SmartGuard: Leveraging Large Language Models for Network Attack Detection through Audit Log Analysis and Summarization

Hao Zhang, Shuo Shao, Song Li, Zhenyu Zhong, Yan Liu, Zhan Qin, Kui Ren, *Fellow, IEEE*

***Abstract*— End-point monitoring solutions are widely deployed in today's enterprise environments to support advanced attack detection and investigation. These monitors continuously record system-level activities as audit logs and provide deep visibility into security events. Unfortunately, existing methods of semantic analysis based on audit logs have low granularity, only reaching the system call level, making it difficult to effectively classify highly covert behaviors. Additionally, existing works mainly match audit log streams with rule knowledge bases describing behaviors, which heavily rely on expertise and lack the ability to detect unknown attacks and provide interpretive descriptions. In this paper, we propose SmartGuard, an automated method that combines abstracted behaviors from audit event semantics with large language models. SmartGuard extracts specific behaviors (function level) from incoming system logs and constructs a knowledge graph, divides events by threads, and combines event summaries with graph embeddings to achieve information diagnosis and provide explanatory narratives through large language models. Our evaluation shows that SmartGuard achieves an average F1 score of 96% in assessing malicious behaviors and demonstrates good scalability across multiple models and unknown attacks. It also possesses excellent fine-tuning capabilities, allowing experts to assist in timely system updates.**

***Index Terms*—Audit log analysis, network attack detection, knowledge graph, large language model.**

I. Introduction

SECURITY incidents in large enterprise systems have been on the rise globally. We have been witnessing the scale and complexity of attacks. Capital One reported that 106 million customers' credit card information was exposed due to unauthorized database access [1]. The recent Twitter attack left dozens of high-profile accounts displaying fraudulent messages to tens of millions of followers [2]. To better prevent and respond to such attacks, endpoint monitoring solutions (e.g., Security Information and Event Management (SIEM) tools [3]) are widely deployed for enterprise security. These monitors continuously record system-level activities as audit logs, capturing many aspects of system execution states.

When responding to security incidents, network analysts perform the causal analysis on audit logs to uncover the root causes of attacks and their scope of damage [4], [5]. However, the volume of audit logs generated by normal systems is not small. Even a desktop computer can easily produce over a million audit events per day [6], [7], not to mention busy servers in cloud infrastructures. To overcome this challenge, recent research solutions have expanded causal analysis by eliminating irrelevant system operations from audit logs [7], [8], [9], [10], [11], [12], [13]. Another research direction aims to improve the efficiency of log query systems [6], [14], [15]. Unfortunately, neither data reduction nor search improvements lead to a significant reduction in the analysis workload. These solutions do not capture the semantics behind the audit data and leave behavior identification to the analysts [16]. Therefore, a substantial amount of manual work is still required to assess relevant but benign and complex events that dominate audit logs. In particular, a major issue analysts face is how to bridge the semantic gap between audit events and system behaviors, thereby achieving more efficient and broader applications in behavior detection.

Existing work aims to bridge this gap by matching audit events with knowledge stores of expert-defined rules describing behaviors, such as tag-based policies [17], [18], query graphs [19], [20], and TTP (Tactics, Techniques, and Procedures) specifications [21], [22]. Essentially, these solutions identify high-level behaviors through tag propagation or graph matching. However, the anticipated bottleneck is the manual involvement of domain experts in specifying such rules. For example, MORSE [18] requires experts to traverse system entities (e.g., files) and initialize their confidentiality and integrity tags for tag propagation. TGMiner [20] requires manual behavior labeling in training log sets before mining distinguishing behavior patterns and searching for their presence in test sets. Despite playing a crucial role in audit log analysis, mapping events to behaviors largely relies on expert knowledge, which may hinder its practical application.

Extracting representative behaviors from audit events for analyst investigation can provide an effective strategy to mitigate this issue. However, the accuracy of abstracting high-level behaviors depends on the matching degree of templates or the universality of similarity algorithms. When facing a large number of heterogeneous audit logs, both templates and similarity algorithms are difficult to achieve good universality.

Recently, the success of large language models in performing complex tasks [3], [17] has provided a promising approach to enhancing abstract high-level behavior analysis. Specifically, large language models can be used to parse large amounts of data, identify relevant information, and generate concise and insightful outputs. This significantly reduces the

Hao Zhang, Shuo Shao, Song Li, Zhan Qin, and Kui Ren are with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, 310007, China, and also with the Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou, 310051, China (e-mail: {haozhang_hz, shaoshuo_ss, songl, qinzhan, kuiren}@zju.edu.cn).

Zhenyu Zhong and Yan Liu are with the Ant Group, Hangzhou, 310063, China (e-mail: {edward.zhong, bencao.ly}@antgroup.com)

burden on analysts to manually sift through large volumes of data, helping them resolve incidents more quickly and effectively. Additionally, large language models can adapt to new and evolving types of incidents, learning from past data to improve future predictions. For example, the OPT model employs an efficient training method, with costs only 1/7 of those of language models with the same parameters, making it well-suited for large-scale inference tasks [23].

More specifically, we can use large language models to automatically abstract high-level behaviors and interpret and label semantically similar behaviors, even without labels to explain what they are. However, since repetitive/comparable behaviors have already been labeled, analysts only need to review the representatives in the labels, resulting in far fewer incidents needing investigation. In addition to reducing the manual workload in behavior analysis, using large language models for automatic behavior abstraction can also enable proactive analysis to detect anomalous behavior patterns in internal threats or external vulnerabilities.

Although adopting large language models for behavior abstraction analysis sounds promising, it also faces the following challenges: complex event semantic differentiation and behavior recognition, as audit events record general system activities and thus lack high-level semantic knowledge. A single event (e.g., process creation or file deletion) can represent different semantics in different contexts. Additionally, due to the large scale and high interleaving of audit events, dividing events and identifying behavior boundaries is like finding a needle in a haystack. Large language models lack intrinsic domain-specific knowledge, especially in specialized fields such as audit log analysis. This lack of understanding of specific contexts may limit their accuracy in predicting incident root causes and generating appropriate explanations.

In this paper, we propose SmartGuard, an automated behavior abstraction and detection method that bridges the semantic gap of audit events and fine-tunes large language models to detect benign and malicious behaviors and provide explanatory narratives. It does not rely on expert knowledge of event semantics to perform behavior abstraction. The semantics are automatically obtained from the context in which the events are used in the audit logs, which we refer to as the contextual semantics of events. More specifically, SmartGuard first preprocesses the input audit logs, including removing redundancies and extracting the subjects, objects, and relationships of behaviors. Unlike previous solutions, it emphasizes and incorporates specific operational behaviors (such as functions) into the records and categorizes the logs according to threads. Therefore, it can obtain more detailed deep behavior characteristics compared to previous solutions. Then, SmartGuard generates natural language summaries for each unit based on the divided units and uses a Graph Neural Network (GNN)-based embedding model to generate Graph Embeddings of behavior semantics according to the context of the logs and divided units. After that, it identifies events connected to related data objects (i.e., the same thread) and aggregates their semantics into representations of high-level behaviors. Finally, SmartGuard fine-tunes the large language model based on the abstracted high-level behavior representations, achieving information diagnosis of log behaviors and providing interpretable narratives.

SmartGuard provides a way to combine large language models with abstract log high-level behaviors. Through interpretable methods, it can reason about behaviors and events, allowing analysts to compare and resolve behaviors. Utilizing the characteristics of large language models, it can also combine multiple behaviors and even predict what specific behaviors should be. These features can form the basis for designing new security solutions, such as anomaly behavior detection, or support existing solutions in selecting appropriate behaviors for in-depth inspection.

We prototyped SmartGuard and used the public DARPA TC dataset released by the DARPA Transparent Computing program [24] to evaluate SmartGuard's accuracy and interpretability in attack investigations. We also conducted comparative experiments on step processing and noted that SmartGuard achieved better results compared to not dividing threads and not using graph embedding model semantics. The experimental results show that SmartGuard accurately identifies system entities with similar usage contexts and achieves an average F1 score of 95.2% in behavior abstraction. And it demonstrates good scalability, achieving favorable results across multiple models.

Our contributions are summarized as follows.

- We propose SmartGuard, a method that combines large language models with the extraction of behaviors from logs, enabling it to effectively detect both known and unknown attacks. Our approach summarizes behaviors guided by information flow and enables large language models to perform information diagnostics by aggregating contextual semantics.
- We propose a new scheme for abstracting contextual semantics, achieving fine granularity down to specific behaviors (function level), and dividing events by threads. This allows integration with large language models to provide explanatory narratives for behavior semantics.
- We conducted a systematic evaluation and scalability analysis of common malicious behaviors. The results show that SmartGuard can effectively abstract high-level behaviors and detect them, demonstrating good scalability across multiple models and unknown attacks. Additionally, it can efficiently collaborate with experts for fine-tuning and real-time system updates.

II. Background and Motivation

In this section, we first introduce audit log analysis and its challenges with an example. We then analyze the problem of behavior abstraction with our insights, as well as describe the threat model.

A. Motivating Example

1) Scenario: Consider the attack scenarios of Barephone Micro. As an attacker, you write a malicious application APK intending to steal database files. The victim user installs and runs the malicious APK, which loads the Micro APT shared object. Micro APT connects to `77.138.117.150:80`

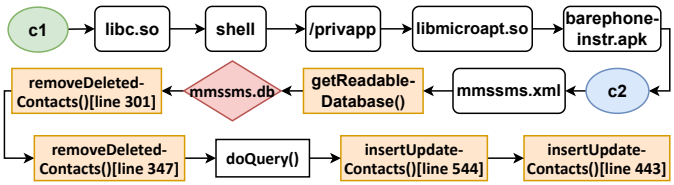

Fig. 1: Scenario example. The nodes in the figure are system entities (rectangles represent functions, rounded rectangles represent addresses and files, ellipses represent sockets, and diamonds represent databases). The edges between the nodes represent system calls. For clarity, we color-code the source data objects in the behavior, with red and yellow representing high-risk behaviors.

as `C2`. The address `128.55.12.114` is used as `C1`, where a benign activity installs an elevation driver and uses the driver for privilege escalation. Finally, the new permissions were used to call `getReadableDatabase` to steal the database files `mmssms.db`. Multiple calls to `removeDeletedContacts` were then made to clear the relevant information from the database. Although this strategy executes high-risk functions multiple times, traditional schemes do not deliberately record specific function behaviors, leading analysts to mistakenly classify it as a normal event. Figure 1 shows the behavior diagram of the strategy.

2) Audit log analysis: System audit logs enable analysts to gain deep insights into network attacks through data sources. Each audit record represents an operating system-level operation (i.e., system call), such as process execution, file creation, and network connection. Specifically, records can be defined as triples (Object, Relation, Object), where Object is a system entity (i.e., process, file, or network socket), and Relation is the system call function. We aggregate records of the same thread within the same process as an event. Each event belongs to a specific thread (marked at the beginning of the event). Note that system entities are associated with a set of identifying attributes, such as tags (e.g., PID) and names (e.g., file path and function). Additionally, each individual record (e.g., the action of writing to a file) represents an information flow between the subject and the object.

3) The Promise of Large Language Models for Incident Inference: Large language models have the ability to infer and analyze event semantics. The rapid advancements in natural language processing and machine learning have led to the development of powerful large language models (LLMs). These models have been reported to be effective in various downstream tasks in zero-shot and few-shot learning scenarios [25], [26], [27]. They have demonstrated exceptional performance in translation, summarization, and semantic understanding. Leveraging the reasoning capabilities of LLMs can transform the way behaviors are identified and detected in audit log analysis. The ability of LLMs to provide explanatory narratives can help alleviate the stress and burden associated with complex tasks for analysts, allowing them to focus more on higher-level work and decision-making.

B. Challenges

When capturing attack sequences and sources, analysts must identify not only malicious behaviors (such as data breaches) but also benign behaviors (such as file uploads). Although provenance graphs provide an intuitive representation to visualize causal dependencies and remove irrelevant events, analysts still spend excessive time investigating relevant but benign events due to the ubiquity of daily activities.

Abstracting behaviors from audit events is an effective strategy for analysts to navigate through a large number of events and focus on specific information. Essentially, behaviors represent an abstraction of audit data. Working at the behavioral level can effectively reduce the analysis workload from the entire event space to the interesting behaviors that attract attention in specific scenarios. However, to automatically abstract high-level behaviors from low-level audit events and accurately classify them, analysts face three major challenges:

- *Refining the semantics of audit events.* Audit records are usually based on processes as the smallest unit, but they overlook the specific functions executed in each step of behavior. This can make certain malicious behaviors difficult to detect. For example, the theft of files from two entities with different names but similar semantics may not be recognized.
- *Inferring the semantics of audit events and identifying behavior boundaries.* Audit events record detailed system execution states but lack the high-level semantic knowledge crucial for recognizing behavior patterns. For example, system entities with the same name might indicate different intentions. Existing work mainly uses expert-defined rules or model knowledge bases to parse audit events to reveal event semantics. However, given the large scale of audit events, manual specifications can easily compromise the scalability of abstracting high-level behaviors, even in moderately sized systems. Additionally, audit data volumes are typically enormous, and audit events are highly interleaved. For example, a single package installation using APT can generate over 30,000 events. Furthermore, all individual behaviors have causal relationships. This makes it difficult for analysts to segment events and distinguish behavior boundaries.
- *Identifying unknown attacks and achieving timely system updates.* Although existing solutions can recognize known attack behaviors, unknown attacks occur more frequently in reality. Unlike known attack types, learning the semantic relationships between behaviors and recognizing unknown attacks is a challenge. More importantly, real-world systems should be capable of interacting with security personnel and promptly updating the types of attacks they recognize.

C. Problem Analysis

Given a large number of audit logs in a user login session, our goal is to identify high-level (benign and malicious) behaviors and provide a quantitative representation of their semantics without analyst involvement. Additionally, we aim to use large language models to identify and provide interpretable narratives for abstract high-level behaviors. Compared to traditional methods of behavior abstraction that heavily

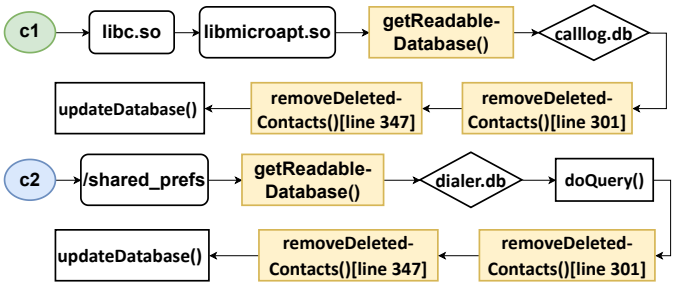

Fig. 2: Attack subgraph for stealing information from different databases. We color-coded the data objects, with yellow indicating similar behavior semantics.

rely on domain knowledge, our goal is to achieve automated behavior abstraction using simple and effective insights.

Our first insight is that function-level behavior recording and thread-based log division can identify more covert log behaviors. This insight comes from: the observation that in traditional audit log analysis, each audit record is generally processed into a triplet (Subject, Relation, Object), where the Subject is the process entity, the Object is the system entity, and the Relation is the system function. Additionally, the Object contains a set of identifying tags. For example, `(cp, read, test.txt)` indicates that the copy process performed a read operation on the file `test.txt`. This triplet division is based on processes. This method can effectively and concisely extract a high-level behavior, but it overlooks more detailed parts of the behavior, such as the specific address of the file and the function usage process in each record. This is because parameters like functions are recorded in the Object tags and are not intuitively reflected in the semantics. In malicious behaviors, besides sensitive and special files, the semantics of certain function usages should also be key monitoring targets. For example, an attacker wanting to steal the contents of the `calllog.db` file can elevate their privileges using an elevation driver and then call the function `CallLogDatabaseHelper.getReadableDatabase()` at the address `/android.providers.contacts/databases` to complete the malicious behavior. Similarly, stealing other files such as `dialer.db` would involve similar behavior. Figure 2 shows similar behaviors in the events. This type of attack is difficult to detect with traditional methods. Additionally, we observed that such behaviors often occur within the same thread space, and the traditional process-based division makes it difficult to reflect the overall characteristics of behaviors within the same thread. Therefore, thread division should also be an important part of audit log analysis.

Our second insight is that the semantics of system entities and relations in audit events can be revealed from their usage context. Similarly, for the other files such as `calllog.db`, the read operation will call the `getReadableDatabase()` function. Even though different behaviors include other nodes, we can still determine their similarity from the overall context. Therefore, we can infer that despite different identifiers, these files may share similar semantics.

The core idea is to reveal the semantics of system entities and relations from the contextual information in audit events, such as by analyzing their correlations in events. A general method for extracting such contextual semantics is to use graph embedding models. The goal is to map system entities and relations into a graph embedding space (i.e., numerical vector space), where the distances between vectors capture semantic relationships. At the same time, generating log summaries from segmented events and integrating them with graph vectors becomes the event semantics. Now we can interpret the semantic information of audit events. The next step is to identify audit events that belong to malicious behaviors.

Our third insight is that large language models can be used to identify high-level log semantics. Based on our observations, large language models have achieved remarkable success in inference tasks. Although large language models have limited knowledge in the area of audit logs, their superior performance in fine-tuning allows for improved semantic recognition through the *Chain-Of-Thought (COT)* approach. For example, by asking guiding questions such as 'What are the sensitive links in the log behavior?', 'What are the sensitive nodes in the log behavior?', and 'What is the malicious type of the log behavior?', we can enhance the accuracy and scalability of large language models in recognizing log semantics.

D. Threat Model

We assume that the underlying operating system, audit engine, and monitoring data are part of the Trusted Computing Base (TCB). Ensuring the integrity of the operating system kernel, endpoint monitors, or the audit logs themselves is beyond the scope of this work. This threat model is shared in related work on system auditing [28], [29], [17], [18], [30], [22]. We also assume that behaviors are audited at the kernel level and their operations are captured as system call audit logs. Although attackers may attempt to perform malicious actions without executing any system calls to hide their tracks, such behaviors appear to be rare and their impact on other parts of the system is limited [31]. In this paper, we focus on behaviors within single-user sessions. Our insights generally apply to cross-session behaviors.

III. Methodology

In this section, we will introduce the Approach Overview of SmartGuard in Section III-A, followed by Knowledge Graph Construction in Section III-B. Next, we will discuss Behavior Abstraction in Section III-C. Finally, we will introduce LLM for Incident Inference in Section III-D.

A. Approach Overview

The overall approach of SmartGuard is illustrated in Figure 3. It consists of three main stages: Knowledge Graph Construction, Behavior Abstraction, and LLM for Incident Inference. SmartGuard takes system audit data as input, such as Linux audit logs [32]. It summarizes behavior instances, abstracts their high-level semantics, and ultimately outputs diagnostic results and explanatory narratives.

Specifically, taking audit logs from user sessions as input, the Knowledge Graph Construction module first parses the

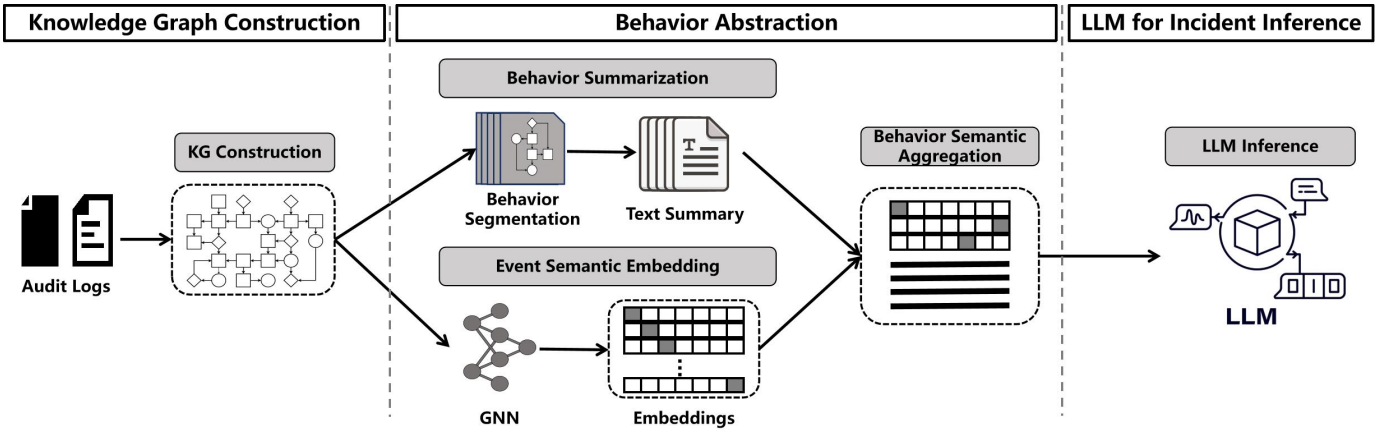

Fig. 3: SmartGuard Overview. First, we extract specific behaviors (function-level) from the logs and construct a knowledge graph. Second, we divide the behaviors according to threads and extract text summaries. Then, we perform embeddings on the extracted behavior subgraphs and combine them with text to form behavior semantics. Finally, we use a large language model to diagnose the behavior semantics and provide explanatory narratives.

logs into triples, divides them by threads, and constructs a log-based Knowledge Graph (KG). Then, the Event Semantic Inference module employs a graph neural model to infer the contextual semantics of nodes in the knowledge graph. Meanwhile, the Behavior Summarization module enumerates subgraphs from the knowledge graph to generate textual summaries of the logs. Combining node semantics, the Behavior Semantic Aggregation module subsequently enhances the subgraphs to encode the semantics of behavior instances. Finally, the Large Language Model (LLM) module performs information diagnosis based on the aggregated semantics and provides explanatory narratives, which can further reduce the workload of downstream tasks. We will introduce the design details of SmartGuard in the following sections.

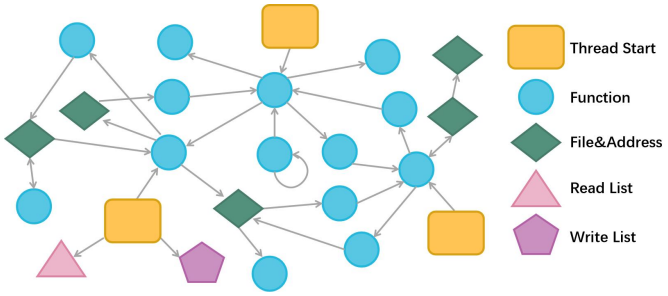

Fig. 4: Example of Graph Construction. We use different colors and shapes to mark different nodes, corresponding to the node types in Table I. The lines between the nodes represent system calls.

B. Knowledge Graph Construction

To analyze the contextual semantics of events, a unified representation is required to present heterogeneous events in a homogeneous manner. We use a knowledge graph (KG)-based representation to integrate heterogeneous information. This allows for capturing relationships beyond provenance in the future [33], [34] (e.g., file metadata such as permissions and owners).

TABLE I: Classification of node types and some specific examples of nodes.

Node Type	Node Name
Thread Start	thread id
Function	query(),setThreadPriority(),execTransact()
File or address	libc.so,dialar.db,/system,/dev/ashmem
Readlist	printlnnative
Writelist	writeEvent

Following the formal description of KG by Färber et al. [35], we define the log-based KG as a Resource Description Framework (RDF) graph [36]. More specifically, the log-based knowledge graph is a set of numerous semantic triples. Each triple corresponds to an audit event and consists of three elements, encoding the semantic relationship between *Head* and *Tail* in the form of (*Head*, *Relation*, *Tail*). Both *Head* and *Tail* can be any type of system entity, mainly including specific file addresses, functions, read/write lists, etc. *Relation* can be any system operation performed on *Tail*, such as READ, LINK, and MODIFY_FILE_ATTRIBUTES. Table I presents some examples of our classification of node types. Additionally, we divide the logs by threads, with each thread subgraph starting with a thread ID and connecting under the process ID that created them. Threads are interconnected through the same entities, such as the same addresses and functions. Figure 4 illustrates an example of one of the attack behavior subgraphs.

C. Behavior Abstraction

1) Behavior Summarization: The first step in behavior abstraction is to identify the semantics of behavior instances from a user login session. We define a behavior instance as a coherent set of audit events that operate on related data. Specifically, accurate understanding depends on appropriate representation and granularity, allowing effective comparison of behavior semantics. Common practices in previous works

[37], [38], [39] are to formulate each log event as the basic unit of analysis or to use individual elements within log events as the basic unit. However, a single audit event includes three elements (Head, Relation, and Tail), and analyzing only from the event or element itself can only obtain the semantics of the one-step behavior from Head to Tail, while the overall semantics of longer behavior trajectories will be fragmented. Meanwhile, we observe that behavior thematic features occur within the same thread, and the thread size is relatively small, mostly containing less than ten events, which can also reflect the semantics of elements compared to the analysis method of single events. Therefore, performing semantic analysis at the thread level can provide a more detailed view than single events, as the context of the entire thread behavior becomes clear. We aggregate events within a thread into thread events. Therefore, summarizing the problem of a single behavior instance can be simplified to extract causally connected subgraphs with thread objects as the root in the session KG. Additionally, considering that a single path may not retain the complete context representing multi-branch data transmission behavior, we also judge whether multiple thread behavior branches need to be considered.

Eigenvector centrality is a method for measuring the importance of nodes in a graph. It considers the importance of a node's neighbors and calculates the centrality value of each node through an iterative method. The formula for eigenvector centrality is:

$$C_E(v) = \frac{1}{\lambda} \sum_{u \in \mathcal{N}(v)} A_{vu} C_E(u). \tag{1}$$

$C_E(v)$ represents the eigenvector centrality of node v. λ is the eigenvalue. $\mathcal{N}(v)$ is the set of neighbors of node v. A_{vu} is the element in the adjacency matrix A that represents the connection between node v and node u. If there is an edge between node v and node u, then $A_{vu} = 1$; otherwise, $A_{vu} = 0$. $C_E(u)$ represents the eigenvector centrality of node u.

For each thread, we first construct the adjacency matrix A of the graph, then calculate the eigenvalues λ and eigenvectors $\mathbf{v}$ of the adjacency matrix, and finally select the eigenvector corresponding to the largest eigenvalue as the key node. Then, We search within a radius of k from the key node, and if there are key nodes from other threads, those threads are also included in the behavior instance; otherwise, the behavior instance only includes the current thread.

This method is adopted considering the following situation: Traditional methods generally use search methods such as Depth-First Search (DFS) to traverse the graph when extracting subgraphs of behavior instances. However, the traversal method is influenced by the frequency of node occurrence, but the frequency does not indicate the importance of the node, resulting in some redundant parts being added to the subgraph. For example, `libc.so` is an important dynamic library file. It is an implementation of the C standard library (libc), providing many basic system calls and functions, and is frequently referenced. Because it is too common, the traditional method often adds `libc.so` to the behavior subgraph during the traversal process, but such nodes are not critical to the behavior and do not have a significant impact on the diagnosis of the behavior, so they should not be added to the behavior subgraph.

Additionally, we also observe that malicious behavior generally ends within a few steps. Therefore, we believe that only when the radius k of a key node contains other key nodes, the behaviors of the two threads are deeply related; otherwise, the behavior in the other thread does not have a significant impact on the behavior of this thread.

Afterward, we first divide the behavior according to the threads, then expand the search to form new behaviors. If one behavior is a subset of another, the two behaviors will be further merged. Then, we generate corresponding natural language summaries based on the behaviors. Considering that we will eventually use large language models to achieve information diagnosis, and the graph embedding vectors do not contain text vector content recognizable by language models, we need to provide text information to the large language model. Thanks to the thread division scheme adopted by SmartGuard, there is not much node information within each division (generally no more than ten). Large language models such as GPT and Copilot can be used to convert graph information into natural language summaries. For example, *(path1, read, getReadableDatabase(), write, b.txt)* can be organized as *'path1 uses getReadableDatabase() to achieve read, writing to b.txt'*.

In summary, we apply a method based on key nodes and thread division to divide the session's KG into subgraphs, where each subgraph describes a behavior instance and generates a natural language summary for each subgraph.

2) Event Semantic Embedding: Understanding the semantics of audit events is the second step in abstracting high-level behaviors. We abstract the semantics of subgraphs according to the divided behavior instances. Although there is a trade-off between scalability and accuracy due to different granularities of semantic analysis, choosing a computationally efficient embedding model allows us to maintain accuracy when handling the large number of events found in logs. Therefore, we choose behavior instances rather than audit events as the basic unit in semantic reasoning.

Since the embedding model can learn the semantics of audit events from the context information of threads, the next question is how to map thread events to vector space. In natural language processing, word embeddings have been successfully used to extract and represent the semantics of words [40], [41]. Inspired by the success of word embeddings in NLP, EKLAVYA [42] demonstrated how to apply it to infer the semantics of binary instructions based on usage context. Zeng et al. [43] showed that mapping contextual semantics into a translation-based embedding model achieved good results. This indicates that the context in which audit events occur is related to their semantics. For example, the triplets *(path1, read, getReadableDatabase())* and *(path2, read, getReadableDatabase())*, although path1 and path2 belong to different events, the context in which *(read, getReadableDatabase())* occurs suggests that they may share similar semantics. Intuitively, our goal is to transform each thread event into a vector, and we expect the embeddings of similar behaviors to be close to each other. To achieve this, we use a GAT model based on

graph neural networks to learn the mapping from thread events to embedding space.

Graph Attention Networks (GAT) are implemented by stacking simple graph attention layers, introducing self-attention mechanisms during propagation, and calculating the hidden state of each node by focusing on its neighboring nodes. Compared to general GCN models, GAT can consider the feature vectors of edges. Therefore, in the GAT model, the graph embedding space can describe the semantic relationships between thread behaviors. Specifically, the GAT model can consider the feature vectors of nodes and edges, and based on the provenance graph, it can autonomously learn and assign weights to edges and nodes. Our guiding principle for choosing GAT is that its graph attention-based model perfectly matches our understanding of the context semantics of audit events. For example, *(path1, read, getReadableDatabase(), write, a.txt)* and *(path2, read, getReadableDatabase(), write, b.txt)* exhibit similar semantics in GAT. Theoretically, the embedding model in GAT reflects our expectations of the semantics and similarities of thread events. Section IV-B demonstrates through experiments that GAT indeed learns the context semantics of audit events that match our domain knowledge.

During embedding, we used the word2vec model. We used the Gensim [44] library to generate node embeddings for training the graph. Gensim is a Python library for natural language processing, particularly adept at handling large-scale text data. Since each node in the source graph contains two attributes, type and name, the input feature f_u of the node u can be calculated as follows:

$$f_u = \text{Concat}(w_u^{\text{type}}, w_u^{\text{name}}), \tag{2}$$

where w_u^{type} is the vector representation of the word for the type of node u, such as a function or address, and w_u^{name} is the vector representation of the word for the name of node u. Table I presents some examples of our classification of node types.

The result of word2vec is an embedding vector that expresses the features of each node. We store it as the input to the graph neural network.

Next, we train the GAT model. Considering the difficulties brought by overly complex models, we prefer to implement our model with a simple network structure. We use contrastive learning in unsupervised learning to train the GAT model. During training, GAT minimizes the distance to the nearest node of the same type found in the KG, while maximizing the distance to randomly sampled nodes in the KG that are not connected and not of the same type. The loss function for optimizing the embedding model is:

$$\mathcal{L} = -\sum_{(i,j)\in\mathbf{KG}} \log\sigma(\mathbf{h}_i, \mathbf{h}_j) - \sum_{(i,j)\notin\mathbf{KG}} \log(1-\sigma(\mathbf{h}_i, \mathbf{h}_j))], \tag{3}$$

where $\mathbf{KG}$ is the set of edges in the graph. $\mathbf{h}_i$ and $\mathbf{h}_j$ are the embedding vectors of nodes i and j, respectively. $\sigma(\cdot)$ represents the cosine similarity function, which maps the cosine similarity to the range [0, 1] for easier computation.

Additionally, we also optimize the attention pooling model. This is a single-layer Fully Connected layer (FC) that obtains the vector representation of the entire graph by performing a weighted sum of the node embedding vectors. We use the same loss function as in Equation (2) and optimize it together with the GAT model during training.

In summary, the result of GAT is an $n \times m$ embedding matrix that maps n nodes into an m-dimensional embedding space. In our example, m is 128. Afterward, through the pooling model, we can obtain the graph embedding of the entire thread event.

3) Behavior Semantic Aggregation: After summarizing the behavior instances, we proceed to extract the semantics of the behavior instances. Recall that each behavior instance consists of audit events, whose semantics have been represented using high-dimensional vectors in an embedding matrix. We then naturally derive the semantics of the behavior instances by combining the partitions of the behavior instances. Additionally, natural language summaries of the behavior instances are generated to provide textual information for the semantics. To obtain the semantic representation of the final combined behavior instances, a simple approach is to aggregate the text summaries of the constituent events with the embedding vectors. However, this method is not suitable when the text summaries are too long, as it would cause the summaries and vectors to have a biased influence on the final judgment of the large language model, whereas they should actually contribute equally. Therefore, to further simplify the behavior summaries, we integrate the noise events in the behavior.

Noise events. It is also known as trivial events, are file operations that are periodically executed by an operation. Examples of such events include those used for file edit history caching *(vim, write, .viminfo)* and for shell program settings *(bash, read, /etc/profile)*. We classify trivial events as noise events because they are related to system routines rather than specific behaviors. Typically, trivial events have two characteristics: (a) they always occur for a given action, and (b) their occurrence order is fixed. To identify and filter them, we first enumerate all possible operations that a program can perform. We find that the vast majority of trivial events mainly involve reading and writing specific files, which are meaningless in themselves. Therefore, we add Read-List and Write-List nodes at the beginning of each behavior graph, then extract the trivial events from the graph and uniformly add them to the labels of these two nodes. We note that NODEMERGE [13] first proposed identifying common events (i.e., data access patterns) to reduce data. However, it focuses on file read operations during process initialization (e.g., loading libraries and retrieving configurations). In contrast, SmartGuard targets all types of file operations in more general operations.

After denoising the behavior instances, the same noise behaviors will no longer appear in the event behavior summaries, significantly shortening the length of the text summaries without affecting the expression of key information. In the embeddings of the behaviors, although there are Read-List and Write-List nodes that can express certain trivial event semantics, they do not affect the importance of the overall behavior embeddings.

Next, we aggregate the behavior text summaries with the

Instruction: The following description shows the log information of the event. It contains a summary of the log text and behavior Embeddings. Please focus on the Embeddings to determine whether this event is a malicious event. If it is, please provide the classification of the malicious event.
Input: The following is a summary of the log: [*Behavior Text Summary*], where the key nodes are [*Key Node*], followed by the embedding of the behavior graph [*Graph Embedding*].
Output: No. / Yes, the category is [*Attack Type*].

Fig. 5: The prompt to predict incident category.

embedding vectors. We construct the following sentence pattern: 'The following is a summary of the log: [*behavior text summary*], where the key point is [*Key Node*], followed by the embedding of the behavior graph <*Graph Embedding*>.' By simply aggregating and adding prompt language, we finally generate a log behavior description that the large language model can understand. Figure 5 shows the input format of the large language model during behavior inference.

In summary, the behavior abstraction phase uses a log-based knowledge graph as input to generate corresponding text summaries and graph embedding vectors, which are ultimately aggregated into descriptive information that the large language model can reason with.

D. LLM for Incident Inference

Recently, LLMs have demonstrated exceptional capabilities in understanding the context of downstream tasks and generating relevant information from demonstrations, making them a potential choice for event reasoning. However, inferring the root cause of events is not a simple task, and LLMs may struggle with long-tail or domain-specific tasks without any guidance [45], [46]. Chain of Thought (CoT) prompting is a gradient-free technique that guides LLMs to generate intermediate reasoning steps leading to the final answer. In few-shot CoT prompting, some manual demonstrations consist of questions and chains of reasoning, with the reasoning chain providing answers for each question. Inspired by the above idea, SmartGuard processes behavior abstraction information to be used as components in the event reasoning process.

In CoT prompts, multiple demonstrations include a question and a reasoning chain with a guiding answer. Drawing inspiration from the automatic prompt construction for reasoning chains [47], we can treat the summarized diagnostic information and labeled root cause categories as questions and reasoning. Therefore, finding the sensitive node links in behavior is an automatic reasoning chain construction, which fits well with the CoT prompt context. Note that we use graph embeddings and text summaries of event behavior to perform the reasoning process, and use the corresponding summarized information as part of the demonstration in the prompt. We construct the prompt as shown in Figure 5, asking the LLM to judge the malignancy and sensitive links of the current event, and we explicitly push the LLM to reason by using the 'give your explanation' instruction in the prompt.

In summary, the reasoning stage of large language models diagnoses abstract behavioral information and provides explanations through the CoT (Chain of Thought) concept.

IV. Evaluation

In this section, we use the DARPA TC EN5 dataset and experimentally evaluate four aspects of SmartGuard: 1) implementation; 2) the interpretability of event semantic inference; 3) the accuracy and scalability of behavior inference; 4) the interpretability of LLMs and solutions for hallucinations

A. Implementation

SmartGuard is implemented in Python 3.8 with around 2.5K lines of code (LoC). In this section, we discuss important technical details in the implementation.

Audit Log Input Interface. SmartGuard takes system audit data as input. We defined a general interface for audit logs and built input drivers to support different log formats, such as the Linux Audit [32] format (cdm20 and DARPA dataset).

Knowledge Graph Construction. To construct a log-based Knowledge Graph (KG), SmartGuard first sorts audit events in chronological order. Then, it divides the records of different threads according to the thread ID. It then converts each event into a triplet using system entities as the Head and Tail, and system call functions as the Relation. To facilitate the comparison of different nodes, we classify each type of element, such as file_path and function in system entities, and execute, link, check_attribute in system calls. Therefore, in the graph neural network, different weights can be assigned according to the categories of nodes and edges. After parsing the audit events into triplets, the NEO4J [48] tool is used to store and construct the KG.

Embedding Generation. We first use the *word2vec* model and the *Gensim* library [44] to generate node embeddings for training the graph. The length of the initial node embedding vectors is limited to 100 to reduce the model size. A two-layer *GAT* model is used to aggregate graph information. The input and output dimensions of the node vectors in the model are 100 and 128, respectively. We optimize the model parameters using the *Adam* optimizer. We train the model for 20 epochs with a batch size of 64. The learning rate changes exponentially with a rate of 0.98, starting at 0.01.

Computational Hardware. We use Pytorch [49] as the backend. All experiments are performed on an Ubuntu 20.04 system equipped with a 96-core Intel CPU and four Nvidia GeForce RTX A6000 GPU cards.

Dataset. The DARPA TC dataset [24] is a publicly available APT attack dataset released under the DARPA Transparent Computing (TC) program [50]. The dataset originates from the host network during a two-week red team vs. blue team engagement 3 in April 2018. During the engagement, enterprises simulated various security-critical services such as web servers, SSH servers, email servers, and SMB servers [22]. The red team conducted a series of nation-state and common attacks on target hosts while performing benign activities such as SSH logins, web browsing, and email checking.

TABLE II: Overview of attack cases in DARPA TC dataset with scenario descriptions

Attack Case	Scenario Description	$V\|G\|$	$E\|G\|$
APK Java	The Java APT automatically connects to the target network and runs privilege escalation software to gain root access and steal files.	25	36
Barephone Micro	An APT that runs on a mobile phone typically connects to the target network by loading `libmicroapt.so`.	10	16
CADETS Nginx	Nginx vulnerability was exploited to attack CADETS FreeBSD. The attacker downloaded a file, elevated it to a new process running as root, and attempted to inject it into the `sshd` process.	17	28
Firefox Drakon	The attacker exploits a Firefox vulnerability to attack CADETS through malformed HTTP requests, downloads the `libdrakon` implant file `*.so`, and injects it into the `sshd` process, causing CADETS to crash.	26	31
Metasploit APK	The attacker uses Metasploit as malware to send a malicious executable file to the target host and attack ClearScope.	35	56
Micro BinFmt-Elevate	The attacker uses the `ta1-pivot-2` tool to achieve BinFmt elevation and gain access to the root directory.	14	34
AppStarter APK	Typically disguised as a legitimate AppStarter APK file, it can use benign activities to install privilege escalation programs.	12	19
Webshell	The shell connection is made to the operator console via HTTP. The attacker executed the apt implant without root privileges.	31	49
Firefox DNS FileFilter	The attack triggered a Firefox backdoor to connect to the target network via DNS. By searching for processes that open specific non-existent files, the attacker elevated privileges.	23	29

The DARPA TRACE dataset consists of 726,072,596 audit events, forming 211 graphs. Overall, we use nine of these attack scenarios to evaluate the interpretability and accuracy of SmartGuard. We introduce the scenarios and attributes in Table II. The first column represents the name of the attack scenario, the second column provides a brief description of the scenario, and the third and fourth columns represent the average number of edges and nodes in the corresponding behavior graph.

For each attack scenario, we construct corresponding subgraphs, including benign and malicious source graphs. Each type of traceability graph (including benign traceability graphs) has a sample size of 4000, with 90% used as the training set.

B. Explicability of Event Semantic Inference

We measure the semantics learned by SmartGuard for audit events in both visual and quantitative ways: Visually, we use t-SNE to embed sampled behavior vectors into a 2D plane, giving us an intuitive understanding of the embedding distribution, and we construct five example behaviors to specifically demonstrate the similarity between individual behaviors; Quantitatively, for real audit events, we use normal behavior abstraction (log text summary + embedding vector) and behavior abstraction using only text summary, respectively, to fine-tune the model and compare the detection results.

Embeddings Visualization. The spatial distances between graph vectors encode their semantic similarity. To visualize the distances, we applied t-SNE to project the high-dimensional embedding space into a two-dimensional (2D) space, while largely preserving the structural information between vectors. We randomly sampled behaviors from nine scenarios for visualization, resulting in a scatter plot of 300 points. Figure 6 (a) shows the 2D visualization of the embedding space. Most of the points in the space are distributed in clusters, indicating that the events are indeed grouped according to some similarity measure.

To more specifically demonstrate the similarity of behavior semantics, we constructed five behavior instances as shown in

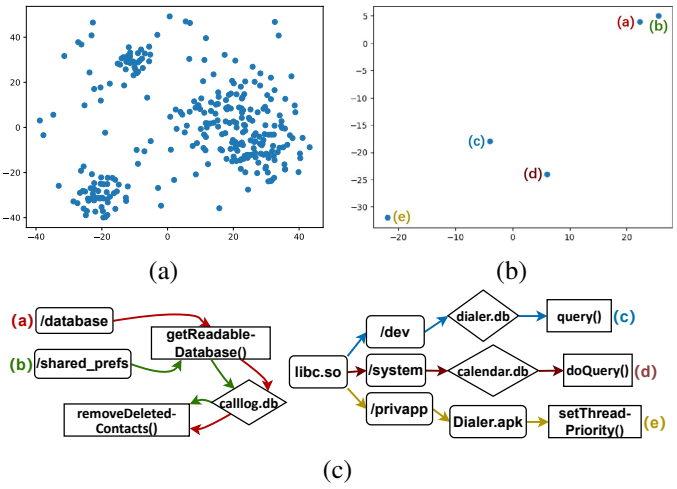

Fig. 6: Visualization of Behavior Abstraction's Embeddings.

Figure 6 (c), each consisting of four nodes. Among them, a and b represent stealing and clearing information from calllog.db from different addresses, respectively. The behaviors of these two instances are highly similar and can be judged as the same type of behavior. Instances c and d represent query operations on different database files at different addresses (libc.so is a common C language library, usually appearing at the beginning and can be removed as a noise event). These two instances have a certain degree of similarity. Instance e represents starting an APK file and setting its priority to the highest. The behavior of instance e has a low similarity to the aforementioned behaviors.

We constructed a scatter plot of the graph vectors for the above five behavior instances as shown in Figure 6 (b). From the plot, it can be seen that the visualization distances of similar behaviors are closer. This also proves that the behavior semantic abstraction of SmartGuard is effective.

Impact of Embeddings on Behavior Recognition. In addition to visualization, we also conducted experiments to determine whether the graph embedding vectors affect the behavior abstraction detection results. Specifically, we fine-

TABLE III: Results of the impact of embedding on the accuracy of behavior abstraction. The **bold part** indicates the best detection metric F1-score for each category.

Category	SmartGuard w/o embedding(%)			SmartGuard(%)		
	Precision	Recall	F1-score	Precision	Recall	F1-score
Barephone	85.2	86.9	86.1	95.9	94.5	**95.2**
CADETS Nginx	89.1	87.2	88.2	100	100	**100**
Metasploit	81.5	83.2	82.4	87.6	94.9	**91.1**

tuned a large language model in three attack scenarios using normal behavior abstractions and behavior abstractions with only log text summary content. During the inference phase, we used the corresponding behavior abstractions and the large language model for inference, and the results are shown in Table III below. We used the OPT-1.3b model for the experiments.

Table III provides a summary of the experimental results. Compared to detecting only log text summaries, the average F1-score for detecting normal behavior abstractions is 10% higher. Although text summaries can express some behavioral characteristics of logs, the length and complexity of the summaries limit the large language model's contextual understanding. Therefore, using only text summaries does not yield good inference results. In contrast, graph embedding vectors can express the overall characteristics of behavior with a fixed input length, complementing text summaries and compensating for the inability to express contextual features, thus performing better in behavior diagnosis.

In summary, the behavior abstractions learned by SmartGuard can express the contextual characteristics of events, which is crucial for information diagnosis.

C. Accuracy and Scalability of SmartGuard

To evaluate the accuracy of SmartGuard in behavior detection, we use two fine-tuned large language models, Facebook OPT-1.3b, and Open LLaMa2-3b, to predict sessions with similar malicious behavior in the DARPA dataset. For this purpose, we selected the aforementioned 9 behaviors as candidates for behavior abstraction. For each behavior, Table IV details the evaluation results of SmartGuard in detecting behavior abstraction in sessions. Performance metrics are measured using precision, recall, and F1 score. Intuitively, they provide measures of true positive rate, false positive rate, and overall accuracy, respectively. We define true positives as sessions where malicious behavior is correctly predicted and false positives as sessions where normal behavior is incorrectly predicted.

Table IV provides a summary of the experimental results. SmartGuard shows promising results in behavioral abstraction detection (with an average F1 score of 95.9%, and it comprehensively surpasses the baseline[51]). Even for complex behaviors, SmartGuard achieved an F1 score of 91.6%. This is because by utilizing the contextual information of audit events, SmartGuard can accurately infer the semantics of behavior instances. For example, even if the attacker changes the name of `calllog.db`, and repeatedly calls the `getReadableDatabase` and `removeDeletedContacts` functions, SmartGuard infers that they share similar semantics with malicious operations on `calllog.db` through these contextual relationships. Therefore, SmartGuard improves the similarity of malicious behaviors, even though at first glance they appear to be accessing different files.

Another interesting observation is that in most cases (14/18 in Table IV), the precision rate is higher than or equal to the recall rate. The relatively low recall rate indicates that even after noise removal, the semantics of behavior instances may still be affected by noise events. In other words, high precision indicates that the impact of such noise is limited. Two exceptions occur in Metasploit and Webshell. This is mainly because the former created multiple variations of APKs with Metasploit built into them during use, while the latter has more link and scan operations compared to other attack scenarios. This results in more noise events in their behaviors, affecting behavior recognition. However, overall, SmartGuard still achieves a low false positive rate in behavior abstraction detection.

We also observed that SmartGuard shows good scalability on two different models. The average F1 scores of the two models are 95.6% and 96.3%, respectively, both demonstrating a high level of behavior recognition. This can be attributed to two reasons: the accurate semantics of behavior abstraction and the understanding and reasoning capabilities of large language models. The necessity of behavior abstraction semantics is detailed in the chapter. The reasoning ability of large language models is influenced by their parameter size and expressive power, which makes a significant difference. We also conducted experiments on RoBERTa-Large (with approximately 350M parameters), and the average F1 score was only 82.9%, much lower than the results of the first two models. We believe the main reason is that the parameters and expressive power of the RoBERTa model are limited and insufficient to understand our behavior abstraction semantics. Therefore, SmartGuard should be deployed on large language models with a larger number of parameters (Billion-level). In summary, SmartGuard has shown scalability on different language models.

We observed that SmartGuard exhibits high accuracy in classifying malicious behaviors. For all nine attack scenarios, the average Precision and Recall of SmartGuard can reach 96.7% and 95.3%, respectively. In other words, 3 out of 90 malicious sessions are missed. SmartGuard can identify malicious behaviors and distinguish them from benign behaviors.

To test the scalability of SmartGuard on the dataset, we retrained the model. We used `CADETS Nginx` and `Micro BinFmt-Elevate` from the original dataset as the test set,

TABLE IV: Evaluation results of detecting behavior abstraction in sessions. The **bold part** indicates the best detection metric F1-score for each category.

Category	Extractor[51](%)			SmartGuard w/ OPT-1.3b(%)			SmartGuard w/ LLaMa2-3b(%)		
	P	R	F1	P	R	F1	P	R	F1
JAVA APK	91.0	97.0	93.9	94.7	93.9	94.3	95.6	94.5	**95.0**
Barephone	88.0	100	93.6	95.9	94.5	95.2	97.2	96.8	**97.0**
CADETS Nginx	100	84.0	91.3	100	100	**100**	100	98.9	99.4
Firefox Drakon	100	90.0	94.7	100	95.6	**97.8**	100	95.2	97.5
Metasploit	91.0	91.0	91.0	87.6	94.9	91.1	90.2	94.6	**92.3**
Micro BinFmt	88.0	100	93.6	100	92.3	**96.0**	100	88.2	93.7
AppStarter	100	88.0	93.6	97.8	96.1	**96.9**	94.2	97.4	96.8
Webshell	89.0	89.0	89.0	92.4	93.3	92.8	95.4	98.2	**97.1**
Firefox DNS	-	-	-	100	98.6	**99.3**	100	91.3	95.4

TABLE V: Results of the scalability of SmartGuard on unknown attacks.The results only indicate the accuracy of determining whether it is an **'Attack Behavior'**. The **bold part** indicates the best detection metric F1-score for each category.

Category	OPT-1.3b(%)			LLaMa2-3b(%)		
	P	R	F1	P	R	F1
CADETS Nginx	91.2	85.1	88.1	92.4	87.3	**89.8**
Metasploit	87.6	88.4	88.0	88.1	89.6	**88.8**
Webshell	87.9	86.2	87.1	87.1	87.7	**87.4**

TABLE VI: Results of SmartGuard's detection after fine-tuning for unknown attacks using expert-provided traceability graphs. The model was fine-tuned with expert traceability graphs different from the original data, and the detection metric is the attack type. The **bold part** indicates the best detection metric F1-score for each category.

Category	OPT-1.3b(%)			LLaMa2-3b(%)		
	P	R	F1	P	R	F1
CADETS Nginx	98.4	97.5	97.9	98.9	97.4	**98.1**
Metasploit	87.9	93.6	90.7	90.2	93.7	**91.9**
Webshell	92.9	92.9	92.9	95.5	97.1	**96.3**

and the remaining seven attack types as the training set. This was to evaluate whether SmartGuard could classify unseen malicious behaviors as attacks by learning the relationships between behaviors. The specific experimental results are shown in Table V. On the scalability of the dataset, SmartGuard showed promising results. The average F1-scores of the two types reached 89.4%, which is close to the recognition accuracy of the baseline on the full dataset. This indicates that SmartGuard can learn the semantic relationships between different behaviors, demonstrating good scalability on different datasets.

Additionally, to test the scalability of SmartGuard in detecting unknown attacks, we retrained the model. We used `CADETS Nginx`, `Metasploit`, and `Webshell` from the original dataset as the test set, while the remaining six attack types were used as the training set. This is because, according to the baseline, these three datasets performed the worst, suggesting they are more complex attack types and thus better suited for detection testing. The specific experimental results are shown in Table V. SmartGuard showed promising results. The average F1 scores for the three types reached 89%, which is close to the recognition accuracy of the baseline on the full dataset. This indicates that SmartGuard can learn the semantic relationships between different behaviors and achieve good recognition accuracy for unknown attacks that are not present in the training set.

Additionally, thanks to the fine-tuning capability of LLMs, SmartGuard can accurately recognize unknown attacks by fine-tuning the model based on traceability graphs provided by experts. Using the aforementioned model, we fine-tuned it with traceability graphs for three types of attacks (different from the original data) provided by experts and used the original data for detection (detecting the specific attack type). The results in Table VI demonstrate SmartGuard's flexible fine-tuning ability, achieving an average F1 score of 94.3%, surpassing the baseline and being within 2-3% of the model trained on the full dataset. In real-world scenarios, SmartGuard can flexibly incorporate new attack types and achieve efficient detection.

In summary, SmartGuard can identify malicious behaviors and distinguish them from benign ones. It can also learn the semantic relationships between different behaviors, achieving good recognition accuracy for unknown attacks not present in the training set. Analysts can further enhance SmartGuard's ability to recognize specific behaviors by fine-tuning with instructions and expert-provided traceability graphs for unknown malicious behaviors. This offers a new approach to solving real-world problems.

D. Interpretability of LLMs and Solutions for Hallucinations

LLMs have demonstrated exceptional ability in understanding the context of downstream tasks and generating relevant information from demonstrations. Therefore, we use the CoT (Chain of Thought) approach to enable large language models to interpret audit events. We designed a hierarchical instruction structure to help the large language model better understand the behavior summary, make accurate judgments, and provide explanatory narratives of specific steps. Specifically, we let the large language model judge and summarize the sensitive nodes, sensitive links, and overall characteristics of the behavior. This is because such CoT instructions provide the large language model with sufficient contextual logic, enabling it

TABLE VII: Results of mitigating hallucinations of SmartGuard. The **bold part** indicates the best detection metric F1-score for each category.

Category	Single-turn(%)			Multi-turn(%)		
	P	R	F1	P	R	F1
JAVA APK	94.7	93.9	94.3	97.6	93.9	**95.7**
Metasploit	87.6	94.9	91.1	93.2	95.8	**94.4**
Firefox Drakon	100	95.6	97.8	100	96.1	**98.0**
Webshell	92.4	93.3	92.8	94.8	94.2	**94.5**

to answer most of the behavior detail questions needed by analysts and make accurate judgments. Finally, we let the large language model determine the attack category and provide more explanations for this attack type.

We take the behavior graph in Figure 1 as an example to illustrate the CoT process shown in Figure 7. As can be seen from the figure, the large language model can make accurate and interpretable narratives of the behavior summary, thereby further reducing the workload of analysts.

Furthermore, CoT can mitigate the issue of hallucinations in large models to some extent. We tested single-turn response (the original mode of SmartGuard, which directly judges the attack type) and multi-turn response (following the mode in Figure 7, first judging sensitive nodes and links, then determining if it is an attack behavior, and finally identifying the specific attack type) on the OPT-1.3b model during the behavior detection phase. Table VII shows the experimental results. The results indicate that the CoT-based multi-turn response can effectively improve the accuracy of SmartGuard and reduce false positives caused by hallucinations. We observed a significant increase in the precision metric compared to single-turn response, mainly due to the reduction in false positive samples, i.e., fewer samples were mistakenly classified as attack behaviors.

In summary, our results show that SmartGuard can provide a certain level of explainability and mitigate the misjudgments caused by hallucinations to some extent.

Instruction: The following description shows the log information of the event. It contains a summary of the log text and behavior Embeddings. Please determine whether there are any sensitive nodes in this event. If so, please specify them.
LLM Output: libmicroapt.so,barephoneinstr.apk,mmssms.db, removeDeletedContacts(),insertUpdateContacts().

Instruction: The following description contains a summary of the log text and behavior Embeddings. Please indicate the **sensitive behavior** path based on the sensitive nodes. The answer template is 'From...to...'.
LLM Output: From libmicroapt.so to insertUpdateContacts().

Instruction: The following description contains a summary of the log text and behavior Embeddings. Please indicate the **overall characteristics** of the sensitive behavior.
LLM Output: Load libmicroapt, install barephoneapk, and perform multiple deletions and updates on the mmssms.db database.

Instruction: The following description contains a summary of the log text and behavior Embeddings. Please focus on the Embeddings to determine **whether this event is a malicious event**. If it is, please provide the classification of the malicious event.
LLM Output: Yes, the category is Barephone Micro.

Instruction: The following description contains a summary of the log text and behavior Embeddings. Please provide **more explanations** about this attack behavior.
LLM Output: Barephone Micro Attack generally loads Micro APT by installing an apk, and the target of the attack is usually to steal database information.

Fig. 7: Interpretive narrative of the behavior summary by the large language model.

V. Discussion

In this section, we will introduce some design choices, implications, and possible extensions of this work.

RQ1: Why is it better to evaluate using 9 scenarios from the DARPA dataset?

The DARPA dataset is a widely recognized test benchmark, and its data quality and authenticity have been verified by the industry. In addition, the DARPA dataset covers multiple attack scenarios and can comprehensively evaluate the performance of the model under different types of attack behaviors. The scenarios in this data set are complex and challenging, which helps to verify the stability and accuracy of the model in dealing with complex and diverse attack behaviors. The 9 scenarios we use are scenarios in which manual traceability maps are given in the ground truth of the dataset. They are more accurate and are also a commonly used choice in the industry.

RQ2: How is this methodology different from previous methods?

In addition to the use of LLM, compared with previous process-level behavior extraction solutions, SmartGuard reduces the granularity of extraction to the thread level and adds function-level behavioral semantic records, not just system scheduling. In the traditional audit log analysis process, triples (Subject, Relation, Object) are recorded at the process level. This method can effectively and concisely extract a high-level behavior, but it will ignore the more detailed parts behind the behavior and improve the difficulty of detection. Then we added LLM for pattern recognition, and aggregated text summaries and graph embedding vectors into description information so that LLM can perform reasoning and large language models can more smoothly understand behavior summaries and make accurate judgments while also being able to give explanatory narratives of specific steps in a behavior. Additionally, SmartGuard has advantages in detecting unknown attacks and supports timely system updates.

RQ3: How is the robustness of Behavior Abstraction?

To evade behavior abstraction, attackers may attempt to confuse behavior by deliberately introducing irrelevant events. However, the impact of such events on behavior semantics is limited. In Sections III-C2 and III-C3, we introduce two strategies (thread-based behavior partitioning and noise events) to enhance SmartGuard's robustness to abstract behavior. Specifically, while SmartGuard aggregates the contextual semantics of behavior, irrelevant events will be assigned low-importance weights or not be partitioned into specific behaviors, and may even be deleted as noise events. Another potential method to counteract behavior obfuscation is to incorporate additional auxiliary information (e.g., semantically rich parameters of audit events) into SmartGuard's KG. We believe this can empower SmartGuard with more capabilities to filter out uninteresting events for security analysis.

RQ4: Why is the graph embedding space and updating?

Most learning methods using embedding techniques are common [52], and due to semantic changes and the inclusion of previously unseen data (e.g., data format changes), SmartGuard needs to periodically retrain the embedding space. Additionally, compared to natural language embedding models, graph embedding models have a natural advantage in focusing attention on key nodes and restoring the entire event scenario. That is, the graph embedding method we chose can fully express behavior semantics for pattern recognition.

VI. Related Works

Causal Analysis. Causal analysis is an orthogonal but important issue related to behavior abstraction, using causal reasoning to analyze log events. King and Chen [5] first introduced the construction of dependency graphs on system audit logs to trace a given security event and find its root cause. Jin et al. [53] improved causal tracking by capturing forward and cross-host dependencies. To mitigate the dependency explosion problem and high storage overhead in causal analysis, a large amount of research work has been further conducted. Recent work has proposed fine-grained unit partitioning [54], [55], [56], model-based reasoning [57], [53], record and replay [58], [59], and general provenance [60] techniques to achieve more precise causal tracking. Another line of research is dedicated to reducing the overall log volume used for analysis through graph compression [8], [9], [10], [13] and data reduction [7], [61], [62]. Although the scope of SmartGuard differs from these solutions, its effectiveness relies on accurate causal analysis when correlating data to summarize behavior.

Behavior Abstraction. It has been proven that abstracting behavior into graphs or causal dependencies is very effective for understanding operating system-level activities and detecting potential threats and risks. TGMiner [20] mines discriminative graph patterns from behaviors of interest and uses them as templates to identify similar behaviors. Based on cyber threat intelligence reports, POIROT [19] extracts query graphs of APT behaviors and proposes an alignment algorithm to search for their presence in source graphs. SLEUTH [17] and MORSE [18] propose labeling strategies to model information leakage behaviors. HOLMES [22] and RapSheet [21] view multi-stage attacks as causal event chains conforming to TTP specifications. Compared to previous work, SmartGuard abstracts behavior as the result of context-based embeddings (graph embedding vectors) and log text summary integration. Our research results show that this quantitative representation of behavior can preserve behavior semantics and enable advanced behavior analysis.

Embedded Space and Large Language Model Analysis. There is a large body of literature on applying embedding techniques to other log analysis tasks. Such tasks include anomaly-based IDS [63], [64], [65], malware identification [66], [67], and understanding the evolution of cyber attacks [39]. Many previous works use machine learning models (e.g., neural networks, word embeddings, and n-grams) to embed logs into vectors. For example, DeepLog [37] is a neural network-based approach that uses Long Short-Term Memory (LSTM) to learn execution patterns in normal log entry streams. PROVDETECTOR [66] applies the neural word embedding model doc2vec to quantify the behavior of processes running on a system. Similarly, ATTACK2VEC [39] uses temporal word embedding models to quantify the context of cyber attack steps over time. Extractor[51] automatically constructs provenance graphs based on the semantic and syntactic characteristics of log languages. UNICORN [68] proposes a graph sketching algorithm to summarize long-running system executions. WATSON [43] uses the translation-based embedding TransE to reveal contextual semantics. SmartGuard uses a combination of text summaries and graph vectors as the context for audit events. Essentially, we use a combination of graphs and text to represent semantic information, which is an innovative point compared to the above methods. Additionally, RCACopilot [69] uses GPT-4 to summarize log information and cluster it, based on the premise that GPT-4 itself has a large corpus of log data. In contrast, SmartGuard can be deployed on smaller models without a log corpus, making it more convenient for analysts to flexibly operate and analyze different corpus logs with lower training costs.

VII. Conclusion

Abstracting high-level behaviors from low-level audit logs is a critical task in security response. It helps bridge the semantic gap between audit events and system behaviors, thereby reducing the manual effort in log analysis. In this paper, we propose an automated method, SmartGuard, to extract behaviors from audit events and analyze them using large language models. Specifically, SmartGuard utilizes context information from log-based knowledge graphs for semantic inference. To differentiate representative behaviors, SmartGuard provides a representation method that combines behavior semantics graph vectors with text summaries. It uses this to perform reasoning with a large language model and provides interpretive narratives. We evaluated SmartGuard against adversarial engagement behaviors organized by DARPA. The experimental results show that SmartGuard can accurately abstract benign and malicious behaviors, demonstrating good scalability for unknown attacks. Additionally, the provided interpretive narratives and timely system updates effectively address real-world scenarios.

ACKNOWLEDGMENTS

This research is supported by data security and privacy protection research phase III (Zhejiang University-Ant Group Fintech Centre).

REFERENCES

[1] C. One, "Information on the capital one cyber incident," 2019. [Online]. Available: https://www.capitalone.com/facts2019

[2] T. Guardian, "Twitter hack," 2020. [Online]. Available: https://www.theguardian.com/technology/2020/jul/15/twitter-elon-musk-joe-biden-hacked-bitcoin

[3] LogRhythm, "Endpoint monitoring and forensics," 2024. [Online]. Available: https://logrhythm.com/products/endpoint-monitoring

[4] S. T. King and P. M. Chen, "Backtracking intrusions," in *ACM Symposium on Operating Systems Principles*, 2003, pp. 223–236.

[5] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen, "Enriching intrusion alerts through multi-host causality," in *Network and Distributed System Security Symposium*, 2005.

[6] P. Gao, X. Xiao, D. Li, Z. Li, K. Jee, Z. Wu, C. H. Kim, S. R. Kulkarni, and P. Mittal, "Saql: A stream-based query system for real-time abnormal system behavior detection," in *USENIX Security Symposium*, 2018.

[7] K. H. Lee, X. Zhang, and D. Xu, "Loggc: Garbage collecting audit log," in *ACM Conference on Computer and Communications Security*, 2013.

[8] C. Chen, H. T. Lehri, L. K. Loh, A. Alur, L. Jia, B. T. Loo, and W. Zhou, "Distributed provenance compression," in *ACM International Conference on Management of Data*, 2017, pp. 203–218.

[9] W. U. Hassan, L. Aguse, N. Aguse, A. Bates, and T. Moyer, "Towards scalable cluster auditing through grammatical inference over provenance graphs," in *Network and Distributed System Security Symposium*, 2018.

[10] M. N. Hossain, J. Wang, O. Weisse, R. Sekar, D. Genkin, B. He, S. D. Stoller, G. Fang, F. Piessens, E. Downing *et al.*, "Dependence-preserving data compaction for scalable forensic analysis," in *USENIX Security Symposium*, 2018.

[11] S. Ma, J. Zhai, Y. Kwon, K. H. Lee, X. Zhang, G. Ciocarlie, A. Gehani, V. Yegneswaran, D. Xu, and S. Jha, "Kernel-supported cost-effective audit logging for causality tracking," in *USENIX Annual Technical Conference*, 2018.

[12] N. Michael, J. Mink, J. Liu, S. Gaur, W. U. Hassan, and A. Bates, "On the forensic validity of approximated audit logs," in *ACM Annual Computer Security Applications Conference*, 2020.

[13] Y. Tang, D. Li, Z. Li, M. Zhang, K. Jee, X. Xiao, Z. Wu, J. Rhee, F. Xu, and Q. Li, "Nodemerge: Template based efficient data reduction for big-data causality analysis," in *ACM Conference on Computer and Communications Security*, 2018.

[14] J. Gui, D. Li, Z. Chen, J. Rhee, X. Xiao, M. Zhang, K. Jee, Z. Li, and H. Chen, "Aptrace: A responsive system for agile enterprise level causality analysis," in *IEEE International Conference on Data Engineering*, 2020.

[15] O. Setayeshfar, C. Adkins, M. Jones, K. H. Lee, and P. Doshi, "Graalf: Supporting graphical analysis of audit logs for forensics," 2019, arXiv preprint arXiv:1909.00902.

[16] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal, "Towards a timely causality analysis for enterprise security," in *Network and Distributed System Security Symposium*, 2018.

[17] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. Stoller, and V. N. Venkatakrishnan, "Sleuth: Real-time attack scenario reconstruction from cots audit data," in *USENIX Security Symposium*, 2017.

[18] M. N. Hossain, S. Sheikhi, and R. Sekar, "Combating dependence explosion in forensic analysis using alternative tag propagation semantics," in *IEEE Security and Privacy*, 2020.

[19] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. N. Venkatakrishnan, "Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting," in *ACM Conference on Computer and Communications Security*, 2019.

[20] B. Zong, X. Xiao, Z. Li, Z. Wu, Z. Qian, X. Yan, A. K. Singh, and G. Jiang, "Behavior query discovery in system-generated temporal graphs," in *Proceedings of the VLDB Endowment*, vol. 8, no. 13. VLDB Endowment, 2015, pp. 1534–1545.

[21] W. U. Hassan, A. Bates, and D. Marino, "Tactical provenance analysis for endpoint detection and response systems," in *IEEE Security and Privacy*, 2020.

[22] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "Holmes: real-time apt detection through correlation of suspicious information flows," in *IEEE Security and Privacy*, 2019.

[23] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.

[24] DARPA, "Transparent computing engagement 5 data release," https://github.com/darpa-i2o/Transparent-Computing/blob/master/README-E5.md, 2024.

[25] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 1877–1901.

[26] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[27] X. Lian, Y. Chen, R. Cheng, J. Huang, P. Thakkar, and T. Xu, "Configuration validation with large language models," *arXiv preprint arXiv:2310.09690*, 2023.

[28] A. Gehani and D. Tariq, "Spade: support for provenance auditing in distributed environments," in *International Middleware Conference*. Springer-Verlag New York, Inc., 2012, pp. 101–120.

[29] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "Nodoze: Combatting threat alert fatigue with automated provenance triage," in *Network and Distributed System Security Symposium*, 2019.

[30] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal, "Towards a timely causality analysis for enterprise security," in *Network and Distributed System Security Symposium*, 2018.

[31] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *ACM Conference on Computer and Communications Security*. ACM, 2002, pp. 255–264.

[32] Linux Audit, "Linux kernel audit subsystem," https://github.com/linux-audit/audit-kernel, 2024.

[33] X. Wang, X. He, Y. Cao, M. Liu, and T.-S. Chua, "Kgat: Knowledge graph attention network for recommendation," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2019, pp. 950–958.

[34] X. Wang, X. He, F. Feng, L. Nie, and T.-S. Chua, "Tem: Tree-enhanced embedding model for explainable recommendation," in *World Wide Web Conference*. ACM, 2018, pp. 123–132.

[35] M. Färber, F. Bartscherer, C. Menne, and A. Rettinger, "Linked data quality of dbpedia, freebase, opencyc, wikidata, and yago," *Semantic Web*, vol. 9, no. 1, pp. 77–129, 2018.

[36] E. Miller, "An introduction to the resource description framework," *Bulletin of the American Society for Information Science and Technology*, vol. 25, no. 1, pp. 15–19, 1998.

[37] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1285–1298.

[38] Y. Shen, E. Mariconti, P. A. Vervier, and G. Stringhini, "Tiresias: Predicting security events through deep learning," in *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 592–605.

[39] Y. Shen and G. Stringhini, "Attack2vec: Leveraging temporal word embeddings to understand the evolution of cyberattacks," in *USENIX Security Symposium*, 2019.

[40] G. Angeli, M. J. J. Premkumar, and C. D. Manning, "Leveraging linguistic structure for open domain information extraction," in *Annual Meeting of the Association for Computational Linguistics and International Joint Conference on Natural Language Processing*. Association for Computational Linguistics, 2015, pp. 344–354.

[41] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems*, 2014, pp. 3104–3112.

[42] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *USENIX Security Symposium*, 2017.

[43] 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 *Network and Distributed System Security Symposium*, 2021.

[44] R. Řehůřek and P. Sojka, "Software framework for topic modelling with large corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Citeseer, 2010.

[45] I. Chalkidis, "Chatgpt may pass the bar exam soon, but has a long way to go for the lexglue benchmark," *arXiv preprint arXiv:2304.12202*, 2023.
[46] J. Kasai, Y. Kasai, K. Sakaguchi, Y. Yamada, and D. Radev, "Evaluating gpt-4 and chatgpt on japanese medical licensing examinations," *arXiv preprint arXiv:2303.18027*, 2023.
[47] Z. Zhang, A. Zhang, M. Li, and A. Smola, "Automatic chain of thought prompting in large language models," in *International Conference on Learning Representations*, 2023.
[48] "Neo4j," https://neo4j.com/.
[49] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019, pp. 8024–8035.
[50] "Darpa transparent computing," https://www.darpa.mil/program/transparent-computing.
[51] K. Satvat, R. Gjomemo, and V. Venkatakrishnan, "Extractor: Extracting attack behavior from threat reports," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2021, pp. 598–615.
[52] E. Ordentlich, L. Yang, A. Feng, P. Cnudde, M. Grbovic, N. Djuric, V. Radosavljevic, and G. Owens, "Network-efficient distributed word2vec training system for large vocabularies," in *ACM International Conference on Information and Knowledge Management*, 2016.
[53] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. F. Ciocarlie *et al.*, "Mci: Modeling-based causality inference in audit logging for attack investigation," in *Network and Distributed System Security Symposium*, 2018.
[54] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition," in *Network and Distributed System Security Symposium*, 2013.
[55] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu, "Accurate, low cost and instrumentation-free security audit logging for windows," in *Annual Computer Security Applications Conference*, 2015.
[56] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "Mpi: Multiple perspective attack investigation with semantic aware execution partitioning," in *USENIX Security Symposium*, 2017.
[57] Y. Kwon, D. Kim, W. N. Sumner, K. Kim, B. Saltaformaggio, X. Zhang, and D. Xu, "Ldx: Causality inference by lightweight dual execution," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.
[58] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, "Rain: Refinable attack investigation with on-demand inter-process information flow tracking," in *ACM SIGSAC Conference on Computer and Communications Security*, 2017.
[59] Y. Ji, S. Lee, M. Fazzini, J. Allen, E. Downing, T. Kim, A. Orso, and W. Lee, "Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking," in *USENIX Security Symposium*, 2018.
[60] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates, "Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis," in *Network and Distributed System Security Symposium*, 2020.
[61] S. Ma, X. Zhang, and D. Xu, "Protracer: Towards practical provenance tracing by alternating between logging and tainting," in *Network and Distributed System Security Symposium*, 2016.
[62] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang, "High fidelity data reduction for big data security dependency analyses," in *ACM SIGSAC Conference on Computer and Communications Security*, 2016.
[63] T. Chen, L.-A. Tang, Y. Sun, Z. Chen, and K. Zhang, "Entity embedding-based anomaly detection for heterogeneous categorical events," in *International Joint Conference on Artificial Intelligence*, 2016.
[64] 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," in *ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1777–1794.
[65] E. Manzoor, S. M. Milajerdi, and L. Akoglu, "Fast memory-efficient anomaly detection in streaming heterogeneous graphs," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 1035–1044.
[66] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Chen, W. Cheng, C. A. Gunter, and H. Chen, "You are what you do: Hunting stealthy malware via data provenance analysis," in *Network and Distributed System Security Symposium*, 2020.
[67] S. Wang, Z. Chen, X. Yu, D. Li, J. Ni, L.-A. Tang, J. Gui, Z. Li, H. Chen, and P. S. Yu, "Heterogeneous graph matching networks for unknown malware detection," in *International Joint Conference on Artificial Intelligence*, 2019, pp. 3762–3770.
[68] X. Han, T. Pasquier, A. Bates, J. Mickens, and M. Seltzer, "Unicorn: Runtime provenance-based detector for advanced persistent threats," in *Network and Distributed System Security Symposium*, 2020.
[69] Y. Chen *et al.*, "Automatic root cause analysis via large language models for cloud incidents," in *European Conference on Computer Systems*, 2024.