LUMEN: A Systems Approach to LLM-Guided Activation of Hidden Behaviors in Malware

Kevin Valakuzhy[1], Miuyin Yong Wong[2], Douglas M. Blough[1], Mustaque Ahamad[1], Fabian Monrose[1]
[1]Georgia Institute of Technology, Atlanta, GA, USA
[2]University of Maryland, College Park, MD, USA
Email: kevinv@gatech.edu, miuyin@umd.edu, {doug.blough, fabian}@ece.gatech.edu, mustaq@cc.gatech.edu

***Abstract*—Malware increasingly conceals its most damaging behaviors, exposing only a controlled surface to analysis systems. Existing tools rely on expensive path exploration or brittle evasion-specific heuristics, leaving analysts to manually determine how to reveal hidden behavior. Although Large Language Models (LLMs) offer new assistance, they often misidentify the code that suppresses malicious activity.**

We introduce LUMEN, an LLM-assisted system that identifies decision points that hide behavior and constructs activation methods to expose it. LUMEN combines domain-specific pre-processing and post-processing with standard analyst tools, enabling seamless use within existing pipelines. Evaluated on 22 malware families where the CAPE sandbox fails, LUMEN enables CAPE to uncover severe malicious activity in 68% of samples and reveals 4× more behavior than JuanLesPIN, a system designed to counter behavior-hiding techniques. LUMEN also outperforms direct prompting of 12 LLMs in 64% of cases while significantly reducing token usage, providing a practical path for scalable LLM-assisted analysis.

***Index Terms*—Malware, Dynamic Analysis**

I. Introduction

Malicious software continues to pose a severe global threat, with escalating financial and operational impacts. In a single incident in 2024, a ransomware attack caused over $800 million in damages [1] and exposed sensitive data from more than 100 million individuals [2]. While automated defenses are essential for mitigating large-scale threats, human analysts must step in when malware's true behaviors remain hidden. These behaviors may be deliberately obscured through evasion techniques or may depend on additional runtime configuration details. Ugarte-Pedrero et al. [3] found that malware that fails to exhibit malicious behaviors in sandboxes constitutes the vast majority of samples requiring manual analysis. Despite significant automation and 900 hours of manual triage, they estimated that analyzing the remaining samples would still require the equivalent of 250 full-time analysts per day. This daunting workload motivates the need for methods that help uncover hidden malicious behaviors in malware [4].

The recent emergence of advanced Large Language Models (LLMs) presents a promising path to improve malware analysis. Commercially available LLMs have shown potential in reverse engineering [5], binary analysis [6], and summarizing the functionality of malicious code [7]–[9]. These models also offer a flexible approach to locating code related to hidden behaviors, a key advantage over rigid approaches that detect specific techniques malware uses to hide malicious behaviors, such as *JuanLesPIN* [10], MORRIGU [11], and BluePill [12]. Importantly, LLMs can be used directly, without additional fine-tuning challenges, making automated support for analysis workflows significantly more powerful.

Despite their promise, applying LLMs to uncover hidden behaviors in real-world malware remains a significant challenge. Prior work on code summarization for malware [8], [9] requires analysts to manually locate the code of interest. Using LLMs to automatically locate relevant code in malware is a substantially harder problem, as success requires both understanding the binary code's functionality and its operational role within the malware sample, an area where LLMs struggle [6], [13]. In addition, malware analysis is a particularly challenging domain as necessary information is spread across the output of disparate tools and techniques, increasing the risk of performance degradation due to insufficient [14], [15] or extraneous [16] context.

In this work, we introduce LUMEN, a system that adapts LLMs to effectively identify code that hides behaviors in real-world malware and then automatically construct methods to activate these hidden behaviors. Our methodology draws on insights from academic research, industry best practices, and conversations with professional malware analysts to systematically design domain-specific software components that address LLMs' limitations and analysts' needs. We find that these adaptations, which include context synthesis and multi-level verification, significantly improve the ability of state-of-the-art LLMs to locate code related to hidden behaviors.

Our contributions are as follows:

- We present a design and implementation of the first LLM-assisted architecture for locating code responsible for concealing malicious behaviors from dynamic analysis and constructing activation methods, leveraging outputs from common malware analysis tools.
- We introduce a malware dataset of samples from 22 malware families with behaviors identified by professional analysts that remain unexposed by CAPE, a widely used malware sandbox in academic [17], [18] and industry settings [19]. Using this dataset, we demonstrate that LU-

This material contains work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001123C0035. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Department of Defense (DoD).

MEN's domain-specific adaptations improve the LLMs' identification of code responsible for hiding behaviors in 64% of cases across twelve leading models. Additionally, our system correctly identifies relevant code that is completely missed by direct LLM usage in 38% of cases.
- We show that LUMEN's automatically generated activation methods significantly increase the behaviors exposed by existing dynamic malware analysis systems. In the CAPE sandbox, LUMEN uncovers severe malicious behaviors in 68% of evaluated samples, all of which previously revealed none. LUMEN also improves the state-of-the-art *JuanLesPIN* analysis tool [10], which is explicitly designed to neutralize behavior-hiding techniques and maximize observable runtime activity [20]. Using LUMEN's output, *JuanLesPIN* extracts four times more APIs under their own measure of "externally visible effects," demonstrating LUMEN's ability to uncover diverse root causes of hidden behaviors and its utility across multiple analysis systems.

Overall, our results show that practical integration can be achieved through a carefully designed modular architecture instead of relying on increasingly large or costly models.

II. Background

Malware analysts are often called upon for in-depth examination of malicious software, such as samples linked to an ongoing incident or those used to inform future defenses [4]. In these scenarios, analysts rely on controlled execution environments like sandboxes to help expose malicious behaviors and to devise remediation strategies [21]. Unfortunately, this demanding job is made even more difficult by malware samples that do not execute their malicious behaviors in modern analysis environments. As a result, analysts are forced into a tedious cyclic workflow, alternating between static and dynamic analyses to locate and bypass these evasion techniques [4]. To address this challenge, it is critical to know how malware's true behaviors are concealed.

Hidden behaviors can arise for many reasons, but they share a common root: mismatches between the malware's intended runtime environment and the analysis system. These mismatches may result from deliberate tactics meant to hinder analysis or from unintentional failures caused by missing dependencies or unexpected environmental differences. This shared structure produces consistent artifacts. Importantly, malware often queries the execution environment through Windows APIs or system calls[1] to check for features such as sandbox registry keys or language settings used to avoid specific regions. If collected information differs from expected values, intended behaviors often remain dormant as the malware exits or shifts to benign code paths. The choice of path is typically made by a branch instruction, which we call the *decision point*. These artifacts, namely environmental queries, dormant malicious code, and decision points, provide key signals for locating and eventually activating hidden behaviors.

[1]For simplicity, we refer to both as APIs.

Locating hidden behaviors from a single artifact is challenging. Many APIs used to gather environmental information or trigger malicious actions also appear in benign code, making their purpose ambiguous. Environment-querying code may be far removed from both the decision logic and the dormant behaviors it guards, and obfuscation or packing can further obscure functionality until runtime. Lacking clear links between queries, decision points, and latent behaviors, it is difficult to determine which code actually governs evasive logic. Robust identification therefore requires reliably connecting environmental checks, decision points, and their outcomes.

Making these connections is non-trivial. As prior successes using off-the-shelf LLMs for malware-related tasks have shown, selecting the appropriate text to provide the LLM in a prompt, referred to as context, is especially important because LLMs perform poorly when given too little [7] or too much [22], [23] information. Malware analysis complicates this even further because it draws on multiple, complementary sources. Static analysis reveals control flow and decision logic but lacks details about environmental information gathered via APIs; dynamic analysis exposes runtime behavior but omits dormant code paths [4]. Therefore, effective LLM-assisted analysis requires consolidating these disparate sources and following design principles that mitigate known LLM weaknesses.

III. Motivation

Motivated by the success of off-the-shelf LLMs in related domains, we explore how well they can identify decision points for hidden behaviors in real-world malware. This task requires complex reasoning, including understanding the significance of environmental queries, analyzing control flow, and identifying paths that likely lead to malicious behaviors.

As a case in point, we prompt an advanced LLM, Claude Sonnet 4, to identify branch instructions that act as decision points for hidden behaviors. In this example, we use a real-world sample from the `Plead` malware family, previously analyzed by ESET [24]. We provide the LLM data from tools commonly used by analysts: disassembly from IDA Pro and dynamic execution traces (executed APIs and basic blocks) from the CAPE sandbox, a popular open-source malware sandbox used in academic [17], [18] and industry [19] settings. To satisfy input length constraints, we segment the disassembly across multiple requests and aggregate the results.

With this approach, the LLM fails to identify the evasion logic after producing 83 responses, the highest confidence of which are shown in Fig. 1. Many responses point to instructions that are non-branching or never executed, as shown in the columns under "Verified", which disqualifies them from being the correct decision point. These results suggest that, in their raw form, LLMs are inadequate in locating code relevant to hidden behaviors. This is not too surprising a conclusion given that LLMs are known to fail on complex problems [16], especially those requiring extensive context [23], [25].

Rather, this failure illustrates the limitations of naïvely applying LLMs without domain knowledge, even when the necessary context is provided to the model. The problem posed

Claude Sonnet 4 Output (83 Responses)			Verified		
Branch Address	Conf.	Branch Purpose	Branch	Executed	Correct
0xA57B	10	Process termination decision point.	✗	✓	✗
0x3A928	10	Main configuration switch statement that controls various malware features based on input parameters.	✗	✗	✗
0x3E634	9	Checks if malware initialization succeeded - if initialization fails, malware exits early without executing payload.	✓	✗	✗
⋮	⋮		⋮	⋮	⋮

Claude Sonnet 4 Output + LUMEN (1 Response)					
0x4DA7	9	Check if exactly 2 command-line arguments are provided to proceed with password stealing functionality.	✓	✓	✓

Reasoning: This branch instruction checks for exactly 2 command-line arguments. If the condition fails (which is likely in a basic sandbox execution), the malware exits early at 00004DB2 *without executing the core malicious functionality that includes memory allocation, registry access, and Outlook password extraction* (emphasis added). The subsequent code shows clear malicious intent with Outlook-specific strings and registry key access, but this early argument check prevents execution in sandboxes that don't provide the expected command-line parameters.

Fig. 1: Comparison of candidate decision points identified by Claude Sonnet 4 with and without our approach, sorted by confidence. Verification indicates the response's adherence to expected properties of the correct decision point.

to the LLM is compounded by intermediate reasoning steps forced upon it, such as identifying irrelevant code to safely disregard and cross-referencing disparate data sources. After systematically reviewing LLM workflow design guidelines from research and industry, we found that applying sound engineering practices, such as offloading intermediate reasoning to traditional software components, can significantly enhance the performance of off-the-shelf LLMs.

As part of developing our approach, we conducted IRB-approved interviews with five professional malware analysts from major security companies, each with more than ten years of experience, to understand their requirements for a system that assists in exposing hidden behaviors. Each interview lasted about 1 hour. All participants reported that uncovering such behaviors is highly time-consuming, and four expressed that both diagnosing the root causes and devising activation strategies demand substantial manual effort. When asked about the capabilities of an ideal tool, analysts were divided between prioritizing clearer explanations of the underlying causes of hidden behaviors and desiring a dynamic analysis environment that automatically configures itself to the malware's operational requirements. Given that LLM-based systems can introduce subtle or hard-to-detect errors, we also asked what evidence would be necessary for them to effectively use or trust such a system. In short, they consistently emphasized that adoption would require outputs that can be readily cross-validated against existing, trusted analysis tools.

Guided by these insights, we developed LUMEN (**L**LM-assisted **U**nderstanding of **M**alware **E**vasio**N**), *an end-to-end system designed to help uncover malware behaviors that remain hidden even from state-of-the-art dynamic analysis systems*. Rather than introducing an entirely new dynamic analysis system, our goal is to work in tandem with existing runtime environments to expose even more behaviors.

A. Guiding Principles

LUMEN's design is guided by five principles (which influence the steps in Fig. 2) distilled from best practices [26]–[31] and interviews with malware experts.

P1: Leverage domain-specific tools: External tools are often used to offload work from LLMs to more predictable software components [9], [27]. Jin et al. [13] show that integrating tools such as disassemblers improves LLM performance, since models understand disassembled code better than raw binaries. We therefore incorporate both static and dynamic analysis tools to supply essential context. Providing analysts with the resulting tool outputs also offers familiar, trusted evidence that clarifies the LLM's reasoning [30].

P2: Curate task-specific context: Tool outputs can give LLMs the information required to solve a task, but models perform better when provided domain-specific data that support desired outcomes [26]. In particular, adding focused, relevant information [14], [15] and removing unnecessary details [15], [16] improve results. We find that consolidating context from multiple sources to reduce the LLM's reasoning burden also improves output quality for malware analysis.

P3: Generate diverse outputs: Generating multiple, diverse outputs from LLMs can mitigate the impact of hallucinations [27], [30]. In both automated and human-in-the-loop workflows, providing several candidate responses increases the likelihood of producing a useful outcome. As such, we leverage different LLMs to allow us to cover multiple possibilities for decision points when designing activation methods.

P4: Reduce task scope: Because LLM performance degrades on tasks with more complex, long-term planning [26], [32], we deliberately constrain the model's role in assisting malware analysts. Rather than adopting an agentic system, where the LLM can autonomously choose actions, we adopt a two-stage workflow aligned with established analyst practices: first identify the root cause of hidden behaviors, then determine how to activate them. Since analyst workflows involve a predetermined sequence of steps [4], the benefits of agent flexibility are less pronounced. By removing decision-making responsibilities from the LLM, we improve its consistency [28], [29] and more clearly reveal persistent limitations in LLM output that other software components can help mitigate.

P5: Use multi-layered verification: Verifying outputs helps minimize time wasted by both automated systems and human analysts when dealing with incorrect results. To this end, we check for consistency between LLM inputs and intermediate outputs to filter, and sometimes correct, clearly incorrect responses [27], [33]. We also check whether the activation method we construct exposes new malicious behaviors detected by our dynamic analysis environment.

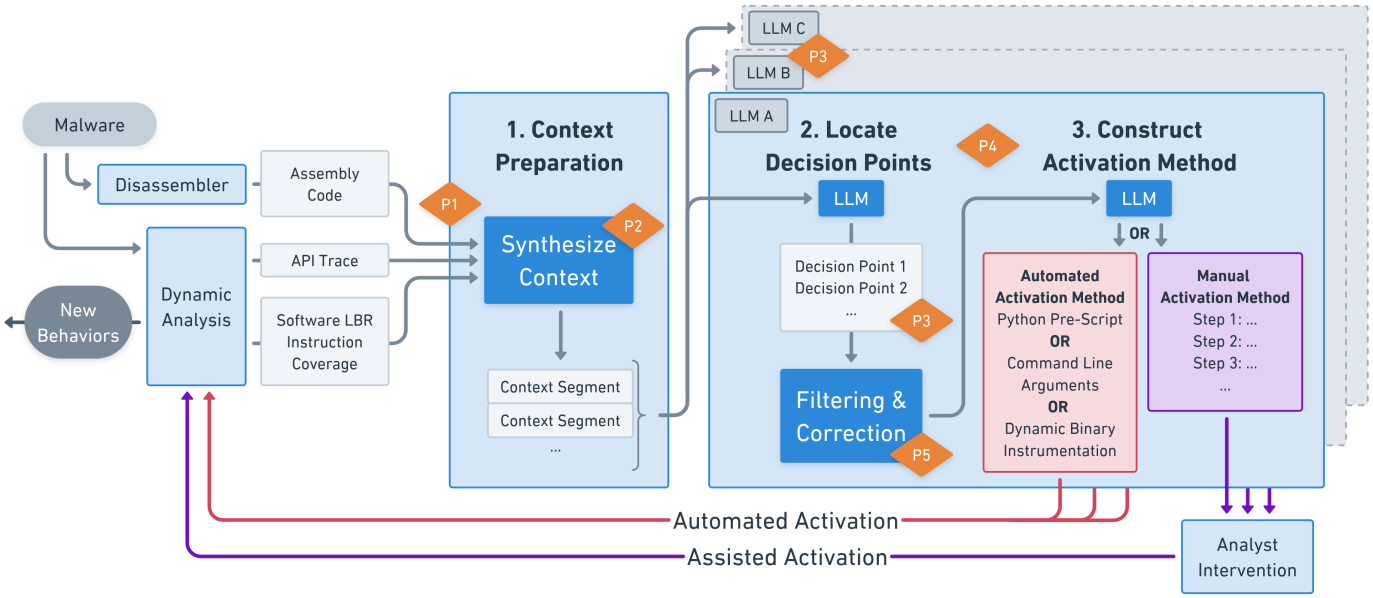

Fig. 2: High-level view of our approach for locating decision points and activating hidden behaviors. We incorporate several guiding principles, including ① leveraging output from domain-specific tools, ② curating task-specific content, ③ generating diverse outputs, ④ reducing task scope, and ⑤ using multi-layer verification.

Returning to the illustrative malware example in Fig. 1 powered by the same Claude Sonnet 4 model, LUMEN returns only a single response, which contains the correct decision point. After automatically generating the expected command-line arguments, the malware exposed malicious behaviors labeled as high severity where previously it exhibited none. In that example, the API cost of querying the LLM to locate the relevant code was also reduced from over $8 to just under $0.20 — underscoring the power of our technique.

IV. Methodology

LUMEN's processing workflow, shown in Fig. 2, proceeds in three stages inspired by the manual process analysts use to locate decision points and activate hidden behaviors [34]. In the first stage, LUMEN synthesizes data from standard static and dynamic analysis tools to give the LLM a unified view of the malware's code and observed behavior. In the second stage, the LLM analyzes this combined context to identify likely decision points that determine whether malicious behaviors will execute. Its predictions are then checked for consistency with the original inputs, and trivial inconsistencies are corrected automatically. In the third stage, LUMEN constructs an activation method that can expose the hidden behaviors during re-execution. When feasible, the system applies this method automatically; otherwise, it generates a clear set of instructions that an analyst can follow to reveal the concealed behaviors.

A. Stage ❶ Context Preparation

Our methodology begins with the context preparation stage. First, we collect relevant information about the malware sample's code and observed runtime behavior to construct the domain-specific context provided to the LLM. We include an overview of our software breakpoint-based instruction coverage technique to efficiently detect regions of executed code without hardware support. Next, these different pieces of information are synthesized to improve the LLM's ability to identify decision points. Finally, the synthesized context is segmented to fit within the LLM's finite context window.

1) Data Collection: LUMEN integrates three types of data to provide complementary perspectives for evasion: executed APIs, disassembled code, and executed instruction addresses. Each source captures key details that other sources may miss.

API calls provide high-level semantic information about the malware's behavior. By collecting API calls along with their arguments and return values, we can infer the resources the malware is trying to access. In some cases, this data alone is sufficient to identify decision points, especially for anti-analysis techniques such as registry-based sandbox detection. However, an API call's purpose is often ambiguous without considering how its output is used. For instance, identifying the reason an API call checks for a file's existence typically requires analyzing the code that depends on the API's result.

In contrast, disassembly reveals the malware's decision-making logic and the consequences of those decisions. Disassembled code exposes which APIs influence control flow decisions, and what code would execute as a result. Modern disassemblers even provide semantic context for instruction operands using dataflow and known types for API arguments and return values. However, static disassembly alone cannot always provide the necessary runtime context to locate evasion. For example, malware may obscure high-level semantics through obfuscation techniques, such as dynamic API resolution. In addition, static disassembly fails to indicate what code

was executed during dynamic analysis, limiting its utility in narrowing the search space for decision points.

To address these limitations, we incorporate the addresses of instructions executed during dynamic analysis. Since decision points must reside in executed code, we can reduce the search space of code to consider. While common tools used by malware analysts can extract disassembly and executed API traces, capturing the set of executed instructions with a sandbox or debugger is challenging due to the severe runtime overhead of instruction-level monitoring. Practical collection of executed instructions typically requires hardware support, such as Last Branch Recording (LBR) registers in modern Intel CPUs. In addition, hypervisor modifications are required to use this specialized hardware within virtual machines [35] where sandboxed malware analysis typically takes place. While dynamic binary instrumentation frameworks such as Intel Pin can efficiently collect runtime information, they are rarely used as a replacement for sandboxes because they lack the system-level visibility and convenience that make sandboxing popular. They are also difficult to use in conjunction with sandboxes because of poor interactions between their instrumentation techniques and sandbox monitoring mechanisms.

To enable collection of executed instructions solely through the sandbox's monitoring capabilities, we developed an approach, coined *Software-based LBR*. Our approach efficiently records code execution coverage using dynamically activated software breakpoints by applying two key optimizations.

First, we track only control-flow transfers. Starting from an initial breakpoint at the executable's entry point, each triggered breakpoint yields the current execution address. From that address, we disassemble forward until reaching the next control-flow instruction (e.g., a branch, call, or return). For direct control-flow instructions, we place breakpoints directly at their known destination addresses. Indirect control-flow instructions require a different approach because their targets depend on runtime register or memory values. Since these destinations cannot be resolved statically, we place a breakpoint on the indirect instruction itself. When that breakpoint triggers, we disassemble the instruction, determine its actual destination, and then set a new breakpoint at the resolved address.

Our second optimization removes breakpoints as soon as they are no longer needed. The main risk in removing a breakpoint is losing coverage of code that may execute later. To prevent this, we enforce that every path from the current instruction pointer to unexecuted code must pass through an active breakpoint. A breakpoint is removed only if this invariant still holds. After dynamic analysis completes, we label as executed all instructions between each triggered temporary breakpoint and the next control-flow instruction, producing a complete and efficiently gathered record of the executed instruction stream. With these two optimizations, LUMEN collects runtime information with software breakpoints far faster than would otherwise be feasible (see Appendix B)

2) Context Synthesis & Segmentation: After collecting the executed instructions, executed APIs, and static disassembly, we synthesize them into a unified representation shown in Listing 1. We link API calls to the responsible instruction within the main module's disassembled code by identifying the return address on the stack. The API name, arguments, and return value are placed after the instruction responsible for the API call. We prefix the virtual address of executed instructions with a `'>>'` delimiter. Labeling executed instructions exposes candidate decision points, specifically any executed branch instruction with an unexecuted path. To reduce input tokens, we discard functions that lack decision points.

```
...
>>00008194: call    ds:InternetOpenUrlA
    HOOKED API: InternetOpenUrlA ARGS: {
     ConnectionHandle: 0x00cc0004,
     URL: http://www.ifferfsodp9ifjaposdfjhgosu...
     Headers: ,
     Flags: 0xffffffff84000000,}
    RETURN: 0xcc000c
>>0000819A: mov     edi, eax
>>0000819C: push    esi; hInternet
>>0000819D: mov     esi, ds:InternetCloseHandle
>>000081A3: test    edi, edi
>>000081A5: jnz     short loc_81BC
  000081A7: call    esi ; InternetCloseHandle
  000081A9: push    0; hInternet
  000081AB: call    esi ; InternetCloseHandle
  000081AD: call    sub_8090
...
```

Listing 1: Example unified context synthesized from static and dynamic analysis showing a successful check for a killswitch domain, causing the function call to `sub_8090` to be skipped.

Even after the filtering from context synthesis, the remaining information can still exceed the input limit for the LLM's context window. In these cases, we group assembly functions into multiple requests, collect the likely decision points from each request, and aggregate the results to find a global answer. We take a first-fit (FF) bin packing approach to group functions into separate requests capped at the LLM's input limit. To rank results across multiple responses, our system prompts the LLM to score its confidence, from 1 to 10, for each decision point in its response. After aggregating the responses, we assign a rank to each prediction after sorting by confidence score and retain only the highest confidence answers across all responses.

B. Stage ❷ Locate Decision Points

For a given segment of synthesized context, we provide it along with a prompt that follows prompt engineering best practices, such as adopting a persona [36] and using zero-shot chain-of-thought [37]. The prompt specifies that the LLM's response should include JSON containing four specific pieces of information for each suspected decision point. The first is the address of the suspected decision point, allowing us to connect the response back to an assembly instruction from the input. The second is a high level summary of the decision point's purpose to increase confidence that the LLM understands the code being analyzed. The third is a confidence score that we use to select the highest confidence decision points across all responses. The fourth is a description of the

reasoning, grounded in the input data, for why this location is suspected to be a decision point.

When LLM output deviates from the requested content, we attempt to recover from errors before discarding blatantly wrong responses. Specifically, we found that LLMs may provide the address of an instruction that is not a conditional branch, often providing the address of the instruction that decides the path the next branch will take. In such cases, we adjust the model's answer to point to the conditional branch at the end of the basic block containing the originally referenced instruction. However, cases in which the answer is clearly incorrect are filtered before the next workflow stage, removing any responses pointing to addresses in basic blocks that do not end with an executed conditional branch.

C. Stage ❸ Construct Activation Method

After filtering and correction, our system sends the candidate decision points to the LLM and tasks it with selecting an activation method for exposing hidden behaviors. To improve reliability, we restrict the LLM to four options. The first method prepares the expected environment using a Python setup script. The second provides command-line arguments to the malware upon execution. The third forces an alternate path at a decision point using dynamic binary instrumentation. For any of these three choices, LUMEN automatically applies the activation during re-execution. If suitable modifications fall outside the scope of the automated methods above, the LLM instead produces a set of manual steps for an analyst to follow. Because confidence scores are provided along with each decision point, the LLM typically chooses its activation method based on the highest-ranked decision point.

Regardless of how the activation method is chosen, any newly observed behaviors both validate LUMEN's output and provide useful insight to analysts. To increase the likelihood of activating hidden behaviors, our system uses multiple LLMs to independently identify decision points and activation methods.

V. Evaluation

We assess LUMEN's benefits by answering two questions:

- ***RQ1:*** *To what extent do the domain-specific adaptations improve the localization of decision points associated with hidden behaviors compared to standalone LLMs?*
- ***RQ2:*** *How effective is LUMEN at revealing previously hidden malicious behaviors during dynamic analysis?*

In addressing these questions, we also analyze the operational challenges and common failure patterns that arise when using LLM-assisted workflows to activate hidden behaviors.

A. Implementation

LUMEN is implemented as an extension to the open source CAPE sandbox [38]. We configured six 64-bit Windows 10 virtual machines with 8GB RAM each. We execute each sample for up to 300 seconds, a conservative setting beyond the two minute minimum runtime recommended by Kuchler et al. [39]. Our sandbox execution times, shown in the Appendix (Fig. 4), support their findings, as 95% of sandbox executions terminated within two minutes. Information about the called APIs and executed instructions is collected during runtime by injecting a monitoring DLL known as `capemon` [40] into each of the malware's processes. We modified capemon to enable our software-based LBR instruction coverage collection.

Once the sample submitted via the CAPE API interface is executed, a python module orchestrates the workflow depicted in Fig. 2. Static disassembly is collected using an IDA Pro (v7.6) script, preserving annotations presented in the IDA Pro GUI. This ensures the LLM has access to the same information that human analysts have through a disassembler. To allow cross-correlation between static and dynamic information, we convert all addresses to Relative Virtual Addresses (RVAs) to be independent of the binary's load address.

Like most dynamic analysis sandboxes, CAPE does not hook every Windows API, resulting in missing annotations for called, but unhooked, APIs in the synthesized context. However, unhooked APIs can internally invoke hooked APIs, enabling us to capture information from nested API calls. For instance, although `ExitProcess` is not hooked directly, it typically invokes the hooked API `NTTerminateProcess`. We associate nested API calls with static disassembly using the same method as for calls made directly from the malware sample's code: parsing return addresses on the stack and identifying the first address located within a memory region belonging to the malicious process. To ensure semantic correctness, we manually verified that each nested API call in our unified data was meaningfully related to its parent API. Across all analyzed samples, we observed only one semantic mismatch: the parent API (`StartServiceCtrlDispatcherW`) was followed by the API (`GetSystemTimeAsFileTime`).

After synthesizing and segmenting the context, we issue a separate query to the LLM for each segment in parallel, providing the segmented context as the "user input" and the prompt as the "system input". The system input, also known as a system prompt, typically consists of guidelines for the LLM that are consistent across queries. In contrast, the user input provides task-specific context [31].

For activating hidden behaviors, CAPE already supports specifying command line arguments and running python scripts before executing malware samples. If the dynamic binary instrumentation method is chosen, we leverage CAPE's built-in debugging capabilities to set a hardware breakpoint at the specified decision point's address. Once the breakpoint is triggered, we force control flow from the decision point towards the opposite path from the one observed during the sample's initial run. We perform this instruction pointer manipulation using CAPE's existing debugger actions feature (e.g., `bp3=0x11d8,action3=Jmp`).

B. Experimental Setup

To evaluate LUMEN's practical value [26], we focus on samples that both interest analysts and conceal malicious behaviors from modern sandbox environments. A natural starting point is malware described in public threat reports published by leading security firms. We therefore draw on the MOTIF

TABLE I: Evaluated LLMs grouped by openness of model parameters and ordered by release date. Other than temperature, set to 0 to maximize determinism, we use default settings for model parameters, including reasoning support. SWE-bench Verified results pulled from [49] if present, '-' otherwise.

Model	Release	SWE-bench Verified	Open	Reasoning	Max Token Context
GPT 5	Aug '25	65.0	✗	✓	400K
Gemini 2.5 Pro	Jun '25	53.6	✗	✓	1.05M
Gemini 2.5 Flash	Jun '25	28.7	✗	✓	1.05M
Claude Sonnet 4	May '25	64.9	✗	✗	200K
Claude Sonnet 3.7	Feb '25	52.8	✗	✗	200K
GPT 4o Mini	Jul '24	-	✗	✗	128K
GPT 4o	May '24	21.6	✗	✗	128K
GLM 4.5	Jul '25	64.2	✓	✓	131K
GLM 4.5 Air	Jul '25	-	✓	✓	131K
Qwen3 Coder	Jul '25	55.4	✓	✗	262K
Kimi K2	Jul '25	43.8	✓	✗	131K
DeepSeek V3	Dec '24	-	✓	✗	164K

dataset [41], a diverse collection of Windows PE malware captured in the wild and linked to threat reports from major firms (e.g., CrowdStrike and Palo Alto Networks). MOTIF has been used extensively in research papers [42]–[45].

We analyzed these reports to find samples with explicitly described conditions required to exhibit malicious behaviors to serve as the foundation of our dataset. Our focus is on malware that does not reveal behaviors in modern sandbox environments. Thus, we remove samples that exhibit malicious behaviors when executed under default settings for either the open-source CAPE sandbox or the commercial VMRay [46] sandbox, which is widely used in industry, government [47], and academic research [48]. In particular, we exclude samples that trigger high-severity behavioral signatures, since these indicate that the sandbox already observes key malicious behaviors. The remaining samples are those for which current sandboxing techniques *do not* surface these behaviors. To limit engineering effort for our prototype, we restrict our dataset to x86 and x64 Windows executables, exclude bytecode-based .NET executables, and use the `UnpacMe` service to unpack binaries when possible. Finally, we only evaluate on malware samples for which all required inputs were successfully collected, ensuring that the system receives all expected data.

Filtering left us with 53 samples across 22 malware families where both sandboxes missed malicious behaviors that analysts had previously documented as evasive. To establish ground truth for evasion localization, we manually reverse-engineered one sample per family, identifying decision points responsible for the evasion technique described in OSINT reports. When multiple samples existed, we selected one at random. For cases where the sandbox's evasion behavior was not covered in reports, we validated our findings by confirming that bypassing the suspected evasion caused the sandbox to reveal new malicious activity. This process produced a final dataset of 22 samples used in our evaluations.

1) Models: We started with GPT-4o and GPT-4o-mini, two widely used LLMs during LUMEN's initial development. To assess the impact of model choice, we expanded our evaluation to include the five most popular open- and closed-weight LLMs for programming, based on Openrouter usage. We include open-weight models for their ability to be run locally, allowing analysts to work with sensitive data they cannot share with third-party LLM providers. Our selected LLMs, shown in Table I, include reasoning models, which can build internal chains of thought to improve responses, as well as Qwen3 Coder, a model trained specifically for coding tasks. These models vary widely in capability, with GPT 5 producing 3x as many fixes for real-world GitHub issues from SWE-bench Verified [50] as GPT 4o, despite only a one-year gap in release.

To provide consistent context segments to each model, we limit each segment to fit within GPT 4o's context window, the smallest among the models considered. Since output tokens also count towards the context window, we further reduce the tokens per segment to reserve at least 16K tokens for output, the smallest output token size across models. We also reserve space for our system prompt and account for tokenization differences, as token counts can vary up to 30% between models. As a result, we cap each context segment to 75K tokens, measured using OpenAI's `tiktoken` package.

s

C. RQ1: Assessing LUMEN's Domain Adaptations

To demonstrate LUMEN's benefits in terms of localizing decision points that relate to hidden behaviors, we compare it to a baseline that uses the same underlying LLMs, but omits our domain-specific adaptations. In the baseline, we construct the input context by concatenating the raw inputs (e.g. disassembly, API calls, executed instructions), separating each with a descriptive heading. This baseline extends beyond the strings and statically derived API calls that comprise the context from Wang et al. [8] to include valuable execution data, such as API outputs and locations of executed code. When segmentation is necessary, we only segment the disassembly, as the dynamic information we collect consumes relatively few tokens that are appended to each segment. This avoids any cross-correlation between static and dynamic information.

Our approach is compared with the baseline in Table II. We show how LUMEN consistently improves performance across all models. Considering all 22 malware families, ***in 38% of cases, LUMEN identifies correct decision points that the baseline did not even consider*** ("+"). Overall, our approach improves the ranking of the correct decision point in 64% of cases. For example, with Claude Sonnet 4, LUMEN enables the model to correctly identify seven decision points it previously missed. It also improves the ranking of the correct decision point for eight samples, preserves the ranking for five samples, but causes the correct decision point to be dropped entirely ("−") for `Deathransom` and `Triton`. In the baseline, the decision points for these two samples appear plausible to the model but are also ranked near the bottom. After LUMEN's context filtering, the model tries to evaluate each

TABLE II: Comparison of decision point ranking of LUMEN versus baseline LLM performance. Cases where LUMEN improves results are highlighted in grey. The symbols · / ↑ / ↓ indicate unchanged/better/worse ranking of the correct decision point by the specified amount. The + / − symbols indicate LUMEN adds/removes the correct answer, while '×' means both approaches fail. Families are presented in increasing order by the number of executed branches.

Malware Family	Babymetal	Avatar	Pirpi	Skimer	Triton	Remexi	Typeframe	Kghspy	Multigrain	Plead	Vivaciousgift	Varenyky	Balkanbackdoor	Deathransom	Freenki	Artfulpie	Obliquerat	Cerber	Electricfish	Indigodrop	Mamba	Shamoon
# Executed Branches	3	14	14	35	41	45	101	215	265	281	294	309	328	356	382	401	454	574	580	609	658	690
GPT 5	·	−	↑1	·	↓1	·	·	·	↑5	↑4	↑2	·	↑9	+	↑1	+	+	↑2	↑5	↑2	↑1	−
Gemini 2.5 Pro	·	·	↑7	·	×	↑2	·	·	↑2	↑9	↑4	·	↑2	↑3	↑2	↑1	+	·	↑14	+	↑3	↑12
Gemini 2.5 Flash	·	·	↑60	·	+	+	↑2	·	+	+	+	·	↑7	×	↓1	+	×	·	+	↑21	↑10	+
Claude Sonnet 4	+	·	↑11	·	−	↑3	↑1	·	↑6	+	+	↑1	+	−	+	↑3	×	+	↑34	×	↑12	+
Claude Sonnet 3.7	·	+	+	·	×	×	↑1	·	+	+	×	·	+	+	+	↑1	×	+	×	+	↑19	×
GPT 4o Mini	+	+	+	−	×	+	↑1	↑6	+	+	+	↑4	↑13	×	×	+	×	+	×	×	↑3	×
GPT 4o	+	+	+	+	+	+	+	+	+	+	+	·	↑11	+	+	×	×	+	+	↑3	↑8	×
GLM 4.5	·	·	+	·	×	↑2	·	+	×	+	+	↑2	+	−	↑2	↑6	×	+	↑58	×	↑4	+
GLM 4.5 Air	×	·	↑18	+	×	+	↑2	+	+	+	+	↑2	+	−	−	×	×	+	↑32	+	×	+
Qwen 3 Coder	+	·	+	·	+	·	↑2	·	+	+	+	·	+	↑34	+	↑2	×	·	+	×	+	+
Kimi K2	+	·	↑24	+	×	↑4	↑4	+	+	↑39	+	↑1	+	−	+	+	×	×	+	×	↑7	×
Deepseek V3	+	·	+	·	×	+	↑6	+	+	+	+	↑8	+	×	+	+	×	×	×	↑14	↑13	×

decision point with more relevant information, but because its confidence in other decision points improves, it incorrectly discards its original low-confidence predictions — leading to the correct decision point being dropped in its final output. We provide further discussion of malware characteristics that pose challenges for the LLM-guided approach in Section VI.

Takeaway 1: LUMEN substantially improves the precision with which LLMs identify and rank the correct decision point, outperforming direct prompting across most samples and even boosting the accuracy of the most advanced reasoning models.

For `Pirpi`, LUMEN improves the rank of the correct decision point for Gemini 2.5 Flash by 60, which exceeds the number of branches the sample executed. This outcome highlights Gemini 2.5 Flash's failure to synthesize execution-relevant context, causing it to suggest that decision points exist in unexecuted code despite our directions. In contrast, LUMEN's context synthesis and output filtering constrain the model's output to relevant, executed code, preventing analysts from wasting time and effort on these errors.

To understand how LUMEN impacts each model, we compare ranking quality using the Mean Reciprocal Rank (MRR). An MRR of 1.0 denotes perfect retrieval, where the correct answer is always first, while lower values reflect later positions on average. For instance, an MRR of 0.33 is equivalent to correct answers being consistently ranked 3rd. Queries with no correct response receive a reciprocal rank of zero.

To isolate the impact of our design decisions, Table III shows model performance with and without multi-layer verification. LUMEN improves MRR across all models. Gemini 2.5 Pro, for example, increases from a baseline MRR of 0.40—ranking the correct decision point between 2nd and 3rd on average—to an MRR of 0.84 with LUMEN, placing the correct decision point first in nearly all cases. With LUMEN, every model except GPT 4o mini returns the correct decision point within the top two ranks on average, and even GPT 4o mini surpasses the best-performing baseline model.

Comparing full models to their "lightweight" variants (Gemini 2.5 Flash, GPT-4o Mini, GLM-4.5 Air) reveals that stronger models benefit from LUMEN's context curation alone, whereas verification is essential for lightweight models. For stronger models (GPT-5, Gemini 2.5 Pro), verification has little effect, which we conjecture is due to better prompt adherence and reasoning capabilities. Nevertheless, these models still gain substantially from the curated synthesized context.

Takeaway 2: State-of-the-art models, including those with advanced reasoning capabilities, benefit substantially from LUMEN's curated context. Lighter-weight or older models can achieve similar gains, but require output verification to ensure reliability.

Another key benefit of LUMEN is cost efficiency. To compare open- and closed-weight models consistently, we sourced usage costs from OpenRouter. LUMEN's filtering of unexecuted code reduces API costs by an order of magnitude — reducing the per-sample cost of the top-performing Gemini 2.5 Pro from $2.23 to $0.17. These reductions are especially valuable in large-scale malware triage. LUMEN even enables the open-weight Qwen3 Coder model (MRR 0.70) to outperform the state-of-the-art lightweight model Gemini 2.5 Flash (MRR 0.68), while operating at 25% lower cost.

TABLE III: MRR for each model with $n = 5$ outputs per sample, followed by its reciprocal representing the average rank of the correct decision point. Higher MRR is better. Results from the best performing closed- and open-weight models are highlighted. p-values (Wilcoxon signed-rank, Pratt method) give the probability of observing improvements to the reciprocal rank at least this large under the null hypothesis that LUMEN did not improve upon baseline ranking.

Model	Baseline	LUMEN No Verification	LUMEN	p-value
GPT 5	0.43 (2.3)	0.76 (1.3)	0.76 (1.3)	0.004
Gemini 2.5 Pro	0.40 (2.5)	0.84 (1.2)	0.84 (1.2)	<0.001
Gemini 2.5 Flash	0.29 (3.4)	0.42 (2.4)	0.68 (1.5)	<0.001
Claude Sonnet 4	0.23 (4.4)	0.56 (1.8)	0.76 (1.3)	<0.001
Claude Sonnet 3.7	0.21 (4.8)	0.40 (2.5)	0.59 (1.7)	<0.001
GPT 4o Mini	0.10 (10.0)	0.23 (4.4)	0.46 (2.2)	<0.001
GPT 4o	0.05 (20.0)	0.44 (2.3)	0.61 (1.6)	<0.001
GLM 4.5	0.22 (4.5)	0.54 (1.9)	0.64 (1.6)	<0.001
GLM 4.5 Air	0.09 (11.1)	0.17 (5.9)	0.61 (1.6)	<0.001
Qwen 3 Coder	0.26 (3.8)	0.53 (1.9)	0.70 (1.4)	<0.001
Kimi K2	0.10 (10.0)	0.52 (1.9)	0.62 (1.6)	<0.001
Deepseek V3	0.09 (11.1)	0.39 (2.6)	0.57 (1.8)	<0.001

Takeaway 3: LUMEN reduces token usage by 90% across all evaluated models, substantially improving scalability and lowering operational costs. This efficiency enables practical deployment with cheaper or open-weight models, which still deliver strong performance and provide viable alternatives for cost- or privacy-constrained deployments.

D. RQ2: Exposing Malicious Behaviors with LUMEN

Analysts aim to understand a malware sample's malicious behaviors and associated Tactics, Techniques, and Procedures (TTPs) [4]. We therefore evaluate how effectively LUMEN uses its ranked decision points to construct activation methods that expose previously hidden behavior. For this evaluation, we form an LLM ensemble from the three top-performing models by MRR (Table III): Gemini 2.5 Pro, GPT-5, and Claude Sonnet 4. For simplicity, each model proposes a single activation method, and then any fully automated methods are independently executed in the dynamic analysis engine to test whether they reveal additional behaviors. If no new behaviors are revealed through these executions, the first author plays the role of an analyst and follows the steps specified in the detailed instructions provided by the manual activation method(s).

1) Comparison to CAPE Sandbox: For a first evaluation, we apply LUMEN's activation methods to the CAPE sandbox. We reiterate that except for fulfilling manual activation requests, the entire process in Fig. 2 is fully automated. Activation methods are considered effective if they cause the malware to trigger any of CAPE sandbox's high-severity malicious behavior signatures used in Section V-B Table IV presents the results, grouping samples by the purpose of the decision point to give context to the necessary activation method. ***Our results show that LUMEN exposes new high-severity malicious behaviors in 68% (15/22) of samples.*** In 14 of these samples, all except `Deathransom`, the matching signatures correspond to TTPs in the MITRE ATT&CK framework, a common language used by analysts to describe malicious behaviors. Additionally, the best individual model achieves only 11/22 (50%), demonstrating the value of our ensemble approach. Exposing previously hidden malicious behaviors can significantly reduce the manual effort required by malware analysts. We provide time savings estimates based on real-world workload measurements in Appendix A.

TABLE IV: Performance of LUMEN at exposing hidden malicious behaviors in CAPE. Auto indicates if we relied solely on automated bypasses, or we attempted at least one manual activation method. The ATT&CK techniques (displaying the name of first high severity, followed by how many additional ones) linked to new behaviors are listed.

Malware Family	Auto	Exposed	Exposed Techniques of High-Severity Malicious Behaviors
Anti-Analysis Technique			
`Deathransom`	○	●	
`Indigodrop`	○	○	
API Hook Error			
`Obliquerat`	○	●	T1547: Logon Autostart Execution
Command Line Arguments			
`Babymetal`	●	●	T1070: Indicator Removal: File Deletion
`Pirpi`	●	○	
`Remexi`	●	○	
`Typeframe`	○	●	T1547: Logon Autostart Execution
`Multigrain`	●	●	T1003: OS Credential Dumping + 1
`Plead`	●	●	T1539: Steal Web Session Cookie
`Vivaciousgift`	●	○	
`Balkanbackdoor`	●	●	T1547: Logon Autostart Execution
`Freenki`	●	●	T1547: Logon Autostart Execution
`Electricfish`	○	○	
`Mamba`	●	●	T1543: Create System Process + 4
`Shamoon`	●	●	T1547: Logon Autostart Execution
Check for Target Environment			
`Avatar`	●	●	T1202: Indirect Command Execution
`Skimer`	●	●	T1055: Process Injection + 1
`Varenyky`	●	●	T1564: Hide Artifacts + 3
`Cerber`	●	●	T1486: Data Encrypted for Impact + 9
Missing Dependency			
`Triton`	○	○	
`Kghspy`	○	●	T1041: Exfiltration Over C2 Channel + 1
`Artfulpie`	●	○	

While each model in our ensemble successfully applied every automated method at least once, the models differed in their ability to determine when manual activation was appropriate. Rather than requesting manual activation, Claude Sonnet 4 disproportionately relied on dynamic instrumentation to force execution, applying this method four times whereas the other two models did so only once. While forcing execution beyond a decision point guarantees short-term progress, applying an activation method that ignores the root cause, such as a missing dependency, can cause execution to fail shortly afterwards. On the other hand, Gemini 2.5 Pro and GPT 5 proposed manual methods that enabled activation of four samples (25% of successful activations), shown in Table IV. This indicates the value in including models in the

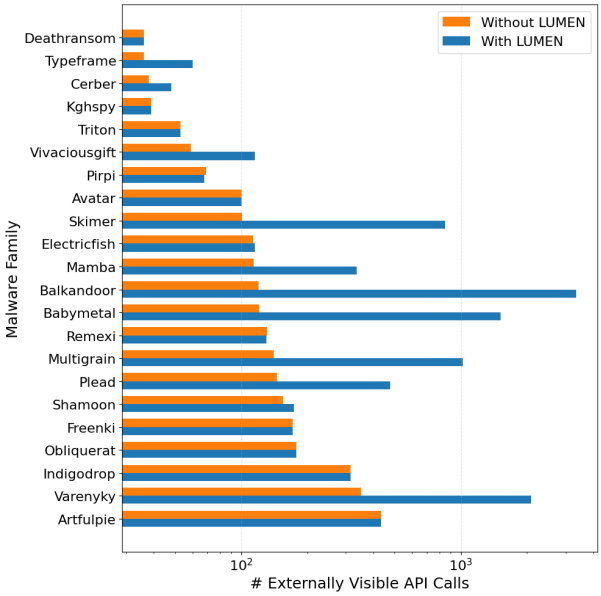

Fig. 3: Counts of APIs with externally visible effects when run with JuanLesPIN, with and without LUMEN.

ensemble that recognize when activation is outside the scope of automated methods and request manual intervention instead.

2) Comparison to JuanLesPIN: For a second evaluation, we apply our activation methods to the dynamic analysis system by Maffia et. al. [10] designed to extract runtime information from malware [20]. That system, *JuanLesPIN*, relies on Intel Pin, a dynamic binary instrumentation framework, to neutralize known evasion techniques and maximize observed behaviors. However, Pin's use of JIT translation for dynamic instrumentation/monitoring conflicts with Capemon's approach of injecting code into the malware's main process to intercept API calls and exceptions. Thus, we cannot apply our CAPE-specific dynamic instrumentation activation method to samples executed with *JuanLesPIN*. Likewise, we cannot use CAPE's malicious behavior signatures with this tool's output.

Instead, we adopt Maffia et al.'s metric for exposed malicious behavior: the number of API calls with externally visible effects. Fig. 3 shows that LUMEN's output significantly enhances *JuanLesPIN*. On average, applying the most effective activation method from our ensemble reveals over four times as many such API calls as with JuanLesPIN alone.

Takeaway 4: LUMEN's structured reasoning, use of multiple LLMs, and support for diverse activation strategies create a flexible and effective approach for strengthening existing dynamic analysis systems. By automating the diagnosis of root causes and the construction of activation methods, LUMEN can significantly reduce the burdens that analysts face when trying to uncover hidden behaviors in malware.

VI. Where LLM-Guided Workflows Fall Short –Lessons Learned and Opportunities

To understand where LLM-based analysis breaks down in practice, we examine how models fail at both identifying decision points and constructing effective activation methods.

1) Localization: We begin by revisiting the samples in Table II where models failed to correctly localize the relevant decision point. A recurring pattern is that LLMs seize on features that appear plausible in isolation but fail to verify them against the actual execution trace. For instance, in the `Deathransom` case, multiple models incorrectly treat certain API calls as indicators of evasion, even though these calls receive null inputs and their outputs are ignored. Because the models do not fully utilize evidence from the executed code, they overlook the real issue: these APIs are invoked repeatedly to stall execution until the analysis environment times out.

LLMs also struggle with cases that violate common malware-analysis expectations. A representative example is `Obliquerat`, where the hidden behavior is not intentionally concealed but instead obscured by a faulty sandbox API hook. The incorrect hook changes the error code returned to the malware, causing premature termination. Most models produce speculative explanations that do not account for this atypical scenario, leading them away from the true source of the failure.

While older or smaller models cannot consistently overcome these weaknesses, more advanced models (e.g., GPT-5 and Gemini 2.5 Pro) reason more accurately when provided LUMEN's structured context. With `Obliquerat`, Gemini correctly identifies the faulty API hook that likely causes the observed behavior in the execution trace. For `Deathransom`, both advanced reasoning models identify the API-based stalling technique. These examples illustrate how structured, execution-grounded context benefits even sophisticated LLMs.

Even with LUMEN's structured guidance, advanced models still struggle when the true decision point is far removed from the dormant malicious code it ultimately controls. Tracing this relationship requires reasoning across long, interdependent chains of logic. At present, this task remains difficult for LLMs, particularly when important clues appear late in execution or when LUMEN's filtering unintentionally removes supporting context. The `Triton` sample illustrates this challenge. Multiple values can satisfy an external dependency, but only one leads to malicious behavior, and the evidence for that link emerges much later in the trace. Although several models correctly identified the relevant decision point, none ranked it as the most important. The challenges that advanced models face for this sample further emphasize the value of carefully selected context to enable long-range and reliable analysis.

2) Activation: The primary reason why LUMEN failed to expose severe malicious behaviors in the samples where a correct decision point is localized is because of sequential decision points. Although our activation methods can address more than one decision point at once, such as `babymetal` which requires three specific command line arguments, activation methods struggle to address multiple decision points that

require independent changes to the environment. For example, `Indigodrop` employs multiple checks to evade analysis systems, however each requires an independent change to the sandbox leading the activation method to only pass a single one. Further research is needed on designing efficient activation methods to handle multiple decision points.

More substantial challenges arise with samples that rely on missing files or unreachable C2 servers, such as `Remexi`, `Electricfish`, `Triton`, and `ArtfulPie`. These dependencies fall outside the scope of LUMEN's current automation, since the system cannot reconstruct unavailable resources or emulate remote infrastructure. In principle, heavier-weight symbolic or dependency-reconstruction frameworks [51] could assist in these cases, and LUMEN's modular design allows such techniques to be incorporated as additional activation methods. In practice, however, manual activation may be more effective for certain scenarios. Analysts working in incident-response settings often have access to missing artifacts or captured network traffic that can directly satisfy the malware's requirements. In the case of `ArtfulPie`, the manual activation instructions pointed to a single missing dependency, allowing us to acquire this file, include it in the sandbox, and uncover its malicious behaviors. Our findings suggest that future systems should integrate automated strategies with mechanisms that let analysts seamlessly supply real-world dependencies, allowing the approaches to complement each other in uncovering hidden behaviors.

VII. Other Related Work

One common approach is to identify hidden behaviors by comparing malware behavior across execution environments [52]–[56]. These methods assume that malware successfully executes in at least one of the included environments, but in practice, many malware use evasion techniques to suppress behaviors when they detect analysis systems. While dynamic analysis systems incorporate dedicated mechanisms to neutralize known evasion techniques and enable execution of evasive malware samples [10]–[12], they must be continually updated as new evasion techniques emerge, and they still struggle with malware that targets specific environments. For example, ATM malware may terminate when expected ATM-specific file paths are absent.

To address these "targeted" malware samples, symbolic execution [57] and forced execution [58]–[61] explore potential execution paths to expose hidden behaviors, even if the target environment is unknown. Unfortunately, these approaches have limited practicality due to the well-known "path explosion" problem. While alternative approaches [62], [63] have shown promise in reducing the cost of path exploration, they are constrained to exploring behaviors that stem from environmental queries with enumerable outcomes [62] or predefined mutation strategies [63]. Rather than exploring all possible execution paths, we use LLMs to prioritize behavior-hiding decision points to guide deeper investigation.

LLMs already show promise in assisting malware analysts by improving summarization [5], [7], [9] and decompilation [33], [64] of complex code. Wang et al. [8] take a first step in using LLMs to detect basic blocks of assembly code that might contain API calls indicative of evasive behavior. That work, however, does not take advantage of execution context, and as such, is limited in its ability to uncover hidden behaviors relevant to the execution environment.

Our Work: In contrast, we provide analysts with a practical systems pipeline that automatically locates and activates hidden behaviors, or generates detailed, tool-grounded instructions for manually exposing them. Although prior work in related domains has identified useful data sources for their respective tasks [15], [65]–[67], these approaches do not integrate inputs from multiple analysis tools. In this paper, we show how to systematically select, condense, and combine context from the tools analysts already rely on (such as disassemblers and sandboxes), enabling LLMs to more effectively support end-to-end malware analysis workflows.

VIII. Open Science

To support future work, code for LUMEN can be accessed from [68]. We believe LUMEN's ability to aid analysts and strengthen defenses outweighs the potential for misuse.

IX. Conclusion

This work examines how domain-specific systems adaptations can overcome limitations of LLMs in uncovering malicious behaviors concealed from modern dynamic analysis environments. Building on established practices in academic and industrial malware analysis, we develop a multi-stage system that identifies the decision points responsible for hiding behaviors and constructs activation methods to expose those behaviors, either automatically or with structured analyst guidance. Using a curated dataset of malware samples with analyst-confirmed hidden behaviors, we show that our design improves the precision of decision-point prioritization, reduces the volume of code requiring manual review, and significantly lowers model usage costs. Our activation methods also reveal behaviors missed by both the popular CAPE sandbox and the *JuanLesPin* evasive analysis tool. Collectively, these results highlight the value of a practical, systems-oriented pipeline that reduces analyst effort, enhances the capabilities of existing dynamic analysis tools, and improves the scalability of workflows needed to address emerging malware threats.

X. Acknowledgment

We thank the anonymous reviewers and Athanasios Moschos for their constructive feedback, as well as Aaron Faulkenberry and Joseph Reilly for their assistance setting up infrastructure for LUMEN. We also thank the five interviewees (discussed in Section III) who contributed to LUMEN's design as well as those who gave feedback on our early prototype (discussed in Appendix C). These participants, who include Jeff Archer, Elliot Chernofsky, Kyle Cucci, Christopher Gardner, Jeremy Hedges, Coleman Kane, Kevin O'Reilly, Mike Sconzo, Pim Trouerbach, and Roman Vasilenko, were each given a copy of the final paper. All listed participants provided explicit consent to be named in the acknowledgments.

REFERENCES

[1] U. Group. (2025) Unitedhealth group reports 2024 results. https://www.unitedhealthgroup.com/content/dam/UHG/PDF/investors/2024/2025-16-01-uhg-reports-fourth-quarter-results.pdf.

[2] Z. Whittaker. (2024) Unitedhealth says change healthcare hack affects over 100 million, the largest-ever us healthcare data breach. https://techcrunch.com/2024/10/24/unitedhealth-change-healthcare-hacked-millions-health-records-ransomware.

[3] X. Ugarte-Pedrero, M. Graziano, and D. Balzarotti, "A close look at a daily dataset of malware samples," *ACM Transactions on Privacy and Security (TOPS)*, 2019.

[4] M. Yong Wong, M. Landen, M. Antonakakis, D. M. Blough, E. M. Redmiles, and M. Ahamad, "An inside look into the practice of malware analysis," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS'21)*, 2021.

[5] H. Pearce, B. Tan, P. Krishnamurthy, F. Khorrami, R. Karri, and B. Dolan-Gavitt, "Pop quiz! can a large language model help with reverse engineering?" *arXiv preprint arXiv:2202.01142*, 2022.

[6] X. Shang, G. Chen, S. Cheng, B. Wu, L. Hu, G. Li, W. Zhang, and N. Yu, "Binmetric: A comprehensive binary code analysis benchmark for large language models," in *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence (IJCAI'25)*, 2025.

[7] C. Fang, N. Miao, S. Srivastav, J. Liu, R. Zhang, R. Fang, Asmita, R. Tsang, N. Nazari, H. Wang, and H. Homayoun, "Large language models for code analysis: Do LLMs really do their job?" in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[8] H. Wang, N. Luo, and P. Liu, "Unmasking the shadows: Pinpoint the implementations of anti-dynamic analysis techniques in malware using llm," *arXiv preprint arXiv:2411.05982*, 2024.

[9] S. Fujii and R. Yamagishi, "Feasibility study for supporting static malware analysis using llm," in *Computer Security. ESORICS 2024 International Workshops*, 2024.

[10] L. Maffia, D. Nisi, P. Kotzias, G. Lagorio, S. Aonzo, and D. Balzarotti, "Longitudinal study of the prevalence of malware evasive techniques," *arXiv preprint arXiv:2112.11289*, 2021.

[11] A. Mills and P. Legg, "Investigating anti-evasion malware triggers using automated sandbox reconfiguration techniques," *Journal of Cybersecurity and Privacy*, 2020.

[12] D. C. D'Elia, E. Coppa, F. Palmaro, and L. Cavallaro, "On the dissection of evasive malware," *IEEE Transactions on Information Forensics and Security*, 2020.

[13] X. Jin, J. Larson, W. Yang, and Z. Lin, "Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models," *arXiv preprint arXiv:2312.09601*, 2023.

[14] S. B. Hossain, N. Jiang, Q. Zhou, X. Li, W.-H. Chiang, Y. Lyu, H. Nguyen, and O. Tripp, "A deep dive into large language models for automated bug localization and repair," in *Proceedings of the ACM on Software Engineering*, 2024.

[15] P. Liu, J. Liu, L. Fu, K. Lu, Y. Xia, X. Zhang, W. Chen, H. Weng, S. Ji, and W. Wang, "Exploring ChatGPT's capabilities on vulnerability management," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[16] S. I. Mirzadeh, K. Alizadeh, H. Shahrokhi, O. Tuzel, S. Bengio, and M. Farajtabar, "Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models," in *The Thirteenth International Conference on Learning Representations*, 2025.

[17] B. Cheng, E. A. Leal, H. Zhang, and J. Ming, "On the feasibility of malware unpacking via hardware-assisted loop profiling," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[18] I. Tsingenopoulos, J. Cortellazzi, B. Bošanskỳ, S. Aonzo, D. Preuveneers, W. Joosen, F. Pierazzi, and L. Cavallaro, "How to train your antivirus: Rl-based hardening through the problem space," in *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, 2024.

[19] Y. Elhamer, W. Ballenthin, M. Raabe, and M. Hunhoff. (2024) Dynamic capa: Exploring executable run-time behavior with the cape sandbox. https://cloud.google.com/blog/topics/threat-intelligence/dynamic-capa-executable-behavior-cape-sandbox.

[20] S. Dambra, Y. Han, S. Aonzo, P. Kotzias, A. Vitale, J. Caballero, D. Balzarotti, and L. Bilge, "Decoding the secrets of machine learning in malware classification: A deep dive into datasets, feature extraction, and model performance," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[21] S. Aonzo, Y. Han, A. Mantovani, and D. Balzarotti, "Humans vs. machines in malware classification," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[22] T. Li, G. Zhang, Q. D. Do, X. Yue, and W. Chen, "Long-context LLMs struggle with long in-context learning," *Transactions on Machine Learning Research*, 2025.

[23] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, 2024.

[24] A. Cherepanov. (2018) Certificates stolen from taiwanese tech-companies misused in plead malware campaign. https://www.welivesecurity.com/2018/07/09/certificates-stolen-taiwanese-tech-companies-plead-malware-campaign.

[25] J. Chen, Z. Pan, X. Hu, Z. Li, G. Li, and X. Xia, "Reasoning runtime behavior of a program with llm: How far are we?" in *IEEE/ACM 47th International Conference on Software Engineering*, 2025.

[26] C. Sypherd and V. Belle, "Practical Considerations for Agentic LLM Systems," *arXiv preprint arXiv:2412.04093*, 2024.

[27] S. Glazunov and M. Brand. (2024) Project naptime: Evaluating offensive security capabilities of large language models. https://googleprojectzero.blogspot.com/2024/06/project-naptime.html.

[28] Anthropic. (2024) Building effective agents. https://www.anthropic.com/engineering/building-effective-agents.

[29] OpenAI. (2024) Optimizing llm accuracy. https://platform.openai.com/docs/guides/optimizing-llm-accuracy.

[30] J. D. Weisz, J. He, M. Muller, G. Hoefer, R. Miles, and W. Geyer, "Design principles for generative AI applications," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024.

[31] Microsoft. (2024) Prompt engineering techniques. https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/prompt-engineering.

[32] G. Deng, Y. Liu, V. Mayoral-Vilches, P. Liu, Y. Li, Y. Xu, T. Zhang, Y. Liu, M. Pinzger, and S. Rass, "PentestGPT: Evaluating and harnessing large language models for automated penetration testing," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[33] P. Hu, R. Liang, and K. Chen, "Degpt: Optimizing decompiler output with LLM," in *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2024.

[34] M. Y. Wong, M. Landen, F. Li, F. Monrose, and M. Ahamad, "Comparing malware evasion theory with practice: results from interviews with expert analysts," in *Twentieth Symposium on Usable Privacy and Security (SOUPS 2024)*, 2024.

[35] W. Liu, X. Liu, Z. Li, B. Liu, R. Yu, and L. Wang, "Retrofitting lbr profiling to enhance virtual machine introspection," *IEEE Transactions on Information Forensics and Security*, 2022.

[36] A. Kong, S. Zhao, H. Chen, Q. Li, Y. Qin, R. Sun, X. Zhou, E. Wang, and X. Dong, "Better zero-shot reasoning with role-play prompting," in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2024.

[37] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *Advances in neural information processing systems*, 2022.

[38] Cape: Malware configuration and payload extraction. https://github.com/kevoreilly/CAPEv2.

[39] A. Küchler, A. Mantovani, Y. Han, L. Bilge, and D. Balzarotti, "Does every second count? time-based evolution of malware behavior in sandboxes," in *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2021.

[40] capemon: The monitor dll for cape: Config and payload extraction. https://github.com/kevoreilly/capemon.

[41] R. J. Joyce, D. Amlani, C. Nicholas, and E. Raff, "Motif: A malware reference dataset with ground truth family labels," *Computers & Security*, 2023.

[42] J. Li, Y. Zhang, Y. Huang, and K. Leach, "Malmixer: Few-shot malware classification with retrieval-augmented semi-supervised learning," in *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*, 2025.

[43] X. Wu, W. Guo, J. Yan, B. Coskun, and X. Xing, "From grim reality to practical solution: Malware classification in real-world noise," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[44] R. J. Joyce, T. Patel, C. Nicholas, and E. Raff, "Avscan2vec: Feature learning on antivirus scan data for production-scale malware corpora," in *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, 2023.

[45] A. Vitale, S. Aonzo, S. Dambra, N. Rani, L. Ippolito, P. Kotzias, J. Caballero, and D. Balzarotti, "The polymorphism maze: Understanding diversities and similarities in malware families," in *European Symposium on Research in Computer Security*, 2025.

[46] "Advanced malware sandbox: In-depth malware & phishing threat analysis," https://www.vmray.com.

[47] VMRay. (2025) Customer success stories. https://www.vmray.com/customer-success-stories.

[48] L. Pirch, A. Warnecke, C. Wressnegger, and K. Rieck, "Tagvet: Vetting malware tags using explainable machine learning," in *Proceedings of the 14th European Workshop on Systems Security*, 2021.

[49] "Swe-bench leaderboards," https://www.swebench.com, 2025.

[50] N. Chowdhury, J. Aung, C. J. Shern, O. Jaffe, D. Sherburn, G. Starace, E. Mays, R. Dias, M. Aljubeh, M. Glaese, C. E. Jimenez, J. Yang, L. Ho, T. Patwardhan, K. Liu, and A. Madry. (2024) Introducing swe-bench verified. https://openai.com/index/introducing-swe-bench-verified.

[51] L. Borzacchiello, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Reconstructing c2 servers for remote access trojans with symbolic execution," in *Cyber Security Cryptography and Machine Learning*, 2019.

[52] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song, "Differential Slicing: Identifying Causal Execution Differences for Security Applications," in *IEEE Symposium on Security and Privacy (SP)*, 2011.

[53] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti, "Detecting Environment-Sensitive Malware," in *Recent Advances in Intrusion Detection*, 2011.

[54] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna, "Efficient Detection of Split Personalities in Malware," in *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2010.

[55] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song, "Emulating emulation-resistant malware," in *Proceedings of the 1st ACM workshop on Virtual machine security*, 2009.

[56] D. Kirat and G. Vigna, "MalGene: Automatic Extraction of Malware Analysis Evasion Signature," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[57] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," *Botnet Detection: Countering the Largest Security Threat*, 2008.

[58] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," in *IEEE Symposium on Security and Privacy*, 2007.

[59] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[60] W. You, Z. Zhang, Y. Kwon, Y. Aafer, F. Peng, Y. Shi, C. Harmon, and X. Zhang, "Pmp: Cost-effective forced execution with probabilistic memory pre-planning," in *IEEE Symposium on Security and Privacy (SP)*, 2020.

[61] D. He, D. Xie, Y. Wang, W. You, B. Liang, J. Huang, W. Shi, Z. Zhang, and X. Zhang, "Define-use guided path exploration for better forced execution," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024.

[62] Z. Xu, J. Zhang, G. Gu, and Z. Lin, "GoldenEye: Efficiently and Effectively Unveiling Malware's Targeted Environment," in *Research in Attacks, Intrusions and Defenses*, 2014.

[63] F. Gorter, C. Giuffrida, and E. Van Der Kouwe, "Enviral: Fuzzing the Environment for Evasive Malware Analysis," in *Proceedings of the 16th European Workshop on System Security*, 2023.

[64] H. Tan, Q. Luo, J. Li, and Y. Zhang, "Llm4decompile: Decompiling binary code with large language models," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2024.

[65] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, "LLMs cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks," in *IEEE Symposium on Security and Privacy (SP)*, 2024.

[66] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *IEEE Symposium on Security and Privacy (SP)*, 2023.

[67] D. Xie, Z. Zhang, N. Jiang, X. Xu, L. Tan, and X. Zhang, "Resym: Harnessing LLMs to recover variable and data structure symbols from stripped binaries," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2024.

[68] "Open science artifacts," https://github.com/kvalakuzhy-gatech/lumen-sandbox.

[69] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016.

APPENDIX

A. End-to-End Malware Analysis Time Reduction

To quantify how our approach reduces analyst workload, we estimate time savings using measurements from Ugarte-Pedrero et al. [3] on a real-world malware dataset. The paper distinguishes three analysis outcomes. Malware samples first go through an initial round of automated filtering to determine whether manual analysis is required. For samples that require manual analysis, the study tracks whether professional analysts conduct a quick cursory analysis, averaging 2.25 minutes, or whether a more in-depth analysis is required, estimated to take at least an hour on average. When sufficient executed behavior is evident, cursory review suffices in over 95% of cases.

However, the authors estimate that around 2000 samples per day require in-depth malware analysis, largely due to the lack of observed malicious behaviors. Under the assumption that LUMEN sustains its current automated activation success rate (i.e., 50% per Table IV), the additional behavioral coverage converts many in-depth analyses into cursory reviews. The resulting reduction in manual analysis corresponds to approximately 108 workdays of time savings. This estimate does not include time savings from improved automated classification due to the expanded behavioral coverage, which would further reduce the number of cases requiring even cursory analysis.

B. Software LBR Runtime Performance

To contextualize Software LBR overhead, we compare execution times of our malware samples under multiple instruction-level tracing approaches, including JuanLesPIN (Fig. 4), using Windows 10 virtual machines from prior evaluations. When measuring JuanLesPIN's runtime, we disable sandbox API monitoring to exclude unrelated sandbox hooking overhead. For each run, we define execution time as the duration between the first and last recorded API timestamp in the API trace collected by the corresponding approach.

We first evaluate the impact of breakpoint deactivation on Software LBR. Incorporating breakpoint deactivation decreases execution timeouts from 75% of samples to just 5% under the two minute sandbox time limit recommended by Kuchler et al. [39]. Even with a generous 300-second limit, breakpoint deactivation reduces timeouts from four samples to just one. Completing execution within the time limit ensures the decision point is reached, allowing the collection of relevant dynamic context for activating behaviors.

We also compare the performance of Software LBR's basic block coverage with instruction-level tracing in *JuanLesPIN*. For PIN-based instrumentation, the code for recording executed instructions is inserted during the "Just-in-time compilation" conducted by PIN, which attempts to apply minimal modifications to the malware's assembly code to ensure the PIN tool can recover control after executing contiguous regions of code. These inserted instruction recording hooks are executed before the "compiled" version of each instruction to record the instruction's virtual address. While this instruction tracing collects more than just executed basic blocks, it incurs a large performance penalty. Software LBR offers efficient collection of the information used by LUMEN, achieving an average runtime (66 seconds) similar to the performance of JuanLesPIN without any tracing enabled (61 seconds).

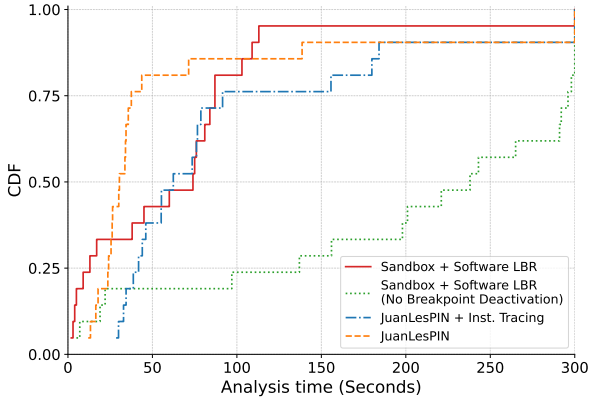

Fig. 4: Runtime for collecting APIs and instructions.

C. Malware Analyst Feedback

We conducted a qualitative study on an early-stage prototype of LUMEN with eight senior malware analysts (avg. ∼11 years of experience) from six organizations. Following an IRB-approved protocol, participants inspected output from the prototype and predicted how its core components—a high-level summary, automated bypass, and annotated control-flow graph—could reduce manual effort when analyzing evasive malware. Across 15 evaluation instances, we collected both quantitative ratings and open-ended feedback, which guided iterative refinements of the system's interface and outputs.

Although the feedback from analysts was based on an earlier GPT-4o-based prototype, the results were quite encouraging: Fig. 5 shows that 8/15 instances contained at least one component that was rated "very helpful", particularly in guiding attention to relevant code regions and accelerating follow-up analysis. This positive reception was especially noteworthy given participants' extensive experience and prior evidence that expert analysts are often skeptical of automated tools [69].

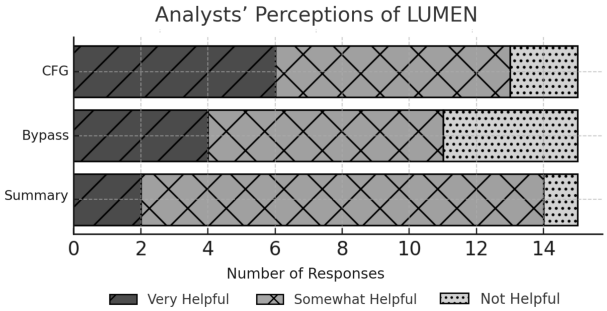

Fig. 5: Participant's perception of early prototype