HiveMind: Enabling Emergent Workflows for Deep Firmware Security Analysis

Xiangrui Zhang*
Beijing Jiaotong University
China

Zeyu Chen*
Trinity University
USA

Haining Wang
Virginia Tech
USA

Qiang Li†
Beijing Jiaotong University
China

ABSTRACT

Security auditing of large-scale firmware is a critical, yet notoriously difficult, task for safeguarding the Internet of Things (IoT) ecosystem. While Large Language Models (LLMs) show promise in code analysis, existing agent-based approaches fundamentally fail when applied to firmware. This failure stems from a core paradigm mismatch: the rigid, predefined workflows of current agents cannot handle the dynamic, exploratory nature of firmware analysis, leading to context overload, fragmented reasoning, and missed vulnerabilities. To resolve this mismatch, we propose **HiveMind**, a novel agent system that enables **structured emergence** in the analysis process. HiveMind introduces two key innovations: a **Recursive Delegation Engine (RDE)** that dynamically generates a Tree of Agents (ToA) to adaptively decompose and explore the firmware, and a **Proactive Knowledge Hub (PKH)** that acts as a collective intelligence to correlate findings across the emergent agent hierarchy. This synergy allows LLMs to perform deep, large-scale reasoning without succumbing to context limitations. Our experiments on 49 real-world firmware images show that HiveMind autonomously discovers and verifies 1,612 high-confidence alerts with 80.5% precision, significantly outperforming both existing LLM-based agents and state-of-the-art (SOTA) security tools. Our work demonstrates a new paradigm for building autonomous agents capable of tackling complex, open-ended exploratory tasks.

1 INTRODUCTION

Large language models (LLMs) have shown advanced abilities in domains such as knowledge acquisition, software development, game simulation, multi-robot systems, and flexible adaptation to new scenarios [7]. In cybersecurity applications, LLMs have been increasingly used to automate tasks such as vulnerability scanning, malware detection, and the enforcement of security protocols [14, 23, 24, 28, 30, 33]. More recently, this has evolved towards employing LLMs as the "brain" for autonomous agents [37, 39, 41, 42], which can interact directly with security tasks without human intervention, holding significant promise for automating complex cyber defense operations.

This paper tackles one such complex operation: firmware security analysis. Firmware, the foundational software embedded in billions of Internet of Things (IoT) devices, is a critical yet notoriously difficult target for security auditing. Its analysis typically demands substantial manual effort and expert knowledge, posing a major barrier to scalable automation. While LLM-based agents offer a path to automation, they face fundamental limitations when confronted with the unique structure of firmware. Firmware images often contain vast, decentralized filesystems where binaries, scripts, and configuration files are intricately interwoven.

Consequently, directly applying LLM-based agents to this domain fails. We identify three core technical challenges that lead to shallow analysis and missed vulnerabilities: (1) Long-term reasoning, where tracing a single vulnerability requires following a deep, multi-step logic chain across numerous files. (2) Large-scope reasoning, where agents must intelligently navigate thousands of files and directories under strict token and attention constraints, a problem we term *context overload*. (3) Cross-component dependency understanding, where critical vulnerabilities emerge from the subtle interactions of disparate components, requiring a holistic view that is easily fragmented. Existing agent architectures, which often rely on static workflows or a single monolithic agent, are ill-equipped to handle this combination of depth, breadth, and complexity.

To address these challenges, we argue for a fundamental paradigm shift: moving away from static workflows and predefined tools towards a dynamic, autonomous intelligent system that mimics the exploratory and adaptive reasoning of a human security expert. This paper introduces HiveMind, the first autonomous agent system engineered specifically for the open-ended nature of firmware security analysis.

Specifically, HiveMind consists of two core technical components: the Recursive Delegation Engine (RDE) and the Proactive Knowledge Hub (PKH). RDE addresses the limitations of LLMs in long-term and large-scale reasoning by dynamically generating a structured and predictable Tree of Agents (ToA). This is not an arbitrary process; agent creation is guided by the inherent hierarchical structure of the firmware itself, allowing the system to adaptively explore newly discovered components and validate emerging hypotheses, just as a human expert would. PKH complements this process by serving as a centralized knowledge base. Crucially, PKH is not a passive database; it proactively correlates new findings with existing knowledge and leverages a hierarchical aggregation strategy to ensure information consistency, preventing the context loss and overwriting issues common in multi-agent communication. Together, RDE

*Both authors contributed equally to this research.
†Corresponding author.

and PKH form a cohesive framework that allows HiveMind to analyze firmware autonomously, uncover vulnerabilities, and generate high-confidence alerts without human intervention.

To validate HiveMind's effectiveness, we have designed five challenging security analysis tasks: (T1) locating hard-coded credentials, (T2) identifying third-party components and their known CVEs, (T3) tracing interactions with NVRAM and environment variables, (T4) performing web attack chain analysis, and (T5) vulnerabilities detection. We conduct a comprehensive evaluation using a public dataset comprising 49 real firmware images, comparing HiveMind against existing LLM-agent frameworks and state-of-the-art (SOTA) security tools, including Mango [11] and SaTC [3]. Our results show that HiveMind outperforms prior LLM-agent systems and SOTA tools, particularly in handling complex logic chains and reasoning across files. Notably, HiveMind demonstrates the ability of autonomous exploration and verification: it identifies 4,571 potential security alerts, verifies them individually, and finally generates 1,612 high-confidence alerts with a precision of 80.5

The main contributions of this work are summarized as follows:

- We propose two novel techniques, the Recursive Delegation Engine (RDE) and the Proactive Knowledge Hub (PKH), to mitigate the inherent limitations of LLMs in long-term and large-scope reasoning, and we demonstrate their application in the challenging domain of firmware security analysis.
- We design and implement HiveMind, a complete, end-to-end autonomous multi-agent system that leverages RDE and PKH to automate complex firmware security tasks using only high-level natural language instructions. We have released its source code to facilitate further research[1].
- Through extensive experiments on real-world firmware, we demonstrate that HiveMind outperforms existing LLM-based agents and traditional SOTA security tools in overall performance, discovering 1,612 high-confidence vulnerabilities with 80.5% precision.

The remainder of this paper is organized as follows. Section 2 presents the technical challenges of applying LLM-based agents to firmware security analysis. Section 3 describes the system overview of HiveMind. Section 4 illustrates the design details in HiveMind. Section 5 presents the experimental evaluation of HiveMind on real-world firmware images. Section 6 discusses the limitations of our approach. Section 7 provides the related work, and finally, Section 8 concludes.

2 THE CORE CHALLENGE: A PARADIGM MISMATCH

Firmware security analysis is not a linear, predictable process but a dynamic, open-ended exploration. Understanding this is key to seeing why existing agent-based approaches fail. The core challenge stems from a fundamental mismatch:

[1]https://github.com/z-zsstar/firmhive

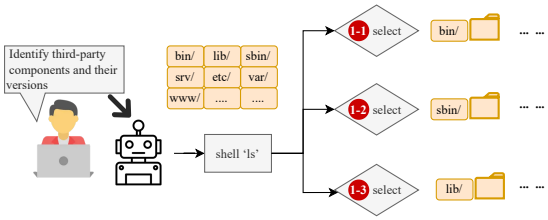

Figure 1: The need for large-scale reasoning: The analysis must cover a vast filesystem to explore a wide range of potential attack surfaces.

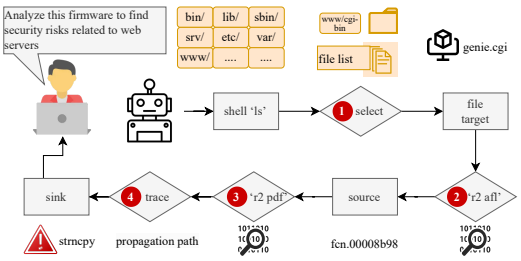

Figure 2: The need for long-term reasoning: Firmware analysis requires tracing complex logic chains across multiple files and layers of abstraction.

the task demands a fluid, adaptive strategy, while current agent paradigms offer only rigidity.

The workflow of a human expert illustrates the task's dynamic nature. It is an unscripted investigation that pivots between high-level reconnaissance and deep forensic analysis. An expert must first navigate a vast and opaque filesystem to build a mental map of the system, identifying potentially interesting components like web assets or network services, as suggested in Figure 1. Upon finding a clue, the focus shifts instantly to tracing a single, deep logical chain that might span multiple files and layers of abstraction, from a high-level script down to a specific binary function, as shown in Figure 2. Success hinges on uncovering the intricate and implicit dependencies between these components, as this is where critical vulnerabilities often hide.

While Large Language Models (LLMs) possess the reasoning power to serve as the "brain" for such a task, they are hobbled by a cripplingly narrow context window. An entire firmware image cannot be processed at once, forcing the LLM to operate in a piecemeal fashion that directly contradicts the need for a holistic view. This *context overload* problem means that an agent can easily lose track of crucial information across files, rendering it blind to the very cross-component vulnerabilities it is supposed to find.

Existing agent frameworks, designed to overcome these context limits, are themselves too rigid to handle this exploratory task. A single, monolithic agent, for instance, is easily overwhelmed [44]. Its long reasoning chains are brittle, prone to compounding errors [29, 35], and a single hallucination can derail the entire analysis with no mechanism

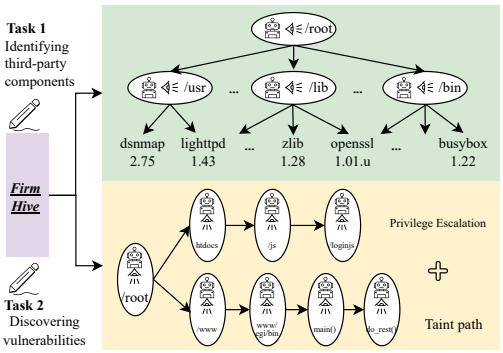

Figure 3: Given a user prompt, HiveMind dynamically generates a Tree of Agents to solve the task. The green box shows the tree for identifying third-party components, while the yellow box depicts the tree for discovering a vulnerability.

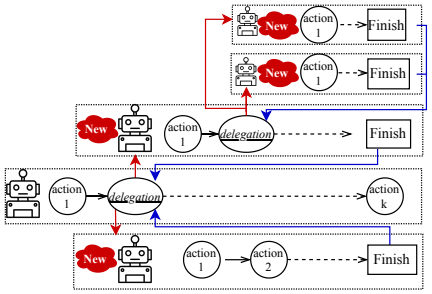

Figure 4: Delegation is the basic operation in HiveMind. The red arrow shows a parent delegating to a child agent, while the blue arrow indicates the child agent returning its result.

for recovery. Static multi-agent systems like AutoGen [36], on the other hand, operate like a fixed assembly line. They excel when the workflow is predictable, but the path of a firmware investigation is precisely what needs to be discovered. Relying on a predefined workflow to solve a problem of exploration is a losing strategy from the outset.

In essence, the central challenge is that we are applying tools built for structured, deterministic problems to a task that is fundamentally unstructured and emergent. What is needed is a system that can dynamically adapt its structure and focus at runtime, a capability that existing agent paradigms lack.

3 OVERVIEW

To address the challenges of firmware analysis, we propose HiveMind, a system designed around a core philosophy: **structured emergence**. This philosophy acknowledges that while the analysis workflow must be dynamic and *emergent* to handle the unpredictable nature of the task, the underlying agent collaboration must be highly *structured* to prevent chaos and ensure reliable, deep analysis. HiveMind implements this philosophy through two synergistic core components: the Recursive Delegation Engine (RDE), which orchestrates the emergent workflow, and the Proactive Knowledge Hub (PKH), which serves as the system's collective intelligence.

3.1 Recursive Delegation Engine (RDE): Orchestrating the Emergent Workflow

The RDE is the engine that drives HiveMind's dynamic capabilities. Its primary function is to take a high-level goal (e.g., "find vulnerabilities in this firmware") and recursively decompose it into a hierarchy of concrete, manageable subtasks. This process dynamically generates a **Tree of Agents (ToA)**, a hierarchical structure of agents tailored specifically to the task at hand. As shown in Figure 3, different analysis goals naturally lead to the emergence of distinct ToA structures, mirroring how a human expert might structure their investigation.

The fundamental operation enabling this process is **delegation**. We define it as the process where an agent formulates a focused sub-task, dynamically generates a new sub-agent, and assigns the sub-task to it. As illustrated in Figure 4, this creates a parent-child relationship where the parent agent reasons over the results returned by its children to determine the next step. Every agent in HiveMind possesses this delegation capability, allowing the ToA to grow organically at runtime.

This delegation is not arbitrary; it is guided by the inherent structure of the firmware itself. For instance, when analyzing a directory, an agent might delegate the analysis of individual files to multiple parallel sub-agents (*Parallel Delegation for Breadth*), enabling scalable exploration. When tracing a vulnerability, an agent might delegate the tracking through a single function call to a sequential sub-agent (*Chained Delegation for Depth*), extending the reasoning chain vertically. This structured approach ensures that the emergent workflow remains coherent and efficient, preventing the combinatorial explosion that cripples monolithic agents.

3.2 Proactive Knowledge Hub (PKH): The Proactive Collective Intelligence

If RDE provides the structure for collaboration, the PKH provides the shared consciousness. In any complex multi-agent system, a critical challenge is preventing information loss. As findings are passed up the hierarchy, crucial details and intermediate discoveries from lower-level agents are often compressed or lost. The top-level agents therefore miss a large number of intermediate key discoveries.

The PKH solves this by acting as a centralized collective memory. More than just a passive database, the PKH functions as the system's **proactive collective intelligence**. Its role is to "connect the dots" across distributed agents, uncovering insights that no single agent could infer in isolation. For instance, if one agent finds a buffer overflow in a library and another discovers that the same library processes high-privilege input, the PKH can link these two seemingly unrelated findings to identify a critical vulnerability.

This proactive nature is what fundamentally distinguishes the PKH from standard knowledge stores used in Retrieval-Augmented Generation (RAG) systems. While RAG systems *passively retrieve* information upon request, the PKH *actively reasons* over new findings as they are submitted. It immediately analyzes new information in the context of its existing knowledge base, enabling not only active knowledge correlation but also facilitating high-level synthesis and risk rollup, where scattered, low-confidence findings can be aggregated into a single, high-impact insight.

To maintain the integrity of this collective intelligence, the PKH employs a **hierarchical aggregation and access control strategy**. Write access is exclusively granted to agents at the Specific Medium Layer. Before submitting, a file-level agent must first consolidate, deduplicate, and summarize all findings related to its specific file. This file-centric approach ensures that the global knowledge base remains coherent and consistent, effectively preventing the context loss and overwriting issues common in large-scale multi-agent communication.

4 DESIGN

This section details the system design of HiveMind, focusing on how it translates its core philosophy into a working architecture. We first explain our paradigm of hypothesis-driven task formulation, then elaborate on the mechanisms of the RDE, and finally, discuss the design of the PKH.

4.1 Hypothesis-Driven Task Formulation

We formulate the analysis process not as a static task decomposition, but as a dynamic, **hypothesis-driven goal exploration**. This paradigm mirrors the workflow of a human security expert, who iteratively forms hypotheses about potential vulnerabilities (high-level goals) and then investigates specific artifacts (low-level objects) to gather evidence.

At its core, an agent's operation revolves around advancing a high-level goal. Guided by its current context C and a working hypothesis, the agent reasons about and generates a set of subtasks $\mathcal{T}$. Each subtask is a pair of a target object and a corresponding analysis goal. This process is formalized as:

$$C \xrightarrow{\text{LLM}} \mathcal{T} = \{(o_i, g_i)\}_{i=1}^{n} \tag{1}$$

where $o_i \in O$ is an *object* selected from the observable environment (e.g., a file or function). The analysis goal g_i is derived by the LLM based on o_i and the comprehensive context C. This context C is crucial as it encapsulates all information available to the agent at that moment, including: **(1) the user's overall objective**, which sets the global direction for the analysis; **(2) the current analysis focus**, defining the specific object or scope the agent is currently examining (e.g., the content of a file or the functions within a binary); **(3) the agent's conversation history**, contains the outputs from previous tool executions, which provide concrete, factual information gathered from exploring the current environment (e.g., the file listing of a directory). This rich, hypothesis-driven formulation is the primary driver for all delegation decisions within HiveMind.

4.2 Structured Delegation via Hierarchical Blueprint

Delegation in HiveMind is not an unconstrained process but is strictly governed by a **Recursive Hierarchical Blueprint**. This blueprint determines the *type* of agent to delegate at runtime, ensuring that the dynamic ToA remains structured and aligned with the analysis task.

4.2.1 The Three-Layer Blueprint. The blueprint mirrors the inherent structure of firmware, organizing the analysis into three distinct layers as illustrated in Figure 5:

1. Directory Analysis Layer: As the entry point for exploration, agents at this level navigate the filesystem and delegate further analysis of interesting files or sub-directories. They can also query the PKH to prioritize targets.

2. File Analysis Layer: This layer is activated to analyze a specific file. A *File Planner Agent* first generates a detailed plan, then instantiates multiple parallel *File Executor Agents* for specific subtasks. For complex binary files, the planner delegates the task to the next layer.

3. Call-Chain Analysis (CCA) Layer: This layer performs deep, code-level analysis of binary functions. A top-level *Function Agent* identifies source and sink functions and then spawns tracing agents to follow the call paths between them, as shown in Figure 6.

To safeguard against the uncontrolled growth and combinatorial explosion of agents, the blueprint incorporates pragmatic depth limits for delegation. These constraints are essential for ensuring that the analysis remains computationally tractable while still allowing for the necessary analytical depth.The Directory Analysis Layer and Call-Chain Analysis Layer, which involve true recursive exploration, are configured to recursively invoke themselves to analyze subsets (sub-directories or called functions). This allows the system to traverse deep directory structures and trace complex function call graphs without risking infinite loops or excessive resource consumption. The File Analysis Layer, however, employs a different form of delegation focused on parallel decomposition. At this layer, a File Planner Agent breaks down the analysis of a single, complex file into multiple concurrent sub-tasks handled by File Executor Agents—for instance, simultaneously checking multiple function entry points or scanning different sections of the file for vulnerabilities. The File Executor Agents are configured not to recursively invoke themselves, a design choice to prevent excessive decomposition and calls.

4.2.2 A Walkthrough Example: From Web Interface to Command Injection. To make these concepts concrete, let's trace a simplified, yet representative, analysis scenario. Imagine a user gives HiveMind the high-level goal: "Analyze this firmware for web-related vulnerabilities."

- **Layer 1: Directory Analysis.** (High-Level Reconnaissance) The initial agent starts at the root of the firmware's filesystem. It identifies a promising directory, /www, and queries the PKH for any known vulnerabilities associated with common web servers found within. Based on this reconnaissance, it delegates a new, specialized task: "Analyze the contents of the /www directory."

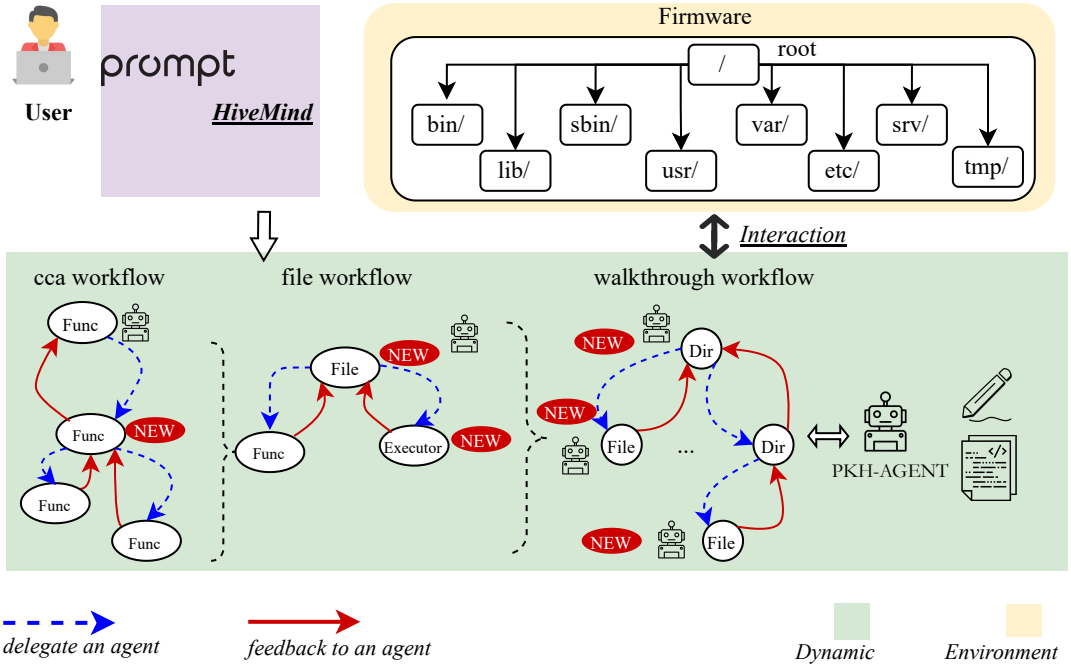

Figure 5: The walkthrough workflow for firmware analysis, illustrating the interaction between the user, HiveMind, and the dynamic agent environment. The diagram shows three key workflows: directory walkthrough (right), file analysis (middle), and call-chain analysis (left), all driven by the core operations of delegation and feedback.

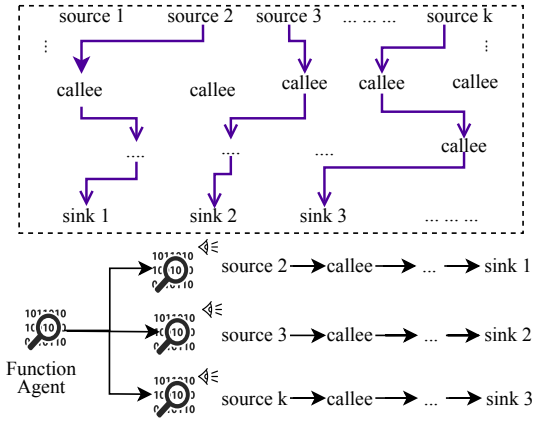

Figure 6: The Call-Chain Analysis (CCA) layer in HiveMind, where a Function Agent delegates tracing tasks to sub-agents to analyze paths from sources to sinks.

- **Layer 2: File Analysis.** (Planning the Attack Trace) The agent responsible for /www discovers a CGI script, apply.cgi, which seems to handle user requests. A *File Planner Agent* is activated. It determines that apply.cgi is a binary and formulates a plan: (1) check for hardcoded secrets, and (2) trace how it handles external data. It delegates the second, more complex task to the CCA layer with a clear starting point: "Identify all web-data input sources in apply.cgi and begin tracing their data flow."
- **Layer 3: Call-Chain Analysis (The Taint Relay Race).** This layer demonstrates how a complex vulnerability emerges from a series of local, recursive delegations.
  (1) **The Entry-Point Agent:** A *Function Agent* is instantiated for the main function of apply.cgi. It identifies a *source* function that reads HTTP POST data into a buffer, post_data, which is now considered **tainted**. The agent observes that post_data is passed as an argument to another function, process_input(). With its visibility ending here, it delegates a new, focused task to a sub-agent: "Analyze the function process_input(). Your goal is to track the data flow of the tainted variable post_data."
  (2) **The Intermediate Agent:** This new sub-agent focuses solely on the process_input() function. It discovers that the tainted post_data is used to construct a command string in a new variable, cmd_string. The taint has now propagated. This agent then sees that cmd_string is passed to yet another function, execute_command(). It delegates again: "Analyze the function execute_command() and track the tainted variable cmd_string."

(3) **The Sink-Discovery Agent:** The third agent examines only the execute_command() function. It finds that the tainted variable cmd_string is passed directly to a sensitive *sink* function, system(), without any sanitization. This is the moment of discovery.
(4) **Reporting Back:** The final agent reports its finding up the hierarchy: "Success. Tainted data reached the system() sink." This result, now complete with the full call stack (main->process_input->execute_command) and variable trace, is passed back to the top-level *File Planner Agent*. It consolidates this emergent attack chain and submits it to the PKH as a high-confidence command injection vulnerability.

This example illustrates how HiveMind's structured delegation process transforms a vague initial prompt into a precise, verifiable security finding. The emergent workflow is not random; it is a focused descent through the layers of the blueprint, guided at each step by the system's evolving hypotheses.

4.2.3 Delegation as a Tool and Agent Configuration. To implement this structured process, we encapsulate delegation into a reusable **tool** available to all agents. When an agent calls this tool with a subtask pair (o, g), the tool's core responsibility is to instantiate a new sub-agent within a strictly controlled operational context. Specifically, for the target object o in the task, the tool constructs a new, independent context containing only the necessary subset of information for analysis. This process effectively creates a **sub-sandbox environment** for the newly generated sub-agent. The configuration for this agent is defined by the following tuple:

$$\textit{AgentConfig} = (\textit{llm}, \textit{schema}, \textit{memory}, \textit{task}, \textit{type}, \textit{tool}) \quad (2)$$

This tuple specifies the agent's LLM, its output schema, an isolated memory for its conversation history, its assigned *task*, and its *type* and available *tools* as prescribed by the blueprint. Crucially, by creating this new, **isolated sub-environment**, the sub-agent is confined to its sandbox, restricted to analyzing only downward from its designated starting point. It cannot directly access its parent's broader context. This architectural choice is fundamental to preventing context overload and enhancing error containment. Each agent is implemented using a ReAct pattern, allowing it to reason and act flexibly within its designated scope.

4.3 PKH Design: Ensuring Proactive and Consistent Collective Intelligence

The Proactive Knowledge Hub (PKH) is architected to serve as HiveMind's collective intelligence. Its design fulfills two critical, intertwined functions: first, it ensures the integrity and consistency of the shared knowledge base through a strict, hierarchical protocol; second, it leverages this protocol to foster proactive knowledge discovery, driven not by the hub itself, but by the agents interacting with it.

4.4 PKH Design: Ensuring Proactive and Consistent Collective Intelligence

The Proactive Knowledge Hub (PKH) is architected to serve as HiveMind's collective intelligence. Its design fulfills two critical, intertwined functions: first, it ensures the integrity and consistency of the shared knowledge base through a strict, hierarchical protocol; second, it leverages this protocol to foster proactive knowledge discovery, driven not by the hub itself, but by the agents interacting with it.

4.4.1 Gatekeeping via Hierarchical Aggregation and Access Control. A key challenge for any shared memory in a multi-agent system is preventing information conflicts and overwriting. The PKH addresses this through a robust, two-pronged strategy inspired by the firmware's own structure: context isolation and hierarchical aggregation.

1. Context Isolation: The delegation process, guided by the hierarchical blueprint, naturally creates orthogonal scopes of operation. For instance, two *File Agents* analyzing two different files operate in almost complete isolation. This inherent separation of concerns, rooted in the firmware's modular structure, fundamentally minimizes the chance of direct conflicts at the data entry level.

2. Hierarchical Aggregation and Access Control: To enforce consistency, write access to the PKH is strictly controlled. **Only agents at the *File Analysis Layer* are granted write permissions.** A file-level agent acts as a gatekeeper for all information generated beneath it (e.g., by multiple *Function Agents*). Before any interaction with the PKH, the File Agent is responsible for performing a local aggregation: it collects all findings from its sub-agents, deduplicates redundant information, resolves minor discrepancies, and summarizes the results into a single, consolidated report for the file it analyzed. This gatekeeping ensures that only coherent and vetted information is ever considered for submission to the global knowledge base.

4.4.2 Agent-Driven Proactive Correlation. The "proactivity" of the PKH is not an intrinsic property of the database, but rather an emergent outcome of a structured interaction protocol. After a *File Agent* has prepared its consolidated local report, but *before* committing it, the agent initiates a mandatory ReAct loop designed to contextualize its findings against the global knowledge.

This process is orchestrated through a correlation-centric schema and a set of specialized tools:

- **Correlation-Centric Schema:** Findings in the PKH are stored with a critical **linking** field. This field contains semantic identifiers (e.g., variable names like `wan_ifname` or shared configuration paths) that act as anchors for correlation.
- **Proactive Correlation Loop:** The File Agent is compelled by its prompt to follow an ***{explore → query → enrich → store}*** cycle.
  (1) It uses the **Explore** tool to scan the PKH for any existing 'linking' identifiers related to its current findings.
  (2) If relevant anchors are found, it uses the **Query** tool to retrieve details about these existing entries.

(3) The agent then reasons over this retrieved information to **enrich** its own report, potentially elevating a low-priority finding to a critical one by connecting it with existing knowledge. For example, a finding of a hardcoded password might be enriched to a critical "Remote Code Execution via Admin Login" if the PKH links that password to a known web interface binary.
(4) Finally, the agent uses the **Store** tool to submit its final, contextualized, and enriched report.

This agent-driven loop transforms agents from passive RAG users into **proactive knowledge builders**. The PKH remains a consistent, centralized repository, while the intelligence of correlation is dynamically generated by the agents at the precise moment it is needed. Together, the gatekeeping mechanism and this correlation loop guarantee the integrity and utility of HiveMind's collective intelligence.

5 EVALUATION

This section demonstrates the effectiveness of HiveMind via real-world experiments. Specifically, we attempt to answer the following research questions:
•*RQ*1 : How does HiveMind perform compared to different agent architectures? (Section 5.2)
•*RQ*2 : How does HiveMind's effectiveness compare to the SOTA firmware analysis tools? (Section 5.3)
•*RQ*3 : How do HiveMind's key deisgn components contribute to its overall performance? (Section 5.4)

5.1 Experimental Setup

Implementation. To demonstrate the HiveMind's feasibility, we use the Python language to implement its prototype system. We provide a set of interactive tools listed by Table 1, because LLMs cannot directly interact with the firmware filesystem.

System settings. We use the DeepSeekv3 as the LLM to reason about the firmware analysis task. The basic requirements for the LLM are: the reasoning capability to correctly follow the requirements in the prompt. The temperature is set to 0. In our experiment, we set the maximum number of reasoning steps as 30 for each agent.

Datasets. We used the dataset from Karonte [26], where the labels of the dataset are also used by Mango [11] and SaTC [3]. Table 2 shows our dataset of 49 firmware images, which are collected from four major IoT vendors (NETGEAR, D-Link, TP-Link, and Tenda). Firmware typically comprises a vast number of directories and files, reflecting the need for both large-scale exploration and long-term reasoning—challenges that static agent workflows struggle to address.

Baselines. To rigorously evaluate HiveMind's performance, we establish a comprehensive set of baselines. A core objective of our comparison is to highlight the fundamental difference between architectures based on **static, predefined workflows** and HiveMind's dynamic, **emergent workflow**. We argue that the key differentiator is not merely the number of agents, but the system's intrinsic **adaptability** to the exploratory nature of firmware analysis. To this end, all agent-based baselines utilize the same tools (Table 1) and are initiated with the same high-level prompt as HiveMind to ensure a fair comparison across our five evaluation tasks.

(1) *Single ReAct Agent (SRA):* This represents the most basic autonomous agent architecture. A single ReAct agent is tasked with completing the entire analysis from start to finish. This baseline is designed to test the limits of a monolithic agent when faced with the long-term, large-scale reasoning challenges inherent in firmware.

(2) *Single ReAct Agent with Knowledge Base (SRA+KB):* We augment the SRA with an external knowledge base, allowing it to save and retrieve alerts. This setup evaluates whether simply adding external memory can overcome the context limitations of a single agent, or if the core reasoning process itself remains a bottleneck.

(3) *Multi-Agent System with Static Workflow (MAS):* This baseline directly embodies the static workflow paradigm. Inspired by frameworks like AutoGen [36], we implement a two-agent system with a fixed division of labor, specifically designed to mimic a common human analysis pattern: autonomous navigation coupled with focused analysis. An **Explorer Agent** navigates the filesystem and identifies targets, which are then passed to a dedicated **Analyzer Agent**. The Analyzer performs deep analysis in isolation (with its memory reset) and reports back. Its performance serves as a direct counterpoint to HiveMind's emergent approach, allowing us to precisely measure the value of dynamic workflow generation.

(4) *Multi-Agent System with Knowledge Base (MAS+KB):* We augment the static MAS baseline with a shared knowledge base, which allows both the Explorer and Analyzer agents to persist and access alerts. Similar to the SRA+KB setup, this baseline aims to mitigate the loss of crucial intermediate discoveries during long-horizon tasks. It specifically

Table 1: Interactive Tools for Firmware File System Analysis.

Tool	Description
GetContextTool	Fetches context (e.g., from a specific root path) to build a "sub-sandbox" view.
ShellExecuteTool	Executes non-invasive shell commands (e.g., 'file', 'grep', 'cat') for general inspection.
Radare2Tool	Wraps core Radare2 commands for static binary analysis, including 'pdg' and all other commands.

Table 2: The evaluation dataset. Each row includes the vendor name, the number of firmware samples, and the number of binaries in the firmware.

Vendor	Product	# Firmware	# Dir	# Files
NETGEAR	R/XR/WNR	17	2,406	34,603
D-Link	DIR/DWR/DCS	9	1,175	16,383
TP-Link	TD/WA/WR/TX/KC	16	1,154	9,524
Tenda	AC/WH/FH	7	459	3,394

tests whether a persistent, shared memory can overcome the limitations of a fragmented workflow where context is not dynamically passed between agents, especially in tasks that require synthesizing information gathered across multiple, isolated analysis steps.

(5) *State-of-the-Art (SOTA) Tools:* To contextualize HiveMind's performance within the broader field of security analysis, we compare it against two recognized SOTA tools for automated firmware vulnerability discovery: SaTC [3] and Mango [11]. These tools employ sophisticated, specialized techniques like taint analysis and symbolic execution. They represent a different paradigm altogether—one based on highly optimized, hard-coded "expert knowledge" rather than the flexible, semantic reasoning of LLMs. This comparison highlights the trade-offs between deep, specialized analysis and broad, adaptive exploration.

We provide a set of tasks for the system to complete, designed to cover firmware security analysis. These tasks represent a diverse range of common, real-world security challenges, from simple data discovery to complex, multi-step vulnerability analysis.

T1: Find Hard-coded Credentials

Report all hard-coded credentials and other sensitive information.

T2: Identify Third-party Components

Identify third-party components and their versions, generating a Software Bill of Materials (SBOM).

T3: Tracing NVRAM/Environment Variable Interactions

Analyze how the firmware interacts with configuration systems like NVRAM or environment variables (e.g., via `getenv`) to identify unsanitized data flow paths from these variables to dangerous function calls.

T4: Web Attack Chain Analysis

Identify how untrusted input from external HTTP requests is processed and trace data flows to find paths where this input reaches dangerous functions (e.g., `system`, `strcpy`).

T5: Vulnerability Detection

Perform a comprehensive analysis to discover and report complete, end-to-end attack chains. The task requires tracing data flows from all potential untrusted input points(e.g., network requests, NVRAM variables) to sensitive sink functions (e.g., `system`, `strcpy`).

5.2 Comparing with Alternative Agent Architectures (RQ1)

To answer RQ1, we compare HiveMind's performance against four baseline agent architectures: SRA, SRA+KB, MAS, and MAS+KB. We evaluate their ability to complete five distinct security tasks (T1-T5), ranging from locating hard-coded secrets (T1) to tracing end-to-end exploit chains (T5), using the number of discovered alerts as the primary performance metric.

Table 3 summarizes the results across three metrics: alerts per firmware, time cost, and token consumption. The data reveals a consistent trend: **HiveMind substantially outperforms all baseline architectures in alert coverage across every task.** For instance, in T2 (third-party component detection), HiveMind detects 32.52 alerts per firmware, over 2.5 times more than the best-performing baseline, MAS+KB (12.80). This advantage is even more pronounced in complex reasoning tasks like T3 and T5.

This performance gap stems from a fundamental architectural difference. HiveMind's recursive and stateful coordination enables it to maintain context and reason deeply across numerous files and components. Although this deep analysis incurs higher token usage (e.g., 5.66M in T2 vs. 0.88M for MAS+KB), the significant gain in discovered vulnerabilities justifies the cost. Conversely, the short runtimes of SRA-based models (e.g., 0.1h) are not indicative of efficiency but rather a sign of premature termination due to severe context limitations.

Accuracy through Analytical Depth. The most critical advantage of HiveMind is not just the quantity of alerts, but their quality. Our manual verification of 250 alerts revealed that **HiveMind achieved the highest accuracy at 72%**, substantially outperforming MAS+KB (42%). This leap in precision is a direct consequence of the unparalleled analytical **depth** enabled by the Recursive Delegation Engine (RDE).

Conventional agent approaches, whether monolithic or static, are ill-suited for deep, multi-step reasoning. A single LLM attempting to trace a complex attack chain inevitably suffers from context overload. HiveMind fundamentally solves this by transforming a long, deep analysis task into a **structured recursive relay**. This is not a recovery mechanism, but a deliberate architectural choice. When an agent must delve into a complex subunit (e.g., a called function), it orchestrates a "baton pass": it packages the current context and a new, precise objective, then delegates the task to a fresh sub-agent. This new agent begins with a **clean, focused context**, allowing it to dedicate its full reasoning capacity to the sub-task.

This recursive process of "context reset and goal passing" creates a chain of expert agents, allowing the analysis to reach a depth that mirrors the code's actual complexity. This **deep penetration capability** is what enables HiveMind to fully trace vulnerabilities hidden deep within the code and minimize the false positives that arise from incomplete, assumption-based tracing in other methods.

The Inherent Flaws of Static Architectures. Our experiments highlight three cascading failures in conventional agent designs:

Table 3: Performance comparison between HiveMind and baselines across five tasks (T1-T5). The metrics include the average number of valid alerts per firmware, the average time cost per firmware, and the average token usage per firmware. Token refers to the average total tokens (input + output) per firmware run, in Millions (rounded to two decimal places).

	T1			T2			T3			T4			T5		
	Alerts	*Time*	*Token*	*Alerts*	*Time*	*Token*	*Alerts*	*Time*	*Token*	*Alerts*	*Time*	*Token*	*Alerts*	*Time*	*Token*
SRA	0.96	0.1 h	0.12M	1.86	0.1 h	0.30M	0.20	0.1 h	0.28M	0.12	0.1 h	0.20M	0.3	0.1 h	0.21M
SRA+KB	1.40	0.1 h	0.17M	1.20	0.1 h	0.30M	0.24	0.1 h	0.35M	0.10	0.1 h	0.21M	0.4	0.1 h	0.36M
MAS	3.96	0.5 h	0.30M	9.72	0.7 h	0.54M	0.87	0.4 h	0.30M	1.68	0.5 h	0.36M	2.1	0.9 h	1.06M
MAS+KB	5.84	0.9 h	0.65M	12.80	0.9 h	0.88M	1.94	0.6 h	0.58M	1.80	0.4 h	0.32M	3.3	1.2 h	2.06M
HiveMind	**8.24**	0.6 h	2.00M	**32.52**	1.4 h	5.66M	**9.06**	0.8 h	5.03M	**2.60**	0.9 h	1.55M	**24.5**	2.3 h	17.43M

(1) **Single-agent systems are ineffective.** SRA and SRA+KB halt prematurely due to context overload, finding only the most superficial issues.
(2) **Static multi-agent systems offer shallow parallelism.** While MAS and MAS+KB show improvement, their rigid workflows cannot adapt to the hierarchical nature of firmware analysis. When deep analysis is required, the task degrades back to a single, isolated agent, which then fails for the same reasons as a monolithic one.
(3) **Knowledge bases cannot fix a flawed foundation.** A KB is only as useful as the information fed into it. Since static agents are incapable of deep exploration, they can only contribute fragmented, surface-level snippets. This disjointed data fails to form meaningful correlations, rendering the KB almost useless in this paradigm.

These observations underscore that simply adding more agents or a memory store is insufficient. The core problem lies in the static workflow itself, which cannot handle the dynamic, exploratory nature of deep security analysis.

> *alert 1: HiveMind overcomes the fundamental limitations of existing agent architectures by introducing a dynamic, recursive delegation model. This enables LLMs to perform the deep, long-context reasoning required for complex, large-scale security analysis tasks.*

5.3 Comparison with SOTA Tools via T5 (RQ2)

To answer RQ2, we leverage Task T5 (vulnerability detection) to compare HiveMind with two state-of-the-art vulnerability analysis tools: Mango and SaTC. This task represents the most comprehensive form of firmware security analysis, requiring the detection and validation of full exploit chains from untrusted inputs to critical sinks.

Unlike traditional tools that follow a single exploration workflow, HiveMind employs a two-stage **Explore then Verify** process. The **exploration stage** first autonomously scans the firmware to identify a broad set of **initial alerts**, aiming for maximum **recall**. Subsequently, the **verification stage** treats each initial alert as a distinct, focused task to produce a smaller set of high-confidence **verified alerts**, prioritizing high **precision**.

Exploration Stage: Breadth of Discovery. As shown in Table 4, HiveMind's exploration stage discovers a total of 4,571 initial alerts—far exceeding Mango (2,310) and SaTC (144). This highlights HiveMind's ability to capture a broader spectrum of vulnerabilities. While Mango generates a higher number of CI/BOF alerts in a few key binaries due to its high-density detection capability, HiveMind identifies 259 other binary alerts (e.g., hardcoded credentials) and 2,361 non-binary alerts (e.g., XSS). These categories are completely missed by Mango and SaTC, which are limited to binary-level analysis.

Verification Stage: Precision and Autonomous Filtering. The key innovation of HiveMind lies in its verification stage. From the initial 4,571 alerts, the system autonomously validates and distills them into **1,612 verified alerts**. This is not performed by a separate, handcrafted module. Instead, HiveMind dynamically repurposes its agent architecture for this task. When an alert is identified, it is transformed into a concrete verification prompt (e.g., "*Given this specific alert..., assess its credibility...*"). The agent's role shifts from a free-roaming *explorer* to a rigorous *judge*, tasked with scrutinizing evidence to confirm a complete and exploitable attack chain. This autonomous workflow construction enables HiveMind to scale its verification capabilities and significantly reduces hallucinations.

To assess the accuracy of this autonomous verification, we manually analyzed a random sample of 300 verified alerts, adopting the threat model from Mango [11] where a TP is exploitable with legitimate device access. The results, summarized in Table 5, reveal two key findings. First, the precision of the verified alerts is **80.5%**, as 161 of the 200 alerts HiveMind flagged as vulnerable were confirmed to be genuine threats (161/200). This strong precision confirms that HiveMind's verified alerts are highly reliable. Second, the measured false negative rate, while non-zero, indicates that the verifier successfully filters out the vast majority of non-critical alerts, achieving a strong balance between accuracy and noise reduction in a fully automated workflow.

A Fundamental Paradigm Shift. The superior performance of HiveMind in both breadth and precision stems from a fundamental paradigm shift, as summarized in Table 6. The core limitation of traditional tools is that they analyze *syntax* without understanding *semantics*; they analyze *code* without understanding the *system*. In contrast, HiveMind, powered by the comprehension capabilities of

Table 4: Comparison of HiveMind's discovery pipeline against SOTA tools.

	Binary CI+BOF	Binary Others	Non-Binary	Initial Alerts	Verified Alerts
SaTC[3]	144	0	0	144	N/A
Mango[11]	2,310	0	0	2,310	N/A
HiveMind	**1,951**	**259**	**2,361**	**4,571**	**1,612**

LLMs and orchestrated by its "structured emergence" architecture, is the first to achieve automated analysis by genuinely 'understanding' the function of a software system. This fundamental shift is what enables HiveMind to surpass existing technologies, transforming a mere discovery tool into a true intelligent system for firmware security analysis.

Table 5: Manual Evaluation of HiveMind's Verification Output (Sample of 300 Alerts).

Verfication Output	Manual		Sample Size (n)
	Vulnerable	Not Vulnerable	
Classified as 'Tps'	**161**	39	200
Classified as 'Fps'	26	**74**	100

Table 6: Paradigm and Performance Comparison: HiveMind vs. Mango.

Dimension	Mango	HiveMind
Paradigm		
Model	Expert-Driven	**LLM-Driven**
Workflow	Exploration-Only	**Explore then Verify**
Performance		
Initial Alerts	2,310	**4,571**
Verified Alerts	—	**1,612**
Precision	47.9%	**80.5%**
Final True Positives	~1,107	**~1,291**

> *alert 2: HiveMind presents a future direction for vulnerability discovery in firmware security analysis, belonging to an autonomous, AI-driven analysis paradigm rather than a traditional expert system.*

5.4 Component Contribution Analysis (RQ3)

LLM Selection. We evaluate the following LLMs as the backbone for HiveMind: GPT-4o, DeepSeek v3, Claude-3.7-sonnet, and Gemini-2.5-flash on a representative subset of 10 firmware images. All models are accessed through their official APIs and integrated into HiveMind. Specifically, we use the task T5 to evaluate the performance of these LLMs in the context of firmware analysis. The labeling and verification process follows the methodology described in Table 5.

Table 7: Performance of HiveMind with different backbone LLMs on a representative subset of 10 firmware images.

LLM Model	Initial Alerts	Verified Alerts	Verification Precision (%)
GPT-4o	985	338	78.5
Claude-3.7-Sonnet	912	328	83.0
Gemini-2.5-Flash	845	311	81.3

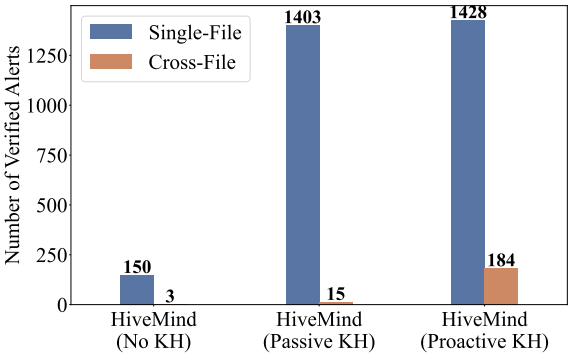

Figure 7: Performance comparison of HiveMind on Task T5 with and without the knowledge agent.

Table 7 summarizes the performance of these LLMs for alert vulnerabilities in firmware. Overall, our alerts suggest that as long as the selected LLM possesses sufficient basic reasoning capability, the performance of HiveMind remains largely robust to the specific choice of backbone model.

Impact of the Knowledge Hub. To quantify the contribution of this component, we conduct an ablation study evaluating the impact of the knowledge hub. We compare three configurations of HiveMind: (1) **without a knowledge hub**, where the agent relies solely on its limited context window; (2) **with a passive knowledge hub**, which only has the ability to store and retrieve alerts; and (3) **with a proactive knowledge hub (PKH)**, which can actively correlate information. Figure 7 presents the performance of these configurations on Task T5 (vulnerability discovery).

The results clearly demonstrate the necessity of the knowledge hub. The configuration without a knowledge hub performs the poorest, as it is severely hampered by context window limitations. This causes the agent to lose track of intermediate alerts and fail to connect related pieces of information across different files, rendering it ineffective for discovering complex vulnerabilities.

As shown in the figure, the introduction of a knowledge hub significantly improves the performance of HiveMind, particularly in identifying vulnerabilities that span multiple files. While the passive hub provides a notable improvement by simply preserving alerts, the proactive knowledge hub substantially outperforms it when multi-file reasoning is required. This highlights the proactive hub's critical ability to not only store but also persistently correlate and leverage intermediate analysis results across the entire analysis workflow, which is essential for deep firmware security analysis.

Table 8: Reasoning steps, number of delegations, and deep analysis scope for tasks (average for each firmware)

Metric	T1 (Credentials)	T2 (Components)	T3 (NVRAM)	T4 (Web Attack)	T5 (Explore then Verify)
Steps & Delegations					
Total Reasoning Steps	692.52	1718.20	1521.80	443.20	5515.1
Total Delegations	53.36	89.52	70.48	29.20	291.4
Deep Analysis Scope					
Directory Count	6.80	9.12	4.84	4.80	13.12
File Count	57.20	96.48	64.56	22.24	71.69

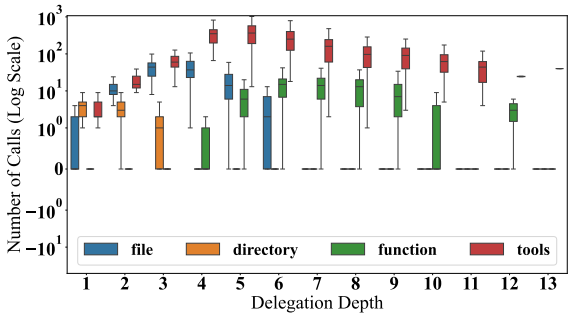

Figure 8: The distribution of the number of agent delegations and basic tool executions in T5 across layers for exploration.

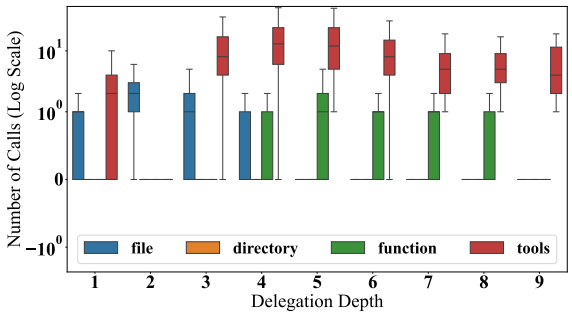

Figure 9: The distribution of the number of agent delegations and basic tool executions required for verifying one alters in T5 across different layers.

Long-Term/Large-Scale Reasoning. To quantify the complexity of different firmware analysis tasks, we measure the total steps required by the agent system to complete each task. As reported in Table 8, this metric reflects the cumulative number of steps and delegations across the agent hierarchy.

Tasks such as T2 (third-party component analysis) and T5 (vulnerability detection and verification) involve over 1700 and 6000 steps on average, indicating the significant depth and breadth required for effective firmware analysis.

In addition to reasoning depth, firmware tasks pose substantial challenges in reasoning breadth. Each task often requires traversing complex file system hierarchies, parsing different file types, and linking semantic relationships across files. Table 8 also reports the average number of directories and files accessed per task.

Task T5 covers over 15 directories and nearly 80 files,showing that comprehensive firmware analysis requires both deep multi-turn reasoning and wide-ranging file system coverage.

Dynamic Workflow Adaptation. Figure 8 and Figure 9 visualize HiveMind's dynamic workflow adaptation by the number of agent delegations and tool invocations across different layers of the hierarchy. The patterns differ significantly depending on their tasks.

In the T5 exploration phase, the workflow reveals a clear hierarchical structure Agent delegation, particularly file delegation, peaks at mid-level depths (3-5), alongside heavy use of basic tools. This indicates that higher-level agents delegate complex analysis to specialized lower-level agents, dynamically expanding the agent tree to cover the problem space in a wide-ranging search for potential vulnerabilities.

In contrast, the targeted T5 verification phase exhibits a different dynamic. Here, the workflow is less about broad expansion and more about focused, evidence-based analysis. Delegation activities are minimal, while function delegation and basic Tools are invoked more intensively. This indicates a strategic shift from structural exploration to a deep, forensic investigation of a specific lead. This comparison demonstrates that HiveMind constructs a ToA for each stage of a complex task. Discovery involves broad delegation and agent expansion, whereas verification triggers a narrower but deeper invocation of analysis tools.

> *alert 3: The power of HiveMind lies in its architecture, not just the LLM. The dynamic generation of a unique Tree of Agents (ToA) for each task empirically proves that complex, exploratory problems cannot be solved by rigid, static workflows; they require an architecture that can adapt its own structure to the problem at hand.*

6 DISCUSSION: LIMITATIONS AND FUTURE DIRECTIONS

In this section, we discuss the primary limitations of our approach not as standalone weaknesses, but as direct catalysts for the future evolution of the HiveMind paradigm. Each challenge naturally maps to a promising research direction, framing a clear course from immediate, practical enhancements to a long-term vision for AI-driven analysis.

A core challenge inherent to any LLM-based system is the risk of *hallucination*—generating syntactically plausible

but semantically incorrect outputs—and *omission*—failing to identify relevant information. Our design philosophy of structured emergence directly confronts these risks. The *structured* aspect, enforced by the RDE's hierarchical blueprint and the PKH's evidence cross-referencing, breaks down tasks into smaller, verifiable steps, thus constraining the LLM and mitigating hallucination. This high-level structure is complemented by robust engineering practices at the agent level, such as strict environmental and path constraints to prevent scope creep, validation of communication formats to ensure coherent inter-agent dialogue, and retry mechanisms for tool interactions to handle transient errors from sub-agents.

However, the risk of *omission* remains particularly challenging when an LLM fails to recognize the significance of non-obvious targets. This points to the need for integrating more specialized, high-efficacy analysis tools not just for deep analysis, but for initial triage. For instance, while an LLM might overlook a custom binary in a large directory, a preliminary scan with a dedicated tool for cryptographic constant scanning or dependency analysis could authoritatively flag it as a high-priority component, forcing the agent's attention where it might otherwise have been missed. This shifts the burden from the LLM's fallible open-ended reasoning to the reliable output of a domain-specific tool. Closely related is the fact that the current implementation of HiveMind operates exclusively through static analysis tools (`radare2`, `grep`, etc.). This fundamentally limits its scope, as it cannot detect vulnerabilities that only manifest at runtime. These intertwined challenges of reliability and analytical scope point directly towards a powerful solution: evolving HiveMind from an LLM-centric analyzer into a hybrid reasoning system. This directly addresses the current limitations of static-only analysis and improves verification integrity. Future work will focus on integrating powerful, deterministic analysis engines as specialized tools. For instance, upon identifying a high-risk data path, an agent could delegate a sub-task to a symbolic execution engine like `angr` for concrete exploitability validation. For runtime analysis, an agent could orchestrate firmware emulation in an environment like QEMU. In this powerful model, the LLM graduates from a mere analyzer to a "master strategist," using its high-level understanding to guide the focus of these narrow-but-powerful expert tools and interpret their results.

Another practical limitation is cost and efficiency. As demonstrated in our experiments, HiveMind's deep and broad analysis comes at a significant computational cost, reflected in its high token consumption. To provide a concrete measure, executing the comprehensive vulnerability analysis task with our primary model costs approximately $5 per firmware. It is worth noting, however, that this cost is not fixed. As our evaluation in Section 5.4 shows, more cost-effective models like `DeepSeek-v3` can achieve strong, comparable performance at a fraction of the price. This highlights a clear path toward optimization, though the trade-off between cost and peak analytical capability requires further study. Nevertheless, the current cost may be a practical barrier for large-scale deployment. Addressing this requires a strategic shift.

Firmware analysis is fundamentally a search problem: identifying promising leads from a vast sea of noise. In this context, the LLM excels as a hypothesis generator. A conventional optimization strategy would involve a *multi-model approach*: delegating routine, high-frequency tasks to smaller models while a powerful LLM acts as a high-level orchestrator. However, we argue that a more fundamental and ultimately more effective solution lies in creating a unified, specialized intelligence. This philosophy posits that the ultimate goal is to train a single, dedicated model. Such a model would learn *effective information extraction and rational action-taking* directly from domain data. The emergent, hierarchical decision-making process, currently guided by the external HiveMind framework, would be internalized as an intrinsic capability of the model itself. This ambitious path aims not just to manage cost, but to distill the broad reasoning of generalist models into a lean, potent, and purpose-built intelligence, fusing the creative capabilities of large models with the learned efficiency of a domain expert.

Finally, a potential threat to external validity lies in whether the dataset used in our evaluation adequately represents the diversity and complexity of the broader firmware ecosystem. While the dataset has been widely used in prior works, including Karonte [26], Mango [11], and SaTC [3], we acknowledge that it may still underrepresent rare architectures, proprietary firmware formats, or newer devices. To mitigate this, we plan to continuously supplement our experiments with a wider range of firmware from public sources in the future.

The ultimate answer to this challenge, however, is not just more data, but a more generalizable paradigm. A crucial question is whether the HiveMind paradigm extends beyond firmware analysis. We posit that its core principles—the RDE and PKH—form a generalizable framework for any task requiring deep exploration of structured targets. The philosophy of structured emergence is broadly applicable to domains like large-scale software repository auditing (navigating dependency graphs) or network penetration testing (recursively exploring subnets, hosts, and services). The key prerequisite for such generalization is the *abstraction* of new problem domains into the hierarchical blueprints HiveMind can operationalize. For instance, a complex network penetration task could be abstracted into a `network segment -> host -> service -> vulnerability` hierarchy, enabling HiveMind to systematically decompose the problem and unlock the power of recursive delegation.

These evolutionary paths culminate in our ultimate vision: HiveMind as the "brain" of a general-purpose, hybrid analysis team. This system would leverage the LLM's unparalleled capabilities in hypothesis generation and planning while delegating the "heavy lifting" to specialized "expert tools"—whether they are deterministic analyzers or other delegated LLM-agents. This synergy combines the creative, exploratory power of LLMs with the rigorous precision of traditional analysis, establishing a scalable paradigm for applying AI-driven, emergent workflows to solve the most complex exploratory problems in science and engineering.

7 RELATED WORK

7.1 LLM Agents

Large Language Models (LLMs) based on Transformer architectures (e.g., GPT-4 [21], Claude [17], LLaMA [18], Codex [20]) have demonstrated strong reasoning, planning, and generalization capabilities. Their performance depends heavily on how prompts are crafted. Advanced reasoning methods include task decomposition [34], chain-of-thought prompting [35], and decision trees via Tree-of-Thought [43]. Techniques like Reflexion [29] and Chain of Hindsight [15] introduce self-feedback to improve output quality.

LLM-based Agents use LLMs as a central controller to perceive environments, decompose tasks, and take actions through external tools [22, 32, 38]. Key modules include: (1) *Planners* that create task graphs [31]; (2) *Perception modules* enabling multimodal input [44]; and (3) *Action modules* that expand capabilities via tool usage [13, 16, 27]. LLMs have shown strong potential in software engineering (e.g., GitHub Copilot [19]) and security tasks [4, 10, 24, 25]. For instance, Feng and Chen [10] used LLMs to replay Android bugs, while others have explored vulnerability discovery and reverse engineering [14, 23].

7.2 Embedded Firmware

Security Analysis. Firmware poses numerous attack vectors: (1) *Third-party libraries* with known CVEs [45, 46]; (2) *Hardcoded secrets* like passwords and keys [5]; (3) *Config files and certificates* that can expose services [1]; (4) *Open services* such as HTTP or FTP that attackers can exploit [6]. Chen et al.[3] proposed a novel input-centric strategy that identifies bugs in embedded systems by extracting and sharing common input keywords across different devices, reducing reliance on heavyweight static analysis.

Vulnerability Discovery. Prior work uses static and dynamic analysis to detect firmware flaws [2, 9, 26]. MANGO [11] introduced a scalable taint-style analysis pipeline to identify vulnerabilities in binary firmware services at scale. Redini et al.[26] explored inter-binary taint propagation, while others applied symbolic execution and modular program analysis[8, 12, 40].

Prior works lack the adaptive planning and deeper semantic understanding offered by LLM-based agents. Our work complements this direction by enabling dynamic, task-driven analysis.

8 CONCLUSION

In this paper, we introduced HiveMind, a novel agent system that pioneers a new paradigm of structured emergence to master the complexities of deep software system analysis. We identified a fundamental mismatch between the dynamic, exploratory nature of tasks like firmware security auditing and the inherent rigidity of existing agent frameworks. HiveMind resolves this mismatch through its synergistic architecture. The Recursive Delegation Engine (RDE) orchestrates an *emergent* and adaptive workflow, enabling analysis with depth and breadth. This dynamic process is anchored by the Proactive Knowledge Hub (PKH), which provides the *structural* foundation for consistent, evidence-driven collective intelligence.

Our extensive experiments validated the superiority of this new paradigm. HiveMind not only outperformed existing agent architectures but also demonstrated a unique capability for autonomous exploration and verification. It successfully navigated real-world firmware to discover over 4,500 initial security alerts and then autonomously distilled them into over 1,600 high-confidence, actionable vulnerabilities with 80.5% precision.

Our results highlight that agent systems like HiveMind can transform cybersecurity analysis from a labor-intensive manual process into an automated workflow that scales with the reasoning capabilities of human experts.

REFERENCES

[1] Firmware walker for collecting firmware sensitive information. https://github.com/craigz28/firmwalker, accessible 2024.

[2] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, 2016.

[3] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 303–319, 2021.

[4] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

[5] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium*, 2014.

[6] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016.

[7] OpenAI et al. Gpt-4 technical report, 2024.

[8] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th USENIX Security Symposium*, 2020.

[9] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491, 2016.

[10] Sidong Feng and Chunyang Chen. Prompting is all your need: Automated android bug replay with large language models. *arXiv preprint arXiv:2306.01987*, 2023.

[11] Wil Gibbs, Arvind S Raj, Jayakrishna Menon Vadayath, Hui Jun Tay, Justin Miller, Akshay Ajayan, Zion Leonahenahe Basque, Audrey Dutcher, Fangzhou Dong, Xavier Maso, et al. Operation mango: Scalable discovery of {Taint-Style} vulnerabilities in binary firmware services. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7123–7139, 2024.

[12] René Helmke and Johannes vom Dorp. Towards reliable and scalable linux kernel cve attribution in automated static firmware analyses. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 201–210. Springer, 2023.

[13] Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. Large language models can self-improve. *arXiv preprint arXiv:2210.11610*, 2022.

[14] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. The hitchhiker's guide to program analysis: A journey with large language models. *arXiv preprint arXiv:2308.00245*, 2023.

[15] Hao Liu, Carmelo Sferrazza, and Pieter Abbeel. Languages are rewards: Hindsight finetuning using human feedback. *arXiv preprint arXiv:2302.02676*, 2023.

[16] Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. Chameleon: Plug-and-play compositional reasoning with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[17] Anthropic Org. Claude is a next generation AI assistant. https://claude.ai/, 2023.

[18] Meta Org. Llama 2: open source, free for research and commercial use. https://llama.meta.com/llama2/, 2023.

[19] Microsoft Org. Copilot: The AI developer tool. https://github.com/features/copilot, 2023.

[20] OpenAI Org. OpenAI Codex: AI system that translates natural language to code. https://openai.com/blog/openai-codex, 2023.
[21] OpenAI Org. The OpenAI API is used for a range of models and fine-tune custom models. https://platform.openai.com/docs/introduction, 2023.
[22] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22, 2023.
[23] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE, 2023.
[24] Hammond Pearce, Benjamin Tan, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, and Brendan Dolan-Gavitt. Pop quiz! can a large language model help with reverse engineering? *arXiv preprint arXiv:2202.01142*, 2022.
[25] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. Can large language models reason about program invariants? In *International Conference on Machine Learning*, pages 27496–27520. PMLR, 2023.
[26] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *IEEE Symposium on Security and Privacy (SP)*, 2020.
[27] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.
[28] Xiuwei Shang, Shaoyin Cheng, Guoqiang Chen, Yanming Zhang, Li Hu, Xiao Yu, Gangyang Li, Weiming Zhang, and Nenghai Yu. How far have we gone in stripped binary code understanding using large language models. *arXiv preprint arXiv:2404.09836*, 2024.
[29] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.
[30] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. *arXiv preprint arXiv:2312.12575*, 2023.
[31] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.
[32] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024.
[33] Xinchen Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. A repository-level dataset for detecting, classifying and repairing software vulnerabilities. *arXiv preprint arXiv:2401.13169*, 2024.
[34] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.
[35] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.
[36] Yizhou Wu, Diyi Yang, Kexin Wang, Vincent Y. F. Tan, Canwen Xu, Zhenyu Yang, Xiang Li, Xiaoxiao Tan, et al. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2309.12307*, 2023.
[37] Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. Os-copilot: Towards generalist computer agents with self-improvement, 2024.
[38] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.
[39] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024.
[40] Can Yang, Zhengzi Xu, Hongxu Chen, Yang Liu, Xiaorui Gong, and Baoxu Liu. Modx: binary level partially imported third-party library detection via program modularization and semantic matching. In *44th International Conference on Software Engineering*, 2022.
[41] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024.
[42] John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback, 2023.
[43] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*, 2023.
[44] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.
[45] Binbin Zhao, Shouling Ji, Jiacheng Xu, Yuan Tian, Qiuyang Wei, Qinying Wang, Chenyang Lyu, Xuhong Zhang, Changting Lin, Jingzheng Wu, et al. A large-scale empirical analysis of the vulnerabilities introduced by third-party components in iot firmware. In *31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022.
[46] Binbin Zhao, Shouling Ji, Xuhong Zhang, Yuan Tian, Qinying Wang, Yuwen Pu, Chenyang Lyu, and Raheem Beyah. {UVSCAN}: Detecting {Third-Party} component usage violations in {IoT} firmware. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3421–3438, 2023.

A CORE AGENT PROMPTS

This appendix provides the core system prompts and instruction templates used to guide the behavior of agents within the HiveMind framework. These prompts are crucial for enforcing the principles of structured emergence, evidence-driven analysis, and hierarchical delegation.

A.1 Directory-Level Agent Prompt

The following system prompt is used for the top-level agents responsible for navigating and exploring the firmware filesystem. It emphasizes structured exploration and delegation to specialized sub-agents.

> You are a firmware filesystem static analysis agent. Your task is to explore and analyze based on the current analysis focus (a specific directory). Please focus on the current focus, and when you believe the analysis is complete or cannot be further advanced, proceed to the next task or end the task.
>
> **Core Workflow:**
>
> **Understand the Requirements**: Always focus on the specific task, while also referring to the user's overall or initial requirements. Do not perform cross-directory analysis.
>
> **Understand the Context**: Use tools to precisely understand your current analysis focus and location.
>
> **Delegate Tasks**: When in-depth analysis is needed, call the appropriate analysis assistants: Explore Directory or Analyze File.
>
> **Summarize and Complete**: After completing all analysis tasks for the current focus, summarize your findings. If all tasks are

completed, use the ‘finish‘ action to end.

A.2 File-Level Agent Prompt

This prompt governs the behavior of *File Agents*. It instructs the agent to perform a deep analysis of a specific file, manage delegation for code-level tasks, and adhere to a strict, evidence-based reporting format.

> You are a specialized file analysis agent. Your task is to deeply analyze the currently specified file and provide detailed, evidence-backed analysis results. Please focus on the current file.
>
> **Workflow:**
>
> **Understand the Task**: Focus on the specific task for the current analysis file and fully reference the user's overall requirements. Do not perform cross-directory analysis.
>
> **Perform Analysis**: Ensure your analysis has sufficient depth. For complex tasks, decompose them into multiple sub-tasks and call assistants or tools. For complex call chains, use ‘Delegator‘ and provide detailed taint information.

A.3 Function-Level Agent Prompt

This highly constrained prompt is for *Function Agents*, which perform deep call-chain tracing. Its strict rules are designed to prevent hallucination and ensure the agent performs as a focused, deterministic tracer.

> You are a highly specialized firmware binary function call-chain analysis assistant. Your sole and exclusive mission is: starting from the currently specified function, to strictly, unidirectionally, and forward-trace the specified tainted data until it reaches a sink.
>
> **Strict Behavioral Directives (Must Adhere):**
>
> **Absolute Focus**: Your analysis scope is LIMITED ONLY to the currently specified function and the sub-functions it calls. You are STRICTLY PROHIBITED from analyzing any other functions or code paths unrelated to the current call chain.
>
> **Unidirectional Tracing**: Your task is FORWARD TRACING. Once the taint enters a sub-function, you must follow it in. You are STRICTLY PROHIBITED from returning or performing reverse analysis.
>
> **Delegate for Depth**: To trace inside a sub-function, you MUST delegate the task to a new child *Function Agent*. Do not analyze the sub-function's internals yourself; your only job is to create the new sub-task.
>
> **No Assessment**: You are STRICTLY PROHIBITED from providing any form of security assessment, remediation advice, or any subjective commentary.
>
> **Complete Path**: You must provide the COMPLETE, REPRODUCIBLE propagation path from the taint source to the sink.
>
> **Final Report Format:**
> At the end of the analysis, present all discovered complete taint propagation paths. The code snippet MUST be real and verifiable.

A.4 Knowledge Hub Agent Prompt

This prompt is for the Proactive Knowledge Hub (PKH) agent, which manages the system's collective memory. It emphasizes data quality, proactive correlation, and strict validation before storing information.

> You are a firmware analysis knowledge base agent, responsible for efficiently and accurately handling the storage, querying, and correlation analysis of firmware analysis findings. When there is no valid and risk information, or it is irrelevant to the user's request, do not perform any storage or query operations.
>
> **Tool Usage Guide**
>
> **1. Store Findings (StoreStructuredFindings)**
> **Purpose**: Store structured analysis findings in the knowledge base. Strictly filter for findings that are practically exploitable and have a complete, verifiable attack chain.
> **Key Requirements**:
> Establish correlations by storing lists of keywords with the same meaning.
> If you discover more credible, deeper findings through correlation, you must proactively store them, especially for tracing taint flow between components to determine the complete vulnerability chain, provided you are certain they are truly related.
>
> **2. Query Findings (QueryFindings)**
> **Purpose**: Query for findings in the knowledge base based on specific criteria.
>
> **3. List Unique Values (ListUniqueValues)**

Purpose: Explore the unique values of a specific field in the knowledge base.

A.5 Verification Task and Instruction Template

The verification stage uses a distinct set of prompts to repurpose the agent architecture from exploration to validation. The template below defines the rigorous, evidence-driven task for validating a potential security alert.

> Your sole mission is to rigorously and objectively verify the following security alert. Your analysis must be based exclusively on the provided evidence.
>
> *{alert_details}*
>
> **Core Principles**:
> 1. **Evidence-Driven**: Every claim in the alert must be validated by analyzing the provided evidence. Speculation or analysis of unrelated information is forbidden.
> 2. **Logic Scrutiny**: Do not just confirm code existence; you must understand its execution logic. Scrutinize conditional statements, data sanitization, and other factors that determine if a code path is reachable.
> 3. **Exploitability Validation**: Verify that the vulnerability is actually EXPLOITABLE by confirming: Input Controllability, Path Reachability, and Real Impact.
> 4. **Complete Attack Chain**: Verify the COMPLETE propagation path from an attacker-controlled input to a dangerous sink, with every step supported by evidence.