Attack Effect Model based Malicious Behavior Detection

Limin Wang, Lei Bu(✉), Muzimiao Zhang, Shihong Cang, Kai Ye
State Key Laboratory of Novel Software Techniques, Nanjing University, Nanjing, Jiangsu 210023, China
Email: bulei@nju.edu.cn,

***Abstract*—Traditional security detection methods struggle to keep pace with the rapidly evolving landscape of cyber threats targeting critical infrastructure and sensitive data. These approaches suffer from three critical limitations: non-security-oriented system activity data collection that fails to capture crucial security events, growing security monitoring demands that lead to continuously expanding monitoring systems, thereby causing excessive resource consumption, and inadequate detection algorithms that result in the inability to accurately distinguish between malicious and benign activities, resulting in high false positive rates.**

To address these challenges, we present FEAD , an attack detection framework that improves detection by focusing on identifying and supplementing security-critical monitoring items and deploying them efficiently during data collection, as well as the locality of potential anomalous entities and their surrounding neighbors during anomaly analysis. *FEAD* incorporates three key innovations: (1) an attack model-driven approach that extracts security-critical monitoring items from online attack reports, enabling a more comprehensive monitoring items framework; (2) an efficient task decomposition mechanism that optimally distributes monitoring tasks across existing collectors, maximizing the utilization of available monitoring resources while minimizing additional monitoring overhead; (3) a locality-aware anomaly analysis technique that exploits the characteristic of malicious activities forming dense clusters in provenance graphs during active attack phases, guiding a vertex-level weight mechanism in our detection algorithm to better distinguish between anomalous and benign vertices, thereby improving detection accuracy and reducing false positives.

Evaluations show *FEAD* outperforms existing solutions with an 8.23% higher F1-score and 5.4% overhead. Our ablation study also confirms that *FEAD* 's focus-based designs significantly boost detection performance.

I. Introduction

Modern computing systems across all scales are increasingly targeted by advanced cyber threats such as APT attacks [1], posing significant security challenges. Traditional security measures often fail to address these evolving threats, prompting academia and industry to focus on lightweight attack detection systems. These systems typically utilize runtime log auditing and analysis (e.g., syscall logs)[2], [3] to achieve effective security monitoring and attack detection. Among these, anomaly-based detection approaches[4] have emerged as a promising solution by analyzing historical logs to identify deviations from established benign patterns [3], [5], [6], thus effectively addressing the challenge of detecting novel and evolving attack variants. While these approaches show promise, they face three critical challenges:

❶ Lack of Security-Oriented Efficient Monitoring. Existing monitoring tools (e.g., auditd [7], [8], [9]) are primarily designed for performance or fault diagnosis rather than security. They focus on system failures and resource usage, offering limited visibility into security-relevant activities. Although efforts like eAudit [10] try to improve auditd by using eBPF and incorporating capabilities from existing monitors (e.g., Trace [11]), they still lack the capability to capture higher-level security events. Enterprise tools such as auditbeat [12] provide some enhancements but heavily rely on expert tuning. The absence of a systematic framework for identifying critical security monitoring points leads to incomplete coverage and security blind spots.

❷ High Deployment Costs of Security Monitoring Tasks. Evolving threats demand frequent updates to monitoring strategies, often requiring new monitoring modules for each new security monitoring requirement and increasing system complexity. Moreover, no single tool covers all needs, forcing organizations to deploy multiple systems, resulting in redundancy and performance degradation.

❸ High False Positive Rates in Anomaly-Based Attack Detection. Current detection systems often struggle to accurately differentiate between legitimate system changes and genuine security threats, as benign and malicious nodes frequently exhibit similar behavioral patterns in monitoring data. For instance, routine updates or maintenance activities that alter network traffic may closely resemble attack signatures, resulting in false alarms. Monitoring blind spots further exacerbate this issue by limiting visibility and degrading the system's ability to distinguish normal variations from actual threats. This poor discrimination leads to alert fatigue among security teams and reduces the practical utility of security monitoring in enterprise environments.

To address these challenges, we present *FEAD* (Focus-Enhanced Attack Detection), a framework that improves detection by focusing on two key aspects: identifying and supplementing security-critical monitoring items, and then deploying them efficiently during data collection, as well as analyzing the locality of potential anomalous entities and their surrounding neighbors during anomaly detection. This dual focus ensures targeted, effective attack detection while minimizing system

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.[23|24]xxxx
www.ndss-symposium.org

overhead. Specifically, **For challenge 1**, *FEAD* uses an *Attack Effect Model* and large language models to analyze attack reports, breaking down attack steps and assessing their impact on systems and software to identify key monitoring items. This approach ensures comprehensive monitoring by systematically extracting monitoring requirements from real-world attacks. **For challenge 2**, *FEAD* introduces a novel task decomposition mechanism that breaks down complex tasks and distributes them across existing collectors, thereby collaboratively achieving target monitoring objectives. This maximizes the use of built-in monitoring capabilities while minimizing new module deployment, reducing system overhead. **For challenge 3**, *FEAD* exploits attack locality in provenance graphs, where malicious activities form dense clusters during active phases and sparse ones across other phases. This insight guides a vertex-level weighting mechanism in the detection algorithm, focusing on anomalous vertices and their neighbors. Unlike previous methods that analyze all vertices indiscriminately, *FEAD* 's approach enhances discrimination between benign and malicious behaviors, targets high-risk areas, and improves accuracy while reducing false positives.

Evaluation demonstrates *FEAD* 's superior performance, achieving an average 8.23% higher F1-score compared to existing solutions, with a low overhead of 5.4%. Our ablation study further validates the effectiveness of our focus-based design, showing that the integration of a security-oriented monitoring framework improves the F1-score by over 12.63%, while leveraging attack locality patterns achieves a 9.52% improvement. These results highlight *FEAD* 's ability to enhance detection accuracy while maintaining system efficiency, making it a promising solution for real-world attack detection.

We summarize our core contributions as follows:

- Propose an attack detection framework that integrates security-oriented monitoring and locality-aware anomaly analysis for enhanced detection accuracy.
- Develop an efficient task decomposition mechanism to optimize monitoring coverage while minimizing overhead, achieving comprehensive security monitoring with 5.4% overhead.
- Evaluate *FEAD* showing a 8.23% improvement in F1-score over existing solutions, with ablation studies confirming significant gains from monitoring enhancements (12.63%) and locality-based detection (9.52%).

II. Background and Related Work

A. Preliminaries

Attack Effect Model. An *Attack Effect Model* is defined as an ordered sequence of attack steps $T = [(T_1, E_1), (T_2, E_2), ..., (T_n, E_n)]$, where each T_i represents a triple $\langle actor_i, action_i, target_i \rangle$ describing who performed what action on which target, and E_i represents the corresponding system impact (e.g., Program Execution, File Modification). Each attack step (T_i, E_i) can be mapped to a set of monitoring items M_i required for detecting the system impact. The steps are ordered chronologically to reflect the temporal progression of the attack.

Provenance Graph. In system security monitoring, provenance graphs model system behaviors by transforming audit logs (e.g., Linux Auditd [7]) into a graph capturing causal relationships between system entities and activities. A provenance graph is a directed graph $G = (V, E)$, where V represents system entities (e.g., processes, files, sockets) and E represents interactions between them. Specifically, $V = \{v | v \in (Process \cup File \cup Socket)\}$, and $E \subseteq V \times V \times T$ captures relationships between entities, where T denotes the type of entity behavior (e.g., read, write, execute). This structure enables detailed tracking of system behaviors, aiding in attack detection, forensic analysis, and attack investigation.

B. Threat Model

Similar to prior research on provenance tracking and threat detection [13], [14], [15], [16], [17], [18], [19], we consider the OS kernel and security monitoring components as part of our trusted computing base (TCB). We assume that the collected provenance data is reliable and has not been tampered with by attackers. Although attackers may attempt to subvert the system or compromise the monitoring components, such subversion activities can be captured in logs before they are compromised. Our focus is on detecting attacks that exploit application vulnerabilities or leverage social engineering techniques to gain unauthorized access to victim systems for data exfiltration or manipulation, rather than hardware-based or side-channel attacks.

We assume that the system is initially in a benign state, with the attack originating from outside the enterprise network. Attackers typically gain initial access through remote network exploitation, compromised credentials, or social engineering, and proceed with multi-stage operations that may include information gathering, exploitation of vulnerable software, payload deployment, privilege escalation, and lateral movement. Our approach focuses on identifying these attack patterns within system behavior data.

C. System Monitoring Tools

We motivate our research with an study of existing system monitoring tools to identify their capabilities and limitations for security monitoring applications.

1) Systematic Tool Collection Using Snowball Technique: System monitoring tools are continuously evolving and scattered across platforms, often lacking unified indexing. To address this, we developed a snowball-based retrieval methodology [20], [21], consisting of two main phases.

This method began with *Linux Auditd* [7] as the initial seed due to its prevalence as Linux's default audit tool. We constructed a structured query template combining ① scenario-related terms (e.g., "Provenance graph", "Causal graph"), ② functionality-related terms (e.g., "Data collection", "Log collection"), and ③ tool names (e.g., "Auditd"). We then executed systematic searches across multiple academic databases (i.e., *Google Scholar*, *IEEE Xplore*, and *ACM Digital Library*) using this template (e.g., *(Provenance graph OR causal graph OR forensic analysis OR investigation) AND (data collection*

TABLE I
SYSTEM MONITORING TOOLS USAGE STATISTICS

Monitoring Tool	Usage Instances	Count	Percentage	Platform
Auditd [7]	[8], [9], [22], [14], [23], [11], [24] [25], [26], [27], [17], [28], [29]	13	28.26%	Linux
CamFlow [30]	[19], [31], [32], [6], [33], [34] [18], [26], [35], [36], [37]	11	23.91%	Linux
ETW [38]	[9], [29], [24], [39], [27], [28] [40], [41], [33]	9	17.39%	Windows
SPADE [42]	[26], [14], [23], [43], [44], [35]	6	13.04%	Linux
Auditbeat [12]	[3], [45]	2	4.35%	Linux
PASSv2 [46]	[47], [48]	2	4.35%	Linux
UBSI [11]	[11]	1	2.17%	Linux
eAudit [10]	[10]	1	2.17%	Linux
sysdig [49]	[50]	1	2.17%	Linux
strace [51]	[52]	1	2.17%	Linux
Total	-	47	100%	-

OR log collection) AND Auditd), and removed irrelevant literature to build a normalized database.

Two co-authors with 3–5 years of relevant experience then extracted monitoring tools from the collected literature. Newly discovered tools were added to the query template (i.e., component ③) and used in subsequent search iterations, thus expanding our search scope to capture literature referencing these newly discovered tools. The process terminated when no new tools appeared in two consecutive rounds or when the search queue was exhausted. This iterative approach effectively addressed the fragmentation and rapid evolution of monitoring tools.

2) Tool Collection Results: Our snowball search technique yielded comprehensive results as shown in Table I. The statistics indicate that in the Linux ecosystem, *Auditd* is the most widely used monitoring tool, appearing in 13 papers (28.26%) and ranking first. *CamFlow* follows closely, used in 11 papers (23.91%). For Windows platforms, *ETW* (Event Tracing for Windows) is the predominant tool, used in 9 papers (17.39%). From our analysis, we classified these monitoring tools into three main categories:

1) **Whole-system Provenance Collection Tools** (*CamFlow*, *SPADE*, *PASSv2*): These focus on system-level data flow and causality tracking. For example, *CamFlow* implements efficient monitoring by integrating Linux Security Modules (LSM) and NetFilter, while *SPADE* utilizes *Linux Auditd* logs to build provenance graphs supporting distributed environments. *PASSv2* is a layered provenance architecture based on Linux 2.6 kernel (circa 2009) that integrates provenance across multiple abstraction layers through a unified Disclosed Provenance API, demonstrating early approaches to cross-layer provenance collection despite being constrained by legacy technology.
2) **Audit Tools** (*Auditd*, *Sysdig*, *ETW*, *Auditbeat*, *eAudit*): These record system behaviors to support security analysis and compliance auditing. *Auditd* is Linux's default audit framework, *Sysdig* uses kernel modules for event capture, *ETW* is Windows' standard audit tool, *Auditbeat* extends *Auditd* with modern features, and *eAudit* combines *Auditd* with *eBPF* technology.
3) **Fine-grained Information Collection Tools** (*UBSI*, *strace*): These focus on precise monitoring of specific system behaviors. *UBSI* provides unit-level behavior monitoring through static analysis, while *strace* records detailed system call-level interactions between processes and the operating system.

TABLE II
SYSTEM CALL AND RELATIONSHIP SCENARIO CLASSIFICATION

Scenario	No.	Monitoring Event	Description	Source
File Operations	1	read	Read file content	Auditd
	2	RL_READ	Read inode	CamFlow
	3	write	Write file content	Auditd
	4	RL_WRITE	Write inode	CamFlow
	5	open	Open file	Auditd
	6	close	Close file	Auditd
Directory Operations	7	creat	Create new empty file	Auditd
	8	unlink	Delete file	Auditd
	9	link	Create hard link	Auditd
	10	linkat	Create relative path hard link	Auditd
	11	unlinkat	Delete file in relative directory	Auditd
	12	rmdir	Remove directory	Auditd
	13	mkdir	Create directory	Auditd
	14	RL_INODE_CREATE	Create inode	CamFlow
Process Operations	15	fork	Create new process	Auditd
	16	clone	Create new process (shared address space)	Auditd
	17	execute	Execute new program	Auditd
	18	RL_CLONE_MEM	Memory copy during cloning	CamFlow
	19	RL_SETUID	Set user ID	CamFlow
	20	RL_SETGID	Set process group ID	CamFlow
	21	kill	Send signal	Auditd
IO Control	22	RL_READ_IOCTL	IO control read operation	CamFlow
	23	RL_WRITE_IOCTL	IO control write operation	CamFlow
	24	pipe	Create pipe	Auditd
	25	fcntl	File control operation	Auditd
Network Operations	26	socket	Create socket	Auditd
	27	RL_SOCKET_CREATE	Create socket	CamFlow
	28	RL_SOCKET_PAIR _CREATE	Create socket pair	CamFlow
	29	connect	Connect to remote host	Auditd
	30	RL_CONNECT	Socket connection operation	CamFlow
	31	RL_BIND	Socket binding operation	CamFlow
	32	RL_LISTEN	Socket listening operation	CamFlow
	33	RL_ACCEPT	Socket accept connection operation	CamFlow
	34	sendto	Send data to specified address	Auditd
	35	recvfrom	Receive data from specified address	Auditd
	36	sendmsg	Send message	Auditd
	37	sendmmsg	Send multiple messages	Auditd
	38	recvmsg	Receive message	Auditd
	39	recvmmsg	Receive multiple messages	Auditd
	40	getpeername	Get remote address of connected socket	Auditd
Memory Operations	41	dup	Duplicate file descriptor	Auditd
	42	dup2	Duplicate file descriptor to specified descriptor	Auditd
	43	RL_MMAP	Memory mapping mount	CamFlow
	44	RL_MMAP_PRIVATE	Private memory mapping mount	CamFlow
	45	RL_SH_READ	Shared memory read operation	CamFlow
	46	RL_PROC_READ	Read process memory	CamFlow
Message Queue	47	mq_open	Open message queue	Auditd
	48	RL_MSG_CREATE	Create message	CamFlow
System Loading	49	RL_LOAD_FILE	Load file to kernel	CamFlow
	50	RL_LOAD_FIRMWARE	Load firmware to kernel	CamFlow
	51	RL_LOAD_MODULE	Load module to kernel	CamFlow
	52	RL_VERSION	Connect entity object version	CamFlow

In practical applications, whole-system provenance collection and audit tools can directly build provenance graphs, while fine-grained information collection tools typically supplement the former by providing detailed parameter information for key nodes in the provenance graph.

3) Security Monitoring Capability Analysis: To assess the monitoring capabilities of these tools, we selected two of the most widely used system monitoring tools and analyzed their monitoring capabilities. Based on the statistical data in Table I, *Auditd* and *CamFlow* are the most widely adopted tools for provenance graph construction. These two tools have

complementary monitoring capabilities: *Auditd* uses system call tracing mechanisms to monitor basic events such as file operations, process behaviors, and network communications, while *CamFlow* leverages the LSM framework to provide more granular tracking of entity relationships, covering memory operations, IO control, and system loading scenarios. We first categorized their monitoring capabilities into eight major dimensions: file operations, directory operations, process control, IO management, network communications, memory operations, message queues, and system loading. Table II shows a detail of the monitoring events covered by these tools. Then, through our analysis, we identified the monitoring focus and limitations of each tool. Based on the comprehensive analysis of the monitoring events presented in Table II, we can draw the following conclusions:

Requirement Gaps: Original design objectives misaligned with security monitoring needs. Our analysis reveals that existing tools demonstrate a significant misalignment with security monitoring requirements. As evident from Table II, both *Auditd* and *CamFlow* were originally designed for general-purpose system monitoring rather than security-specific monitoring. The table shows that neither tool adequately covers critical security-relevant operations such as environment variable manipulations, which are frequently exploited in attacks [53].

Fragmented Monitoring Ecosystem: No unified system provides comprehensive coverage. As illustrated in Table II, no single monitoring system comprehensively implements all necessary monitoring capabilities. For instance, *Auditd* demonstrates weak coverage in the *System Loading* dimension (items 49-52), while *CamFlow* excels in this area. Conversely, *CamFlow* shows incomplete monitoring for *Directory Operations* (items 7-14), which *Auditd* covers extensively. Our deeper investigation found that security-optimized monitoring tools like *Auditbeat* have attempted to address these gaps by augmenting *Auditd* with support for system events (user logins, etc.) and file integrity monitoring [54]. However, these enhancements are primarily guided by expert experience rather than systematic methodology, resulting in inevitable blind spots for common security-relevant operations. For example, despite its security focus, *Auditbeat* still cannot monitor environment variable manipulations. This fragmentation highlights the urgent need for a more systematic approach to security monitoring capability design.

Our analysis of existing monitoring tools reveals critical limitations that demand immediate attention. The requirement gaps and fragmented monitoring ecosystem pose significant challenges for effective security monitoring. If left unaddressed, these issues will lead to persistent monitoring blind spots, resulting in reduced accuracy during attack detection and analysis. Furthermore, as new monitoring requirements emerge, organizations are forced to either develop custom monitoring modules (increasing development costs) or deploy multiple overlapping systems simultaneously (causing redundant data collection and performance degradation). These challenges urgently necessitate:

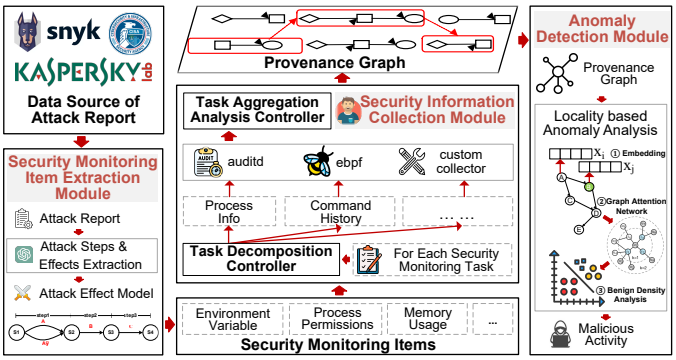

Fig. 1. The workflow of *FEAD*.

- ***A unified monitoring framework*** that systematically covers security-relevant operations to eliminate blind spots and enhance the quality of collected data and resulting provenance graphs, thereby improving attack detection accuracy.
- ***A lightweight deployment solution*** that consolidates essential monitoring capabilities while minimizing resource consumption, enabling organizations to adapt to evolving security requirements without prohibitive operational overhead.

D. Provenance Graph-based Anomaly Detection

Provenance graphs are widely used for anomaly detection due to their ability to capture system behaviors and causal relationships. Early methods, such as StreamSpot [55] and Unicorn [19], relied on graph kernels for clustering, but struggled with stealthy threats and subtle structural differences in rare anomalies. Machine learning-based methods [56], [57], [58] aim to learn complex patterns from provenance graphs using only benign data, but often fail to capture critical structural information, leading to poor detection of subtle or novel threats. With the rapid development of graph neural network (GNN) techniques [59], they have become a popular choice for anomaly detection in provenance graphs [18], [17], [60]. GNNs can capture complex structural patterns, adapt to dynamic systems, and scale to large datasets, improving detection accuracy. However, many GNN methods treat vertices and edges uniformly, making it difficult to distinguish between benign and malicious behaviors, leading to false positives. These challenges urgently necessitate:

- ***A context-aware detection framework*** that intelligently differentiates between normal and suspicious behaviors based on their semantic context and relationships.

III. FEAD: Focus-Enhanced Attack Detection

As shown in Fig. 1, *FEAD* addresses the challenges of non-security-oriented data collection, high deployment costs, and high false positive rates in attack detection with three key components: **(1) The Security Monitoring Item Extraction Module**, which utilizes the *Attack-Effect Model* to extract critical security monitoring items from attack reports

TABLE III
INCLUSION AND EXCLUSION CRITERIA

Type	Description
Inclusion	- Contains technical details on attack steps - Relevant to target systems, environments, or industry - Published within the last 5 years
Exclusion	- Duplicate attack reports - Reports with insufficient length (less than 200 words)

by analyzing attack steps and impacts; **(2) The Security Information Collection Module**, which employs *Adaptive Security Monitoring* to decompose complex tasks, distribute subtasks among existing collectors, and aggregate data for comprehensive analysis, minimizing system overhead; and **(3) The Anomaly Detection Module**, which applies *Locality-based Anomaly Analysis* by leveraging the dense clustering of malicious activities while remaining sparse across different attack phases.

A. The Security Monitoring Item Extraction Module

To tackle the lack of security-oriented efficient monitoring, we propose a systematic approach to identify security-critical monitoring items from online attack reports. This approach involves two main steps: **Attack Report Collection** and **Key Information Extraction**.

1) Attack Report Collection: Our attack report collection followed two main steps: **Report Crawling** and **Report Filtering**.

Report Crawling. During the data acquisition phase, we crawled attack reports from multiple sources to build a comprehensive dataset. Guided by previous studies [61], [62], [63], we developed web crawlers to gather reports from platforms such as Snyk [64], Microsoft Security Intelligence Center [65], and CISA [66], among others (due to space limitations, the complete list is available on our website [67]). To enhance coverage, we also included attack cases from the MITRE ATT&CK knowledge base [68].

Report Filtering. To ensure relevance and quality of collected reports, we rigorously filtered reports based on predefined criteria Table III. This process involved collaboration among four authors (with 2–3 years of attack detection experience) and two industry experts (with 7–8 years in cybersecurity). After manual review, the final dataset comprised 260 APT reports and 7,098 attack cases spanning 268 MITRE ATT&CK techniques (Ref. [67] for detail).

2) Key Information Extraction: Recent advancements in Large Language Models (LLMs) have made them highly effective for information extraction due to their vast knowledge and language understanding [69], [70], [71]. However, LLMs still face challenges like hallucinations [72], [73], which affect extraction accuracy. To address this, we use the Chain-of-Thought (CoT) prompting technique [74], [75], [76]. By designing a reasoning process, we break down complex extraction tasks into manageable, smaller steps that LLMs can directly process, thereby reducing hallucinations and improving the transformation of unstructured attack reports into

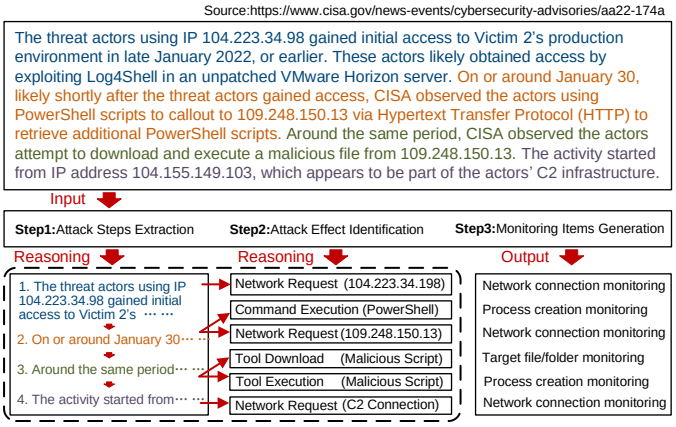

Fig. 2. The workflow of monitoring items generation.

Attack Effect Model and reason out corresponding security monitoring items.

In this work, we implement CoT by instructing LLM to follow our defined steps (in Fig.2) while providing explicit reasoning for each step's output (Ref. [67] for detailed prompt).

Step 1: Attack Steps Extraction We ask LLM to analyze the syntactic structure of input attack report text (I) to identify subject-verb-object relationships, forming triples (T = $\langle\ actor_i,\ action_i,\ target_i\rangle$), where each triple represents a distinct attack step. As shown in the Fig.2, from "The threat actors using IP 104.223.34.98 gained initial access to Victim 2's production environment", we form ($\langle$ threat actors, Network Request, Victim 2's production environment$\rangle$).

Step 2: Attack Effect Identification Attack effects represent the impact of an action on targets, we focus on ($\langle\ action_i$, $target_i\rangle$) pairs from (T). For each pair, we further ask the LLM to analyze the context to determine its corresponding attack effect (E_i). For example, ($\langle$Tool Execution, PowerShell$\rangle$) triggers (E = Program Execution), indicating a behavioral effect on the system.

Step 3: Monitoring Items Generation For each ($\langle action_i$, $target_i$, $E_i\rangle$), the LLM finally generates corresponding monitoring item (M_i). As shown, when (E = Program Execution), the LLM generates (M = Process creation monitoring).

By combining the attack report text (I) with our designed CoT prompt (Ref. [67] for detail) as input to the LLM, we guide the LLM to decompose attack descriptions into structured steps, reason about their system impacts, and derive corresponding monitoring items.

B. The Security Information Collection Module

In the previous section, we identified security-critical monitoring targets by analyzing attack reports and applying our attack effect model. However, identifying what to monitor is only part of the solution—implementing these monitors remains costly.

To address this, we propose a lightweight collaborative security monitoring architecture. It systematically decomposes

complex monitoring tasks and intelligently assigns them to existing monitors. By integrating collected data to meet original goals, this approach reuses existing capabilities, reduces the need for new monitoring modules, and therefore minimizes performance overhead.

The following sections first introduce key ***Definitions***, then present our task decomposition, integration, and deployment methodology.

C. Symbolic Definition of Security Monitoring Capabilities

To accomplish our monitoring objectives, we establish a symbolic definition framework that standardizes the representation of existing monitoring tools' capabilities and our monitoring goals, enabling automated analysis, task decomposition, appropriate allocation of subtasks, and result integration.

Through analysis of existing information collection tools, we found that these tools typically collect data to build provenance graphs through specialized log parsing algorithms. Based on this observation, we adopt a symbolic representation format compatible with both these tools and provenance graphs, enabling a unified framework for defining monitoring capabilities. This design ensures compatibility with existing systems while supporting standardized decomposition and integration of sub monitoring tasks.

Specifically, based on the *Provenance Graph* model in Section II-A, we define the monitoring capability set as $C = \{c_1, c_2, ..., c_n\}$. Each monitoring capability c_i is characterized by a triple $\langle V_c, O_c, T_c \rangle$:

- V_c represents the set of system entities observable by this monitoring capability
- O_c represents property descriptions of monitored entities and their output category sets
- T_c represents the set of system event types observable by this monitoring capability

For the entity property set O_c in monitoring capability c_i, we define:

$$O_c = \{(a_1, t_1), (a_2, t_2), ..., (a_m, t_m)\} \tag{1}$$

where a_i represents the property of the monitored entity, and t_i represents that property's data type. Specifically, property data types include:

- Basic data types: integers ($\mathbb{Z}$), real numbers ($\mathbb{R}$), boolean values ($\{\text{true}, \text{false}\}$), strings ($\Sigma^*$)
- Composite data types: lists, sets, key-value pairs, etc.
- Time-series data types: representing continuously sampled metric values in form (t, v), where t represents a timestamp and v represents the sampled value

To achieve our monitoring objectives, we designed logical operations to decompose complex tasks and integrate results. We define operators (λ) based on data types to flexibly combine existing system monitors. These logical operators (λ) are as follows:

- Logical operations: AND ($\wedge$), OR ($\vee$), NOT ($\neg$)
- Set operations: Element contains ($\in$), Subset relationship ($\subseteq$), Union operation ($\cup$), Intersection operation ($\cap$)

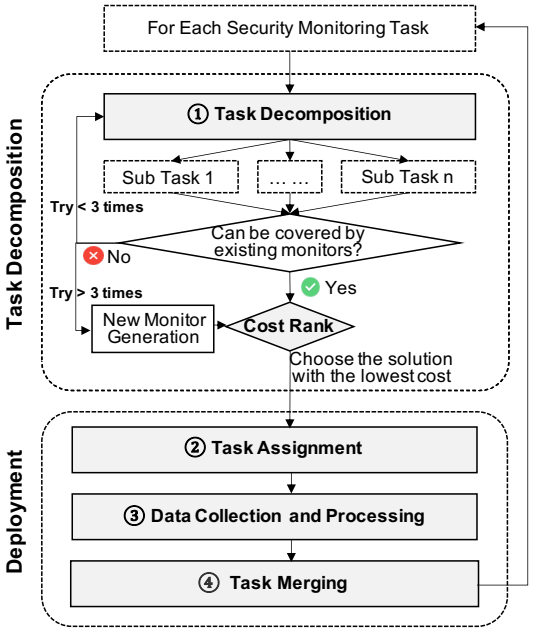

Fig. 3. The workflow of Lightweight Collaborative Security Monitoring Framework.

- String operations: String matching (match), String concatenation (concat), String splitting (split), Substring contains (contains)
- Numeric operations: Greater than ($>$), Less than ($<$), Equal to ($=$), Sum operation (sum), Average operation (avg)

These operators ensures that after breaking tasks into subtasks for existing monitors to collect information and then integrating this data, the resulting monitoring entities and their properties align with our original monitoring goals.

D. Lightweight Collaborative Security Monitoring Framework

Based on the definitions established earlier, we propose a systematic task decomposition and data integration methodology as illustrated in Figure 3. Our lightweight collaborative security monitoring framework transforms complex monitoring requirements into manageable subtasks through recursive decomposition. The workflow begins with *Task Decomposition* where security monitoring tasks are broken down and evaluated against existing capabilities. If existing monitors can solve the subtasks, a *Cost Rank* function evaluates implementation solutions, selecting only the lowest-cost option. When existing capabilities cannot fulfill monitoring requirements after multiple attempts, *New Monitor Recommendation* suggest appropriate new monitoring components. Once optimal solutions are identified, they proceed to *Task Assignment*, where decomposed subtasks are allocated to corresponding existing system monitors. In the *Data Collection and Processing* phase, these monitors gather data to fulfill their assigned subtasks. Finally, during *Task Merging*, the collected data is combined according to the original decomposition logic, effectively implementing the complete monitoring task while

Algorithm 1 Task Decomposition DecomposeTask

Input: Complex monitoring task T, Existing collector capabilities C
Output: Subtask set S, New collector requirements N, Integration logic P

1: **function** DECOMPOSETASK(T, C)
2: $S \leftarrow \emptyset$, $N \leftarrow \emptyset$, $P \leftarrow \emptyset$
3: $(T_{sub}, P_{compose}) \leftarrow$ GenerateSubtasks(T, C) ▷ Subtasks and integration logic
4: **for** each t_i in T_{sub} **do**
5: **if** $\mu(t_i, C) = 1$ **then** ▷ Mappable to existing collectors
6: $S \leftarrow S \cup \{t_i\}$
7: $P \leftarrow P \cup$ GenerateIntegrationLogic($t_i, P_{compose}, C$)
8: **else**
9: $(S', N', P') \leftarrow$ DecomposeTask(t_i, C)
10: **if** $S' \neq \emptyset$ **then** ▷ If subtask set non-empty
11: $S \leftarrow S \cup S'$
12: $N \leftarrow N \cup N'$
13: $P \leftarrow P \cup P'$
14: **else** ▷ Otherwise requires new collector
15: $N \leftarrow N \cup \{t_i\}$ ▷ Requires new collector
16: $P \leftarrow P \cup$ GenerateIntegrationLogic(t_i,$P_{compose}$)
17: **end if**
18: **end if**
19: **end for**
20: **return** (S, N, P)
21: **end function**

Algorithm 2 Generate Subtasks

Input: Monitoring task T, Existing collector set C
Output: Subtask set T_{sub}, Integration logic $P_{compose}$

1: **function** GENERATESUBTASKS(T, C)
2: $T_{sub} \leftarrow \emptyset$, $P_{compose} \leftarrow \emptyset$
3: **for** each collector $c_j \in C$ **do**
4: **if** $T.V_T \subseteq c_j.V_c \wedge T.O_T \subseteq c_j.O_c \wedge T.T_T \subseteq c_j.T_c$ **then** ▷ ***Target entities covered by existing monitors***
5: $T_{sub} \leftarrow T_{sub} \cup \{T\}$
6: $P_{compose} \leftarrow P_{compose} \cup T_{sub}$
7: **end if**
8: **end for**
9: **if** T_{sub} is empty **then** ▷ If no directly satisfying collector
10: $prompts \leftarrow$ Create LLM Prompt
11: **for** each $c_j \in C$ **do**
12: $prompts \leftarrow prompts + \{c_j\}$ ▷ ***Add existing collector capability descriptions***
13: **end for**
14: $prompts \leftarrow prompts$ + "Task requirements: target entities V_T, attribute output type O_T, event type T_T, please generate integration logic that can combine existing monitoring capabilities to satisfy these requirements." ▷ ***Utilize LLM for task decomposition***
15: $P'_I, C'_I \leftarrow$ LLM-generated integration logic, integrated monitoring capabilities
16: **if** C'_I satisfies T's target entities V_T, attribute output type O_T and system events T_T **then** ▷ ***Determine if generated integration logic satisfies monitoring requirements***
17: Add subtasks involved in C'_I to T_{sub} and generate integration logic
18: $P_{compose} \leftarrow P_{compose} \cup P'_I$
19: **else**
20: Request further optimization of integration logic (attempt 3 times)
21: **end if**
22: **end if**
23: **return** $T_{sub}, P_{compose}$
24: **end function**

minimizing resource utilization. This section elaborates the core mechanisms of this methodology, encompassing **Task Decomposition** and **Deployment**.

1) Task Decomposition: This phase establishes standardized constraints for decomposable tasks and introduces an algorithm for systematically decomposing complex monitoring tasks into manageable subtasks with integration logic. This integration logic enables subtasks to combine their monitoring outputs, reconstructing results for the original complex task.

Specifically, given a monitoring task T, we represent it symbolically as a triple $\langle V_T, O_T, T_T \rangle$, where:

- V_T represents target system entities requiring observation by this monitoring task
- O_T represents property descriptions of target entities and their output types, $O_T = (a_T, t_T)$
- T_T represents system events requiring observation by this monitoring task

Based on this definition, the monitoring task decomposition problem can be formalized as: given a monitoring task T and existing monitoring capabilities set $C = \{c_1, c_2, ..., c_n\}$ in the system, our objective is to identify a monitoring capabilities subset $C' \subseteq C$ and corresponding integration operations set λ, such that the combination satisfies task T requirements.

Based on these constraints, we propose Algorithm 1, which processes complex monitoring task T through recursive decomposition and integration. Initially, *GenerateSubtasks(T, C)* (line 3) decomposes T into subtasks set T_{sub} and integration logic $P_{compose}$.

Specifically, as Algorithm 2 *GenerateSubtasks* demonstrates, this algorithm describes an adaptive monitoring task decomposition and integration process. Algorithm 2 first traverses existing collector set C in lines 3-8, examining whether a single collector can directly satisfy monitoring task T requirements, including target entity (V_T), attribute output type (O_T), and event type (T_T) coverage. When no directly satisfying collector is found, Algorithm 2 introduces a large language model for task decomposition in lines 9-23, constructing prompts containing existing collector capability descriptions to generate integration logic P'_I and corresponding monitoring capabilities C'_I. The algorithm verifies generated integration scheme feasibility; if satisfying original monitoring requirements, relevant subtasks are added to T_{sub} and integration logic $P_{compose}$ is updated, otherwise up to three optimization iterations are performed. Finally, the algorithm outputs optimized subtask set T_{sub} and corresponding integration logic $P_{compose}$, achieving automated monitoring task decomposition and dynamic integration.

Subsequently, Algorithm 1 lines 4-19 evaluates each subtask using matching function $\mu(t_i, C)$, determining whether a monitoring task can be fulfilled by existing collectors:

Algorithm 3 Generate Integration Logic

Input: Subtask t_i, Existing integration logic $P_{compose}$, Optional parameter existing collector capabilities C (default: None)
Output: Integration logic P

1: **function** GENERATE_INTEGRATION_LOGIC($t_i, P_{compose}, C =$ None)
2: $P \leftarrow \emptyset$
3: **if** $C \neq$ None **then**
4: **for** each basic information collector $c_i \in C$ **do**
5: **if** collector c_i corresponding to subtask t_i found **then** Update $P_{compose}$, appending corresponding collector c_i to t_i forming P
6: **end if**
7: **end for**
8: **else** ▷ No existing basic information collectors available, must construct new ones
9: $P_{new} \leftarrow$ CreateNewCollector(t_i) ▷ Manually construct new monitor
10: Update $P_{compose}$, appending newly constructed P_{new} to t_i forming P
11: **return** P
12: **end if**
13: **end function**

$$\mu(t_i, C) = \begin{cases} 1 & \text{if } \exists c_j \in C : t_i.V_T \subseteq c_j.V_c \\ & \wedge\, t_i.O_T \subseteq c_j.O_c \wedge t_i.T_T \subseteq c_j.T_c \\ 0 & \text{Otherwise} \end{cases} \tag{2}$$

Here, mappable subtasks ($\mu(t_i, C) = 1$, Algorithm 1 lines 5-7) are added to S with corresponding integration logic via *GenerateIntegrationLogic* (Algorithm 1 line 7), which derives integration operations λ required for combining their outputs. The obtained integration logic is recorded in P. Specifically, as Algorithm 3 lines 3-8 illustrate, the algorithm traverses existing basic information collectors $c_i \in C$, identifying collectors matching subtask t_i, and integrates them into existing integration logic $P_{compose}$ to form final integration logic P.

For unmappable subtasks (Algorithm 1 lines 8-18), we attempt further decomposition using the previously described method. If further decomposition is possible (line 10), resulting subtasks are recursively processed, with integration logic incorporated into P (Algorithm 1 lines 11-13).

If further decomposition is impossible, the task is marked as requiring new collector capabilities (Algorithm 1 line 15, equivalent to Algorithm 3 lines 9-12), and its integration logic is added to P (line 16). This process continues until all subtasks are either mapped to existing collectors or identified as new capability requirements in N.

Cost Rank To formalize our approach to cost optimization within the security monitoring framework, we introduce a comprehensive cost function that quantifies the tradeoffs involved in monitoring task decomposition. This function serves as a critical component in our optimization process, guiding the selection of implementation strategies that minimize resource utilization while maintaining effective security coverage.

The cost function $C(T)$ for a monitoring task T decomposed into subtasks $\{t_1, t_2, ..., t_n\}$ is formulated as:

$$C(T) = \sum_{i=1}^{n} \left(D_{\text{deploy}}(t_i) + D_{\text{dev}}(t_i) \right) + C_{\text{complex}}(T) \tag{3}$$

Where D_{deploy} denotes deployment costs, D_{dev} denotes development costs, and C_{complex} captures integration complexity.

For deployment costs, we differentiate between existing and new monitoring components:

$$D_{\text{deploy}}(t_i) = \begin{cases} \alpha \cdot P_{\text{overhead}}(t_i) & \text{if } t_i \text{ maps to existing monitor} \\ \beta_{\text{imp}} \cdot P_{\text{overhead}}(t_i, \text{imp}) & \text{if new monitor required} \end{cases} \tag{4}$$

Where α is a weighting factor for existing monitors, P_{overhead} quantifies performance impact, and β_{imp} represents implementation-specific weights that vary according to implementation approach (hardware, kernel, or user-space). Notably, the relationship $\beta_{\text{hw}} < \beta_{\text{kernel}} < \beta_{\text{user}}$ reflects our observation that hardware implementations typically introduce less runtime overhead than kernel-level implementations, which in turn impact performance less than user-space implementations.

Development costs are structured to minimize resource usage by prioritizing existing capabilities: zero cost for reusing existing monitors, with increasing costs for user-space, kernel-level, and hardware implementations respectively (i.e., $0 < \gamma_{\text{user}} < \gamma_{\text{kernel}} < \gamma_{\text{hw}}$).

With weighting factors satisfying $\gamma_{\text{hw}} > \gamma_{\text{kernel}} > \gamma_{\text{user}}$, reflecting the relative development effort associated with each implementation approach. This formulation encourages the reuse of existing monitoring capabilities when possible, as these components incur zero additional development cost.

Integration complexity is modeled using a logarithmic function to reflect the sub-linear growth in complexity as the number of components increases, i.e., $C_{\text{complex}}(T) = n$, Where n represents the number of subtasks in the decomposition.

After generating potential implementation solutions through task decomposition, we evaluate each solution using the cost function defined above. Let $S = \{S_1, S_2, ..., S_m\}$ represent the set of candidate solutions, where each solution S_j consists of a specific decomposition of the original monitoring task. The cost evaluation function assigns a numerical cost score to each solution, i.e., $\text{Score}(S_j) = C(S_j)$

We then rank all candidate solutions by their cost scores in ascending order. The solution with the minimum cost score is selected as the optimal implementation strategy. This process ensures that our monitoring implementation balances comprehensive security coverage with practical resource constraints. By systematically evaluating and comparing different implementation strategies, we achieve optimal resource utilization while maintaining effective security monitoring capabilities.

2) Deployment: After systematically decomposing monitoring tasks and optimizing their implementation strategy through our cost-aware approach, we proceed to the deployment phase. This phase encompasses three key stages: (1) **Task Assignment** to appropriate collectors, (2) **Data Collection and Processing** according to task specifications, and (3) integration of collected data through **Task Merging**.

Task Assignment. Based on $\mu(t_i, C)$, subtasks $t_i \in S$ are assigned to existing collectors C or marked as new collector requirements N. Subtasks with $\mu(t_i, C) = 1$ are mapped to compatible collectors for direct information collection, while subtasks with $\mu(t_i, C) = 0$ are added to N, and corresponding custom collectors are implemented through expert intervention to fulfill monitoring requirements.

Data Collection and Processing. Collectors execute tasks based on t_i, producing data outputs O_t.

Task Merging. Using integration logic P from *Task Decomposition*, collected outputs $\{O_t\}$ are combined to reconstruct original task result O_T. Integration operations $\lambda \in P$ ensure consistency with T's output requirements.

Through this systematic approach, our methodology maximizes utilization of existing collectors, reducing requirements for new modules and deployments, thereby optimizing resource efficiency.

E. Monitor Construction and Deployment Case Study

Consider a *Log4Shell zero-to-root* attack scenario (Figure 4), where attackers exploit the Log4Shell vulnerability for initial access and then manipulate environment variables (EnvVar) for privilege escalation. A critical monitoring requirement is tracking *environment variable modifications*. Traditional security monitoring relies on system call events, but this approach has major limitations: environment variable operations typically execute through shell built-in functions that don't trigger system calls. This creates monitoring blind spots since traditional system call tracking cannot detect these operations. As environment variable manipulation is often critical in privilege escalation attacks, these blind spots significantly impact system security awareness capabilities.

Therefore, based on the real-world attack scenario analysis, we must enhance the existing security monitoring framework to address these critical requirements. Using Algorithm 1's task decomposition methodology, we decompose *environment variable modification monitoring* into fundamental implementable subtasks. Specifically, we decompose it into two basic subtasks T_{sub}: command history monitoring (t_1) that tracks command line activities with output type $O_{t1} = \{e_1, ..., e_n\}$, where each e_i contains command string, process ID and timestamp; and environment variable list monitoring (t_2) that tracks the system's environment variable names, with output type $O_{t2} = \{v_1, ..., v_m\}$, where each v_i represents an environment variable name.

This task decomposition implements two key monitoring components: t_1 is responsible for command line history monitoring, implemented through eBPF technology. Specifically, we utilize eBPF's dynamic tracing capabilities to perform probe instrumentation at shell program key functions (such as readline, execute_command, etc.), capturing user input command sequences in real-time. t_2 focuses on environment variable monitoring, where we developed an independent collector program that periodically obtains and records system environment variable snapshots at predetermined intervals.

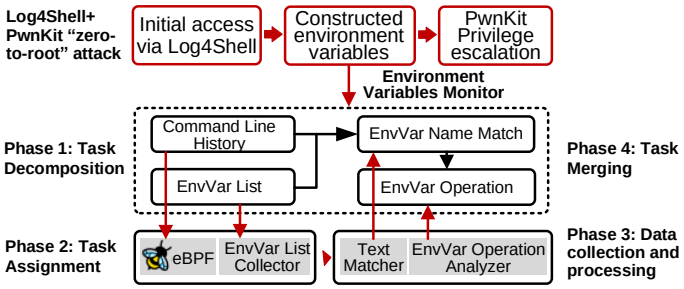

Fig. 4. SMART method Log4Shell Zero-to-Root Attack example.

The integration logic $P_{compose}$ implements data association analysis as follows: first through operation $\lambda_{name}(O_{t1}, O_{t2}) = \{cmd \in O_{t1} | \exists var \in O_{t2} : var$ appears in $cmd.cmd\_str\}$, it performs name matching between command strings and environment variable names, then analyzes matched commands to identify current environment variable operations.

F. The Anomaly Detection Module

Building upon our observations of attack locality, we introduce a weighted mechanism with attack locality awareness to enhance threat detection effectiveness. This mechanism leverages endpoint anomaly information embedded in the provenance graph mentioned earlier and guides weight allocation through anomaly scoring, thereby improving the identification and response to potential threats. The following sections detail our approach through three stages: provenance graph node embedding, graph neural network design, and data post-processing.

Provenance Graph Node Embedding. Given a provenance graph $G = (V, E)$ where $V = \{v | v \in (Process \cup File \cup Socket)\}$ and $E \subseteq V \times V \times T$, each vertex $v \in V$ is characterized by a feature vector that encodes its entity-behavior interaction patterns:

$$h_v = [f_{in}(v) \parallel f_{out}(v) \parallel S(v)] \tag{5}$$

where $f_{in}(v)$ and $f_{out}(v)$ represent the distributions of incoming and outgoing edge types (i.e., behavior types) respectively, and $S(v)$ is the anomaly score of node v as described in Section II-A.

$$f_{in}(v)[t] = |(u, v, type) \in E | type = t, t \in T| \tag{6}$$

$$f_{out}(v)[t] = |(v, u, type) \in E | type = t, t \in T| \tag{7}$$

Specifically, $f_{in}(v)[t]$ (Equation 6) counts the number of edges of type t that are directed toward node v. This captures the frequency of specific types of behaviors performed by other nodes on v. Similarly, $f_{out}(v)[t]$ (Equation 7) counts how many edges of type t originate from node v. This captures the frequency of specific types of behaviors that node v performs on other nodes.

In essence, these formulas compute the frequency of each behavior type associated with node v, categorized as incoming

(behaviors from other nodes toward v) and outgoing (behaviors from v toward other nodes). The resulting feature vector encodes the behavioral patterns of nodes in the provenance graph, aiding in the identification of anomalous or potentially malicious activities.

Furthermore, the inclusion of the anomaly score $S(v)$ enhances the discriminative capability of the feature representation, effectively guiding branch weight allocation during the global provenance analysis process, thereby improving the overall accuracy and reliability of attack detection.

Graph Neural Network Design. In this paper, we use a two-layer Graph Attention Network (GAT) [77] to learn vertex features while capturing vertex-edge (i.e., entity-behavior) patterns. During the learning process, the feature update for vertex i is computed by:

$$h_i = \sum_{j \in \mathcal{N}(i)} \alpha_{ij} \mathbf{W}_2 \text{ELU} \left(\sum_{k \in \mathcal{N}(j)} \alpha_{jk} \mathbf{W}_1 h_k \right) \tag{8}$$

Here, $\mathbf{W}_1$ and $\mathbf{W}_2$ are the weight matrices for the first and second layers. The attention coefficients α_{ij} (α_{jk}) determine the importance of neighbor j's (k's) features to vertex i (j). Exponential Linear Unit (ELU) serves as the inter-layer activation function, which enhances gradient flow, improves expressive capability, and provides non-linear transformations across negative value ranges, thereby improving model performance.

For anomaly detection, we first train our GAT on benign provenance graphs. By adjusting the vertex weights based on vertex-edge (i.e., entity-behavior) patterns in benign graphs, we obtain a benign GAT entity-behavior model. In the prediction phase, we apply a multi-class classification approach to the output of the GAT layer. Specifically, after feature aggregation, we apply the *softmax* function to the vertex feature h_v to produce class probabilities, i.e., $z_v = \text{softmax}(h_v)$. Subsequently, we employ the argmax function to obtain prediction results from the softmax output. Specifically, we select the class with the highest probability as the model's final predicted class, i.e., $\hat{y}_v = \arg\max_c z_v^c$, where z_v^c represents the probability that vertex v belongs to class c.

Data Post-processing. We define $is\_anomalous(v)$ to check vertex v. If its predicted entity type differs from its actual type, we consider it anomalous, having deviated from our trained benign entity-behavior model. When anomalies are detected, we analyze their neighbors, where attention weights accumulate among connected anomalous vertices while becoming diluted among benign ones.

To further reduce false positives, for each predicted anomalous vertex v, we analyze its k-hop neighborhood $N_k(v)$ (where $k = 2$ in our implementation) and compute a benign density score:

$$\text{Benign Density}(v) = \frac{|\{u \in N_k(v) : \neg \text{is\_anomalous}(u)\}|}{|N_k(v)|} \tag{9}$$

If Benign Density(v) exceeds a threshold (80% in our implementation), indicating that the majority of its neighboring vertices are benign, we consider v to lack attack locality and correct it to benign, further reducing false positives.

TABLE IV
EVALUATION DATASETS OVERVIEW

Dataset	Scenario	Benign Vertices	Anomalous Vertices	Edges
DARPA TC	THEIA	3,505,326	25,362	102,929,710
	Trace	2,416,007	67,383	6,978,024
	CADETS	706,966	12,852	8,663,569
	Fivedirections	569,848	425	9,852,465
CSE-CIC-IDS2018	-	212,628	89,228	501,856

IV. EVALUATION.

We evaluate *FEAD* by answering the following research questions:

- **RQ1: (Monitoring Coverage)** What security-critical monitoring items from real-world attacks are captured, and do existing tools miss any of these?
- **RQ2: (Effectiveness)** How effective is *FEAD* in detecting attack events and anomalies in resource-constrained environments?
- **RQ3: (Ablation Study)** How do the components and design choices of *FEAD* impact its effectiveness in attack detection and anomaly identification?
- **RQ4: (Deployment Costs)** What are *FEAD*'s development costs, and is it feasible for real-world deployment?

Evaluation Datasets We evaluate our approach using the DARPA TC [78] and CSE-CIC-IDS2018 [79] datasets, as in previous work [18], [17], [13], [80]. The DARPA TC dataset consists of four scenarios—THEIA, Trace, CADETS, and Fivedirections—covering various attack steps and environments. The CSE-CIC-IDS2018 dataset, from the Canadian Institute for Cybersecurity (CIC), includes data on various attacks, such as Brute Force, Heartbleed, etc. Table IV summarizes these datasets.

Since existing datasets do not incorporate our constructed monitoring system, we create custom datasets to evaluate *FEAD* 's monitoring designs and anomaly detection capabilities. We collaborate with two industry experts to reproduce common exploits, such as the Log4j vulnerability (CVE-2021-44228)[81], [82], [83] and the OpenSMTPD vulnerability (CVE-2020-7247)[84], [85], [86], along with the associated attack activities based on real-world application scenarios. The datasets include: **(1) Log4j+ENV Attack Dataset:** This dataset simulates attacks related to the Log4j vulnerability, covering scenarios like initial access to the target, privilege escalation via environment variables, and the establishment of reverse shell connections. **(2) OpenSMTPD+Malicious Execution:** This dataset demonstrates an attack chain exploiting the OpenSMTPD vulnerability, including unauthorized command execution, downloading malicious scripts, and executing additional malware.

Deployment Environment. (1) For attack report analysis and security monitoring item extraction, we use Microsoft's Azure OpenAI API [87] with the GPT-3.5-16K model and set the temperature parameter to 0.4 to balance creativity and

Registry Operations	Host System Metrics	User Activities	File Operations	Network Traffic	Process Monitoring	System Security
Registry Key Abnormal Access Records	Host Abnormal Restart Records	User Command Execution	File Rename and Move Records	Network Traffic Metrics (Bytes/Packets)	Process Core Metrics (Name, PID, Tree)	Script File Execution
Registry Key Deletion Records	Host System Resource Usage Monitoring	User Login Records and Statistics	File Modification Time	DNS Resolution Records	Process DLL and Path Records	Drive Mount Operations
Registry Key Permission Change Records	Host System Configuration Backup Records	User Account Management	File Deletion Records	Protocol Type	Process Execution Details	Sensitive Location Access (MBR, Boot Sector, etc.)
Registry Key Value Change Records	Host System Crash Records	User Authentication Records	File Signature Verification	Source/Destination IP	Process Resource Usage	
Registry Key Read Records	Host Software Installation/ Uninstallation Records	User Privilege Escalation Records	File Path and Name	Connection Status and Duration	Process File Operations	
Registry Key Creation Records	Host Boot/Shutdown Time	User Session Monitoring	File Encryption Status	Port Numbers	Process Network Activity	
	Host Kernel Module Loadin Records	User Operation History	File Hash Value Changes	Encrypted Traffic Records	Process Security (Signature, Permissions)	
	Host Memory/CPU Usage	User Created Processes	File Access Time and Permissions		Process Environment Variables	
	Host Service Status Change Records		File ACL Changes		Process Performance Metrics	
	Host Firewall Rules Change Records		File Size Changes			
	Host Scheduled Task Execution Records		File Creation Time			
	Host Security Settings Change Records					

Fig. 5. Security-focused monitoring items from real-world attacks.

TABLE V
CUSTOM DATASET INFORMATION

Scenario	Benign Vertices	Anomalous Vertices	Edges
Log4j+ENV	812	37	1,975
OpenSMTPD+Malicious Execution	3,384	34	7,989

accuracy. **(2) For our GAT implementation**, we use a two-layer GAT architecture with 8 attention heads and a hidden layer of 128 units. We set the batch size to 500, learning rate to 0.01, weight decay to 5e-4, and a dropout rate of 0.5 to prevent overfitting. **(3) For *FEAD* deployment and anomaly detection experiments**, we use Debian 10.8 (Linux 5.10) with an Intel(R) Core(TM) i5-12500 processor and 48GB RAM, hosting all information collection tools and attack detection systems. **(4) For our cost function implementation**, we determined parameters through expert evaluation. Four co-authors with 2-3 years of attack detection research experience, plus two industry experts with 7-8 years of cybersecurity experience, collaboratively established parameter values using majority voting to resolve disagreements. We set $\alpha = 0.2$ for existing monitor weights, reflecting minimal overhead when reusing components. For new implementations, we assigned $\beta_{user} = 0.7$, $\beta_{kernel} = 0.5$, and $\beta_{hw} = 0.3$, capturing our observation that user-space implementations typically introduce higher runtime overhead than kernel-space, while hardware-accelerated monitors showed the lowest impact. Development costs were parameterized as $\gamma_{user} = 10$, $\gamma_{kernel} = 25$, and $\gamma_{hw} = 50$, representing relative implementation effort in person-days based on previous projects, with hardware implementations requiring more specialized expertise and time. This methodology ensured our cost function balanced theoretical soundness with practical considerations, producing task decompositions that maximized existing monitor utilization while minimizing development overhead.

A. RQ1: Monitor Items and Monitoring Coverage Evaluation

Fig. 5 presents the 85 security-relevant monitoring items our methodology has extracted from real-world attack reports and ATT&CK cases. While traditional monitoring tools primarily focus on basic system elements (process PIDs, names, arguments, file paths/names, and network IPs and ports). In contrast, our approach systematically broadens the monitoring scope. Beyond these fundamental elements, our framework incorporates detailed monitoring of system security configurations (e.g., firewall rule changes, security setting modifications, and service status updates), user authentication patterns (e.g., login records, privilege escalation events, and session tracking), process behaviors (e.g., DLL loading, resource usage, and network activity), and file integrity metrics (e.g., hash values, encryption status, and changes to access control lists), which are frequently neglected by conventional monitoring tools.

As shown in Table VII, while *Auditd* and *Camflow* are widely used to generate system logs for provenance graphs, they were not originally designed for security, resulting in low coverage—49.40% and 40.00% respectively—for the monitoring requirements derived from real-world attacks. *Auditbeat*, Elastic's security-enhanced version of *Auditd*, demonstrates how expert knowledge can close these gaps, achieving 75.30% coverage. More recently, eBPF has gained traction for its low overhead and real-time capabilities in Linux environments. We evaluated *eAudit*, an academic extension of *Auditd* using eBPF, which improved coverage to 56.47%, though with room for further improvement. Focusing on Linux (excluding Windows-specific tools like ETW), our approach applies the methodology from Section III-B (with case studies in Section III-E). By decomposing complex monitoring tasks, leveraging existing monitors, and integrating collected data, we extended coverage to 83.50%. We further validated the effectiveness and practicality of our monitoring framework

TABLE VI
COMPARISON OF DETECTION EFFECTIVENESS

Dataset	Our Approach				ThreaTrace			
	Precision	Recall	FPR	F1-Score	Precision	Recall	FPR	F1-Score
Cadets	97.92%	99.88%	0.08%	**98.89%**	93.84%	99.96%	0.24%	96.81%
Fivedirections	72.53%	95.06%	0.04%	**82.28%**	75.21%	84.94%	0.032%	79.78%
Theia	99.81%	99.91%	0.02%	**99.86%**	95.49%	99.90%	0.37%	97.65%
Trace	98.24%	99.996%	0.02%	**99.11%**	81.59%	99.99%	1.33%	89.86%
CSE-CIC	98.99%	93.55%	0.04%	**96.19%**	92.58%	95.12%	7.58%	93.44%
Log4j+ENV	99.46%	100%	0.02%	**99.73%**	76.09%	99.99%	1.36%	86.42%
OpenSMTPD	94.97%	100%	0.05%	**97.42%**	77.50%	91.18%	0.27%	83.78%
Average	94.41%	98.91%	0.03%	**96.76%**	84.61%	95.87%	1.57%	88.53%

TABLE VII
COVERAGE ANALYSIS OF DIFFERENT MONITORING TOOLS

Metrics	**Auditd**	**Auditbeat**	**eAudit**	**Camflow**	**Our Method**
Coverage Quantity	42	64	48	34	71
Coverage Rate (%)	49.40%	75.30%	56.47%	40.00%	83.50%

through ablation experiments in ***RQ3***.

B. RQ2: Effectiveness of FEAD

To evaluate the effectiveness of our proposed approach, we conducted comprehensive experiments on multiple datasets and compared our method against ThreaTrace [18], a state-of-the-art GraphSAGE-based detection method for provenance graph anomaly detection that has received significant citations and provides complete open-source implementation.

Table VI shows that our approach demonstrates superior detection performance (average 8.23% higher F1-score) compared to ThreaTrace. A detailed analysis of performance across individual datasets reveals the following improvements:

DARPA Dataset Performance Analysis: Our method demonstrates remarkable consistency across the four *DARPA datasets*, achieving notable improvements on the *Theia* dataset with up to 9.25% higher F1-scores and up to 1.31% lower false positive rates (FPR). This translates to a 16.65% precision increase, highlighting our locality-aware anomaly detection approach's effectiveness in established complex system environments. Similarly impressive results on the *Trace* dataset show that our method achieved an outstanding 99.11% F1-score compared to ThreaTrace's 89.86%, primarily due to our substantially higher precision (98.24% versus 81.59%) while maintaining comparable recall. These improvements demonstrate how our attack locality-based vertex weighting mechanism effectively distinguishes between benign and malicious activities in diverse system behaviors.

CSE-CIC Dataset Performance Analysis: On this dataset, our method achieves a 96.19% F1-score, outperforming ThreaTrace's 93.44%. Most remarkably, our approach drastically reduces the FPR to merely 0.04% compared to ThreaTrace's 7.58% - a 189-fold improvement that would significantly reduce the number of false alarms security analysts must investigate. This substantial performance gap further validates that our locality-aware anomaly detection mechanism effectively leverages the clustering characteristics of malicious activities, providing more precise differentiation between normal network traffic and genuine attacks.

Custom Attack Dataset Performance Analysis: To validate our security monitoring framework's effectiveness, we collected data from real-world attacks using our proposed monitoring system. Here, our approach shows remarkable improvements, with the Log4j+ENV dataset showing a 13.31% higher F1-score compared to ThreaTrace. Our approach achieves a near-perfect 99.73% F1-score versus ThreaTrace's 86.42%, demonstrating exceptional detection capability for this sophisticated attack vector. For the OpenSMTPD dataset, we observe similarly impressive results, reaching 97.42% F1-score while maintaining a remarkably low 0.05% FPR. These results on modern attack scenarios clearly show how our combined approach - using both security-oriented monitoring framework and locality-aware analysis - successfully captures important attack signatures while correctly distinguishing benign nodes from the provenance graph.

These results confirm that our approach effectively combines enhanced security monitoring with locality-aware anomaly detection to significantly improve detection accuracy while reducing false positives across diverse environments.

C. RQ3: Ablation Study of FEAD

To evaluate the effectiveness of our key design components, we conducted two ablation experiments: **(1) whether utilizing our security-oriented monitoring items framework enhances detection capabilities** - testing *FEAD*'s detection performance on our reproduced Log4j and OpenSMTPD vulnerability exploitation scenarios, with and without our monitoring items for data/log collection, and **(2) whether considering attack locality improves detection performance** - evaluating *FEAD*'s detection effectiveness on the widely-used DARPA datasets with and without attack locality consideration.

Monitoring Framework Impact Analysis. As shown in Table VIII, our monitoring items framework significantly improves detection capability, achieving an average F1-score of 98.58% compared to 85.95% without it, showing a 12.63% improvement. The FPR drops by 0.91% with our framework, corresponding to a 20.93% precision increase. Examining the individual datasets reveals that the most dramatic improvement occurs on the Log4j+ENV scenario, where precision increases from 74.00% to 99.46% (a 25.46% gain) while maintaining perfect recall. Similarly, for OpenSMTPD, precision increases

TABLE VIII
ABLATION STUDY: MONITORING FRAMEWORK AND ATTACK LOCALITY

Data Source	With Our Monitoring Framework				Without Our Monitoring Framework			
	Precision	Recall	FPR	F1-Score	Precision	Recall	FPR	F1-Score
opensmtpd	94.97%	100%	0.05%	**97.42%**	78.57%	97.06%	0.27%	86.84%
log4jEnv	99.46%	100%	0.02%	**99.73%**	74.00%	100%	1.6%	85.06%
Average	97.22%	100%	0.035%	**98.58%**	76.29%	98.53%	0.94%	85.95%
Scenario	With Attack Locality				Without Attack Locality			
	Precision	Recall	FPR	F1-Score	Precision	Recall	FPR	F1-Score
Cadets	97.92%	99.88%	0.08%	**98.89%**	84.14%	99.89%	0.70%	91.34%
Fivedirections	72.53%	95.06%	0.04%	**82.28%**	44.30%	95.06%	0.14%	60.43%
Theia	99.81%	99.91%	0.02%	**99.86%**	92.11%	99.91%	0.68%	95.85%
Trace	98.24%	99.996%	0.02%	**99.11%**	89.47%	99.995%	0.11%	94.44%
Average	92.13%	98.71%	0.04%	**95.04%**	77.51%	98.71%	0.41%	85.52%

TABLE IX
DEPLOYMENT COST ANALYSIS

Metrics	**Baseline**	***FEAD* (Cost)**	**Non-*FEAD* (Cost)**
SPEC2006 (Execution time)	2,724.6s	2,871.85s (**5.40%**)	3043.65s (**11.71%**)
STREAM (Throughput)	19,036.75 MB/s	18,988.28 MB/s (**0.26%**)	18860.30 MB/s (**0.94%**)
Application (Execution time)	4,498ms	4,514ms (**0.36%**)	5,814ms (**29.26%**)
Lines of Code	-	59,203	86,257

from 78.57% to 94.97% (a 16.4% improvement). These results validate that our systematically extracted monitoring items from real-world attacks are crucial for accurate attack detection, providing visibility into attack behaviors that would otherwise be missed.

Attack Locality Impact Analysis. *FEAD* achieves an average F1-score of 95.04% when considering locality patterns, versus 85.52% without them. This demonstrates locality's effectiveness in improving detection precision by 9.52% while maintaining a significantly lower false positive rate (0.04% vs 0.41%). The impact is most pronounced on the Fivedirections dataset, where F1-score increases from 60.43% to 82.28% (a 21.85% improvement), primarily through enhanced precision (44.30% to 72.53%). Even on datasets where our approach already performs well, such as Theia (F1-score improvement from 95.85% to 99.86%), incorporating locality patterns further reduces false positives by 0.66%. This consistent pattern across all datasets demonstrates that attack locality is a fundamental characteristic that effectively distinguishes genuine attacks from isolated anomalies.

These results confirm that both our security-oriented monitoring framework and attack locality consideration substantially enhance detection effectiveness while minimizing false alarms. This framework ensures comprehensive visibility into security-relevant system activities, while the locality-based analysis differentiates between benign anomalies and actual attack patterns, creating a multiplicative rather than merely additive improvement in overall detection capabilities.

D. RQ4: Deployment Cost Analysis

To evaluate the deployment costs of our approach, we measured both performance overhead and development complexity. For performance metrics, we assessed CPU performance using the widely-recognized SPEC2006 benchmark [88], [89], memory throughput using STREAM benchmark [90], [91], and application runtime overhead through instrumentation tests across 100 runs. For development complexity, we measured implementation effort in lines of code. While our FEAD approach decomposes complex monitoring tasks and distributes them to existing collectors, the non-FEAD implementation requires developing new monitoring capabilities from scratch.

Table IX shows that *FEAD* incurs minimal deployment costs. It introduces only a 5.40% overhead in CPU performance (SPEC2006), a 6.31% improvement over traditional methods. Memory throughput impact is even smaller, with a 0.26% degradation, sustaining 18,988.28 MB/s. Most notably, *FEAD* introduces a mere 0.36% application runtime overhead, reduced by 28.90%. In terms of development effort, *FEAD* requires 59,203 lines of code, cutting implementation complexity by 31.4%. These results highlight *FEAD*'s monitoring strategy as an effective solution that minimizes both performance impact and development effort, making it highly suitable for production environments.

V. CONCLUSION

We proposed *FEAD*, a novel framework for detecting sophisticated cyber threats in resource-constrained systems. We introduced an attack model-driven monitoring items identification approach that systematically extracts security-critical items from attack reports, proposed an efficient monitoring framework deployment via a complex task decomposition mechanism, and developed a locality-aware anomaly analysis technique that leverages the clustering characteristics of malicious activities. We conducted extensive evaluations on multiple real-world datasets and custom attack scenarios. The experimental results demonstrate that *FEAD* outperforms existing solutions with an 8.23% higher F1-score while maintaining only 5.4% overhead, validating its effectiveness for efficient and accurate attack detection.

VI. ACKNOWLEDGMENT

We are grateful for all the anonymous reviewers. The authors from Nanjing University are supported in part by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (No. BK20202001), the National Natural Science Foundation of China (No. 62232008, 62172200), and the Postgraduate Research & Practice Innovation Program of Jiangsu Province (No. KYCX24_0237)

Ethical Considerations

This research followed ethical guidelines throughout all phases of data collection and analysis. In Section III-A, threat intelligence was gathered exclusively from publicly accessible web pages while fully complying with site policies and ethical web crawling practices, ensuring no unauthorized access occurred. The evaluation in Section IV utilized publicly available DARPA and CSE-CIC datasets, which are widely accepted benchmarks in the security research community for assessing attack detection capabilities without introducing new ethical concerns. For the specialized dataset developed in Section IV, all domain experts were clearly informed about our research goals before participating, received strong assurances that no user privacy would be compromised, and explicit acknowledgment of their right to withdraw participation at any time without consequence, thus maintaining ethical integrity throughout the study.

References

[1] "Lazarus apt continues to exploit log4j vulnerability," 2024, https://www.secureworld.io/industry-news/lazarus-continues-exploit-log4j.

[2] T. Zhu, J. Wang, L. Ruan, C. Xiong, J. Yu, Y. Li, Y. Chen, M. Lv, and T. Chen, "General, efficient, and real-time data compaction strategy for apt forensic analysis," *IEEE TIFS*, vol. 16, 2021.

[3] T. Zhu, J. Yu, C. Xiong, W. Cheng, Q. Yuan, J. Ying, T. Chen, J. Zhang, M. Lv, Y. Chen *et al.*, "Aptshield: A stable, efficient and real-time apt detection system for linux hosts," *IEEE TDSC*, vol. 20, no. 6, 2023.

[4] Z. Yang, X. Liu, T. Li, D. Wu, J. Wang, Y. Zhao, and H. Han, "A systematic literature review of methods and datasets for anomaly-based network intrusion detection," *Comput. Secur.*, vol. 116, 2022.

[5] F. Yang, J. Xu, C. Xiong, Z. Li, and K. Zhang, "Prographer: An anomaly detection system based on provenance graph embedding," in *USENIX Security '23*, 2023.

[6] H. Ding, J. Zhai, Y. Nan, and S. Ma, "Airtag: Towards automated attack investigation by unsupervised learning with log texts," in *USENIX Security '23*, 2023.

[7] "auditd(8) - linux man page," 2024, https://linux.die.net/man/8/auditd.

[8] N. Michael, J. Mink, J. Liu, S. Gaur, W. U. Hassan, and A. Bates, "On the forensic validity of approximated audit logs," in *ACSAC '20*, 2020.

[9] K. Kurniawan, A. Ekelhart, E. Kiesling, G. Quirchmayr, and A. M. Tjoa, "Krystal: Knowledge graph-based framework for tactical attack discovery in audit data," *Comput. Secur.*, vol. 121, 2022.

[10] R. Sekar, H. Kimm, and R. Aich, "eaudit: A fast, scalable and deployable audit data collection system," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 3571–3589.

[11] H. Irshad, G. Ciocarlie, A. Gehani, V. Yegneswaran, K. H. Lee, J. Patel, S. Jha, Y. Kwon, D. Xu, and X. Zhang, "Trace: Enterprise-wide provenance tracking for real-time apt detection," *IEEE TIFS*, vol. 16, 2021.

[12] "Auditbeat: Lightweight shipper for audit data," 2024, https://www.elastic.co/cn/beats/auditbeat.

[13] A. Alsaheel, Y. Nan, S. Ma, L. Yu, G. Walkup, Z. B. Celik, X. Zhang, and D. Xu, "Atlas: A sequence-based learning approach for attack investigation," in *USENIX Security '21*, 2021.

[14] L. Yu, S. Ma, Z. Zhang, G. Tao, X. Zhang, D. Xu, V. E. Urias, H. W. Lin, G. F. Ciocarlie, V. Yegneswaran *et al.*, "Alchemist: Fusing application and audit logs for precise attack provenance without instrumentation." in *NDSS '21*, 2021.

[15] F. Dong, S. Li, P. Jiang, D. Li, H. Wang, L. Huang, X. Xiao, J. Chen, X. Luo, Y. Guo *et al.*, "Are we there yet? an industrial viewpoint on provenance-based endpoint detection and response tools," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2396–2410.

[16] Y. Shen and G. Stringhini, "{ATTACK2VEC}: Leveraging temporal word embeddings to understand the evolution of cyberattacks," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 905–921.

[17] J. Zengy, X. Wang, J. Liu, Y. Chen, Z. Liang, T.-S. Chua, and Z. L. Chua, "Shadewatcher: Recommendation-guided cyber threat analysis using system audit records," in *S&P '22*. IEEE, 2022.

[18] S. Wang, Z. Wang, T. Zhou, H. Sun, X. Yin, D. Han, H. Zhang, X. Shi, and J. Yang, "Threatrace: Detecting and tracing host-based threats in node level through provenance graph learning," *IEEE TIFS*, vol. 17, 2022.

[19] X. Han, T. Pasquier, A. Bates, J. Mickens, and M. Seltzer, "Unicorn: Runtime provenance-based detector for advanced persistent threats," *arXiv:2001.01525*, 2020.

[20] M. A. Nainna, J. Bass, and L. Speakman, "Cyber threat intelligence sharing in nigeria," *Communications of the IIMA*, vol. 22, no. 1, p. 1, 2024.

[21] M. R. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A literature review on mining cyberthreat intelligence from unstructured texts," in *2020 International Conference on Data Mining Workshops (ICDMW)*. IEEE, 2020, pp. 516–525.

[22] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates, "Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis," in *NDSS '20*, 2020.

[23] M. A. Inam and W. Ul, "Forensic analysis of configuration-based attacks," in *NDSS '22*, 2022.

[24] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting," in *CCS '19*, 2019.

[25] M. N. Hossain, S. Sheikhi, and R. Sekar, "Combating dependence explosion in forensic analysis using alternative tag propagation semantics," in *S&P '20*. IEEE, 2020.

[26] X. Chen, H. Irshad, Y. Chen, A. Gehani, and V. Yegneswaran, "{CLARION}: Sound and clear provenance tracking for microservice deployments," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3989–4006.

[27] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Chen, W. Cheng, C. A. Gunter *et al.*, "You are what you do: Hunting stealthy malware via data provenance analysis." in *NDSS*, 2020.

[28] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "Holmes: real-time apt detection through correlation of suspicious information flows," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1137–1152.

[29] W. U. Hassan, A. Bates, and D. Marino, "Tactical provenance analysis for endpoint detection and response systems," in *S&P '20*. IEEE, 2020.

[30] T. Pasquier, X. Han, T. Moyer, A. Bates, O. Hermant, D. Eyers, J. Bacon, and M. Seltzer, "Runtime analysis of whole-system provenance," in *CCS '18*. ACM, 2018.

[31] T. Pasquier, D. Eyers, and M. Seltzer, "From here to provtopia," in *VLDB Workshop on Data Management and Analytics for Medicine and Healthcare*. Springer, 2019, pp. 54–67.

[32] A. Goyal, G. Wang, and A. Bates, "R-caid: Embedding root cause analysis within provenance-based intrusion detection," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 257–257.

[33] Z. Cheng, Q. Lv, J. Liang, Y. Wang, D. Sun, T. Pasquier, and X. Han, "Kairos: Practical intrusion detection and investigation using whole-system provenance," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 3533–3551.

[34] U. Satapathy, R. Thakur, S. Chattopadhyay, and S. Chakraborty, "Disprotrack: Distributed provenance tracking over serverless applications," in *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*. IEEE, 2023, pp. 1–10.

[35] L. Yu, Y. Ye, Z. Zhang, and X. Zhang, "Cost-effective attack forensics by recording and correlating file system changes," in *33rd USENIX Security Sym@inproceedingsshen2019attack2vec, title={ATTACK2VEC}: Leveraging temporal word embeddings to understand the evolution of cyberattacks, author=Shen, Yun and Stringhini, Gianluca, booktitle=28th USENIX Security Symposium (USENIX Security 19), pages=905–921, year=2019 posium (USENIX Security 24)*, 2024, pp. 1705–1722.

[36] Z. Jia, Y. Xiong, Y. Nan, Y. Zhang, J. Zhao, and M. Wen, "{MAGIC}: Detecting advanced persistent threats via masked graph representation learning," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5197–5214.

[37] Y. Xie, D. Feng, and Y. Hu, "Pagoda: A hybrid approach to enable efficient realtime provenance based intrusion," *IEEE Transactions on Dependable and Secure Computing, 17 (6)*, 2020.

[38] "Event tracing for windows (etw)," 2024, https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw-.

[39] R. Yang, X. Chen, H. Xu, Y. Cheng, C. Xiong, L. Ruan, M. Kavousi, Z. Li, L. Xu, and Y. Chen, "Ratscope: Recording and reconstructing missing rat semantic behaviors for forensic analysis on windows," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1621–1638, 2020.

[40] C. Yagemann, M. A. Noureddine, W. U. Hassan, S. Chung, A. Bates, and W. Lee, "Validating the integrity of audit logs against execution repartitioning attacks," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3337–3351.

[41] J. Zeng, C. Zhang, and Z. Liang, "Palantír: Optimizing attack provenance with hardware-enhanced system observability," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3135–3149.

[42] A. Gehani and D. Tariq, "Spade: Support for provenance auditing in distributed environments," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2012, pp. 101–120.

[43] Y. Wu, Y. Xie, X. Liao, P. Zhou, D. Feng, L. Wu, X. Li, A. Wildani, and D. Long, "Paradise: real-time, generalized, and distributed provenance-based intrusion detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 2, pp. 1624–1640, 2022.

[44] S. Hussain, M. Musa, T. Neeshat, R. Batool, O. Ahmed, F. Zaffar, A. Gehani, A. Poggio, and M. K. Yadav, "Towards reproducible ransomware analysis," in *Proceedings of the 16th Cyber Security Experimentation and Test Workshop*, 2023, pp. 1–9.

[45] J. Zeng, Z. L. Chua, Y. Chen, K. Ji, Z. Liang, and J. Mao, "Watson: Abstracting behaviors from audit logs via aggregation of contextual semantics." in *NDSS*, 2021.

[46] K.-K. Muniswamy-Reddy, U. J. Braun, D. A. Holland, P. Macko, D. Maclean, D. W. Margo, M. I. Seltzer, and R. Smogor, "Layering in provenance systems," in *Proceedings of the 2009 USENIX Annual Technical Conference (USENIX'09)*. USENIX Association, 2009.

[47] Y. Xie, Y. Wu, D. Feng, and D. Long, "P-gaussian: provenance-based gaussian distribution for detecting intrusion behavior variants using high efficient and real time memory databases," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 6, pp. 2658–2674, 2019.

[48] Y. Xie, D. Feng, Y. Hu, Y. Li, S. Sample, and D. Long, "Pagoda: A hybrid approach to enable efficient real-time provenance based intrusion detection in big data environments," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 6, pp. 1283–1296, 2018.

[49] "Sysdig," https://sysdig.com/, 2025, (Accessed on 02/05/2025).

[50] P. Fang, P. Gao, C. Liu, E. Ayday, K. Jee, T. Wang, Y. F. Ye, Z. Liu, and X. Xiao, "Back-propagating system dependency impact for attack investigation," in *USENIX Security '22*, 2022.

[51] "strace," https://strace.io/, 2025, (Accessed on 02/05/2025).

[52] P. Datta, I. Polinsky, M. A. Inam, A. Bates, and W. Enck, "{ALASTOR}: Reconstructing the provenance of serverless intrusions," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2443–2460.

[53] "Analyzing the hidden danger of environment variables for keeping secrets," 2022, https://www.trendmicro.com/en_hk/research/22/h/analyzing-hidden-danger-of-environment-variables-for-keeping-secrets.html.

[54] "Auditbeat reference:," 2025, https://www.elastic.co/guide/en/beats/auditbeat/current/auditbeat-modules.html.

[55] E. Manzoor, S. M. Milajerdi, and L. Akoglu, "Fast memory-efficient anomaly detection in streaming heterogeneous graphs," in *KDD '16*, 2016.

[56] F. Liu, Y. Wen, D. Zhang, X. Jiang, X. Xing, and D. Meng, "Log2vec: A heterogeneous graph embedding based approach for detecting cyber threats within enterprise," *CCS '19*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:207958857

[57] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *CCS '17*, 2017.

[58] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li *et al.*, "Robust log-based anomaly detection on unstable log data," in *ESEC/FSE '19*, 2019.

[59] A. Bessadok, M. A. Mahjoub, and I. Rekik, "Graph neural networks in network neuroscience," *IEEE TPAMI*, vol. 45, no. 5, 2022.

[60] J. Tang, F. Hua, Z. Gao, P. Zhao, and J. Li, "Gadbench: Revisiting and benchmarking supervised graph anomaly detection," *NeurIPS '23*, vol. 36, 2023.

[61] M. Sills, P. Ranade, and S. Mittal, "Cybersecurity threat intelligence augmentation and embedding improvement-a healthcare usecase," in *ISI '20*. IEEE, 2020.

[62] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "Sok: Taxonomy of attacks on open-source software supply chains," in *S&P '23*. IEEE, 2023.

[63] Z. Li, J. Zeng, Y. Chen, and Z. Liang, "Attackg: Constructing technique knowledge graph from cyber threat intelligence reports," in *ESORICS '22*. Springer, 2022.

[64] "Snyk," 2024, https://snyk.io/blog/.

[65] "Microsoft security intelligence center," 2024, https://www.microsoft.com/en-us/security/blog/topic/threat-intelligence/.

[66] "The cybersecurity and infrastructure security agency," 2024, https://www.cisa.gov/.

[67] "Homepage of fead," 2024, https://sites.google.com/view/feadx.

[68] "Att&ck tactics, techniques, and procedures," 2024, https://attack.mitre.org/matrices/enterprise/.

[69] J. Li, Z. Jia, and Z. Zheng, "Semi-automatic data enhancement for document-level relation extraction with distant supervision from large language models," *arXiv:2311.07314*, 2023.

[70] L. Sun, K. Zhang, Q. Li, and R. Lou, "Umie: Unified multimodal information extraction with instruction tuning," in *AAAI '24*, vol. 38, no. 17, 2024.

[71] X. Chen, N. Zhang, X. Xie, S. Deng, Y. Yao, C. Tan, F. Huang, L. Si, and H. Chen, "Knowprompt: Knowledge-aware prompt-tuning with synergistic optimization for relation extraction," in *WWW '22*, 2022.

[72] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Comput. Surv.*, vol. 55, no. 12, 2023.

[73] Y. Zhang, Y. Li, L. Cui, D. Cai, L. Liu, T. Fu, X. Huang, E. Zhao, Y. Zhang, Y. Chen *et al.*, "Siren's song in the ai ocean: a survey on hallucination in large language models," *arXiv:2309.01219*, 2023.

[74] G. Feng, B. Zhang, Y. Gu, H. Ye, D. He, and L. Wang, "Towards revealing the mystery behind chain of thought: a theoretical perspective," *NeurIPS '24*, vol. 36, 2024.

[75] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *NeurIPS '22*, vol. 35, 2022.

[76] Z. Chu, J. Chen, Q. Chen, W. Yu, T. He, H. Wang, W. Peng, M. Liu, B. Qin, and T. Liu, "A survey of chain of thought reasoning: Advances, frontiers and future," *arXiv:2309.15402*, 2023.

[77] "Graph attention networks," in *ICLR '18*, 2018.

[78] "Darpa tc dataset," 2024, https://github.com/darpa-i2o/Transparent-Computing.

[79] "Cse-cic-ids2018," 2024, https://www.unb.ca/cic/datasets/ids-2018.html.

[80] W. W. Lo, S. Layeghy, M. Sarhan, M. Gallagher, and M. Portmann, "E-graphsage: A graph neural network based intrusion detection system for iot," in *NOMS '22*. IEEE, 2022.

[81] Y. Wu, N. Jiang, H. V. Pham, T. Lutellier, J. Davis, L. Tan, P. Babkin, and S. Shah, "How effective are neural networks for fixing security vulnerabilities," in *ISSTA '23*, 2023.

[82] Z. Chen, X. Hu, X. Xia, Y. Gao, T. Xu, D. Lo, and X. Yang, "Exploiting library vulnerability via migration based automating test generation," in *ICSE '24*, 2024.

[83] M. S. Haq, T. D. Nguyen, A. Ş. Tosun, F. Vollmer, T. Korkmaz, and A.-R. Sadeghi, "Sok: A comprehensive analysis and evaluation of docker container attack and defense mechanisms," in *S&P '24*. IEEE, 2024.

[84] B. Ruan, J. Liu, W. Zhao, and Z. Liang, "Vulzoo: A comprehensive vulnerability intelligence dataset," *arXiv:2406.16347*, 2024.

[85] G. Wang, W. Zhang, L. Yan, L. Tang, H. Qu, K. Liu, Y. Zhao, and H. Li, "Research on automated anomaly localization in the power internet of things based on fuzzing and semantic analysis," in *IoTML '23*, vol. 12937. SPIE, 2023.

[86] N. Becker, D. Reti, E. V. Ntagiou, M. Wallum, and H. D. Schotten, "Evaluation of reinforcement learning for autonomous penetration testing using a3c, q-learning and dqn," *arXiv:2407.15656*, 2024.

[87] "Azure openai service," 2024, https://azure.microsoft.com/en-us/products/ai-services/openai-service.

[88] "Spec2006," 2024, https://www.spec.org/cpu2006/.

[89] M. Hassan, C. H. Park, and D. Black-Schaffer, "A reusable characterization of the memory system behavior of spec2017 and spec2006," *ACM TACO*, vol. 18, no. 2, 2021.

[90] "Stream benchmark," 2024, https://www.amd.com/en/developer/zen-software-studio/applications/spack/stream-benchmark.html.

[91] I. Peng, R. Pearce, and M. Gokhale, "On the memory underutilization: Exploring disaggregated memory on hpc systems," in *SBAC-PAD '20*. IEEE, 2020.