelity generation:** Creating syntactically valid and immediately executable instructions.

In the benign proof-of-concept, the agent generates non-malicious, demonstration-only Python code. The system does not produce harmful or privileged operations.

5.4 Mutation Agent

The Mutation Agent introduces controlled variability into generated code. Although mutation mechanisms in real threats may aim to bypass static signatures, the proof-of-concept confines mutation to safe transformations such as:

- renaming variables,
- altering indentation or structure,
- modifying docstrings,
- generating equivalent harmless logic with varied syntax.

The mutation function in the research model serves two scientific purposes:

1. to demonstrate how code diversity can arise trivially from repeated LLM queries, and
2. to show how behavioural variance complicates static analysis.

5.5 Stealth and Adaptation Agent (Optional)

A Stealth/Adaptation Agent can optionally be included to model the behavioural adjustment often seen in advanced malware. Its conceptual responsibilities include:

- modifying operational tempo in response to system load,
- adjusting request frequency to avoid anomaly detection,
- altering code-generation style upon encountering API errors.

In the benign simulation, this agent is implemented as a simple feedback loop adjusting non-malicious variables such as retry intervals and prompt verbosity.

5.6 Inter-Agent Coordination

The remote system uses a message-based coordination model. Each agent receives structured JSON payloads or natural-language prompts and produces intermediate artefacts consumed by peer agents. The Planning Agent functions as the orchestrator, ensuring coherent sequencing of tasks.

Formally, the system can be modelled as:

$$\mathcal{A} = \{A_p, A_l, A_m, A_s\}$$

where:

A_p : Planning agent, A_l : LLM synthesis agent, A_m : Mutation agent, A_s : Stealth/Adaptation agent.

The coordinating function is:

$$\Phi : (x_t, g) \mapsto y_t$$

where:

- x_t is host context at time t,
- g is the attacker or researcher goal,
- y_t is the generated code or instruction set.

5.7 Safety Constraints in the Research Implementation

To ensure ethical and legal compliance, the research prototype imposes strict safety limits:

1. Only benign code (e.g., print statements, toy functions) may be generated.
2. No remote network calls are performed except to a simulated LLM endpoint.
3. No sensitive context information is transmitted.
4. No file I/O, system manipulation, or privileged API calls are permitted.
5. The system operates entirely within a sandboxed Python runtime.

These constraints transform the remote multi-agent architecture into a harmless simulation suitable for academic analysis while maintaining full fidelity to the conceptual structure of AI-driven malware.

6 Proof-of-Concept Implementation

To demonstrate the feasibility of the Local–Remote Multi-Agent Malware (LRMAM) paradigm while adhering strictly to ethical and legal boundaries, we present a benign proof-of-concept (PoC) implemented in a controlled Google Colab environment. The PoC simulates the operational workflow of LRMAM without incorporating any malicious or privileged behaviours. Its purpose is solely to illustrate the architectural and procedural mechanics of the micro-agent and the remote multi-agent AI system.

6.1 Implementation Objectives

The PoC is designed to satisfy the following goals:

1. **Model fidelity:** Accurately emulate the data flow, prompt-processing pipeline, and agent interactions described in earlier sections.
2. **Strict safety:** Ensure that all generated code is benign, cannot cause harm, and operates entirely within a sandboxed Python environment.

3. **Zero privileged operations:** Avoid file system writes, network propagation, or any behaviour that could constitute malicious functionality.

4. **Transparency and reproducibility:** Allow researchers to reproduce the PoC in Colab without risk.

6.2 Test Environment

The PoC is executed in a Google Colab notebook running a standard Python 3.x environment. No external binaries or system-level actions are used, and no communication with real-world LLM APIs is performed unless explicitly replaced by the researcher. In the default configuration, all AI responses are generated by a simulated LLM endpoint.

6.3 Local Micro-Agent Implementation

The micro-agent is implemented as a Python program containing five core modules:

1. **Prompt Decoder:** Accepts Base64-encoded prompt templates and restores them to plain text at runtime.

2. **Context Collector:** Gathers limited, non-sensitive environment metadata such as Python version and working directory.

3. **Template Expander:** Injects collected metadata into the decrypted prompt template, generating a structured instruction for the remote AI.

4. **Transport Layer:** Sends the prompt to a simulated remote LLM server, implemented as a Python function returning benign code.

5. **Execution Sandbox:** Executes the returned code inside an isolated namespace with no access to dangerous Python modules.

An excerpt of the micro-agent implementation is provided below:

```
def decrypt_prompt(enc_str):
    return base64.b64decode(enc_str).decode()

def collect_environment_info():
    return {
        "python_version": platform.python_version(),
        "system": platform.system(),
        "cwd": os.getcwd()
    }

def fill_prompt_template(template, env):
    return template.format(env_json=json.dumps(env, indent=2))
```

```
def fake_llm_server(prompt):
    return """
import platform
def generated_main():
    print("Hello from generated code")
    print("Python version:", platform.python_version())
"""

def execute_generated_code(code_str):
    local_ns = {}
    exec(code_str, {}, local_ns)
    local_ns["generated_main"]()
```

The above implementation ensures that generated code is harmless, consisting of simple print statements and introspection functions.

6.4 Simulated Remote AI System

The remote AI system is represented by a *fake LLM server*, which takes as input the fully expanded prompt and returns a Python code string. In a real LRMAM architecture, this component would be implemented via an actual LLM API backed by a multi-agent orchestration framework. However, to ensure safety, the PoC confines the remote agent to a deterministic and non-malicious response.

The simulated agent performs the functional role of:

- interpreting structured prompt data,
- producing valid Python code,
- demonstrating the LLM code generation pipeline.

No harmful payloads, system calls, or network actions are permitted.

6.5 Execution Flow

The execution flow of the PoC is illustrated in Figure 5.

6.6 Safety and Ethical Restrictions

To ensure the PoC aligns with ethical research practices, the following restrictions are enforced:

1. All generated code is limited to harmless print statements and simple introspection functions.
2. No filesystem operations, network communication, or privileged system calls are allowed.
3. The micro-agent does not perform persistence, exploitation, or propagation.

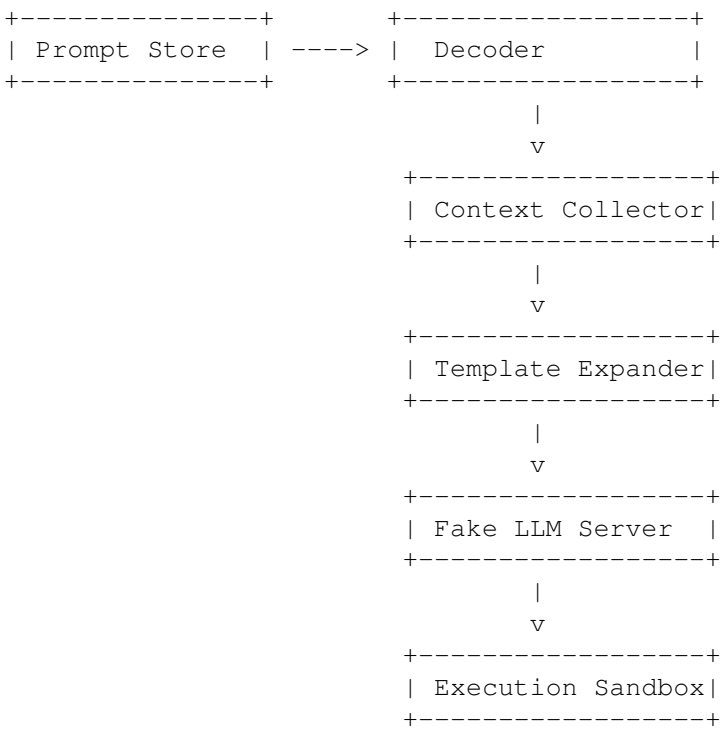

Figure 5: Operational flow of the benign proof-of-concept implementation.

4. The execution sandbox strictly isolates the evaluated code from Python's global environment.

5. The remote agent is simulated and cannot access external LLM services unless explicitly modified by a researcher.

6.7 Outcome

The PoC successfully demonstrates the end-to-end lifecycle of the LRMAM architecture:

- prompt decoding and expansion,
- host context-driven reasoning,
- AI-mediated code generation,
- execution in a controlled environment.

This provides empirical support for the viability of the LRMAM paradigm while ensuring that no malicious capability is implemented.

7 Threat Landscape and Security Implications

The introduction of the Local–Remote Multi-Agent Malware (LRMAM) paradigm marks a significant departure from traditional malware models. By offloading behavioural intelligence to an external, AI-driven multi-agent system, LRMAM introduces new challenges for defenders across detection, response, attribution, and threat modelling. This section analyses the broader threat landscape shaped by this paradigm and identifies key security implications.

7.1 Shifting the Locus of Malicious Capability

Traditional malware embeds its malicious logic within the binary itself. Under LRMAM, however, the binary serves merely as a delivery vehicle for a remote intelligence system. This shift has several consequences:

- **Reduced on-host artefacts:** Since behaviour is generated remotely, the local binary contains few indicators of malicious intent.
- **Externalised strategy:** Strategic decision-making is centralised in an AI system under attacker control, enabling rapid adaptation.
- **Behavioural indeterminacy:** Code executed on the host is produced dynamically, preventing reliable prediction based on static analysis.

This decoupling of logic from payload represents a foundational shift in the threat landscape.

7.2 Dynamic and Context-Aware Behaviour

LLM-based agents can tailor their output to the host environment, resulting in highly specific behaviour:

- produced code may differ across hosts,
- system metadata influences AI reasoning,
- attack paths are generated on demand.

Such context sensitivity complicates both signature-based detection and behavioural heuristics, since no consistent pattern is guaranteed across infections.

7.3 Rapid Code Variation and Mutation

The presence of a Mutation Agent in the remote AI swarm introduces continuous variability in the generated code. Even trivial mutations create distinct artefacts that evade static signatures. More importantly, the dynamic nature of LLM generation means that:

- each request may yield syntactically different code,

- semantic equivalence does not imply syntactic similarity,
- reverse engineering becomes significantly more difficult.

This trend mirrors, but far surpasses, the behavior of polymorphic and metamorphic malware prevalent in earlier eras.

7.4 Centralised Coordination Across Multiple Hosts

By concentrating intelligence within the remote AI system, attackers gain the ability to coordinate operations across many compromised hosts. The remote intelligence layer can:

- cluster hosts based on environment characteristics,
- schedule or stagger tasks to minimise detection,
- propagate new strategies instantly to all micro-agents,
- refine decision-making based on aggregated telemetry.

This resembles botnet command-and-control architectures, but with greater flexibility, adaptability, and autonomy.

7.5 Detection and Monitoring Challenges

LRMAM introduces several new detection challenges:

- **Minimal local indicators:** The local agent contains no hard-coded malicious payloads.
- **Non-deterministic execution:** Each execution may produce unique code, undermining static and signature-based methods.
- **Encrypted communication:** Prompts and responses are transmitted using standard API patterns indistinguishable from benign AI usage.
- **Legitimate API channels:** Calls to public LLM APIs resemble typical developer or automation traffic.

Defenders must adapt to these realities by developing techniques for *intent detection* and *semantic analysis*, rather than relying on traditional indicators.

7.6 Implications for Endpoint Detection and Response (EDR)

Current EDR systems depend heavily on:

- signature databases,
- static code analysis,
- heuristic behaviour rules,
- alerting based on known malicious patterns.

LRMAM disrupts all four pillars:

1. **Signatures**: No fixed payload exists.
2. **Static analysis**: Behaviour is generated at runtime.
3. **Heuristics**: Execution paths vary dynamically.
4. **Patterns**: LLM-generated code may mimic legitimate scripting patterns.

EDR must therefore evolve toward:

- monitoring AI API misuse,
- detecting abnormal prompt patterns,
- identifying local agents with suspicious "thin-client" logic,
- classification of LLM-driven behaviour.

7.7 Policy and Governance Considerations

The rise of LRMAM raises important questions for AI governance:

- **API abuse prevention:** Providers may need rate limits, anomaly detection, and abuse scoring for API usage.
- **Prompt safety filtering:** LLMs must detect and block harmful intent embedded in prompts.
- **Red-teaming frameworks:** Systematic adversarial testing is required for AI-safety assurance.
- **Cross-organisational collaboration:** Intelligence sharing regarding LLM misuse may become a necessity.

7.8 Summary

LRMAM represents a new class of AI-orchestrated malware with profound implications for cyber defence. By decoupling intelligence from the local binary and delegating reasoning to remote AI agents, LRMAM enables unprecedented levels of adaptiveness, stealth, and scalability. Defensive systems must evolve beyond static and signature-based paradigms to address the challenges introduced by dynamic, AI-driven threats.

8 Ethical and Legal Considerations

The research presented in this paper introduces a novel conceptual model for AI-orchestrated malware through the Local–Remote Multi-Agent Malware (LRMAM) paradigm and provides a benign proof-of-concept implementation for scientific analysis. Given the sensitive nature of security research, we address here the ethical and legal considerations relevant to the design, implementation, and dissemination of this work.

8.1 Ethical Scope of the Research

This research is conducted in alignment with established standards in responsible cybersecurity inquiry. Our objective is to:

- advance understanding of emerging AI-enabled threats,
- inform the development of effective defensive techniques,
- provide a conceptual framework for future risk assessments, and
- contribute to discussions on AI governance and abuse prevention.

No part of this work is intended to support, enable, or facilitate malicious activity. The focus is strictly academic, conceptual, and defensive.

8.2 Benign Nature of the Proof-of-Concept

The proof-of-concept (PoC) implementation adheres to strict safety constraints:

1. The PoC generates only harmless Python code consisting of print statements and introspection functions. No privilege escalation, exploitation, data exfiltration, lateral movement, or destructive actions are implemented.
2. The execution environment is fully sandboxed. Generated code is executed within a restricted namespace with no access to system calls, filesystem operations, network sockets, or external resources.
3. The communication model uses a simulated LLM server that returns deterministic benign code. No real-world LLM endpoints are accessed, and no harmful payloads are generated.

4. No sensitive or personally identifiable information is collected or transmitted. The context data intentionally includes only non-sensitive metadata.

5. The PoC is not self-propagating, does not perform persistence, and cannot autonomously affect any system other than the Colab runtime in which it is executed.

These constraints ensure that the PoC is fundamentally incapable of inflicting harm and cannot be repurposed into a functional weapon without significant redesign.

8.3 Legal Compliance

The publication of this research—including its conceptual model, diagrams, and benign PoC—is compliant with current legal frameworks governing security research. Specifically:

- The PoC does not meet the definition of malicious software under applicable computer misuse statutes, as it lacks capability to cause damage, steal data, or gain unauthorised access.
- All code samples are inert and cannot be executed outside of a sandbox without modification. They are unsuitable for operational misuse.
- The work does not provide actionable instructions for developing harmful malware, nor does it include exploit chains, payloads, backdoors, or operational command-and-control infrastructure.
- No proprietary or sensitive information is used. All referenced materials are publicly accessible threat intelligence or academic analyses.

The publication of conceptual threat models and benign simulations is consistent with common academic practice in security research, including work disseminated at conferences such as IEEE S&P, USENIX Security, NDSS, ACSAC, and Black Hat.

8.4 Responsible Disclosure and Research Transparency

Although this research does not disclose vulnerabilities or weaknesses in specific software systems, we adhere to the principles of responsible communication:

- All descriptions of potential attacker behaviour are abstract and avoid operational detail.
- The remote multi-agent architecture is described conceptually without providing dangerous implementation specifics.
- All code fragments in the paper are explicitly safe and do not enable malicious extension.
- Reproducibility is supported through transparent documentation and strong sandbox enforcement.

8.5 Broader Ethical Context

The LRMAM paradigm highlights the potential for misuse of advanced AI systems, but its purpose is to catalyse discussion on:

- AI safety mechanisms,
- detection of AI-driven threat activity,
- mitigation of LLM abuse risks,
- governance structures for model and API access,
- collaborative defence across public and private sectors.

By articulating this threat model early, the research community can prepare adequate responses before such techniques mature in adversarial contexts.

8.6 Summary

This research complies fully with ethical norms and legal requirements for security research. The contribution is conceptual and defensive, and the provided proof-of-concept is strictly benign, non-operational, and constructed for academic evaluation alone. The insights gained from this work are intended to enable more robust defences against future AI-enabled threats.

9 Conclusion

This paper introduced the Local–Remote Multi-Agent Malware (LRMAM) paradigm, a novel framework that redefines the structure and operation of AI-enabled malicious activity. Unlike conventional malware, where behaviour is statically embedded within the binary, LRMAM distributes intelligence across a minimal local micro-agent and a remotely hosted AI-driven multi-agent system. This decoupling enables a new class of dynamic, adaptive, and context-aware threats, with potential to bypass traditional detection mechanisms and challenge longstanding assumptions in cyber defence.

We formalised the architectural foundations of LRMAM, detailing its two-layer structure and the roles of the local micro-agent, LLM-based synthesis agent, planning agent, mutation agent, and optional adaptive components. We further articulated the operational lifecycle that emerges from this architecture, highlighting how dynamically generated behaviour differs fundamentally from that of prior malware families.

To support the scientific validity and reproducibility of the paradigm, we presented a fully benign and sandboxed proof-of-concept implementation. The PoC simulates the prompt-processing pipeline, context collection, remote orchestration, and execution flow of LRMAM using harmless code produced by a simulated LLM endpoint. Strict safety constraints ensure compliance with ethical and legal standards while enabling practical demonstration of the architectural model.

Our analysis of the broader threat landscape reveals that LRMAM introduces significant implications for endpoint detection, behaviour analysis, and AI governance. Dynamic code generation,

externalised reasoning, and centralised orchestration present challenges that cannot be adequately addressed by existing signature-based or heuristic-based defences. Addressing these challenges will require new security mechanisms capable of detecting intent, modelling AI-driven activity, and recognising misuse patterns in LLM API traffic.

The LRMAM paradigm represents a forward-looking conceptual contribution to the field of cybersecurity. By articulating the structure and risks of multi-agent AI-driven threats at an early stage, this work aims to support researchers, practitioners, and policymakers in developing proactive defences, informed governance frameworks, and improved monitoring systems. Future research should explore detection strategies for AI-orchestrated activity, develop robust LLM-abuse prevention systems, and extend multi-agent modelling to defensive architectures.

References

[1] P. Szor, *The Art of Computer Virus Research and Defense*. Addison-Wesley, 2005.

[2] Google Threat Intelligence Group, "GTIG AI Threat Tracker: Advances in Threat Actor Usage of AI Tools," https://cloud.google.com/blog/topics/threat-intelligence/threat-actor-usage-of-ai-tools, 2025, accessed: 2025-11-22.

[3] A. Delamotte, V. Kamluk, and G. Bernadett-Shapiro, "Prompts as Code & Embedded Keys: The Hunt for LLM-Enabled Malware," https://www.sentinelone.com/labs/prompts-as-code-embedded-keys-the-hunt-for-llm-enabled-malware/, 2025, accessed: 2025-11-22.

[4] L. Constantin, "Novel malware from Russia's APT28 prompts LLMs to create malicious Windows commands," https://www.csoonline.com/article/4025139/novel-malware-from-russias-apt28-prompts-llms-to-create-malicious-windows-commands.html, 2025, accessed: 2025-11-22.

A Appendix A – Benign PoC Code (Google Colab / Python)

Catatan penting (disclaimer): kode di bawah ini hanya demonstrasi akademik. Ia sengaja dibatasi agar *tidak* dapat melakukan aksi berbahaya. Sebelum menjalankan di lingkungan lain, baca dan pahami batasan keselamatannya.

```
% ---------------------------
% Put this in a code block in your Colab notebook.
% ---------------------------
# Benign PoC: Local Micro-Agent (Colab safe demo)
# - No filesystem writes
# - No subprocess/os.system
# - No network calls (fake_llm_server is local)
# - Simple policy check before exec

import base64
```

```
import json
import platform
from textwrap import dedent

############################
# 1) Encrypted Prompt (Base64 - simulation of obfuscation)
############################
plain_prompt_template = dedent("""
You are a safe Python assistant.

You receive this JSON describing the environment:
{env_json}

Task:
Write VALID Python code that defines a function called `generated_main()`.
When called, this function must:
 - Print the string "Hello from generated code".
 - Print the Python version using the platform module.

Important:
 - Only output pure Python code.
 - Do NOT include imports other than `platform`.
 - Do NOT use os, sys, subprocess, socket, open, eval, exec, importlib, ctyp
""").strip()

# simulate obfuscation/encryption
encrypted_prompt = base64.b64encode(plain_prompt_template.encode('utf-8')).d

############################
# 2) Local Micro-Agent primitives
############################
def decrypt_prompt(enc_str: str) -> str:
    """Decode Base64 'encrypted' prompt back to plain text."""
    return base64.b64decode(enc_str.encode('utf-8')).decode('utf-8')

def collect_environment_info() -> dict:
    """Collect only non-sensitive environment info."""
    return {
        "python_version": platform.python_version(),
        "system": platform.system(),
        "release": platform.release(),
        "machine": platform.machine()
    }

def fill_prompt_template(template: str, env: dict) -> str:
```

```
    env_json = json.dumps(env, indent=2)
    return template.format(env_json=env_json)

############################
# 3) Fake LLM Server (deterministic, safe)
############################
def fake_llm_server(prompt: str) -> str:
    """
    Simulated LLM: returns a harmless Python code string.
    In real experiments, researchers may replace this with an API call
    to a controlled LLM endpoint that enforces safety policies.
    """
    # A deliberately simple and explicit safe code block
    code = dedent("""
    import platform

    def generated_main():
        print("Hello from generated code")
        print("Python version:", platform.python_version())
    """).strip()
    return code

############################
# 4) Safety Policy Check before execution
############################
FORBIDDEN_TOKENS = [
    "os.system", "subprocess", "socket", "open(", "eval(", "exec(",
    "import os", "import sys", "ctypes", "importlib", "from os", "from sys",
    "__import__", "shutil", "fork", "popen"
]

def safe_policy_check(code_str: str) -> None:
    """Raise RuntimeError if code contains forbidden tokens."""
    lower = code_str.lower()
    for token in FORBIDDEN_TOKENS:
        if token in lower:
            raise RuntimeError(f"Unsafe token detected in generated code: {t

############################
# 5) Execute in restricted namespace
############################
def execute_generated_code(code_str: str):
    """
    Execute code in a restricted namespace.
    We pass an empty globals dict to reduce possible access to outer scope.
```

```
    The local namespace receives only what exec defines.
    """
    # Policy check
    safe_policy_check(code_str)

    # Prepare very restricted globals (only builtins allowed)
    restricted_globals = {
        "__builtins__": {
            # allow only a minimal safe subset of builtins
            "True": True, "False": False, "None": None, "print": print,
            "len": len, "range": range
        }
    }

    local_ns = {}
    exec(code_str, restricted_globals, local_ns)
    if "generated_main" not in local_ns:
        raise RuntimeError("generated_main not found in generated code")
    # Call the generated function
    local_ns["generated_main"]()

############################
# 6) Full agent run (simulate)
############################
def run_local_agent_demo(enc_prompt: str):
    # 1. decrypt
    template = decrypt_prompt(enc_prompt)
    # 2. collect environment
    env = collect_environment_info()
    # 3. fill template
    filled_prompt = fill_prompt_template(template, env)
    # 4. send to fake LLM server
    generated_code = fake_llm_server(filled_prompt)
    # 5. execute (with safe policies)
    execute_generated_code(generated_code)

# Run demo
if __name__ == "__main__":
    run_local_agent_demo(encrypted_prompt)
```

B Appendix B – Pseudocode: Full LRMAM Lifecycle (Abstract, Safe)

```
# Pseudocode (abstract) for Local-Remote Multi-Agent Malware lifecycle
# NOTE: This is conceptual pseudocode for academic modelling only.

function LRMAM_Lifecycle():
    # 1. Deployment (benign research: simulated)
    deploy_local_micro_agent()

    # 2. Context Collection
    context = collect_non_sensitive_environment_metadata()

    # 3. Prompt Assembly
    template = retrieve_obfuscated_prompt()
    prompt = decrypt_and_fill_template(template, context)

    # 4. Remote Orchestration (simulated / controlled)
    # In research, remote_ai returns benign code or analysis results
    response = remote_ai_orchestration(prompt)

    # 5. Safety Filter (must be enforced in research PoC)
    if not safety_policy_check(response.code):
        abort("Unsafe code detected")

    # 6. Execution (sandboxed)
    result = execute_in_sandbox(response.code)

    # 7. Feedback & Adaptation
    telemetry = collect_execution_telemetry()
    send_safe_telemetry_to_remote_ai(telemetry)

    # 8. Retasking (optional)
    if should_request_new_task(telemetry):
        goto 3
    else:
        terminate()

# End pseudocode
```

C Appendix C – Sequence Diagram (Textual / ASCII)

```
Sequence: Local Agent <-> Remote AI (safe demo)
```

```
LocalAgent -> LocalAgent: decrypt_prompt()
LocalAgent -> LocalAgent: collect_environment_info()
LocalAgent -> LocalAgent: fill_prompt_template()
LocalAgent -> RemoteAI: POST /generate  (simulated local function)
RemoteAI -> RemoteAI: planning_agent() -> llm_agent() -> mutation_agent()
RemoteAI -> LocalAgent: 200 OK (generated code string)
LocalAgent -> LocalAgent: safety_policy_check()
LocalAgent -> Sandbox: execute_generated_code()
Sandbox -> LocalAgent: stdout (print outputs)
LocalAgent -> RemoteAI: (optional) telemetry / feedback
```

D Appendix D – How to Run the PoC (Instructions for Researchers)

1. Create a new Google Colab notebook (or local safe Python environment).
2. Copy the code from Appendix A into a single code cell.
3. Review the FORBIDDEN_TOKENS list and the policy check to ensure it fits your institution's rules.
4. Execute the cell. Expected output:

   ```
   Hello from generated code
   Python version: 3.x.x
   ```

5. If you wish to replace the fake LLM server with a real LLM API, ensure:
   - The API is under your control and has abuse prevention.
   - You perform strict server-side policy checks to forbid harmful responses.
   - You do not transmit any sensitive host data.
   - You have permission from your institution / IRB (if applicable).

E Appendix E – License and Responsible-Use Statement

```
This PoC code and accompanying materials are released for academic and
defensive-research purposes only. By using this code you agree to:
 - Not deploy it in any environment that is not explicitly controlled for re
 - Not modify the code to perform malicious actions.
 - Obtain all necessary approvals from your institution or legal body if you
   plan to extend or publish derivatives that involve live tests.
 - Attribute the original authors when reusing materials.
```

Suggested license: CC BY-NC-SA (for paper + documentation) and MIT with a explicit "defensive-research only" note for code, or your institution’s preferred research-use license.