Reducing False Positives in Provenance-Based Intrusion Detection Systems

Yue Xiao, Xiaoli Zhang, Peiyang Li, Ke Xu, *Fellow, IEEE*, and Qi Li, *Senior Member, IEEE*

***Abstract*—Provenance-based Intrusion Detection Systems (PIDSes) have been widely used to detect Advanced Persistent Threats (APTs). The PIDSes typically construct provenance graphs based on system logs, learn patterns of benign processes from the provenance graphs, and identify processes whose patterns deviate from benign ones as anomalies. However, in real deployment, these PIDSes would produce many false positives (FPs), because they cannot distinguish between abnormal processes and benign processes with unknown patterns. In this paper, we propose SheepHunter to automatically distinguish between benign and malicious processes in PIDS alarms. We observe that benign processes are numerous and their behaviors are similar, while malicious processes are few in number, each with distinct behaviors. Based on the observation, SheepHunter represents processes as high-dimensional embedding vectors that can effectively characterize process behaviors, and then identifies false positives through analyzing the density of embeddings. Specifically, we devise a multi-dimensional process representation that captures execution context (e.g., function arguments), access control attributes (e.g., privilege level), and structural characteristics of the process within the provenance graph (e.g., node connectivity and causal dependencies). Then, we measure process behavior density in each dimension using a community splitting algorithm, and classify processes that exhibit high density across dimensions as false positives. We evaluate SheepHunter on two open-source datasets and the results demonstrate that SheepHunter can reduce FPs by an average of 43.8% for existing PIDSes.**

***Index Terms*—Provenance-based intrusion detection system (PIDS), anomaly-based detection, false positive reduction.**

Received 21 August 2025; revised 9 July 2026 and 30 August 2026; accepted 1 September 2026. Date of publication 17 September 2026; date of current version 2 October 2026. This work was supported in part by the National Natural Science Foundation of China under Grant 62132011, Grant 62425201, and Grant 62302452; in part by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China under Grant JYB2025XDXM114; in part by Beijing-Tianjin-Hebei Natural Science Foundation Cooperation Project under Grant 25JJJJC0003; in part by Beijing Natural Science Foundation under Grant 4262028; in part by the Open Research Fund of State Key Laboratory of Internet Architecture under Grant HLW2025ZD02; in part by the Young Elite Scientists Sponsorship Program of Beijing High Innovation Plan under Grant 20250835; and in part by the Fundamental Research Funds for the Central Universities under Grant FRF-TP-25-037. The associate editor coordinating the review of this article and approving it for publication was Dr. Fabio De Gaspari. *(Corresponding author: Qi Li.)*

Yue Xiao is are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China (e-mail: xiaoy25@mails.tsinghua.edu.cn).

Xiaoli Zhang is with the School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China (e-mail: xiaoli.z@ustb.edu.cn).

Peiyang Li is with Ant Group, Beijing 100081, China, and also with Tsinghua University, Beijing 100084, China (e-mail: peiyangli.20@gmail.com).

Ke Xu is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, and also with Zhongguancun Laboratory, Beijing 100094, China (e-mail: xuke@tsinghua.edu.cn).

Qi Li is with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, and also with the State Key Laboratory of Internet Architecture, Beijing 100878, China (e-mail: qli01@tsinghua.edu.cn).

Digital Object Identifier 10.1109/TIFS.2026.3734778

I. Introduction

With the development of the Internet, cyber-attacks are increasing rapidly. Advanced Persistent Threat (APT) is a kind of sophisticated cyber-attack, which conducts sustained attacks via complex procedures. For example, APT41 [1], observed in July 2024, has extracted sensitive data from multiple organizations since 2023, such as the global shipping and logistics industry.

Provenance-based Intrusion Detection Systems (PIDSes) achieve great performance in detecting APTs. The PIDSes [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12] typically collect benign audit logs to construct provenance graphs and learn benign processes' patterns. They identify processes whose patterns deviate from benign ones as alarms. However, in real deployment, those *anomaly-based PIDSes* produce many false positive processes. The main reason is that they cannot distinguish between benign but unknown processes and malicious ones, e.g., processes that aim to perform a new function are misclassified as anomalies. Traditional methods [13], [14] of dealing with FPs rely on manual analysis and retraining detection models. However, in PIDSes, manual analysis is costly, and even experts can only handle a limited amount of data per day [15]. What's even worse, there is no automated method to help identify FPs.

In this paper, we aim to automatically distinguish between benign and malicious processes in PIDS alarms. We observe that benign processes are numerous and typically exhibit similar behaviors (i.e., similar function invocation sequences), whereas malicious processes are relatively few and display diverse behaviors due to varied attack objectives. For example, a cloud server may spawn many processes to establish connections with different users, resulting in highly similar behavioral patterns. In contrast, attackers typically aim to remain stealthy and conduct complex attacks using only a small number of specialized processes.

Based on this key observation, we propose SheepHunter. Our intuition is to search for processes whose behavior is similar and deem these processes benign. To achieve this, we use process behavior density to quantitatively represent this characteristic of processes. A high behavior density indicates that a process's behavior is common and shared by many other processes, in which case the process is classified as an

1556-6021 © 2026 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:05:57 UTC from IEEE Xplore. Restrictions apply.

FP. SheepHunter applies an unsupervised community splitting approach to obtain the process behavior density. This allows SheepHunter to identify FPs without any knowledge of known benign processes.

In detail, we first devise a multi-dimensional process representation extracted from the raw logs and alarms generated by PIDSes. The representation includes a process's execution context (e.g., function arguments), access control attributes (e.g., privilege level), and structural characteristics within the provenance graph (e.g., node connectivity and causal dependencies). By integrating complementary perspectives from these distinct feature dimensions, the representation highlights behavioral differences between processes, thereby improving the separability of malicious and benign processes. Then, we measure process behavior density in each feature dimension by applying a community splitting algorithm [16], which clusters processes with similar behavior. A process is classified as a false positive only if it appears in high-density clusters across all the above representations, ensuring consistent similarity from multiple behavioral perspectives. This multi-dimensional density analysis not only helps FP identification but also reduces manual effort, as analysts can focus on representative processes within each cluster rather than examining all alarms.

We evaluate the performance of SheepHunter on two widely used datasets, the DARPA TC Engagement [17] datasets and the Arena datasets provided by [7]. The experimental results show that SheepHunter can help existing anomaly-based PIDSes reduce FPs by an average of 43.8%. Additionally, with the help of SheepHunter, analysts only need to analyze 0.39%~4.9% of the alarm processes instead of all of them.

In summary, our contributions are as follows:

- We try to solve the problem of FP detection in provenance-based detection. To achieve this, we conducted an in-depth analysis of the root causes of false positives in PIDSes. We find that a key reason is that PIDSes cannot distinguish between benign but unknown processes and malicious ones.
- We propose SheepHunter, a method that identifies false positive processes from PIDSes' alarms by analyzing process behavior density. The method can automatically distinguish between benign and malicious processes in PIDS alarms.
- We systematically evaluate SheepHunter on two widely used datasets to demonstrate its performance and usability. The results show that SheepHunter can achieve large reductions in FPs for existing detectors. SheepHunter can also help with model adaptation and manual analysis. Our code is available at https://github.com/FIND-Lab/SheepHunter.

The rest of the paper is organized as follows: In §II, we briefly introduce existing PIDSes and use an example to demonstrate how FP arises. Then, we present the problem statement. In §III, we state our key observation and provide an overview of SheepHunter. In §IV, we introduce the details of SheepHunter. In §V, we conduct several experiments to evaluate the performance of SheepHunter. §VI is a discussion of other corresponding problems. §VII shows the related work. We conclude the paper in §VIII.

II. Background and Problem Statement

In this section, we first provide a brief introduction to existing PIDSes. Then, we use an example to explain how FP occurs. Finally, we present the problem statement.

A. Provenance-Based Intrusion Detection Systems

Provenance-based Intrusion Detection Systems (PIDSes) record system logs in the kernel. These logs contain events that occur between entities. PIDSes use entities (e.g., process, file, and socket) as nodes and events (e.g., fork, write, and connect) as edges to construct provenance graphs. For example, if a process *Firefox* connects to a socket *146.153.48.2*, there will be two nodes named *Firefox* and *146.153.48.2* with an edge *connect* between them. PIDSes then conduct detection based on the graph. There are two main categories of detection approaches.

1) Rule-Based PIDSes: These approaches [18], [19], [20], [21], [22], [23] design rules based on expert knowledge to perform detection. Common steps include tag initialization for nodes, tag propagation, and alarm generation. Different approaches design different strategies for these three steps. Rule-based approaches are efficient and cost-effective. Unfortunately, rule-based approaches can only detect known attacks, as determined by expert knowledge. If an attack is not covered by expert knowledge, it will not be detected.

2) Anomaly-Based PIDSes: Anomaly-based approaches [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [24], [25] are designed to deal with unknown attacks. These approaches first extract features from the provenance graphs, including features from graphs [6], [8], [9], [24], paths [2], [3], or nodes [5], [7], [10], [11], [12], [25], learn features of benign processes, and identify unknown processes as anomalies.

In this paper, we focus on the anomaly-based PIDSes as they have great performance in detecting attacks, especially the unknown ones. However, lots of FP processes need to be handled. We should automatically identify these FP processes for different PIDSes.

B. Motivation Example About FPs in Reality

In this subsection, we explain why FPs occur by experimenting with the state-of-the-art (SOTA) PIDS MAGIC [12] and a widely used dataset DARPA-E3 CADETS [17].

The detector (i.e., MAGIC) uses data from April 2nd to April 6th, as well as part of the data from April 12th, as training data. It then tests its performance using the remaining data from April 12th. Also, we remove the data from April 12th from the training set and test the detector's performance for comparison. Without the data from the last day, the F1 score of the detector drops from 92% to 10% and the FPs increase by about 100 times. The main reason is that there exist benign but unknown processes in the last day's data. Figure 1 shows an example. Process-1 and process-2 are both benign processes whose command line is *sendmail*. Process-1 loads the initialization files (*/lib/*.so.**, */etc/hosts*) and communicates with *128.55.12.10* via the *SENDTO* and *RECVFROM* events. Process-2 loads the initialization files first, which is similar to process-1, but unlike process-1, process-2 uses inter-process

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:05:57 UTC from IEEE Xplore. Restrictions apply.

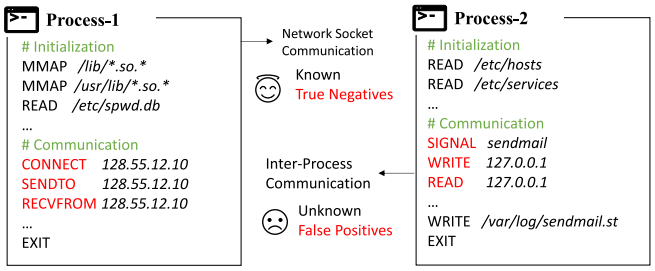


Fig. 1. An example of FP. Process-1 and process-2 are benign *sendmail* processes. But process-2 uses inter-process communication. Process-2 is different from process-1 (red words) and is not in the training data. Thus, *sendmail* processes acting as process-2 are misidentified as FPs.

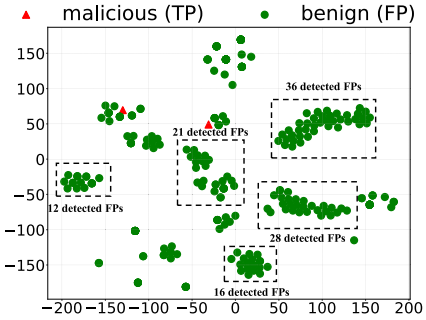


Fig. 2. The t-SNE distribution of process embeddings. The dashed box indicates the detected FPs by SheepHunter.

communication to communicate with *127.0.0.1*, and the events between the process and the socket are *WRITE* and *READ*. The new pattern of *sendmail* becomes an FP as it is not included in the training data. We find that there are 162 *sendmail* processes in the test set, 39 of which are FPs. All the processes with this new pattern (as process-2, 34 in total) are misidentified as anomalies. Furthermore, the detector triggers approximately 23,000 alarms, making it difficult for an analyst to analyze them all. An automatic method of detecting FPs without relying on the known logs and assisting with manual analysis is needed.

C. Problem Statement

The aim of this paper is to automatically identify FP processes from alarms reported by anomaly-based PIDSes. We take alarms generated by PIDSes (i.e., detectors) and the corresponding raw logs as input, and then we output the false positive processes.

1) Threat Model and Assumptions: We assume that attackers can use known or unknown vulnerabilities in the system to perform remote attacks, including gaining unauthorized privileges, stealing information, and so on. In addition, attackers try to hide their behavior, such as blending into the legitimate background and reducing unnecessary activity.

Similar to existing work [7], [11], [12], we assume that the log system is secure. The log system honestly records the behavior of attackers without missing anything. Also, the stored logs are protected and cannot be modified or deleted by attackers. These assumptions can be met by existing work such as [26] and [27].

2) Design Goals: Our FP identification method, SheepHunter, should achieve the following goals.

- **Identification without Prior Knowledge.** SheepHunter can only obtain information from the alarms and the corresponding raw logs. Any prior knowledge about the detected alarms, such as how these alarms are raised, is unknown to SheepHunter. Furthermore, the previous logs, which are always used as training sets, are also unavailable.
- **Black-Box Detector Compatibility.** SheepHunter should be applicable to different detectors (i.e., PIDSes). At the same time, it should not have any information about the detectors, such as the detection algorithms, or the anomaly scores of alarms.
- **Human-Auditable FP Evidence.** Attack investigation and manual validation are important in PIDSes [28], i.e., the alarms should be understandable to humans. The identified FPs should also meet this property. SheepHunter should provide the reason why a process is an FP, and help with manual analysis.

III. Overview of SheepHunter

In this section, we first introduce the key observation of this paper. Then we provide an overview of SheepHunter.

A. Key Observation

Existing PIDSes generate numerous FP processes. While manual analysis for FP identification is resource-intensive, it is possible to identify FPs based on the distribution of benign processes. Specifically, we observe that benign processes are numerous and typically exhibit similar behaviors (i.e., similar function invocation sequences), whereas malicious processes are relatively few and display diverse behaviors due to varied attack objectives. To illustrate this observation, we analyze the distribution of process embeddings generated by [12] on the DARPA-E3 CADETS dataset, shown in Figure 2. The embeddings, which represent process behaviors in a high-dimensional space, reveal the following: For benign processes, the embeddings form several dense clusters (highlighted by dashed boxes), and the processes within each cluster exhibit nearly identical behavioral patterns. In contrast, the malicious processes are scattered far from each other. SheepHunter will identify the processes in dense clusters as FPs and those which are isolated or in sparse clusters as anomalies.

The behavioral difference between the benign and malicious processes causes the density difference. Benign processes are generated by routine system services and user applications, which repeatedly execute similar workflows and interact with common system entities, naturally forming dense behavioral clusters. In contrast, malicious processes are typically task-driven and perform attack-specific operations involving uncommon execution patterns or resource accesses. As a result, they tend to appear as isolated processes or sparse groups in the behavioral space.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:05:57 UTC from IEEE Xplore. Restrictions apply.

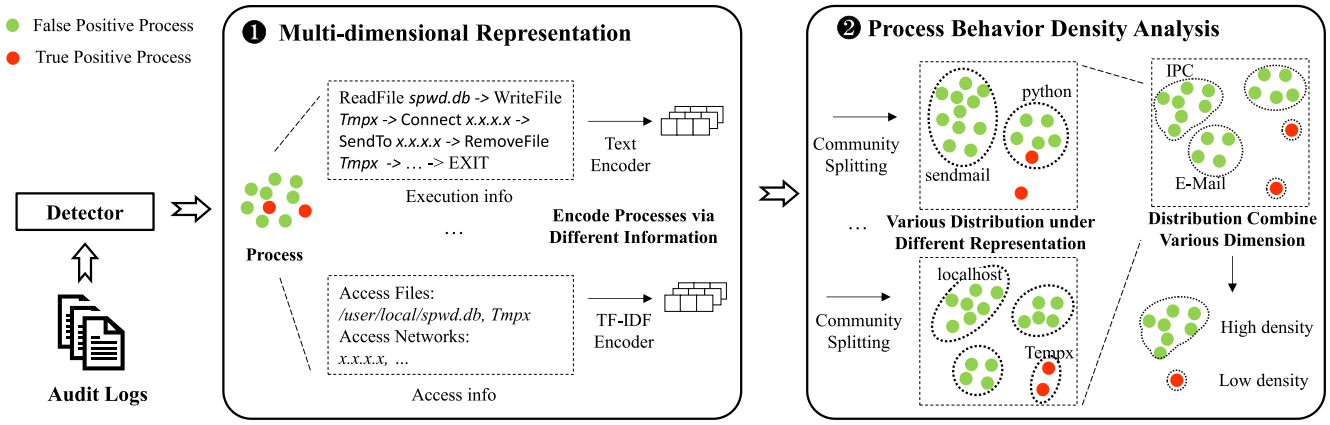


Fig. 3. Overview of SheepHunter.

B. Overview of SheepHunter

Based on the observation, we design an FP detection method, called SheepHunter. Our intuition is to search for processes whose behavior is similar to each other and deem these processes benign. To achieve this, we use process behavior density to quantitatively represent this characteristic of processes. Higher behavior density indicates that a process is more likely to be an FP.

As shown in Figure 3, the workflow of SheepHunter is as follows: (1) **Multi-dimensional process representation**. We take the raw logs and alarms generated by the anomaly detectors as input, and extract the behavior features for processes. We represent each process from multiple dimensions, including its execution context, access control information, and structural information in provenance graphs. The multi-dimensional representation highlights behavioral differences between processes, thereby improving the separability of malicious and benign processes. (2) **Process behavior density analysis.** We design a community splitting algorithm that groups processes with similar representations into the same cluster. Then, we analyze the process behavior density based on community size, i.e., a large community has high density. A process is classified as a false positive only if it appears in high-density clusters across all feature dimensions, ensuring consistent similarity from multiple behavioral perspectives. Process behavior density analysis helps to identify FPs, and also reduces manual effort, i.e., an analyst only needs to analyze representative processes in each community instead of all of them.

IV. Design Details

In this section, we introduce the details of SheepHunter. We first introduce how to preprocess the input raw logs, which is the basis of process representation. Then, we give the details of multi-dimensional process representation. At last, we show how we analyze the process behavior density to identify FPs.

A. Preprocessing for Raw Logs

SheepHunter takes as input alarms reported by PIDSes, along with their corresponding raw logs. In this subsection, we introduce how SheepHunter preprocesses these raw logs to extract the necessary information for process representation.

1) Provenance Graph Construction: System audit logs contain abundant information. To conduct anomaly detection, we should first convert audit logs into provenance graphs. Audit logs contain entities with their attributes and events among these entities. We use entities as nodes, including processes, files, and network sockets. We use command line contents, file names, and IP addresses as node attributes. About the edges, we use events between entities with a type and a timestamp. The resulting provenance graph is a directed graph. For example, if a process p reads a file f at timestamp t, there will be two nodes and one edge between them, as $p \xrightarrow{read,\ t} f$.

Notably, the provenance graph construction here does not violate the goal of black-box detector compatibility. The provenance graph is constructed solely for SheepHunter and is independent of the provenance graph construction used by the detector, which remains unknown to SheepHunter.

2) Formalization for Processes: A process is a node in provenance graphs. We formalize a process node p_i using its one-hop neighbor nodes with edges, as:

$$B_{p_i} \ \{(edge_type, obj_id, obj_name, timestamp)\}, \quad obj \in \mathcal{N}(p_i). \tag{1}$$

where $edge_type$ and $timestamp$ refer to the type of edge (e.g., read) and the timestamp when the corresponding event occurs. obj_id and obj_name mean the universally unique identifier and the name of the neighbor node of the process. We use command line contents, file names, and IP addresses as node names for processes, files, and network sockets, respectively.

B. Multi-Dimensional Process Representation

We devise a multi-dimensional representation for the reported alarm processes based on the information extracted above. Program execution sequence and access control information are two basic attributes of processes. The contextual information of processes in provenance graphs is also useful. Therefore, we choose these three types of information

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:05:57 UTC from IEEE Xplore. Restrictions apply.

to represent processes. Together, they reflect complementary behavioral aspects of a process: how it performs its actions, what resources it accesses, and whether the surrounding entities are common or discriminative in the system. The details are as follows.

1) Execution Information: A process will execute its program, and its semantic information about execution is important. Inspired by [7], we represent a process's features about execution via its one-hop neighbors.

As shown in Equation 1, a process interacts with several entities obj to finish its execution. We use the names of these entities for embedding generation. However, we do not use all of these entities, as some of them are too common and do not provide discriminative information for process representation. For example, system configuration files are commonly loaded by many processes and thus may not help distinguish between processes. Thus, we retain only the more discriminative entities for process representation. We use inverse document frequency (IDF) to identify such entities. We ignore processes and calculate the IDF value for other entities (including files and network sockets), formatted as:

$$IDF(obj) = \frac{|P|}{\sum_{k=0}^{|P|-1} \mathbf{1}_{(obj \in B_{p_k})}}, \ (p_k \in P), \tag{2}$$

where P is the set of processes, $|P|$ is the size of P, and the output of $\mathbf{1}_{condition}$ is "1" if $condition$ is "true" otherwise "0". A higher IDF value means a more unique entity, i.e., interacted with by fewer processes. We filter out entities with an IDF value less than a threshold thr_{IDF}, which are always system files such as system configuration files.

Then we use a language model $Text_Enc$ to generate embeddings for each object of a process p and use the average of these embeddings as the embedding of p, as follows:

$$\mathcal{E}_{execution_p} = \frac{1}{|B'_p|} \sum_{obj_name_k \in B'_p} Text_Enc(obj_name_k), \tag{3}$$

where B'_p is the set obtained by removing objects whose type is process or whose IDF value is low from B_p. We use existing large pretrained text embedding models as $Text_Enc$.

2) Access Information: A process needs to access other entities to finish execution. If processes have the same behavior, they will have similar access control information. Additionally, when processes cooperate to complete a task, the entities with which these processes interact will overlap. Based on this insight, we represent a process's access control information by the entities with which it interacts.

We first filter out the nodes with a low IDF value, which is the same as before. Then we can represent a process p as a $|O|$-dimensional vector:

$$\mathcal{E}_{access_p} = [IDF(obj_k) * \mathbf{1}_{(obj_k \in B'_p)}]_{k=0}^{|O|-1}, \tag{4}$$

where O is the set of all objects and B'_p is the set obtained by removing objects whose type is process or whose IDF value is low from B_p.

This type of embedding seems similar to the execution one, but they are actually not the same. Execution embedding focuses on the execution procedure of a process, while access embedding focuses on the entity with which it interacts. For example, if two processes interact with two files that have the same name, they will have similar execution information but different access information. Similarly, two processes that use *conda* to execute two different *Python* files may have similar access information as they all access entities used for environment setup, but have different execution information which is mainly decided by the name of *Python* files.

3) Structural Information: There is rich and complex contextual information in provenance graphs. We use GAT [29] to learn the structural information in provenance graphs and get process embeddings. In each layer k, the embedding of the node v_i and the structural embedding of process p are formatted as:

$$\begin{aligned} h_i^k &= W_{self} h_i^{k-1} \\ &\quad + GAT(\{h_j^{k-1} : v_j \in \mathcal{N}(v_i)\}), 1 \le k \le l, \\ \mathcal{E}_{structural_p} &= h_p^0 || h_p^1 || \dots || h_p^l, \end{aligned} \tag{5}$$

where l is the number of layers, W_{self} is a learnable parameter, and .||. denotes the concatenation operation. We use a self-supervised way proposed in [12] to obtain structural embeddings. The parameters of the GAT are determined solely by the raw logs being tested and analyzed. The node and edge types are represented using one-hot encodings as the input features of the GAT.

Structural embedding focuses on the contextual information in the whole provenance graph and can be influenced by the entire graph, while the access embedding captures local features from one-hop neighbors. Execution embedding takes node names into consideration, whereas structural embedding only uses node and edge types.

Malicious and benign processes often differ in these three aspects. First, in terms of execution information, malicious processes usually exhibit stronger scripting, batching, and tool-driven patterns. For instance, malicious processes in the Arena-linux dataset show a pattern of *sh -c* wrapped execution of toolchains such as *sqltool* and *fscan* for vulnerability exploration. Second, in terms of access information, malicious processes are more likely to interact with attack-related entities that are rarely accessed by benign processes. For example, in the E3-CADETS dataset, some malicious processes create and access the temporary file *XIM*, which will not be accessed by benign processes. Finally, the structural information is widely used in existing PIDSes [9], [11], [12] to distinguish malicious and benign processes, as it captures the local graph context around processes and helps reveal abnormal relations with key attack entities. Thus, these three dimensions provide complementary signals for distinguishing malicious and benign processes.

C. Process Behavior Density Analysis

After obtaining process representations, we measure process behavior density in each feature dimension by applying a community splitting algorithm [16], which clusters processes with similar behavior. A process is classified as a false positive only if it appears in a high-density cluster across all the above

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:05:57 UTC from IEEE Xplore. Restrictions apply.

representations. The processes in such a cluster also remain grouped together under each individual representation, ensuring consistent similarity from multiple behavioral perspectives. This multi-dimensional density analysis not only helps FP identification but also reduces manual effort, as analysts can focus on representative processes within each cluster rather than examining all alarms.

There are three main steps. For each dimension of the process representation, we first generate a similarity matrix $S = [s_{ij}]_{|P|\times|P|}$ to describe the similarities among processes. A higher s_{ij} means a higher similarity between p_i and p_j. Second, we use an iterative community splitting algorithm to split processes with similar behavior into a community. At last, we combine the results from different dimensions and obtain the final community splitting results. The processes in a community are similar across all dimensions, and the processes in a large-size community have a high behavior density and are classified as FPs. The details are as follows.

1) Similarity Matrix Generation: The similarity matrix depends on how processes are represented. For structural and execution representations, the embeddings are feature vectors and we use cosine similarity. For the access representation, we sum the IDF values of overlapping objects between p_i and p_j. The similarity matrix is formatted as:

$$s_{ij} = \begin{cases} \dfrac{\mathcal{E}_{p_i} \cdot \mathcal{E}_{p_j}}{\|\mathcal{E}_{p_i}\|_2 \|\mathcal{E}_{p_j}\|_2}, & \mathcal{E} \in \{\mathcal{E}_{structural}, \mathcal{E}_{execution}\} \\ \sqrt{\mathcal{E}_{p_i} \cdot \mathcal{E}_{p_j}}, & \mathcal{E} \in \{\mathcal{E}_{access}\} \end{cases} \tag{6}$$

where $\|.\|_2$ is the 2-norm, and we set $s_{ii} = 0$.

2) Iterative Community Splitting: We want to divide processes with similar behavior into a community. Based on the similarity matrix, we use the Louvain algorithm to complete the community detection. The Louvain algorithm partitions communities by maximizing the modularity Q:

$$Q = \frac{1}{2m} \sum_{i,j} \left[s_{ij} - \frac{s_i s_j}{2m} \right] \mathbf{1}_{(c_i == c_j)}, \tag{7}$$

where m is the number of non-zero elements in S, $s_i = \sum_{k=0}^{|P|-1} s_{ik}$, and c_i is the community of process p_i.

Although the Louvain algorithm is accurate and fast, its result is a global optimal solution, which is not appropriate in FP detection. In detail, for a process p whose community is c, the result illustrates that p is more similar to the processes in c than to those not in c. But the result cannot illustrate that processes in c are similar to each other. To deal with this issue, we run the Louvain algorithm again for each community to get smaller communities. We repeat this procedure until a community cannot be split or its size is smaller than a threshold thr_{comm}. Finally, we get communities for all processes under a dimension representation. We think that processes in the same community are similar in the corresponding dimension. The whole procedure is shown as Algorithm 1 and an example is shown in Figure 4.

3) Multi-Dimensional Combination: For each dimensional representation, we run iterative community splitting and get the result C_d. Then we take the intersection of the communities from each dimension for each process p, formatted as:

$$C_{d_p} = \{p, p_{d1}, p_{d2}, \ldots\}, \; C_p = \bigcap_d C_{d_p}. \tag{8}$$

We use the three representations mentioned before as three different dimensions. Then, if the size of a community is greater than the threshold thr_{comm} (i.e., processes with high behavior density), we consider processes in that community to be benign (i.e., FPs), otherwise malicious.

The processes in C_p are consistently assigned to the same community in each individual dimension. This suggests that they are highly similar, so analyzing one process is often sufficient to determine the nature of the entire community, which significantly reduces manual effort. To further guard against exceptional cases, we additionally design a statistics-based refinement step to split the processes within a community into representative processes (i.e., processes that characterize the common behaviors of the community) and special processes (i.e., processes that slightly deviate from the common behaviors). Details are presented in §V-C. This design reduces the manual effort while helping preserve analysis coverage.

Algorithm 1 Iterative Community Splitting

Input: Similarity Matrix S
Output: Community Splitting Result C

```
C ← [ ], Queue ← [S]
while len(Queue) ≠ 0 do
    s = Queue.pop()
    // c is a list of community
    c = community_split(s)
    if len(c) == 1 then
        // community cannot be split
        C.append(c[0])
        continue
    end
    for c_k in c do
        // c_k : {p_{k_0}, p_{k_1}, ...}
        if len(c_k) ≥ thr_comm then
            Queue.append(S[c_k][c_k])
        else
            C.append(c_k)
        end
    end
end
return C
```

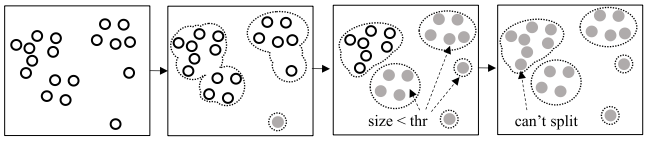


Fig. 4. An example of iterative community splitting. The community splitting will not stop until the size is less than thr_{comm} or the community cannot be split.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:05:57 UTC from IEEE Xplore. Restrictions apply.

TABLE I
STATISTICAL INFORMATION OF DATASETS

Dataset	# of Edge	# of Node	# of Process	# of Malicious Process
E3-TRACE	8,570,857	3,197,278	22,365	8
E3-THEIA	24,063,604	1,558,101	59,933	23
E3-CADETS	15,585,878	1,172,567	94,321	15
Arena-win10	14,041,869	20,563	1,041	19
Arena-linux	7,891,873	61,822	7,929	136
Arena-wins12	8,275,352	10,835	1,928	43

V. EVALUATION

In this section, we evaluate the performance of SheepHunter. We conduct several experiments to answer the following questions:

- Q1 How many FPs can SheepHunter identify for various PIDSes?
- Q2 What is the performance of SheepHunter in real usage when combined with existing PIDSes?
- Q3 Is SheepHunter robust to adaptive attacks?
- Q4 How do different designs work in SheepHunter?
- Q5 How much overhead does SheepHunter bring?

To answer **Q1**, we use 4 different PIDSes to generate alarms and evaluate how SheepHunter identifies FPs for them on two widely used datasets (§V-B). For **Q2**, we simulate a real scenario where PIDSes need to continuously detect attacks and we evaluate how SheepHunter helps with model adaptation and manual analysis (§V-C). For **Q3**, we design several adaptive attacks and evaluate the performance of SheepHunter under these attacks (§V-D). For **Q4** and **Q5**, we conduct an ablation study on different designs of SheepHunter (§V-E), and measure the time and memory usage of SheepHunter (§V-F).

A. Experiment Setting

In this subsection, we show the experiment settings, including the dataset and detectors. Then we show the implementation of SheepHunter.

1) Dataset: We evaluate SheepHunter on 2 open-source datasets with 6 sub-datasets, the statistical information of these datasets is shown in Table I. (1) DARPA-TC-E3 [17]. DARPA Engagement 3 datasets are generated in a Red vs. Blue team exercise. The red team exploits vulnerabilities to perform APTs. CADETS, THEIA, and TRACE are three sub-datasets that are widely used in provenance-based detection. We use these three sub-datasets for the evaluation (denoted as E3-CADETS, E3-THEIA, and E3-TRACE), and the ground truth is from [5]. (2) Arena Dataset. NODLINK [7] builds two datasets that contain the data of internal security evaluation engagement of Sangfor. The Industrial Arena dataset is closed-source. So, we use the In-lab Arena dataset for evaluation. The In-lab Arena dataset contains the logs of 5 attacks on 3 different hosts, including Windows 10 desktop, Windows Server 2012, and Ubuntu 20.04. These three sub-datasets are denoted as Arena-win10, Arena-wins12, Arena-linux.

2) Detector: Detectors are used to generate alarms. We select three SOTA approaches, including NODLINK [7], MAGIC [12], and FLASH [11]. NODLINK applies the Steiner tree to detect anomalous processes. MAGIC and FLASH are two approaches that generate alarms based on GNN embeddings. We use the default settings of these three approaches.

In addition to existing approaches, we design an overfitting approach for comparison (denoted as OVERFIT). Our principle is to try not to leave out any malicious process, i.e., to keep false negatives as low as possible. To achieve this goal, we generate alarms with a strict policy. We decide whether a process is malicious according to its corresponding events. In the training phase, when all processes are benign, we record the events for each process node in the graph, and get a set containing all known and benign events, formatted as $S_{known} = \{(edge_type, obj_name)\}$. In the test phase, only if all of a process's events appear in S_{known} (i.e., all of its events are benign and known) will we consider it as a benign process, or a malicious one.

Note that training on benign logs is only for the detector, and SheepHunter can only see the alarms given by each detector and the raw logs of the test data. The training logs and the graph constructed in the detector are invisible to SheepHunter. In addition, we clarify the mismatches with the results in their original papers: NODLINK uses its own ground truth, which is not publicly available. This difference in ground truth leads to the inconsistency between our reproduced results and those reported in the original paper. By contrast, MAGIC and FLASH use the same ground truth as SheepHunter. The remaining discrepancies mainly arise because we evaluate performance at the process level, whereas their original results are reported at the node level. This process-level evaluation reflects both the central role of processes as the subject nodes and the natural unit of investigation in provenance graphs, and the fact that process-level evaluation directly measures the effectiveness of SheepHunter in reducing false positives. Also, we verified that the node-level results are generally consistent.

3) Implementation of SheepHunter: We implement SheepHunter using the Python language. For the representation of processes, we use an encoder-only large pretrained model [30] for execution representation, and GAT [29] for structural representation. The hyperparameters of the GAT model are set as follows: the embedding dimension is 64, the number of GNN layers is 3, the dropout rate is 0.1, and the number of attention heads is 4. We train the GAT for 50 epochs. For FP detection, we use the Louvain community algorithm with its default settings in the igraph library [31]. For hyperparameters of SheepHunter, we set thr_{IDF} to 0.1 (i.e., we filter out an entity if it is interacted with by more than 10% of processes), and thr_{comm} to 10 (i.e., processes in a community whose size is less than 10 are deemed malicious).

B. Performance of Detecting FPs

In this subsection, we evaluate the performance of SheepHunter in detecting FPs when used in conjunction with different detectors. We use different detectors to generate alarm processes and then SheepHunter detects FPs. We use TPs, FPs, recall, precision, and F1 as metrics. The absolute

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:05:57 UTC from IEEE Xplore. Restrictions apply.

TABLE II

THE PERFORMANCE OF SHEEPHUNTER IN DETECTING FPS WITH DIFFERENT DETECTION APPROACHES ON E3 DATASET. "-" MEANS THAT THE DETECTOR DOES NOT GENERATE ANY ALARM. "ORG" AND "RES" REPRESENT THE RESULTS WITHOUT AND WITH SHEEPHUNTER, RESPECTIVELY. THE LAST ROW FOR EACH DETECTOR REPORTS THE RETENTION RATES OF TPS AND THE REDUCTION RATES OF FPS

		E3-TRACE					E3-THEIA					E3-CADETS				
		R	P	F1	TP	FP	R	P	F1	TP	FP	R	P	F1	TP	FP
MAGIC	org	0.375	0.0029	0.0057	3	1046	0.9565	0.1068	0.1921	22	184	0.1333	0.0066	0.0125	2	302
	res	0.375	0.0074	0.0145	3	403	0.9565	0.1746	0.2953	22	104	0.1333	0.0127	0.0231	2	156
					100%	61.47%				100%	43.48%				100%	48.34%
FLASH	org	0.0667	0.0041	0.0077	1	224	0.0435	0.0075	0.0127	1	133	0.0667	0.0041	0.0077	1	244
	res	0.0667	0.0172	0.0274	1	57	0.0435	0.0147	0.0219	1	67	0.0667	0.0172	0.0274	1	57
					100%	74.55%				100%	49.62%				100%	76.64%
NODLINK	org	-	-	-	-	-	-	-	-	-	-	0.0667	0.0333	0.0444	1	29
	res	-	-	-	-	-	-	-	-	-	-	0.0667	0.0333	0.0444	1	29
OVERFIT	org	0.375	0.0074	0.0145	3	403	0.8696	0.5556	0.6779	20	16	0.8667	0.0053	0.0105	13	2451
	res	0.375	0.0098	0.0192	3	302	0.8696	0.5556	0.6779	20	16	0.8667	0.0095	0.0188	13	1357
					100%	25.06%				100%	0.00%				100%	44.63%

TABLE III

THE PERFORMANCE OF SHEEPHUNTER IN DETECTING FPS WITH DIFFERENT DETECTION APPROACHES ON ARENA DATASET. "-" MEANS THAT THE DETECTOR DOES NOT GENERATE ANY ALARM. "ORG" AND "RES" REPRESENT THE RESULTS WITHOUT AND WITH SHEEPHUNTER, RESPECTIVELY. THE LAST ROW FOR EACH DETECTOR REPORTS THE RETENTION RATES OF TPS AND THE REDUCTION RATES OF FPS

		Arena-win10					Arena-linux					Arena-wins12				
		R	P	F1	TP	FP	R	P	F1	TP	FP	R	P	F1	TP	FP
MAGIC	org	1	0.0561	0.1063	136	2287	1	0.0544	0.1033	19	330	1	0.0351	0.0678	43	310
	res	0.5735	0.0679	0.1214	78	1071	0.6842	0.0714	0.1293	13	169	0.6279	0.0801	0.1421	27	118
					57.35%	53.17%				68.42%	48.79%				62.79%	61.94%
FLASH	org	0.0074	0.0263	0.0115	1	37	-	-	-	-	-	-	-	-	-	-
	res	0.0074	0.0263	0.0115	1	37	-	-	-	-	-	-	-	-	-	-
NODLINK	org	0.7132	0.1394	0.2331	97	599	0.3684	0.0507	0.0892	7	131	0.8837	0.0943	0.1704	38	365
	res	0.625	0.2163	0.3213	85	308	0.3684	0.0569	0.0986	7	116	0.5581	0.0927	0.1589	24	235
					87.63%	48.58%				100%	11.45%				63.16%	35.62%
OVERFIT	org	0.9853	0.1113	0.2	134	1070	0.7895	0.122	0.2112	15	108	0.9767	0.1405	0.2456	42	257
	res	0.5662	0.1339	0.2166	77	498	0.6316	0.1379	0.2264	12	75	0.6046	0.1156	0.1941	26	199
					57.46%	53.46%				80.00%	30.56%				61.90%	22.57%

number of TPs and FPs reflects how SheepHunter detects FPs. The most ideal case is that SheepHunter reduces FPs to "0" and keeps TP unchanged compared to the original alarms. The reason we use the absolute number is that even with high precision, there will be many FPs that need to be analyzed manually. The recall, precision, and F1 score focus on the performance of anomaly detection. The results are shown in Table II (DARPA-TC-E3) and Table III (Arena). The "org" and "res" rows show the results without and with FP detection, respectively. The last row for each detector reports the retention rates of TPs and the reduction rates of FPs.

1) Anomaly Detection and FP Detection Results: For anomaly detection, we compare the results of overfit approaches with FP detection (i.e., our approach) against other existing results without FP detection. Our approach achieves the highest F1 score in most cases. MAGIC and FLASH are two detectors that extract information from nodes. MAGIC achieves high recall, while FLASH performs poorly at detection, e.g., it detects no anomalies on the Arena dataset. NODLINK performs well on the Arena dataset.

For FP detection, we concentrate on the absolute number of TPs and FPs. Excluding the FLASH results on Arena and the NODLINK results on E3, our method reduces FPs by an average of 40.68% (from 11.45% to 61.94%) on the Arena dataset and 47.09% (from 0% to 76.64%) on the E3 dataset across the evaluated detector-dataset pairs. Notably, even when the detector produces a small number of FPs, SheepHunter can still detect FPs. For example, on the E3-TRACE dataset, the FLASH detector produces 224 FPs, and SheepHunter still detects 74.55% of them. The results indicate that SheepHunter is effective for FP detection.

2) False Negative Analysis: SheepHunter detects FPs without prior knowledge, which is aggressive and can lead to additional false negatives, i.e., anomalies that are detected by anomaly detection approaches but considered as false positives by SheepHunter. This may be used by attackers to evade detection. So it is necessary to analyze the false negatives. On the E3 dataset, SheepHunter does not cause any extra false negatives. On the Arena dataset, there are extra false negatives. These false negatives are part of attacks, and they are clustered into a community by SheepHunter due to their similar behavior. These similarities mainly result from the automated attack strategies. For example, in the Arena-win10 dataset, there are 18 processes that execute different attack techniques through an automated script like *Invoke-AtomicTest TXXX.XX*, whereas the Arena-wins12 dataset contains 10 processes that correspond to batch cleaning commands such as *cmd.exe /c del /f /q*. While such a phenomenon does pose a risk of

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:05:57 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
RESULTS OF THE MITIGATION OF FNS

Dataset		TP	FP
Arena-linux	Before	12 (80.00%)	75 (30.56%)
	After	13 (86.67%)	76 (29.63%)
Arena-win10	Before	77 (57.46%)	498 (53.46%)
	After	110 (82.09%)	514 (51.96%)
Arena-wins12	Before	26 (61.90%)	199 (22.57%)
	After	31 (73.81%)	199 (22.57%)

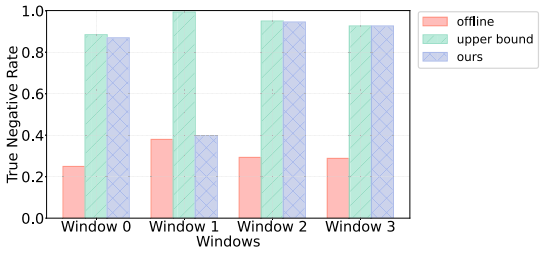


Fig. 5. How SheepHunter helps to improve existing detector. "offline" means that the model is not updated, and "upper bound" means that the model is updated with the entire data.

attack evasion, we still believe our approach is valuable. First, SheepHunter still retains other malicious processes, which can be used to attack detection and investigation. Second, SheepHunter clusters these attack processes together, which can amplify the features of attacks and be beneficial for manual analysis or further detection.

To mitigate the FNs introduced by SheepHunter, we implement a lightweight post-processing method. For each process removed by SheepHunter, we reconstruct its one-hop directed provenance paths, each containing one or two process-to-process edges. We then associate each process with tactic tags based on its command line; for example, a process executing *reg add* is assigned the *Persistence* tag. Following MITRE ATT&CK [32], we define five tactic tags: *Download*, *Execution*, *Persistence*, *Lateral Movement and Credential Access*, and *Defense Evasion*. A process can be assigned multiple tactic tags. We then choose the shortest reconstructed path that covers the largest number of distinct tactic tags as the representative path. The removed process is considered an FN only when the representative path covers at least three distinct tactic tags and the candidate process itself is associated with at least one tactic tag.

Table IV shows that our post-processing method can recover FNs with limited additional FPs. For example, on Arena-win10, it increases the TP rate from 57.46% to 82.09% with only 16 additional FPs, while on Arena-wins12 it recovers five additional TPs without introducing any new FPs. We do not apply this method to E3 because command lines in the datasets contain little semantic information for reliable tactic-tag mapping, and SheepHunter introduces no FNs on E3. We note that this method is one possible strategy for FN mitigation. More sophisticated attacks may require more comprehensive rules for both provenance-path identification and tactic mapping. We further discuss the mitigation of FNs in §VI.

C. Usage of SheepHunter

In real scenarios, PIDSes should continuously detect attacks, and manual analysis results will be used to adapt detectors. In this subsection, we simulate a real scenario and we evaluate how SheepHunter helps with model adaptation and manual analysis.

We use MAGIC as the detector and perform the experiment on the E3-CADETS dataset. We divide the E3-CADETS dataset into multiple windows at 12-hour intervals and get 7 windows in total. Then we use the first 3 windows to train the detector and the others for testing. For each test window, we evaluate the performance of the anomaly detection (case 1) and the manual costs (case 2).

1) Case 1: Model Adaptation: In this case, we show how SheepHunter helps with model adaptation. We focus on the performance of anomaly detection and use the worst and best situations for comparison. The worst situation is that the model is not updated after the training phase. In the best situation, we update the model with the benign data on that day and test on the same day. For SheepHunter, we reconstruct the graph consisting of the detected FP processes and use the reconstructed graph to update the model. Note that FP detection, graph reconstruction, and model adaptation can be done automatically without any manual effort.

The key procedure for model adaptation is to reconstruct a provenance graph from FPs. In the provenance graph to be detected, we drop all process nodes that are alarms detected by detectors and are not FPs. Then we drop the nodes with no edges. In this way, we obtain a reconstructed provenance graph that contains only the behavior of benign processes. The provenance graph can be used to improve detectors in appropriate ways, e.g., a GNN-based detector can be retrained on this new graph. The whole procedure can be done automatically without any manual effort.

Note that there are entities interacted with by both benign and malicious processes, and we just keep these entities in the reconstructed graph. It is reasonable to do so. These entities are always required for program execution and are benign, such as system configuration files, while the malicious entities are not interacted with by benign processes. Thus, we will not misuse malicious data for training. What's more, if entities of benign processes are dropped, the behavior of benign processes is broken, which has a negative impact on training.

Figure 5 shows the true negative rates (TNRs) of the detector in each window. As we can see, when the model is not updated (offline), the TNR is less than 40%. With the help of SheepHunter, the TNR is more than 80% and is close to the upper bound (trained with entire graphs), and at the same time, the TPR on each window does not decrease. The result indicates that SheepHunter can effectively improve the detector. But there is still one failed window. To further explain why SheepHunter works or does not work, we record the node and edge information of the reconstructed graphs for each window. Figure 5 shows the TNRs at different levels

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:05:57 UTC from IEEE Xplore. Restrictions apply.

TABLE V
TRUE NEGATIVE RATES AT DIFFERENT LEVELS AFTER RECONSTRUCTING THE GRAPH FOR EACH WINDOW

	Process-level	Node-level	Edge-level
Window0	88.99%	71.52%	75.12%
Window1	98.60%	27.74%	63.14%
Window2	93.42%	90.09%	88.37%
Window3	76.41%	83.37%	69.97%

TABLE VI
HOW SHEEPHUNTER HELPS WITH MANUAL ANALYSIS

	# of Process	# Need to Analyze	Reduction ↑
Window0	12739	63+50	112.7
Window1	143	5+2	20.4
Window2	14363	46+11	251.9
Window3	357	11+0	32.4

after reconstructing the graph for each window. The result indicates that SheepHunter can detect most of the FPs at the process level (TNRs are 76.4%~98.6%). Intuitively, a higher process-level TNR will result in a higher TNR at the node and edge level, which is consistent with the results in all windows except Window1. Window1 has a long tail effect, i.e., a small number of processes (less than 2%) are connected to a large number of object nodes (about 72%). SheepHunter fails to reconstruct this portion of the nodes, causing the failure of model adaptation.

About FNs of SheepHunter, the attack happens in Window1, so we only show the FNs in Window1. In the offline setting, MAGIC produces 12,816 malicious nodes (2 of them are processes). SheepHunter filters out 98.60% false positive processes, while keeping all the malicious processes. Furthermore, the corresponding malicious nodes will not appear in the reconstructed graph, thus, MAGIC can still detect 12,816 malicious nodes after model adaptation.

2) Case 2: Manual Analysis: SheepHunter divides processes with similar behavior into the same community, reducing the number of processes to analyze. Unlike existing approaches that select only a single entity per community for manual analysis (e.g., WATSON [33]), our method additionally identifies special processes that deviate from the community, so as to avoid overlooking atypical yet potentially malicious processes. We apply a statistics-based refinement step to split the processes within a community into a representative process and several special processes. In a community, almost all processes are similar to the representative process, except for special processes. These two types of processes are selected according to the three-σ principle in each community. Details are as follows.

First, we use statistical information to roughly analyze a community. Specifically, we record the number of entities and events of each process in a community, and calculate the mean and standard error of these two statistics. Then we use the three-σ principle to find special processes in a community, i.e., processes whose number of interacted entities or events is outside the three-σ range are considered as special processes. Finally, only one representative process and a few special processes need to be analyzed for a community.

Table VI shows the results for each window. The second column of the table shows the counts of representative processes and special processes (denoted as "# of representative processes" and "# of special processes"). SheepHunter reduces the number of processes to analyze from 251 times to 20 times. For example, in Window0, 12,739 FP processes are detected and SheepHunter divides them into 63 communities. There are 63 representative processes for each community and 50 selected special processes whose behavior may be different from those in the same community. A researcher only needs to analyze about 113 processes instead of all of them to investigate FPs.

Returning to Case 1, SheepHunter fails to automatically detect FPs and update the detector in Window1, and manual analysis can help solve this problem. We sort the processes by the number of objects they connect to and manually analyze the top-5 processes. We find that a Python2.7 process repeatedly creates and closes different objects (it may malloc and free memory), and this process connects to 172,541 (about 57.4% of all nodes in Window1). Missing this process in the reconstructed graph causes the model adaptation to fail. After manually analyzing the top-5 processes and adding them to the reconstructed graph, the detector achieves 99.7% TNR, close to the upper bound. This result indicates that with the automatic FP detection of SheepHunter and some manual effort, we can adapt the detector well. What's more, the number of detected malicious nodes does not decrease with such manual effort and model adaptation, which means no extra FNs are caused by SheepHunter.

D. Adaptive Attack

SheepHunter may cause false negatives, which can be exploited by attackers to evade detection. In this subsection, we design several adaptive attacks. To exclude the influence of the detector, we use the overfit approach as the detector, which has a very strict policy and can detect almost all malicious processes.

We design four adaptive attacks, including (1) *Subgraph:* [34] points out that attackers can sample a subgraph of the benign graph and let it be part of the attack to make its context similar to benign activities. The *Subgraph* attack is designed based on this idea. We select a benign process (both randomly and by degree) and expand it to a subgraph containing 16 ~ 512 nodes, which is then deemed as part of the attack. (2) *Behavior:* Attackers can make their behavior similar to benign processes to mislead SheepHunter. The *Behavior* attack randomly selects several benign processes and adds their corresponding events to the execution sequences of malicious processes. (3) *Access:* Similar to the *Behavior* attack, the *Access* attack randomly selects several benign processes and accesses the same entities as these benign processes. (4) *Repeat:* This attack aims to make the attack processes form a large community and thus evade detection. To simulate repeated executions of the attack, we extract the 1-hop attack subgraph, namely, all attack processes and the

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:05:57 UTC from IEEE Xplore. Restrictions apply.

TABLE VII
THE PERFORMANCE OF SHEEPHUNTER UNDER DIFFERENT ADAPTIVE ATTACKS

Dataset	Method	R	P	F1	TP	FP
E3-TRACE	-	0.3750	0.0098	0.0192	3	302
	Subgraph	0.3750	0.0108	0.0210	3	274
	Behavior	0.3750	0.0103	0.0201	3	288
	Access	0.3750	0.0108	0.0210	3	275
	Repeat	0.2500	0.0072	0.0141	2	274
E3-THEIA	-	0.8696	0.5556	0.6779	20	16
	Subgraph	0.8696	0.5556	0.6779	20	16
	Behavior	0.8696	0.5556	0.6779	20	16
	Access	0.1739	0.2000	0.1860	4	16
	Repeat	0.0000	0.0000	0.0000	0	16
E3-CADETS	-	0.8667	0.0095	0.0188	13	1357
	Subgraph	0.8667	0.0095	0.0187	13	1361
	Behavior	0.8667	0.0096	0.0189	13	1348
	Access	0.8667	0.0100	0.0198	13	1283
	Repeat	0.8667	0.0096	0.0189	13	1344
Arena-win10	-	0.5662	0.1339	0.2166	77	498
	Subgraph	0.5588	0.1426	0.2272	76	457
	Behavior	0.5588	0.1336	0.2156	76	493
	Access	0.5588	0.1394	0.2232	76	469
	Repeat	0.6250	0.1574	0.2514	85	455
Arena-linux	-	0.6316	0.1379	0.2264	12	75
	Subgraph	0.6316	0.1446	0.2353	12	71
	Behavior	0.6316	0.1379	0.2264	12	75
	Access	0.6316	0.1379	0.2264	12	75
	Repeat	0.1579	0.0526	0.0789	3	54
Arena-wins12	-	0.6046	0.1156	0.1941	26	199
	Subgraph	0.6046	0.1156	0.1940	26	199
	Behavior	0.6046	0.1156	0.1940	26	199
	Access	0.6046	0.1156	0.1940	26	199
	Repeat	0.0000	0.0000	0.0000	0	201

entities directly connected to them, and replay this subgraph multiple times (2 ∼ 16 times). In each replay, the attack processes are assigned new UUIDs, while the connected non-process entities are kept unchanged.

We show the results of these adaptive attacks in Table VII. The shown metric is the lowest TP among different attack settings, which reflects the worst case of SheepHunter. The *Subgraph* and *Behavior* attacks have little influence on the performance of SheepHunter. The reason is that such attacks are hard to mimic benign processes in all dimensions. The *Subgraph* attack changes the structural information, but does not impact other dimensions, while the *Behavior* attack changes the execution information, but the access information is still different from benign processes. The *Access* attack has a significant influence on the E3-THEIA dataset, as the accesses to entities change the three dimensions a lot. The *Repeat* attack is the most effective attack, as it can make the attack processes form a large community and thus evade SheepHunter. However, such an approach will reduce the stealthiness of the attack and make it easier to be detected by other approaches. We will further discuss this in §VI. What's more, SheepHunter still produces TPs on the E3-TRACE, E3-CADETS and Arena-win10 datasets.

E. Ablation Study

In this subsection, we perform ablation studies on SheepHunter. We first look at how the multi-dimensional representation influences the performance. Then, we explore the impact of the most important hyperparameter, thr_{comm}. At last, we compare the performance of different community algorithms. In these studies, we use OVERFIT as the detector, and we remove the E3-THEIA dataset as OVERFIT performs so well that SheepHunter cannot detect any FPs.

TABLE VIII
FP DETECTION UNDER LEAVE-ONE-OUT DIMENSION SETTINGS

Dataset	Dimension	R	P	F1	TP	FP
E3-TRACE	all	0.3750	0.0095	0.0185	3	314
	w/o execution	0.3750	0.0116	0.0226	3	255
	w/o access	0.3750	0.0105	0.0203	3	284
	w/o structural	0.3750	0.0099	0.0193	3	300
E3-CADETS	all	0.8667	0.0093	0.0183	13	1389
	w/o execution	0.8667	0.0128	0.0253	13	1000
	w/o access	0.8667	0.0112	0.0222	13	1145
	w/o structural	0.8667	0.0099	0.0196	13	1297
Arena-win10	all	0.5588	0.1423	0.2268	76	455
	w/o execution	0.4559	0.1640	0.2412	62	316
	w/o access	0.5588	0.1517	0.2386	76	425
	w/o structural	0.3750	0.1146	0.1755	51	394
Arena-linux	all	0.6316	0.1379	0.2264	12	75
	w/o execution	0.6316	0.1846	0.2857	12	53
	w/o access	0.6316	0.1558	0.2500	12	65
	w/o structural	0.6316	0.1379	0.2264	12	75
Arena-wins12	all	0.6046	0.1156	0.1940	26	199
	w/o execution	0.5814	0.1389	0.2242	25	155
	w/o access	0.6046	0.1156	0.1940	26	199
	w/o structural	0.6046	0.1226	0.2039	26	186

1) Lack of Dimension: We perform a leave-one-out study on the three dimensions to show the effect of each dimension. The results are shown in Table VIII. When we remove one dimension, there are more FPs detected and higher precision, but fewer TPs and lower recall. This is because fewer dimensions will lead to fewer constraints, which will cause more processes to be divided into the same community and thus be considered as FPs. However, when a dimension is missing, it increases the risk of FNs. For example, in the Arena-win10 dataset, the lack of execution or structural representation leads to a decrease in TPs. In addition, according to the results in the table, the access dimension seems a bit redundant, but in fact, when attackers attempt semantic mimicry, it will play a role. In the *Behavior* attack in §V-D, there are only 4 TPs on the E3-THEIA dataset without the access dimension, while there are still 20 TPs without the execution dimension. These results indicate that while the lack of a dimension may increase the number of FPs detected, it also increases the risk of FNs. The multi-dimensional features can complement each other to minimize FNs as much as possible.

2) Hyperparameters Study: thr_{comm} is the most important hyperparameter in SheepHunter. We choose E3-CADETS and Arena-win10 where the detector causes the most FPs as datasets, and set thr_{comm} from 0 to 40. As for the metric, instead of the absolute number of TPs and FPs, we use the ratios of TPs and FPs to the original results. For example, if the number of TPs and FPs on Arena-win10 are 134 and 1070, and SheepHunter detects 77 TPs and 498, then the metric is 57% and 47%. This metric can directly reflect the impact of thr_{comm} on TPs and FPs.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:05:57 UTC from IEEE Xplore. Restrictions apply.

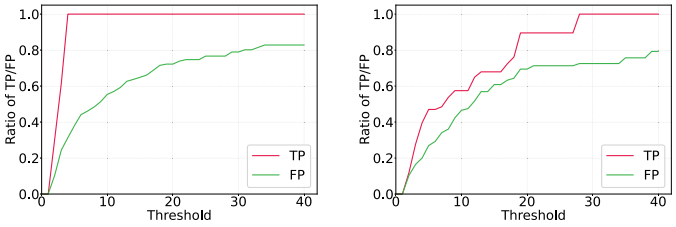


Fig. 6. How thr_{comm} influences the performance of SheepHunter.

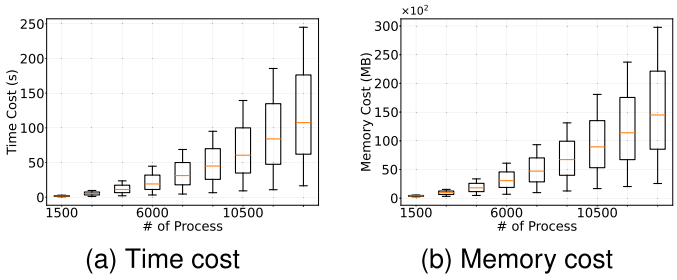


Fig. 7. Time cost and memory cost of SheepHunter.

TABLE IX
INFLUENCES OF DIFFERENT COMMUNITY SPLITTING ALGORITHMS

Dataset		R	P	F1	TP	FP
E3-TRACE	louvain_iter	0.375	0.0098	0.019	3	304
	kmedoids_iter	0.375	0.0074	0.0145	3	403
	dbscan	0.0	0.0	0.0	0	56
E3-CADETS	louvain_iter	0.8667	0.0099	0.0197	13	1295
	kmedoids_iter	0.4	0.0031	0.0061	6	1944
	dbscan	0.0	0.0	0.0	0	4
Arena-win10	louvain_iter	0.5588	0.1431	0.2279	76	455
	kmedoids_iter	0.7206	0.1429	0.2384	98	588
	dbscan	0.0368	0.0641	0.0467	5	73
Arena-linux	louvain_iter	0.6316	0.1379	0.2264	12	75
	kmedoids_iter	0.7895	0.122	0.2112	15	108
	dbscan	0.2632	0.1250	0.1694	5	35
Arena-wins12	louvain_iter	0.6046	0.1156	0.194	26	199
	kmedoids_iter	0.6046	0.1281	0.2114	26	177
	dbscan	0.1395	0.3750	0.2033	6	10

Figure 6 shows the results. In summary, increasing thr_{comm} raises the threshold for a community to be considered as an FP. Thus, as thr_{comm} increases, there are more TPs and FPs. The results on two datasets are different, which mainly depends on the number of benign processes and malicious processes. On E3-CADETS, attackers use 15 processes to connect to a *Nginx* server, download a malicious payload, elevate to root privilege, and attempt (but fail) to inject another payload into *sshd*. These processes do not resemble each other or other benign processes, so a small thr_{comm} can detect all the TPs, while 64.10% FPs are detected. On Arena-win10, there are 40 processes (in 3 communities) that elevate system privileges and 27 processes (14 of which are part of attacks) that delete registries. Thus, only a large thr_{comm} can maintain the detection of TPs, but it also results in a large number of FPs. When all TPs are detected, there are 64.10% and 27.48% FPs detected on E3-CADETS and Arena-win10, respectively.

3) Influence of Community Splitting Algorithms: To demonstrate the effectiveness of the designed community splitting algorithm, we compare it with other algorithms: (1) We replace the Louvain algorithm with the K-Medoids algorithm [35] and run it iteratively as Algorithm 1. We let K-Medoids divide the processes into 2 communities in each iteration. (2) We use a widely used density-based clustering approach, DBSCAN [36], to complete the community splitting. We implement these approaches using the Scikit-learn library [37].

The results are shown in Table IX. In summary, the Louvain algorithm and the K-Medoids algorithm with the iteration design achieve comparable results, and the former is slightly better than the latter, while DBSCAN performs the worst. This result indicates that the iteration design in SheepHunter works, and the Louvain algorithm is good at community finding, so it achieves the best performance. DBSCAN is influenced by its parameters, and a more appropriate parameter setting will result in better performance, but this affects its universality. Our design is less parameter-sensitive and more universal.

F. Overhead of SheepHunter

We evaluate the overhead of Algorithm 1 in this subsection. The algorithm consists of two parts, community splitting and the outer loop. In terms of time complexity, the Louvain community algorithm is $O(nlogn)$, where n is the number of processes. For the outer loop, processes are constantly being divided, and the average number of outer loop executions is $logn$. So the time complexity is theoretically $O(nlog^2n)$. In terms of memory cost, the Louvain algorithm needs to build a graph in which nodes and edges are the main memory consumers.

We conduct FP detection under different numbers of processes and different representations, as shown in Figure 7. As the number of processes increases, the time and memory costs increase, and the increase varies between different representations. When using the access representation, the time cost is from 0.63s to 16s and the memory cost is from 249MB to 2,560MB, while under the structural representation, it is from 3s to 245s and from 551MB to 29,764MB. This difference is due to the similarity of the processes. Under the access representation, there are many processes whose similarity to each other is 0, which speeds up the algorithm and consumes less memory. What's more, the number of processes to detect is generally not so large, e.g., in Table II and Table III it is less than 2,500. The corresponding time cost is no more than 10s and the memory cost is no more than 1,534MB.

VI. DISCUSSION

A. FNs Brought by SheepHunter

SheepHunter is aggressive and can lead to additional false negatives. To mitigate this issue, we implement a lightweight post-processing step to analyze the local provenance paths of processes removed by SheepHunter. As discussed in §V-B, this approach can recover a portion of the introduced FNs with limited additional FPs. However, this is only one possible approach to FN mitigation. More sophisticated attacks

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:05:57 UTC from IEEE Xplore. Restrictions apply.

may require more comprehensive rules for provenance-path identification and tactic mapping. Other possible approaches include semantic analysis of communities based on their representative and special processes. For example, we can apply keyword matching or LLM to identify suspicious patterns such as persistence-related activities, remote download, and trace cleaning. SheepHunter reduces the number of processes needing to be analyzed, so it is feasible to achieve such analysis. Another approach is to record the historical information of communities and assign a malicious score to a community based on its historical behavior. Moreover, designing more powerful representations of processes can also help to mitigate the false negatives brought by SheepHunter. We will explore these approaches in our future work.

B. Attacks Targeted at SheepHunter

Goyal et al. [34] propose a mimic attack in which attackers mimic benign processes to evade detection, and attackers can also try to fool SheepHunter into identifying malicious processes as FPs. In §V-D, we have shown that SheepHunter can defend against the previous mimic attacks while maintaining good performance under several adaptive attacks. The most intuitive and effective way for attackers to fool SheepHunter is to repeat their behavior several times to increase the process behavior density, but this kind of strategy is not so reasonable. On the one hand, it will make their attacks less stealthy and may be easier to be detected by other approaches. On the other hand, SheepHunter can still remain effective in some scenarios (as shown in Table VII). Therefore, this strategy essentially trades off evasion against SheepHunter for a higher exposure risk to other detectors and analysts.

C. Anomaly Detection Using SheepHunter's Representation

We design a multi-dimensional representation for processes. One question is whether we can achieve better detection performance if we apply the representation to anomaly detection. However, the results of anomaly detection will not improve because there are still unknown processes in the test data. To reduce generated false positives, a detector should be able to handle unknown processes.

D. Limitations of SheepHunter

(1) The primary limitation of SheepHunter is that it may introduce additional FNs by mistakenly identifying some malicious processes as FPs. This mainly occurs when multiple malicious processes exhibit highly similar behaviors and are grouped into the same community, making them difficult to distinguish based on behavioral density. As a result, some malicious processes may be filtered out during the refinement stage. Although this may reduce recall, the remaining malicious processes can still indicate the attack, and the grouped malicious processes remain useful for manual investigation. As discussed above, incorporating semantic analysis and historical community information may help mitigate this issue. (2) Another limitation is that SheepHunter relies on the assumption that benign processes generally exhibit higher behavioral similarity and therefore form denser communities than malicious processes. When this assumption does not hold—for example, if malicious processes also form large, behaviorally homogeneous communities—the density difference becomes less discriminative, reducing the effectiveness of FP detection. Nevertheless, such attacks are generally less stealthy and may be detected by complementary approaches. (3) In addition, the effectiveness of SheepHunter depends on the baseline detector. When the detector already produces very few FPs, the room for further FP reduction is naturally limited because only a small number of benign alerted processes remain to be eliminated. In such cases, SheepHunter is more suitable for assisting manual FP analysis than automatic model adaptation. (4) Finally, about the limitations of datasets, our evaluation is conducted on two independent publicly available real-world provenance datasets collected from different environments, both of which exhibit the same density difference between benign and malicious processes, which is consistent with our observation. Nevertheless, two public datasets cannot fully capture the diversity of real-world deployment environments or attack scenarios. Therefore, we do not claim that this observation universally holds under all settings. Future work will further validate SheepHunter on larger-scale production provenance data collected from more diverse real-world systems.

VII. Related Work

A. Rule-Based PIDSes

Rule-based PIDSes [18], [19], [20], [21], [22], [23] always leverage expert knowledge about attacks and make security policies that will be triggered by attacks to generate alarms. SLEUTH [18], MORSE [20], and HOLMES [21] assign tags to each node, then conduct tag propagation to detect and reconstruct attack behavior. Poirot [22] constructs graphs of known attacks and queries them in graphs built from system logs to detect attacks. CAPTAIN [23] proposes adaptive parameters so that tag propagation-based approaches can be adapted to different scenarios. The number of FPs raised by rule-based approaches is small, as the rules are designed according to known attacks and are not triggered by benign behavior. However, rule-based PIDSes can only detect known attacks. If an attack is not covered by the rules, it will escape detection.

B. Anomaly-Based PIDSes

To detect unknown attacks, anomaly-based PIDSes are proposed and extensively explored by existing work [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [24], [25]. Anomaly-based PIDSes learn from benign processes and consider unknown processes as anomalies. The key to these approaches is feature extraction. Some of them extract statistical information [2], extract features from paths [3], and score the rareness of events [4], [8]. For instance, DISTDET [8] builds a hierarchical system event tree to store benign events and assigns anomaly scores to events to complete detection. Most of them [5], [6], [9], [10], [11], [12] apply GNNs to represent provenance graphs. For example, KAIROS [9] uses a temporal graph network (TGN [38]) to generate state embeddings for

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:05:57 UTC from IEEE Xplore. Restrictions apply.

nodes and constructs time window queues to detect anomalies. PROGRAPHER [6] splits provenance graphs based on time, represents each window as a snapshot through graph2vec [39], and detects anomalies according to the obtained snapshot sequences. A drawback common to all anomaly-based PIDSes is the high number of FPs [15]. Although a more fine-grained and powerful detector can reduce FPs, it will also consider unknown but benign processes as anomalies. Thus, updating detectors with FP data is important. However, FP analysis and obtaining data for updating PIDSes is labor intensive. Our work proposes an automatic way to fill this gap.

C. Provenance-Based Investigation

Existing work [33], [40], [41], [42], [43], [44], [45], [46], [47] on provenance-based investigation focuses on two goals: recognizing attacks in the complex provenance graphs and reducing manual analysis effort. DEPIMPACT [44] finds the most related events to the point-of-interest event and recognizes attack nodes and edges. DEEPCOM [45] designs a set of hierarchical walk schemes, splits a provenance graph into multiple communities, and uses an information path to summarize the behavior for each community in a human-readable way. TREC [46] is the first work that attempts to label subgraphs with tactics and techniques. Investigation approaches will select the subgraphs that are most likely to be attacks and exclude other nodes and edges, e.g., RapSheet [42] ranks different tactical provenance graphs and selects only graphs with high scores as alarms.

Unlike all the above investigation methods that focus on identifying attacks or reducing analyst effort, SheepHunter targets a different goal: explicitly identifying false positives among alerted processes. Methods such as NoDoze [41] and WATSON [33] rank alerts or abstract behaviors to prioritize suspicious entities for manual inspection, but do not determine whether a given alert is benign. SheepHunter introduces a fine-grained multi-dimensional representation of process behavior that captures richer information, enabling better separation between benign and malicious processes. Building upon this representation, SheepHunter further performs an iterative community splitting to ensure that processes within each community are as similar as possible. These technical designs distinguish SheepHunter from prior work and enable accurate false positive identification without requiring prior knowledge.

VIII. Conclusion

In this paper, we propose SheepHunter, which automatically distinguishes between benign and malicious processes in PIDS alarms by measuring the process behavior density. We devise a multi-dimensional process representation and measure process behavior density in each dimension to identify FP processes for existing PIDSes. We classify processes that exhibit high density across multiple dimensions as false positives. We evaluate SheepHunter on two open-source datasets. The results show that SheepHunter can reduce FPs by an average of 43.8% for existing PIDSes.

References

[1] M. Stokkel et al. (2024). APT41 Has Arisen From the Dust. [Online]. Available: https://cloud.google.com/blog/topics/threat-intelligence/apt41-arisen-from-dust

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

[3] Q. Wang et al., "You are what you do: Hunting stealthy malware via data provenance analysis," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–17.

[4] J. Zengy et al., "SHADEWATCHER: Recommendation-guided cyber threat analysis using system audit records," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 489–506.

[5] S. Wang et al., "THREATRACE: Detecting and tracing host-based threats in node level through provenance graph learning," *IEEE Trans. Inf. Forensics Security*, vol. 17, pp. 3972–3987, Sep. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9899459

[6] F. Yang, J. Xu, C. Xiong, Z. Li, and K. Zhang, "PROGRAPHER: An anomaly detection system based on provenance graph embedding," in *Proc. 32nd USENIX Secur. Symp. (USENIX Security)*, 2023, pp. 4355–4372.

[7] S. Li et al., "NODLINK: An online system for fine-grained APT attack detection and investigation," 2023, *arXiv:2311.02331*.

[8] F. Dong et al., "DISTDET: A cost-effective distributed cyber threat detection system," in *Proc. 32nd USENIX Secur. Symp. (USENIX Security)*, 2023, pp. 6575–6592.

[9] Z. Cheng et al., "Kairos: Practical intrusion detection and investigation using whole-system provenance," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2024, pp. 3533–3551.

[10] A. Goyal, G. Wang, and A. Bates, "R-CAID: Embedding root cause analysis within provenance-based intrusion detection," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2024, pp. 3515–3532.

[11] M. U. Rehman, H. Ahmadi, and W. U. Hassan, "Flash: A comprehensive approach to intrusion detection via provenance graph representation learning," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2024, pp. 3552–3570.

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

[13] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, "Learning under concept drift: A review," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 12, pp. 2346–2363, Dec. 2019.

[14] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM Comput. Surveys*, vol. 46, no. 4, pp. 1–37, Apr. 2014.

[15] F. Dong et al., "Are we there yet? An industrial viewpoint on provenance-based endpoint detection and response tools," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2023, pp. 2396–2410.

[16] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *J. Stat. Mechanics: Theory Exp.*, vol. 2008, no. 10, Oct. 2008, Art. no. P10008.

[17] Transparent-Computing. (Feb. 24, 2021). *Transparent Computing Engagement 5 Data Release*. [Online]. Available: https://github.com/darpa-i2o/Transparent-Computing

[18] M. N. Hossain et al., "SLEUTH: Real-time attack scenario reconstruction from COTS audit data," in *Proc. 26th USENIX Secur. Symp. (USENIX Security)*, 2017, pp. 487–504.

[19] P. Gao et al., "SAQL: A stream-based query system for real-time abnormal system behavior detection," in *Proc. 27th USENIX Secur. Symp. (USENIX Security)*, 2018, pp. 639–656.

[20] M. N. Hossain, S. Sheikhi, and R. Sekar, "Combating dependence explosion in forensic analysis using alternative tag propagation semantics," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1139–1155.

[21] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "HOLMES: Real-time APT detection through correlation of suspicious information flows," in *Proc. IEEE Symp. Security Privacy (S&P)*, May 2019, pp. 1137–1152.

[22] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "POIROT: Aligning attack behavior with kernel audit records for cyber threat hunting," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2019, pp. 1795–1812.

[23] L. Wang et al., "Incorporating gradients to rules: Towards lightweight, adaptive provenance-based intrusion detection," 2024, *arXiv:2404.14720*.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:05:57 UTC from IEEE Xplore. Restrictions apply.

[24] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 1285–1298.
[25] B. Jiang et al., "Orthrus: Achieving high quality of attribution in provenance-based intrusion detection systems," in *Proc. Secur. Symp. (USENIX Security)*, 2025, pp. 7173–7192.
[26] A. Ahmad, S. Lee, and M. Peinado, "HARDLOG: Practical tamper-proof system auditing using a novel audit device," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 1791–1807.
[27] R. Paccagnella et al., "Custos: Practical tamper-evident auditing of operating systems using trusted execution," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–18.
[28] M. A. Inam et al., "SoK: History is a vast early warning system: Auditing the provenance of system intrusions," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2023, pp. 2620–2638.
[29] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lió, and Y. Bengio, "Graph attention networks," 2017, *arXiv:1710.10903*.
[30] D. Zhang, J. Li, Z. Zeng, and F. Wang, "Jasper and stella: Distillation of SOTA embedding models," 2025, *arXiv:2412.19048*.
[31] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal*, vol. 1695, no. 5, pp. 1–9, 2006. [Online]. Available: https://r.igraph.org/authors.html#citation
[32] MITRE. (Aug. 17, 2026). *MITRE ATT&CK*. [Online]. Available: https://attack.mitre.org/
[33] 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 *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2021, pp. 1–18.
[34] A. Goyal, X. Han, G. Wang, and A. Bates, "Sometimes, you aren't what you do: Mimicry attacks against provenance graph host intrusion detection systems," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2023, pp. 1–18.
[35] H.-S. Park and C.-H. Jun, "A simple and fast algorithm for K-medoids clustering," *Expert Syst. Appl.*, vol. 36, no. 2, pp. 3336–3341,Mar. 2009.
[36] M. Ester et al., "A density-based algorithm for discovering clusters in large spatial databases with noise," *Kdd*, vol. 96, no. 34, pp. 226–231, 1996.
[37] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Nov. 2011.
[38] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein, "Temporal graph networks for deep learning on dynamic graphs," 2020, *arXiv:2006.10637*.
[39] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "Graph2vec: Learning distributed representations of graphs," 2017, *arXiv:1707.05005*.
[40] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates, "OmegaLog: High-fidelity attack investigation via transparent multi-layer log analysis," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–16.
[41] W. U. Hassan et al., "NoDoze: Combatting threat alert fatigue with automated provenance triage," in *Proc. Netw. Distrib. Syst. Secur. Symp*. Washington, DC, USA: Combat, 2019, pp. 1–15.
[42] W. U. Hassan, A. Bates, and D. Marino, "Tactical provenance analysis for endpoint detection and response systems," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1172–1189.
[43] A. Alsaheel et al., "ATLAS: A sequence-based learning approach for attack investigation," in *Proc. 30th USENIX Secur. Symp. (USENIX Security)*, 2021, pp. 3005–3022.
[44] P. Fang et al., "Back-propagating system dependency impact for attack investigation," in *Proc. 31st USENIX Secur. Symp. (USENIX Security)*, 2022, pp. 2461–2478.
[45] Z. Xu, P. Fang, C. Liu, X. Xiao, Y. Wen, and D. Meng, "DEPCOMM: Graph summarization on system audit logs for attack investigation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 540–557.
[46] M. Lv et al., "TREC: APT tactic/technique recognition via few-shot provenance subgraph learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Dec. 2024, pp. 139–152.
[47] S. Mohammadi et al., "Connecting the extra dots (contexts): Correlating external information about point of interest for attack investigation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2025, pp. 130–148.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:05:57 UTC from IEEE Xplore. Restrictions apply.