ProGrapher: An Anomaly Detection System based on Provenance Graph Embedding

Fan Yang[1], Jiacen Xu[2], Chunlin Xiong[3], Zhou Li[2], and Kehuan Zhang[1]

[1]*The Chinese University of Hong Kong*
[2]*University of California, Irvine*
[3]*Sangfor Technologies Inc.*

Abstract

In recent years, the Advanced Persistent Threat (APT), which involves complex and malicious actions over a long period, has become one of the biggest threats against the security of the modern computing environment. As a countermeasure, data provenance is leveraged to capture the complex relations between entities in a computing system/network, and uses such information to detect sophisticated APT attacks. Though showing promise in countering APT attacks, the existing systems still cannot achieve a good balance between efficiency, accuracy, and granularity.

In this work, we design a new anomaly detection system on provenance graphs, termed ProGrapher. To address the problem of "dependency explosion" of provenance graphs and achieve high efficiency, ProGrapher extracts temporal-ordered snapshots from the ingested logs and performs detection on the snapshots. To capture the rich structural properties of a graph, whole graph embedding and sequence-based learning are applied. Finally, key indicators are extracted from the abnormal snapshots and reported to the analysts, so their workload will be greatly reduced.

We evaluate ProGrapher on five real-world datasets. The results show that ProGrapher can detect standard attacks and APT attacks with high accuracy and outperform the state-of-the-art detection systems.

1 Introduction

The long-standing war between defenses and attacks in computing systems keeps evolving. Though defenses like Intrusion Detection Systems (IDS) and anti-malware software have been broadly deployed, sophisticated attacks under the theme of Advanced Persistent Threat (APT) [35] are still able to penetrate organizational networks, causing severe damages [2]. The major reasons for the failures against APT attacks are that 1) the traditional defense systems rely on attack signatures that can be easily changed by attackers, or 2) they perform detection (e.g., on system logs) without sufficiently leveraging the causal relations between different entities in a computing system or network.

To address these two fundamental issues, recently, a number of defense systems were developed based on data provenance [23]. Data provenance converts the system logs into a graph representation, which captures the temporal and causal relations between different types of entities (e.g., processes and files). On this representation, graph operations like graph traversal can be performed to detect ongoing attacks or reason about the root causes of intrusions. With data provenance, detecting APT attacks becomes possible as the rich contextual information embedded in the logs is well utilized.

Yet, based on our review, none of the existing provenance-based systems are able to fulfill all the essential deployment requirements in a complex production environment, including *detection accuracy*, *runtime efficiency*, *"signature-free"*, and *fine-granularity*. 1) Provenance systems relying on signatures, heuristics, or known attack traces can be evaded when the attackers adjust their patterns [1, 19]. 2) Some systems choose to construct a single provenance graph from the logs and detect malicious entities and events [25, 64], but the overhead would be prohibitive when a large volume of logs is to be analyzed, and a large number of false alarms would also be generated in such a setting. 3) A few systems construct temporal-ordered snapshots from the logs in a streaming fashion and try to detect abnormal snapshots [17, 37], but the detection granularity is too coarse as the analysts have to analyze all entities/interactions within the abnormal snapshots.

In this paper, we present ProGrapher, a new provenance-based anomaly detection system that meets the requirements mentioned above simultaneously. It follows the direction of graph-level, learning-based attack detection on provenance graph [17]. When logs are ingested, ProGrapher extracts snapshots to reduce the computation and memory costs on the whole provenance graph. On each snapshot, ProGrapher applies a whole graph embedding technique named *graph2vec* [44] to generate *rooted subgraph (RSG)* as a low-dimensional representation for each node, and learns the graph representation by maximizing the co-occurrence likelihood between normal snapshots and normal RSGs. To capture the temporal dynamics between snapshots, a sequence-learning model named TextRCNN [30] is adopted, so the representation of a future snapshot can be predicted and the abnormal snapshot can be detected when it deviates from the prediction. Previous systems like Unicorn [17] stops at the stage of reporting the abnormal snapshots, but PROGRAPHER moves on to pinpoint the abnormal entities by ranking the RSGs and reporting the most suspicious ones as attack indicators.

We implement and evaluate PROGRAPHER in *5* log datasets, including StreamSpot [36], ATLAS [48], DARPA3 [9], an unpublished DARPA ENGAGEMENT dataset, and logs collected by a commercial Endpoint Detection & Response (EDR) product, which cover a wide range of standard attacks (e.g., sending phishing emails), APT attacks (e.g., Nginx Backdoor), and computing environments (small lab networks and large enterprise networks). The evaluation results show that PROGRAPHER is able to effectively identify the snapshots containing attacks with high precision and recall on *all* datasets (e.g., 1.0 precision, recall and accuracy on DARPA ENGAGEMENT). Compared to the most relevant baseline system [17], PROGRAPHER gains a large margin, especially on the production EDR dataset (e.g., 0.943 vs. 0.542 in AUC). The indicator generation process reduces the workload of analysts by more than 50%. Finally, PROGRAPHER is highly efficient in only taking seconds to perform detection and indicator generation.

The contributions are summarized below:

- We present a novel anomaly detection system PROGRAPHER on the provenance graph. It combines whole graph embedding (through graph2vec) and sequence learning (through TextRCNN) to analyze snapshots of a provenance graph, which effectively and efficiently learns the representations of normal system behaviors.
- We introduce a new technique to identify attack indicators from a detected abnormal snapshot based on rooted subgraph (RSG), which significantly reduces the workload of analysts.
- We implement PROGRAPHER and evaluate it on 5 datasets that include traces of standard or APT attacks in different environments. The results show PROGRAPHER achieves high detection precision and recall on all datasets, and outperforms previous work by a large margin.

2 Background

We first introduce the background of data provenance in the context of attack investigation. Then, we focus on the learning-based approaches that are extensively used by the provenance systems. Finally, we briefly overview graph embedding, the main technique used by PROGRAPHER in modeling provenance graphs.

2.1 Data Provenance for Attack Investigation

To enable attack detection and forensics, system logs are often collected by the system-level auditing tools, such as Windows ETW [10], Linux Audit [52] and FreeBSD Dtrace [12], which describe the interactions between system entities like processes and files. The logs collected on the end hosts within an organization are often analyzed by a central service like Security Information and Event Management (SIEM) [3] to detect sophisticated cross-machine attacks.

On top of system logs, data provenance was proposed to detect and reason about intrusions, and even long-term Advanced Persistent Threat (APT) attacks that consist of multiple stages (e.g., reconnaissance, installation, command & control, and lateral movement) [35] can be detected. In essence, data provenance constructs a *dependency graph* from the system logs to describe the relationship between events, so detection and investigation can be transformed into graph-related operations [23].

Among all the graph-related operations, *graph traversal* is likely the most popular choice. One prominent example is back-tracking, through which the security analyst queries the provenance graph with a point of interest (POI) entity and a time window, and the events with time dependency are returned [28]. However, this simple approach suffers from "dependency explosion" [6], which can be caused by long-running processes that interact with many subjects/objects during their lifetime.

2.2 Learning-based Attack Detection on Provenance Graph

To accurately pinpoint the attack events, a wealth of rule-based approaches [23] were proposed, which leverage the knowledge of known attack behaviors to search the provenance graph. However, writing the rules requires considerable effort from the analysts, and attacks under unseen patterns could be missed. As a result, recent learning-based approaches, which train models with normal system behaviors (and malicious behaviors for supervised learning) to detect abnormal system executions, started to gain more attention. Though applying learning-based approaches to capture cyber-attacks is not new, provenance graph introduces new opportunities to exploit graph structures and apply new graph-learning methods. The existing works can be categorized by the granularity of target: edge/node, path, and graph.

When the target is edge/node, the trained system aims to tell whether the interaction between a pair of entities or the entity itself is malicious. One example is ShadeWatcher, which builds a knowledge graph from the system logs and uses

Graph Neural Networks (GNN)-based recommendation system to detect the malicious interactions [64]. SIGL leverages node embedding and an auto-encoder model to tell whether a process spawned from a software installation graph (SIG) is malicious [18]. However, achieving high accuracy for edge/node-based detection is quite challenging when encountering a large provenance graph[1]. Besides, the detection result does not provide contextual information (e.g., other activities related to the detected edge/node) that is valuable to understand the attack campaign.

For path-based detection, paths that fit certain patterns (e.g., being associated with POI nodes) are selected from the provenance graph and the trained system classifies the paths. For example, ProvDetector identifies stealthy malware by applying word embedding to transform execution paths into vectors and then clustering them [56]. Atlas applies lemmatization and word embedding to generate sequences and uses Long Short-term Memory (LSTM) network to predict whether a sequence is related to attack [1]. However, these approaches rely on heuristics to select POI paths (ProvDetector selects the rare paths tailored to stealthy malware) or nodes (ATLAS assumes some malicious nodes have been known) first and then apply learning-based approaches.

For graph-based detection, the provenance graph is either classified as a whole, or is decomposed into a set of subgraphs, on which the classification is performed. For example, Unicorn [17] slices the logs by a sliding time window and constructs evolving subgraphs from them. For each subgraph, graph sketching [61] is performed to convert a histogram that captures the structural features into a fixed-size vector. ProvGem [25] proposes multi-embedding to capture the varied contexts of nodes and classifies a graph on the aggregated node embeddings in a supervised-learning way. The main problem of graph-based detection is that its detection granularity is too coarse, and the analyst still needs considerable effort to pinpoint the malicious entities/events from the graph, which could include *thousands* of nodes.

PROGRAPHER follows the direction of graph-based detection, but makes prominent improvements in detection accuracy and granularity. In Section 8, we provide a detailed literature review of the data provenance systems.

2.3 Graph Embedding

To capture the key properties of the provenance graph, graph embedding is often used by provenance systems. In other domains like social networks [32], recommendation systems [53], and life sciences [13], graph embeddings have seen prominent successes in improving the performance of downstream tasks like graph classification, clustering and regression. In essence, graph embedding learns to represent nodes, edges, subgraphs, or the whole graph by low-dimensional vectors, which capture the graph structures, vertex-to-vertex relationships, and other relevant information about graphs. Two types of graph embedding techniques have been leveraged by provenance systems, and we describe them below[2]:

- **Node embedding** maps each node of a graph to a low-dimensional vector that preserves its key information, like the node's neighborhood information, the node's structural role, and the node's status. Popular node embedding models include DeepWalk [47], GCN [29], GraphSage [15], etc. Downstream tasks like node classification and edge classification, which are relevant to node-, edge-, and path-based detection, can be performed by computing node/edge scores from the node embedding and comparing them to thresholds.

- **Whole graph embedding** represents the whole graph with a single vector, which aggregates the information from node representations. Popular whole graph embedding models include DiffPool [63], graph2vec [44], graph sketching [61], etc. Downstream tasks like graph classification and clustering, which are relevant to graph-based detection, can be performed by computation on the graph vector.

PROGRAPHER applies whole graph embedding to detect subgraphs that contain attack traces. The key technique leveraged by PROGRAPHER is graph2vec, which extends the neural document embedding models of the NLP domain to the graph domain. Previous graph-based detection systems like Unicorn [17] examined relatively simple embedding models like graph sketching, which only captures the frequency of sub-structures in a graph. We found that with graph2vec, complex non-linear substructures are considered, which leads to more accurate measures of structurally similar graphs. In Section 4.2, we elaborate on how graph2vec is adapted to build provenance graph embedding.

3 Overview and Design of PROGRAPHER

In this section, we first formally define the problem and the threat model. Then, we overview the design of PROGRAPHER and the challenges. The symbols used in the paper are defined in Table 1.

3.1 Problem Statement

Here we define the provenance graph to be analyzed as $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \lambda, \delta, \gamma)$, where $\mathcal{V}$ is the set of nodes in $\mathcal{G}$ and $\mathcal{E}$ is a set of edges. An edge $e = (u, v) \in \mathcal{E}$ exists between two entities

[1]ShadeWatcher achieves high detection accuracy, but it is evaluated against small graphs (most of the graphs only have hundreds of interactions, as shown in Table I of [64]). The SIGs inspected by SIGL are usually small.

[2]Edge embedding has also been proposed for applications like recommendations in social network [54], but we have not found it to be used by any provenance system.

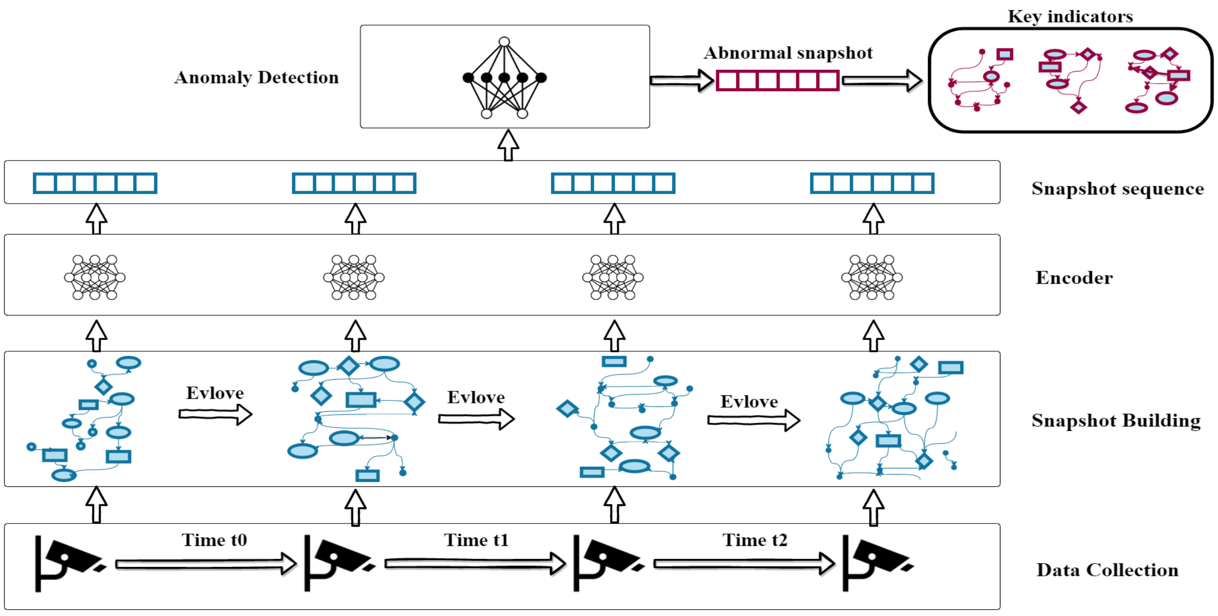

Figure 1: Overview of PROGRAPHER workflow.

Table 1: Main symbols used in the paper.

Term	Symbol
Graph	$\mathcal{G}$
Edges	$\mathcal{E}$
Nodes	$\mathcal{V}$
One edge	e
One node	u, v
Embedding	E
Snapshot	$\mathcal{S}$
Rooted subgraph	$\mathcal{R}$
Snapshot size	n
Snapshot sequence length	L
Forgetting rate	fr

u and v ($u, v \in \mathcal{V}$) when their interactions are logged. We also assume $\mathcal{G}$ is undirected. $\lambda : \mathcal{V} \to \mathcal{T}_{\mathcal{V}}$ is a function that assigns a label from the node type set $\mathcal{T}_{\mathcal{V}}$ to each node $v \in \mathcal{V}$. Similarly, $\delta : \mathcal{E} \to \mathcal{T}_{\mathcal{E}}$ assigns a label from the edge type set $\mathcal{T}_{\mathcal{E}}$ to each edge $e \in \mathcal{E}$. Examples of node types and edge types are shown in Table 3. $\gamma : \mathcal{E} \to \mathbb{Z}$ records the timestamp of each edge $e \in \mathcal{E}$. To notice, we consider a small set of fields in the log events (i.e., node type and edge type). This choice aligns with other related works like Unicorn [17], which only considers node type and edge type as well. We leave the utilization of other event fields as future work. The goal of our system PROGRAPHER is to detect the attack traces by analyzing the provenance graph, and provide the root-cause candidates to reduce analysts' workload.

Threat model. We follow the threat model from the previous works that conduct log-based anomaly detection [1,17,34,64]. We assume attackers do not manipulate the audit logs collected from the end-host monitors. As such, any attacks that deliberately compromise the security of the auditing systems are beyond the scope of this study. Existing works that ensure log integrity [26,45] can be leveraged to defend against such attacks.

3.2 Overview

We envision three design goals (G1 to G3) to be fulfilled by PROGRAPHER. Noticeably, none of the prior works were able to meet them all together and we compare PROGRAPHER to the representative ones in Table 2.

- **G1.** PROGRAPHER should learn the normal behavior patterns from the benign logs, so it increases the chances of detecting attacks that exploit zero-day vulnerabilities. In other words, PROGRAPHER should be built with *unsupervised-learning*, without the knowledge of any attack or event labels.
- **G2.** Since processing the provenance graph for a long period[3] is resource-consuming, PROGRAPHER should be able to process subgraphs of the whole provenance graph that are separated by periods, and leverage the temporal dynamics between periods for detection.
- **G3.** PROGRAPHER should be able to accurately identify the subgraphs with abnormal activities. In addition, PROGRAPHER should point out the entities that are directly

[3]For example, the whole-system provenance graph built upon DARPA THEIA dataset [9] consists of over one million nodes and one hundred million edges.

Table 2: The comparison with other related works in learning-based provenance analysis.

System	Target	Embedding	Setting	Knowledge of Attacks (G1)	Streaming[1](G2)	Report Granularity (G3)
ShadeWatcher [64]	Edge	GNN	Generic	No	No	Edges
SIGL [18]	Node	word2vec	Software installation	No	No	Nodes
ATLAS [1]	Path	word-representations [41]	Generic	Yes	No	Paths
ProvDetector [56]	Path[2]	doc2vec	Stealthy malware	Yes	No	Paths
Prov-Gem [25]	Graph	GCN	Generic	Yes	No	Graphs
Unicorn [17]	Graph	Graph sketching	Generic	No	Yes	Snapshots
PROGRAPHER	Graph	graph2vec	Generic	No	Yes	Nodes

[1] whether prediction can be performed incrementally on the ingested logs;[2] though ProvDetector has a version to classify graphs, it simply checks if the number of malicious paths is over a threshold.

related to attacks, which narrows down the investigation scope.

PROGRAPHER consists of four components to meet G1-G3: 1) snapshot builder, 2) encoder, 3) anomaly detector, and 4) key indicator generator. The workflow of PROGRAPHER is shown in Figure 1 and we present the details of each component in Section 4.

Specifically, the snapshot builder first extracts nodes and edges from the audit logs collected from end hosts and then splits the data into *snapshots* by the timestamps. The encoder generates a whole graph embedding on each snapshot to capture the graph's structural features. The anomaly detector trains a prediction model with the embeddings from the snapshots that are supposed to contain only benign activities, and detects the abnormal snapshots. Finally, the key indicator generator ranks the nodes contained by the abnormal snapshot and reports the top nodes to analysts.

Challenges of provenance graph embedding. The major challenge for building PROGRAPHER is how to choose and adapt the existing graph embedding methods for provenance graphs. Though whole graph embedding has been examined by provenance systems, none of them can meet the three design goals. For example, graph sketching examined by Unicorn cannot meet G3 (i.e., cannot pinpoint the malicious entities). Using more complex models powered by GCN like DiffPool [63] is likely to encounter scalability issues due to the high overhead in training GCN on large-scale graphs. In addition, these embedding models work with static graphs but how to adjust them to capture the temporal dynamics is unclear. In the end, we found that graph2vec strikes a good balance between efficiency and accuracy. Yet, directly applying graph2vec to our problem would lead to unsatisfactory results. Wang et al. actually tested graph2vec for graph-level detection, and found its recall is only 0.452, at 0.899 precision (Table IV of [56]). We speculate that the poor performance of graph2vec is due to the fact that the benign activities in the provenance graph usually far exceed the malicious activities, which may "hide" the malicious activities in the same graph. We elaborate on how we adjust graph2vec to make it practical for provenance graph embedding in Section 4.2 and Section 4.3.

4 Components of PROGRAPHER

4.1 Pre-processing and Snapshot Builder

We consider the log events about the files (e.g., file creation, file reading, file writing), processes (e.g., creation and privilege change), network sockets (e.g., network connection), principal (users or account), etc. The edges are made of events describing the actions performed by the source entities on the destination entities (e.g., a process reads a file). An example is shown in Figure 2. The full list of node types and edge types considered by PROGRAPHER is shown in Table 3.

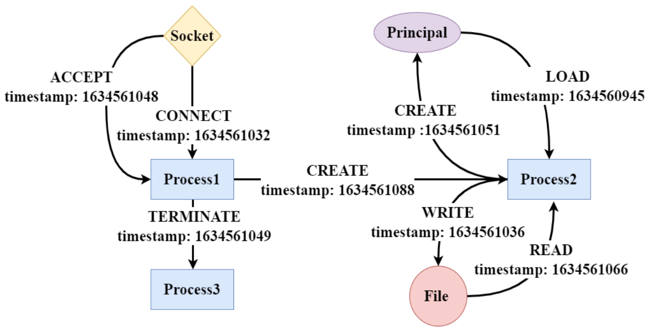

Figure 2: An example that includes four types of entities. Each edge has event type and event timestamp.

To handle the large volume of incoming logs efficiently, PROGRAPHER constructs snapshots ordered by time. It maintains a cache graph when the logs are ingested. For each incoming log, the event source and destination are added to the cache graph as nodes, when they are unseen. An edge is also created between the pair of new nodes, and the log timestamp is assigned to the nodes. For a pair of existing nodes, their timestamps are updated. When the number of nodes reaches n (also termed snapshot size), all n nodes and

Table 3: Graph elements and their types.

Element	Type
Node $\mathcal{T}_{\mathcal{V}}$	PROCESS, NETFLOW, PACKETSOCKET, FILE, PIPELINE, MEMORY, PRINCIPAL
Edge $\mathcal{T}_{\mathcal{E}}$	CONNECT, SEND, ACCEPT, LISTEN, OPEN, READ, WRITE, COPY, LOAD, UNLINK, MODIFY_ATTRIBUTES, CLONE, EXECUTE, TERMINATE, MEMORY_PROTECT, MEMORY_MAP

their edges are saved into the first snapshot. After that, PROGRAPHER adds new nodes from the incoming logs, uses a *forgetting rate* (fr) to retire the $n \times fr$ oldest nodes, and saves the cache graph into a new snapshot when the node number reaches $n \times (1 + fr)$. In other words, a pair of adjacent snapshots always have $1 - fr$ overlap in nodes.

We repeat this process to produce a sequence of snapshots $\{\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_k\}$. With such a design, we ensure that the temporal dynamics can be recovered by comparing the adjacent snapshots and the size of each snapshot is under control (i.e., $\leq n$). Moreover, the ratio between benign and malicious traces in a snapshot is expected to be much smaller than the whole provenance graph, addressing the data imbalance issue. The pseudo-code is shown in Algorithm 1 of the Appendix.

4.2 Encoder

After a snapshot is generated, the encoder component converts it to a low-dimensional embedding to capture its key information for anomaly detection to be performed later. As explained in Section 3.2, we choose *graph2vec* [44] as the encoding model. In essence, graph2vec considers the *rooted subgraphs (RSGs)* that are centered on every node as its *vocabulary*, and applies an NLP embedding technique named doc2vec [31] on the vocabulary to learn the graph representation. Below, we elaborate on the steps for embedding generation.

In the first step, every node will be enumerated to extract RSGs of various degrees, which capture the node's neighborhood information. In graph2vec, *Weisfeiler-Lehman (WL) graph kernel* [50,60], which is used to test graph isomorphism, is leveraged to achieve this goal. Specifically, for a node v, the WL kernel takes its label and the labels of its connected edges and nodes as input labels. Then, a new label, which is termed RSG, is generated for v that is aggregated from the input labels. The whole procedure is repeated d times on every node $v \in \mathcal{V}$ to describe its neighborhoods of depth $1, ..., d$.

To accommodate the format of provenance graph, we consider the node and edge types as labels (the original WL kernel only considers node types). Moreover, we found RSGs generated from a large and dense graph can have a lot of redundant labels. For efficiency, we only keep the unique labels for each RSG. In Algorithm 2 of Appendix, we describe the steps of RSG generation. As a concrete example, the RSGs of the node "Process2" at $d = 0, 1, 2$ in Figure 2 are:

- $d = 0$: [(Process)].
- $d = 1$: [(Process), (File), (Principal), LOAD, WRITE, CREATE, READ].
- $d = 2$: [(Process, File, Principal, LOAD, WRITE, CREATE, READ), (Process, Principal, LOAD, CREATE), (Process, File, READ, WRITE), (Process, Socket, CREATE, CONNECT, TERMINATE, ACCEPT), LOAD, WRITE, CREATE, READ].

Next, the embedding $E_{\mathcal{S}_i}$ of a snapshot $\mathcal{S}_i$ will be generated. $E_{\mathcal{S}_i}$ is initialized as a random vector, and then updated by maximizing the log-likelihood of the RSGs of all nodes, which are also represented by embeddings. The embeddings E of all snapshots $\{\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_k\}$ can be updated together through gradient descent. The updating process follows the skipgram model [39] used by doc2vec. The objective function we use is defined as:

$$J(E) = log\ Pr(E_{r_j}|E_{\mathcal{S}_i}) = log \frac{\exp(E_{r_j} \cdot E_{\mathcal{S}_i})}{\sum_{r_k \in \mathcal{R}_i} \exp(E_{r_k} \cdot E_{\mathcal{S}_i})} \tag{1}$$

where r_j is RSG j of $\mathcal{R}_i = \{r_1, r_2, \cdots, r_{n\times(\mathbb{D}+1)}\}$, $\mathbb{D}$ is the maximum degree of RSGs and E_{r_j} is the embedding of one RSG r_j.

For training efficiency, we apply *negative sampling* [40] like prior works based on unsupervised learning [64]. On a snapshot $\mathcal{S}_i$, we randomly select m RSGs $\mathcal{R}_i' = \{r_1, r_2, \cdots, r_m\}$ from the whole subgraph set as negative samples such that $\mathcal{R}_i' \cap \mathcal{R}_i = \emptyset$. The objective function $J(E)$ will be adjusted to maximize the log-likelihood of $\mathcal{R}_i$ and minimize the log-likelihood of $\mathcal{R}_i'$ at the same time. Since $\mathcal{R}_i'$ is a subset of non-exist RSGs, the training overhead is reduced. In Algorithm 3 of Appendix, we summarize the whole process of embedding generation.

4.3 Anomaly Detector

After generating the representations of the snapshots, PROGRAPHER moves on to detect the abnormal snapshots. We consider the changes between snapshots as an important input for the detection, and examine a sequence of snapshots $\{\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_k\}$ altogether. The bidirectional recurrent structure and convolution neural network model proposed by TextRCNN [30], which has been widely used for text classification, is chosen for this task.

To train the anomaly detector, we take a set of snapshot sequences and their associated embeddings as input. The recurrent structure and convolutional network are used to obtain a latent representation y_i of each snapshot $\mathcal{S}_i$ in the input sequence, which is defined below:

$$y_i = tanh(Wx_i + b) \tag{2}$$

where $x_i = [left(\mathcal{S}_i); E(\mathcal{S}_i); right(\mathcal{S}_i)]$ is the concatenation of left-side context vector, the embedding of itself, and right-side context vector. W is the weight matrix and b is the bias vector. The left and right context vectors are defined as:

$$left(\mathcal{S}_i) = Relu(W^l left(\mathcal{S}_{i-1}) + W^{sl} E(\mathcal{S}_{i-1})) \tag{3}$$

$$right(\mathcal{S}_i) = Relu(W^r right(\mathcal{S}_{i+1}) + W^{sr} E(\mathcal{S}_{i+1})) \tag{4}$$

where $Relu$ is the Relu activation function. W^r, W^l, W^{sl}, W^{sr} are weight matrices.

Then we utilize a maxpool layer and a fully-connected layer to obtain the final representation of each snapshot sequence.

$$F_{\mathcal{S}_{[1:k]}} = W(\max_{i=1}^{k}(y_i)) + b \tag{5}$$

where $F_{\mathcal{S}_{[1:k]}}$ is the final representation of the snapshot sequence, W is the weight matrix, max is the max pooling layer, and b is the bias vector.

In the training phase, given a snapshot sequence $\{\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_k\}$, we predict how likely the sequence would be related to its followed snapshot $\mathcal{S}_{k+1}$. So we define the loss as the distance in terms of the L2 distance as given below.

$$d(\mathcal{S}_{k+1}, \mathcal{S}'_{k+1}) = \left\| E_{\mathcal{S}_{k+1}} - F_{\mathcal{S}_{[1:k]}} \right\|_2 \tag{6}$$

where $E_{\mathcal{S}_{k+1}}$ is the embedding of snaoshot $\mathcal{S}_{k+1}$.

In the testing phase, given a snapshot sequence, we compare its predicted embedding with the ground-truth embedding. If the distance between them exceeds a pre-defined threshold, we will label it as abnormal.

4.4 Key Indicator Generator

After detecting the abnormal snapshot, the key indicator of the malicious activities will be generated. We found this step is missed by other works like Unicorn [17]. As such, their detection results are coarse-grained. But with the adoption of graph2vec, finer-grained attack attribution becomes possible by ProGrapher.

According to Equation 1, the objective function $J(E)$ measures the co-occurrence probability between a snapshot embedding and each RSG. The smaller the value, the smaller the co-occurrence probability. Since the likelihood is computed on *every* RSG of a snapshot, we can order the RSGs by their probabilities and select the key indicators from them. In particular, during the testing phase, given a snapshot $\mathcal{S}_i$, we compare the embedding $E_{\mathcal{S}_i}$ generated from $\mathcal{S}_i$ to the embedding $E'_{\mathcal{S}_i}$ predicted from the sequence of k snapshots, and extract the differences between the two embeddings for every RSG. After that, the RSGs are sorted by the loss differences, and the top K suspicious RSGs are selected. A RSG can be mapped to multiple nodes because it only stores node and edge types. Hence, we search the snapshot to locate all nodes matching the K suspicious RSGs and send them to the analysts.

4.5 A Running Example

Here we use a running example summarized from one attack of the ATLAS dataset ("Malvertising dominate" exploiting CVE-2015-3105 [7]) to illustrate how ProGrapher works in practice. In this attack, a user accesses a malicious IP address and establishes a set of network sessions. Next, a payload file is downloaded to the user's machine and executed to collect additional information to be exploited by the attacker.

As shown in Figure 3, $\mathcal{S}_1$ and $\mathcal{S}_2$ represent the snapshots before the attack, and $\mathcal{S}_3$ represents the initial phase of the attack. For ease of demonstration, we use a subset of the snapshots generated from the provenance graph and simplify each snapshot by removing a lot of unrelated activities and merging entities among snapshots. The red nodes denote the malicious entities labeled by the ATLAS dataset.

ProGrapher detects the anomalous snapshot, $\mathcal{S}_3$, based on snapshot sequence $\mathcal{S}_1 - \mathcal{S}_2$. After that, ProGrapher further selects top RSGs as the indicators, represented by their root nodes. The experiment results show that 3 of them are related to the attack campaign by that (1) the root node is a labeled malicious node ("6479_IP_address" and "6492_session"), or (2) a labeled malicious node can be found in the RSG ("6483_process"). Meanwhile, the remaining one is a false positive.

5 Implementation

We implement all components of ProGrapher in Python 3.7 with about 2000 lines of code. For machine-learning components, we implement the encoder model with Tensorflow 1.4 and the anomaly detection model with PyTorch 1.10.

To ensure that each snapshot contains sufficient information to learn its representation, we set the snapshot size n based on the scale of the dataset. For small graphs with less than 10K nodes (e.g., the provenance graph of software on a machine), n is set to 300. For large graphs with nodes more than 10k (e.g., the provenance graph of OS), n is set to 900. L is configured to 32, 128, and 176 for different datasets. During snapshot building, a forgetting rate fr is used to remove the oldest, and we set it to $\frac{1}{3}$. For the encoder and anomaly detector, we choose the hyper-parameters through grid search, and their optimal values are described below. In Section 6.5, we show the impact of different snapshot size n, snapshot sequence length L and forgetting rate fr. Table 4 lists the parameter values by datasets.

- **Encoder.** The dimension of graph embedding is 256; the depth of WL kernel (d) is 3 in small graphs and 4 in large graphs; the number of negative samples is 15.
- **Anomaly detector.** The dimension of the hidden layer is 128; the number of the hidden layer is 5; the initial learning rate is $3e-4$; early-stopping patience is 30 epochs; the dropout rate is 0.2.

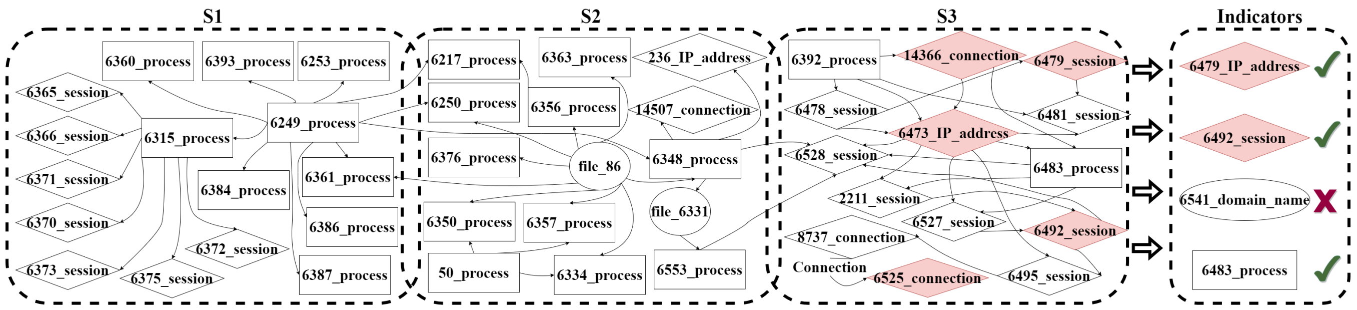

Figure 3: A running Example. The nodes in red are labeled as malicious in the Atlas dataset.

Table 4: Hyper-parameters of different dataset.

Dataset	L	n	fr	d
StreamSpot-DS	176	300	1/3	3
CADETS	128	900	1/3	4
CLEARSCOPE	32	300	1/3	3
THEIA	128	900	1/3	4
ATLAS-DS	32	300	1/3	3
DARPA ENGAGEMENT	128	900	1/3	4

Training and testing are conducted on a server with a 32-core Inter E5-2640 processor, 256 GB physical memory and one Nvidia GTX TITAN X GPU. The operating system is Ubuntu 14.04.6 LTS.

6 Evaluation

In this section, we perform a comprehensive evaluation of PROGRAPHER. We start by describing the datasets used for evaluation. Then, we evaluate the component for indicator generation, the runtime overhead of PROGRAPHER, the impact of parameters, and the robustness of PROGRAPHER against adaptive attacks.

We choose Unicorn [17] as the baseline system to compare with, because it also performs graph-level provenance analysis and its source code is publicly available [16].

6.1 Datasets

We use two log datasets with simulated attacks (StreamSpot and ATLAS) and two DARPA datasets (DARPA3 and DARPA ENGAGEMENT) to evaluate how PROGRAPHER performs in practice. Furthermore, we deploy PROGRAPHER in a production environment to analyze the system logs collected by a commercial EDR product. Below we describe the 5 datasets in detail.

StreamSpot dataset. The StreamSpot [37] dataset (or StreamSpot-DS for short) contains 600 benign and attack graphs derived from 6 scenarios: “YouTube”, “GMail”, “VGame”, “Drive-by-download Attack”, “Download”, and “CNN”. Each scenario contains 100 graphs. Five of these scenarios only have benign system activities, while the “Drive-by-download Attack’ scenario involves a drive-by-download attack that exploits an Adobe Flash vulnerability to gain root access.

ATLAS dataset. To evaluate more attack scenarios and how PROGRAPHER performs when benign and malicious activities are mixed, we use ATLAS dataset [48] (or ATLAS-DS for short), which is collected in a lab environment and contains 10 types of APT attacks, including different tactics like malicious email attachments and lateral movement. Various benign activities, including browsing websites, reading emails, downloading attachments, connecting to other hosts that happened before the attack, etc., are simulated together during the execution of each attack. On average, each scenario has 20,088 unique entities and 249k events. The attack-related entities are labeled as malicious by the data provider.

DARPA3 dataset. The DARPA3 dataset consists of 5 sub-datasets, including Trace, Fivedirection, CLEARSCOPE, THEIA, and CADETS, that are built under the DARPA Transparent Computing (TC) program [9]. Each sub-dataset contains 2-week system logs of specific system events (e.g., file read/write, network connection) on various platforms. Multiple attack campaigns are performed by red teams after a “silent” period (only benign activities are performed). The attack campaigns simulate the known APT attack vectors like Nginx backdoor, Darkon APT and the Firefox backdoor, and common attack vectors like sending phishing emails. We use the same three sub-datasets (CADETS, CLEARSCOPE, and THEIA) as Unicorn, which represent system activities on FreeBSD, Android, and Ubuntu Linux, for a fair comparison.

DARPA ENGAGEMENT dataset. We also obtained another unpublished dataset under the DARPA TC program, called DARPA ENGAGEMENT. The dataset contains 8-hour system logs collected from Linux systems. In total, there are about 3M system entities with 120M system events, and its per-hour system events and entities are much more than the previous DARPA3 sub-datasets, reflecting a more difficult scenario in which to perform accurate attack detection. Two APT attacks, including Firefox backdoor and Loader Drakon APT, were performed.

Production EDR dataset. Finally, we evaluate how ProGrapher performs in a real-world, production environment, by analyzing the logs collected by a commercial EDR product deployed on 18K endpoints (workstations and servers) from over 100 companies. On average, 2,030 events (1.2 MB) were collected per day per one endpoint, and overall there are 332,433,377 events (180 GB) analyzed for the duration of 9 days. The data are all stored in the EDR cloud server with strict access control policies to address privacy concerns. In the training phase, the security analysts examine the logs and give us the data with no reported attacks. In the testing phase, ProGrapher aims to capture the attack that was discovered by the security analysts.

6.2 Effectiveness

We first compare ProGrapher with Unicorn on StreamSpot-DS and three DARPA3 sub-datasets (CADETS, THEIA, CLEARSCOPE) on the effectiveness. Then, we evaluate ProGrapher on ATLAS-DS, DARPA ENGAGEMENT dataset and production EDR dataset. Before showing the results, we describe our evaluation metrics.

Evaluation Metrics. Each dataset has a number of graphs either determined by the data provider (e.g., StreamSpot-DS) or generated by us (e.g., DARPA3, to be elaborated later), and we separate each graph into a number of snapshots. Following the metrics of Unicorn, once we detect at least one abnormal snapshot, *the whole graph* will be considered malicious. Otherwise, the graph is considered benign. In this case, we count a true positive (TP) when an attack graph is detected correctly, and a false negative (FN) when it is not detected. True negative (TN) and false positive (FP) are defined on the benign graphs accordingly. Then, we compute the following metrics: accuracy, precision, recall, and F1 as:

$$\begin{gathered}\text{Accuracy} = \frac{TP+TN}{TP+FP+TN+FN}\\ \text{Precision} = \frac{TP}{TP+FP},\ \text{Recall} = \frac{TP}{TP+FN}\\ \text{F1} = 2\times\frac{\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}\end{gathered} \tag{7}$$

Table 5 summarizes the information about each dataset. Each dataset is separated by training, validation and testing, and we ensure that all of them are disjoint and all benign graphs in the testing set happen after the training and validation graphs.

Results on StreamSpot-DS. In this experiment, we randomly select 75 graphs from each benign scenario as the training set, 5 graphs from each benign scenario as the validation set and the remaining benign and attack graphs as the testing set. In total, we have 375 training graphs, 25 validation graphs and 200 testing graphs. For Unicorn, we use their released code and run the experiment in the same setting.

Table 5: Statistics of the datasets. "#Benign" and "#Attack" are the number of benign and attack graphs. Size is measured on the graphs after pre-processing, not the raw data.

Dataset	#Benign	#Attack	Size (GB)
StreamSpot-DS	500	100	8.3
DARPA3 (CADETS)	127	4	9.2
DARPA3 (CLEARSCOPE)	116	4	2
DARPA3 (THEIA)	66	3	27
ATLAS-DS	10	10	1.1
DARPA ENGAGEMENT	24	2	38
Production EDR	58,692	486	43

We ran the experiment 100 times for different data splits with both ProGrapher and Unicorn. As shown in Table 6 (row 1), our model has better precision, accuracy and F1. Both ProGrapher and Unicorn can capture all the attack graphs, but ProGrapher detects less FP [4]. It reveals that graph embedding and temporal modeling by ProGrapher are important to detect abnormal snapshots accurately.

Results on DARPA3. For each sub-dataset, we identify the attack-related activities and their timestamps with the ground-truth documents [8]. The benign graphs are generated by splitting the logs into 2-hour disjoint windows. Each attack graph captures a complete attack attempt recorded in the ground-truth documents (e.g., "Nginx Backdoor w/ Drakon In-Memory" in CADETS). We use the 80% benign graphs to train ProGrapher, 10% benign graphs for validation, and the remaining 10% benign graphs combined with all attack graphs as the testing set.

Table 6 (row 2-4) shows the results. Specifically, ProGrapher detects all attacks in three datasets (1.0 recall) and only falsely detects one benign graph in CLEARSCOPE as an anomaly. We investigate further and find that the FP is mainly caused by insufficient behavioral information in the training set. Since we only keep the latest event of the same pair of entities, for CLEARSCOPE, hundreds of events can be removed per entity pair. Admittedly, such a strategy removes useful information that can distinguish benign and malicious behaviors. This also explains why the size of CLEARSCOPE dataset after pre-processing is small (only 2GB). For comparison, Unicorn performs worse than ProGrapher in nearly every metric[5].

Results on ATLAS-DS and DARPA ENGAGEMENT. For

[4]For Unicorn, though we have not been able to get the same results as in their paper [17], the results are similar.

[5]We found the Unicorn result here is worse than its reported result on paper, due to two reasons. 1) We learned from the authors they used a non-public benign dataset under DARPA TC for training, to which we do not have access. 2) We found the Unicorn implementation does not enforce that the graphs in testing happen after training and validation, which is a common problem in security systems ("Data Snooping" mentioned by [4]).

Table 6: Experiment results of Sreamspot-DS and DARPA3. The numbers are averaged over the 100 runs.

Dataset	System	Precision	Recall	Accuracy	F1
StreamSpot-DS	Unicorn	0.85	1.0	0.91	0.92
	PROGRAPHER	**0.90**	**1.0**	**0.94**	**0.94**
CADETS	Unicorn	0.31	1.0	0.44	0.47
	PROGRAPHER	**1.0**	**1.0**	**1.0**	**1.0**
CLEARSCOPE	Unicorn	1.0	0.75	0.93	0.89
	PROGRAPHER	**0.8**	**1.0**	**0.93**	**0.89**
THEIA	Unicorn	0.67	0.67	0.8	0.67
	PROGRAPHER	**1.0**	**1.0**	**1.0**	**1.0**

ATLAS-DS, 10 attacks are simulated, resulting in 10 subsets. We split each subset into a benign graph and a malicious graph. The malicious graph has all the attack sequences labeled by the data provider, and the benign graph has the remaining sequences. Because the number of graphs is small, for each run, we choose one attack graph and one benign graph from a subset for testing, one benign graph from another subset for validation, and all remaining benign graphs for training. For DARPA ENGAGEMENT, the data splitting follows the same procedure as DARPA3, but we change the 2-hour window to half-hour, because the logs only span 12 hours and each attack is no longer than half an hour.

Table 7 shows that our system successfully detects all the attacks and does not generate any FP. The result also shows PROGRAPHER can handle different types of APT attacks.

Table 7: Results of ATLAS-DS and DARPA ENGAGEMENT.

Dataset	Precision	Recall	Accuracy	F1
ATLAS-DS	**1.0**	**1.0**	**1.0**	**1.0**
DARPA ENGAGEMENT	**1.0**	**1.0**	**1.0**	**1.0**

Results on Production EDR. We extract around 59K graphs from the 180GB logs by endpoints. From the first 7 days, we use 51,119 benign graphs for training (the benign graphs were selected by the analysts) and 2,889 benign graphs for validation. For the remaining 2 days, we use 4,684 benign graphs and 486 attack graphs for testing. Notably, this dataset has far more graphs than the other datasets.

In Figure 4, we draw the ROC curve to illustrate the relation between TPR and FPR, and compare it with Unicorn. The result suggests that PROGRAPHER can achieve reasonable accuracy in a production environment, e.g., 94% TPR at 14% FPR. The detection accuracy of Unicorn is significantly reduced and lower than PROGRAPHER, e.g., less than 10% TPR at 20% FPR. The area under the curve (AUC) of PROGRAPHER nearly doubles from Unicorn (0.943 vs. 0.542).

Still, we acknowledge that PROGRAPHER is less accurate in the production environment. The root causes could be 1) the training set may contain malicious activities not identified by the analysts, 2) the normal behaviors are more diverse during the 9-day period.

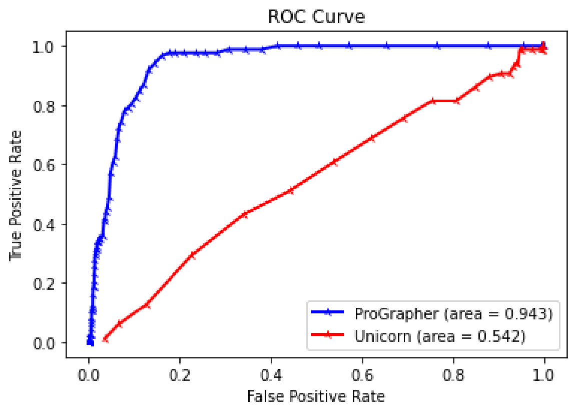

Figure 4: ROC curve for the production EDR dataset.

6.3 Evaluation on Indicator Generation

PROGRAPHER is designed to infer the attack indicators from an abnormal snapshot, which is the major difference from other systems like Unicorn. Here we evaluate this component under three metrics: effectiveness of indicators, coverage of the attack, and workload reduction.

Effectiveness of indicators. As described in Section 4.4, we select top K RSGs from an abnormal snapshot and return all nodes matching these RSGs. We judge whether the selected nodes can be an effective indicator with the following metric. Given a ground-truth attack node, we consider the nodes in the 3-hop neighborhood as valid, and all other nodes as invalid. We choose 3-hop neighborhood as we set the depth of WL kernel to 3 for small graphs (4 for large graphs, see Section 5), as it is a common practice to investigate the neighborhood entities given an alert [20]. If at least one node of the indicator belongs to the attack node or the valid neighbors, the indicator is considered effective. Figure 5 gives an example of these three types of nodes.

For a dataset, we compute the effectiveness rate as the ratio between the effective indicators and all indicators identified by PROGRAPHER. Table 8 shows the rate with various K (from 1 to 5) for the 3 DARPA3 subsets. As we can see, even with a small K between 2 and 3, the effectiveness rate is already quite high (at least 0.94).

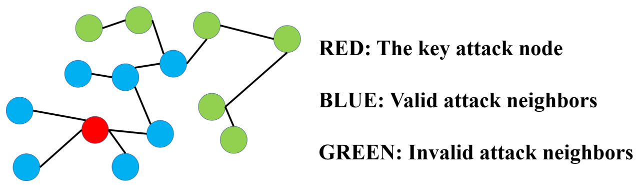

Figure 5: Attack nodes, valid attack neighbors and invalid attack neighbors.

Coverage of attack. We also measure how many ground-truth attack nodes are covered by the indicators. For one dataset,

Table 8: Effective rate of indicators on DARPA3.

Dataset	Effectiveness Rate				
	K=1	K=2	K=3	K=4	K=5
CADETS	0.88	0.94	0.94	1	1
THEIA	0.89	1	1	1	1
CLEARSCOPE	1	1	1	1	1

we define the coverage rate as the ratio between the correctly identified attack nodes and all ground-truth attack nodes. Table 9 shows the coverage rate of each dataset with K ranging from 1 to 5. All attack nodes are identified for THEIA and CLEARSCOPE. For CADETS, only 1 attack node is missing when $K \geq 4$.

Table 9: Coverage rate of attack nodes on DARPA3.

Dataset	Coverage Rate					
	Total	K=1	K=2	K=3	K=4	K=5
CADETS	28	0.61	0.67	0.85	0.96	0.96
THEIA	18	1	1	1	1	1
CLEARSCOPE	28	1	1	1	1	1

Workload reduction. In the previous experiments, we measure the effectiveness and coverage of indicators. Since an indicator only records node and edge types, it can be mapped to multiple nodes in a snapshot. We measure the number of nodes to be investigated and compare it to all nodes of a snapshot. We define the reduction rate as $1 - \frac{\text{Covered}}{\text{Total}}$ ("Covered" and "Total" are the number of nodes mapped to all indicators and all nodes in abnormal snapshots).

Table 10: Workload reduction. "Covered" and "Total" are related to ProGrapher. Unicorn's number is higher than "Total" because Unicorn predicts larger snapshots as abnormal.

Dataset	Covered	Total	Reduction	Unicorn
CADETS	6,794	16,200	58.1%	51,029
CLEARSCOPE	3,460	7,500	53.9%	21,853
THEIA	6,988	17,100	59.2%	51,147
Average	5,748	13,600	57.7%	41,343

Table 10 shows the work reduction of each DARPA3 dataset with K = 4. On average, the indicator generator reduces the workload of security analysts by 58%. By contrast, the baseline system Unicorn only tells if a snapshot is abnormal, and the analyst has to investigate all of the contained nodes. We sum the number of nodes from all alerted snapshots by Unicorn and also show it in Table 10. Unicorn's number of nodes to be investigated is 41,343, which is 7.1 times more than ProGrapher.

6.4 Runtime Performance

In this subsection, we measure the runtime overhead of each component and show the results in Table 11. The results are averaged across different runs on different DARPA3 datasets. Then, we discuss how ProGrapher scales with the input volume.

Data processing and training. On average, ProGrapher takes 8.4 minutes to process the one-day logs from one dataset and 8.3 microseconds to generate snapshot sequences. ProGrapher takes 6.29 hours to train the encoder model for 100 epochs. For the anomaly detector, it takes about 20.6 minutes to train. We note that the training is performed in an one-GPU server, so the overhead is expected to be reduced when distributed training can be performed.

Inference and indicator generation. ProGrapher takes on average 10.3 seconds to predict the abnormal snapshots and 8.3 seconds to generate the sorted RSGs for each abnormal snapshot. This result suggests ProGrapher is able to detect abnormal activities in near real-time.

Table 11: Overhead of each component.

Component	Mean Duration
Snapshot builder	8.4 mins
Training (encoder)	6.29 hours
Training (anomaly detector)	20.6 mins
Inference	10.3 sec
Indicator	8.3 sec

Scalability. We first measure how the memory consumption grows with the data volume, by changing the number of training graphs. For the same data size, the depth of the WL kernel (d) has the biggest impact, so we vary its value from 2 to 4. Figure 6(a) shows the relationship between data volume and memory consumption. Since ProGrapher performs training and inference on snapshots rather than on the whole dataset, the memory consumption is sub-linear to the data volume. For example, even when 40 GB data is processed with d set to 4, the maximum memory usage is 12.7 GB, with 10.2 GB to train and store embeddings.

We also measure how training time is impacted by the data volume, and show the overhead per epoch in Figure 6(b). It turns out the overhead scales linearly with the data size. The overhead only increases faster when $d = 4$ and the data size is more than 30GB.

6.5 Impact of Key Parameters

We now analyze the impact of key parameters on the effectiveness of ProGrapher using StreamSpot-DS, which is relatively small (8.3GB). Our baseline configurations follow the same setting as the StreamSpot-DS experiment in Section 6.2. Then we vary parameters independently to examine the impact of each parameter to justify the choices of parame-

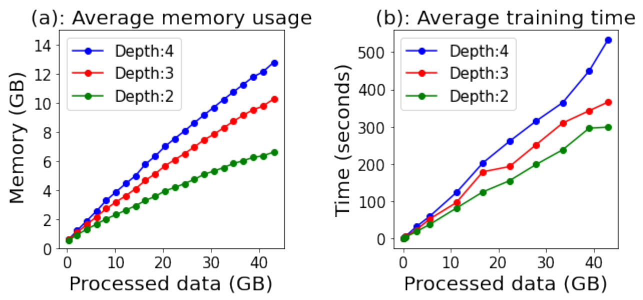

Figure 6: Scalability of PROGRAPHER.

ter values described in Section 5. Figure 7 and Figure 8 show the experimental results.

Snapshot size (n)**.** In general, if the snapshot size is too small, the information contained by the snapshot would be insufficient for accurate detection. But when the size is too large, more events will be merged per edge, which makes the abnormal behaviors less obvious. So the size should be carefully chosen. The result shows that the overall best result is achieved when n is set to 300 for StreamSpot-DS.

Forgetting rate (fr)**.** This determines the proportion of the snapshot to be forgotten each time. If fr is too small, adjacent snapshots will be more similar, thus obscuring the temporal patterns. However, if fr is too large, the useful contextual information will be lost, which leads to a higher FPR. The result shows when $fr = \frac{1}{3}$, best result can be achieved.

Snapshot sequence length (L)**.** Like snapshot size, this parameter has to be carefully chosen to provide sufficient but not excessive contextual information for accurate detection. We found best result can be achieved when $L = 176$ for StreamSpot-DS, as shown in Figure 8. Yet, the optimal L is smaller for other datasets (e.g., 32 for CLEARSCOPE), though StreamSpot-DS is a smaller dataset. We speculate a large L is needed for StreamSpot-DS because the simulated benign activities on the testing set are diverse but the training data are limited. Hence, more information coming from the snapshot sequences is needed.

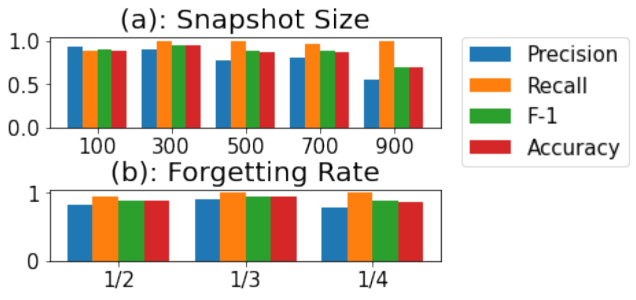

Figure 7: Impact of snapshot size n and forgetting rate fr measured on StreamSpot-DS.

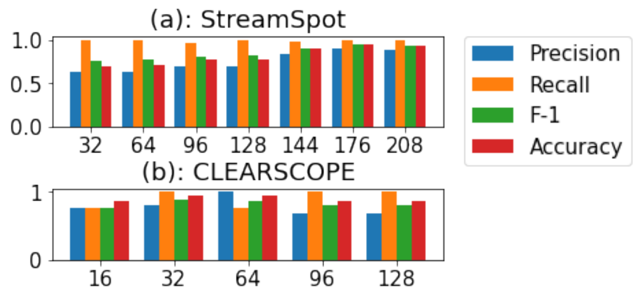

Figure 8: Impact of snapshot sequence length L on StreamSpot-DS and CLEARSCOPE.

6.6 Robustness

The attacker may want to evade PROGRAPHER by adjusting the attack behaviors. One approach is to insert many random events before and after the attack events in order to hide the real attack. We simulate this strategy by conducting a new experiment on ATLAS-DS.

Specifically, for each ground-truth attack node in the testing set, we randomly insert events between it and the other nodes within a 10-minute time window with a probability p. After that, we perform the same training and testing procedures. We examined $p = 10\%$, 20%, and 50%, and found the precision, recall and F1 did not change (all stay at 1.0, as shown in Table 7). We speculate this is because the space for benign behaviors is large, and it is difficult to choose the right ones (and their combinations) that are observed during training. On the other hand, the number of benign snapshot sequences is dropped (60, 58, 57 for $p = 10\%$, 20%, and 50%) because the more benign entities are interacting with the attack nodes.

We acknowledge that the attacker strategies simulated by us are not exhaustive. We discuss this limitation and potential solutions in Section 7.

7 Discussion and Limitation

Changes of normal behaviors. Since PROGRAPHER is designed as an anomaly detection system on the provenance graph, it will issue an alert when unseen behaviors are observed. However, the changes in normal behaviors (e.g., an employee logs onto a new internal server) are likely to be considered as abnormal, which could introduce false positives. This problem can be seen as a concept-drift problem, which can be partially mitigated by retraining PROGRAPHER with updated data. It is critical to detect when concept drift happens, so that the costly retrain does not need to be executed frequently. Recent works like [5, 24, 62] can be leveraged to this end.

Transductive and inductive Learning. The current design of PROGRAPHER follows the *transducitve learning* mode, which assumes all RSGs in the testing stage have been seen in the training stage. When new RSGs are encountered, PROGRAPHER has to be retrained. Though we reduce the chances of seeing new RSGs by constructing them with only node and edge types, retraining is still needed in the production environment. This limitation also persists in the existing graph-learning-based security systems like Euler [27] and ShadeWatcher [64]. To address this problem, we can explore the *inductive learning* mode, which is able to generate the embedding of a new node from its neighborhood on the fly, without retraining. However, in this case, a different encoder model, e.g., GraphSage [14], has to be chosen, and we leave this implementation as our future work.

More adaptive attacks. We have evaluated the robustness of PROGRAPHER against random events injection in Section 6.6. However, the attacker can choose a more advanced strategy by simulating benign behaviors that have been used in the training phase of PROGRAPHER. Such a strategy can be seen as *mimicry attack* [55] being implemented on the provenance graph. Previous works [17] mentioned this threat and argued that it is more difficult to carry out mimicry attacks on the provenance graph because generating valid benign behaviors that can capture the right contextual information is hard. Besides, PROGRAPHER generates the representation of each snapshot based on the extracted RSGs, and considers the possibility of co-occurrence both in spatial and temporal dimensions, which further raises the bar for mimicry attacks.

Another adversarial strategy is to inject duplicate events to fill the cache used to build a snapshot. In Section 6.2, we found that false negatives on the CLEARSCOPE dataset are caused by event aggregation. The issue exists in other provenance systems that aggregate events (e.g., [64]). One potential solution is to keep more information after event aggregation (e.g., distribution of certain event fields).

Finally, we assume the training phase is "attack free", but it is possible that attack events are embedded in the training set. In fact, our production EDR dataset is unlikely to be clean, but PROGRAPHER still achieves satisfactory performance and outperforms the baseline system.

8 Related Work

Learning-based provenance analysis. Section 2.2 has described a few exemplary works based on edge-, path-, and graph-level learning [1, 17, 18, 25, 56, 64]. Here we overview the other relevant works. HERCULE uses a semi-supervised community detection algorithm to correlate attack events and reconstruct attacks [46]. Streamspot extracts local graph features through bread-first search and clusters the snapshots to detect the abnormal ones [37]. P-Gaussian applies the Gaussian distribution principle to compute the similarity between intrusion behavior and its variants [58]. Log2vec constructs a heterogeneous graph from the logs and applies graph embedding to detect abnormal activities [33]. However, as described in Section 3.2, none of the prior works are able to achieve the three goals fulfilled by PROGRAPHER.

Heuristics-based provenance analysis. To solve the problem of "dependency explosion", another direction is to apply human-written rules to prioritize investigations, as described in Section 2.2. A number of prior works perform graph traversal (e.g., breadth-first search) from the POI events and select the suspicious paths by rules. For example, NoDoze uses historical information to assign threat scores to alerts within provenance graphs and then identify anomalous paths [20]. PrioTracker accelerates forward tracing by computing the rareness score of an event to prioritize abnormal events [34]. Padoga [57] considers the anomaly degree of both a single path and the whole provenance graph to identify intrusions. SLEUTH and Morse use tag-based information flow techniques to reconstruct attack scenarios [21, 22].

Alternatively, an analyst can query the provenance graph with the attack signatures, e.g., Indicators of Compromises (IoCs), and analyze similar subgraphs. τ-calculus proposes a new domain-specific language (DSL) to make the query more intuitive and efficient to threat analysts [51]. Poirot models the problem as a graph pattern matching (GPM) problem and proposes a new graph alignment method for it [42].

Finally, a few works were developed to abstract a summary graph from the fine-grained provenance graph to ease the investigation. RapSheet and Holmes rely on a knowledge base of adversarial Tactics, Techniques, and Procedures (TTPs) to construct the summary graph [19, 43]. DepComm [59] summarizes provenance graphs based on process-centric communities and extracts info-paths for attack investigation.

Anomaly detection on logs. PROGRAPHER relies on graph2vec, which adapts NLP techniques like doc2vec and word2vec, for graph embedding. Similar NLP techniques have also been applied to detect abnormal logs. Deeplog treats the audit logs as sentences and utilizes LSTM models to detect abnormal events [11]. LogAnomaly applies word2vec to extract the semantic information hidden in the log templates to detect log anomalies [38]. Attack2Vec uses temporal word embeddings to model and track the evolution of attack steps [49].

9 Conclusion

In this paper, we present PROGRAPHER, a learning-based system that leverages data provenance to detect abnormal activities from system logs. PROGRAPHER employs a novel combination of techniques in graph embedding, sequence learning, and indicator extraction, for accurate and unsupervised anomaly detection at the graph level. We evaluate PROGRAPHER on 4 simulated datasets and 1 dataset from a production environment. The result shows PROGRAPHER is able to achieve high accuracy in finding abnormal snapshots and significantly reduce analysts' workload in finding the root cause of the anomalies.

Acknowledgement. We want to thank our anonymous reviewers for their valuable comments. This work was supported in part by National Key Research & Development Project of China (Grant No. 2019YFB1804400), Hong Kong S.A.R. Research Grants Council (RGC) General Research Fund No. 14209720, and research fund (TS2116226) from Sangfor Technologies Inc. The authors from University of California, Irvine are supported by Microsoft Security AI RFP and Amazon Research Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the sponsors.

References

[1] Abdulellah Alsaheel, Yuhong Nan, Shiqing Ma, Le Yu, Gregory Walkup, Z. Berkay Celik, Xiangyu Zhang, and Dongyan Xu. ATLAS: A sequence-based learning approach for attack investigation. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security Symposium*, pages 3005–3022. USENIX Association, 2021.

[2] Adel Alshamrani, Sowmya Myneni, Ankur Chowdhary, and Dijiang Huang. A survey on advanced persistent threats: Techniques, solutions, challenges, and research opportunities. *IEEE Commun. Surv. Tutorials*, 21(2):1851–1877, 2019.

[3] Amrit T. Williams, Mark Nicolett. Improve it security with vulnerability management. https://www.gartner.com/en/documents/480703, 2005.

[4] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. Dos and don'ts of machine learning in computer security. In *Proc. of the USENIX Security Symposium*, 2022.

[5] Federico Barbero, Feargus Pendlebury, Fabio Pierazzi, and Lorenzo Cavallaro. Transcending transcend: Revisiting malware classification in the presence of concept drift. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 805–823. IEEE, 2022.

[6] Adam Bates and Wajih Ul Hassan. Can data provenance put an end to the data breach? *IEEE Security & Privacy*, 17(4):88–93, 2019.

[7] Brooks Li and Joseph C. Chen. Exploit kits in 2015: Flash bugs, compromised sites, malvertising dominate. https://www.trendmicro.com/en_us/research/16/c/exploit-kits-2015-flash-bugs-compromised-sites-malvertising-dominate.html, 2015.

[8] DARPA Transparent Computing. Ground truth file. https://drive.google.com/file/d/1mrs4LWkGk-3zA7t7v8zrhm0yEDHe57QU/view, 2018.

[9] DARPA I2O. Transparent computing engagement 3. https://github.com/darpa-i2o/Transparent-Computing/blob/master/README-E3.md, 2018.

[10] Don Marshall. Winodws event tracing. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw-, 2021.

[11] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM SIGSAC Conference on Computer and Communications Security, CCS*, pages 1285–1298. ACM, 2017.

[12] George V. Neville-Neil. Dtrace on freebsd. https://wiki.freebsd.org/DTrace, 2018.

[13] Zhen-Hao Guo, Zhu-Hong You, De-Shuang Huang, Hai-Cheng Yi, Kai Zheng, Zhan-Heng Chen, and Yan-Bin Wang. Meshheading2vec: a new method for representing mesh headings as vectors based on graph embedding algorithm. *Briefings in bioinformatics*, 22(2):2085–2095, 2021.

[14] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 1025–1035, 2017.

[15] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*, pages 1024–1034, 2017.

[16] Xueyuan Han. UNICORN. https://github.com/crimson-unicorn, 2018.

[17] Xueyuan Han, Thomas F. J.-M. Pasquier, Adam Bates, James Mickens, and Margo I. Seltzer. UNICORN: runtime provenance-based detector for advanced persistent threats. *CoRR*, abs/2001.01525, 2020.

[18] Xueyuan Han, Xiao Yu, Thomas F. J.-M. Pasquier, Ding Li, Junghwan Rhee, James W. Mickens, Margo I. Seltzer, and Haifeng Chen. SIGL: securing software installations through deep graph learning. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2345–2362. USENIX Association, 2021.

[19] Wajih Ul Hassan, Adam Bates, and Daniel Marino. Tactical provenance analysis for endpoint detection and response systems. In *IEEE Symposium on Security and Privacy*, pages 1172–1189. IEEE, 2020.

[20] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2019.

[21] Md Nahid Hossain, Sadegh M. Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R. Sekar, Scott D. Stoller, and V. N. Venkatakrishnan. SLEUTH: real-time attack scenario reconstruction from COTS audit data. *CoRR*, abs/1801.02062, 2018.

[22] Md Nahid Hossain, Sanaz Sheikhi, and R. Sekar. Combating dependence explosion in forensic analysis using alternative tag propagation semantics. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1139–1155. IEEE, 2020.

[23] Muhammad Adil Inam, Yinfang Chen, Akul Goyal, Jason Liu, Jaron Mink, Noor Michael, Sneha Gaur, Adam Bates, and Wajih Ul Hassan. Sok: History is a vast early warning system: Auditing the provenance of system intrusions. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 307–325. IEEE Computer Society, 2023.

[24] Roberto Jordaney, Kumar Sharad, Santanu K Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting concept drift in malware classification models. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 625–642, 2017.

[25] Maya Kapoor, Joshua Melton, Michael Ridenhour, Siddharth Krishnan, and Thomas Moyer. PROV-GEM: automated provenance analysis framework using graph embeddings. In M. Arif Wani, Ishwar K. Sethi, Weisong Shi, Guangzhi Qu, Daniela Stan Raicu, and Ruoming Jin, editors, *20th IEEE International Conference on Machine Learning and Applications, ICMLA 2021, Pasadena, CA, USA, December 13-16, 2021*, pages 1720–1727. IEEE, 2021.

[26] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. Sgx-log: Securing system logs with sgx. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 19–30, 2017.

[27] Isaiah J King and H Howie Huang. Euler: Detecting network lateral movement via scalable temporal link prediction. In *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS). The Internet Society, San Diego, California, USA*, 2022.

[28] Samuel T. King and Peter M. Chen. Backtracking intrusions. *ACM Trans. Comput. Syst.*, 23(1):51–76, 2005.

[29] Thomas N Kipf and Max Welling. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*, 2016.

[30] Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. Recurrent convolutional neural networks for text classification. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 2267–2273. AAAI Press, 2015.

[31] Quoc V. Le and Tomás Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 1188–1196. JMLR.org, 2014.

[32] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. Pytorch-biggraph: A large scale graph embedding system. *Proceedings of Machine Learning and Systems*, 1:120–131, 2019.

[33] Fucheng Liu, Yu Wen, Dongxue Zhang, Xihe Jiang, Xinyu Xing, and Dan Meng. Log2vec: A heterogeneous graph embedding based approach for detecting cyber threats within enterprise. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM SIGSAC Conference on Computer and Communications Security*, pages 1777–1794. ACM, 2019.

[34] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *Network and Distributed System Security Symposium*. The Internet Society, 2018.

[35] Lockheed Martin. Cyber kill chain. https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html, 2022.

[36] Emaad Manzoor. Streamspot_datasets. https://github.com/sbustreamspot/sbustreamspot-data, 2016.

[37] Emaad A. Manzoor, Sadegh M. Milajerdi, and Leman Akoglu. Fast memory-efficient anomaly detection in streaming heterogeneous graphs. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *Proceedings*

of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016, pages 1035–1044. ACM, 2016.

[38] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, and Rong Zhou. Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In Sarit Kraus, editor, *International Joint Conference on Artificial Intelligence, IJCAI*, pages 4739–4745. ijcai.org, 2019.

[39] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.

[40] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[41] Tomáš Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies*, pages 746–751, 2013.

[42] Sadegh M. Milajerdi, Birhanu Eshete, Rigel Gjomemo, and V. N. Venkatakrishnan. POIROT: aligning attack behavior with kernel audit records for cyber threat hunting. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1813–1830. ACM, 2019.

[43] Sadegh Momeni Milajerdi, Rigel Gjomemo, Birhanu Eshete, R. Sekar, and V. N. Venkatakrishnan. HOLMES: real-time APT detection through correlation of suspicious information flows. In *IEEE Symposium on Security and Privacy*, pages 1137–1152. IEEE, 2019.

[44] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning distributed representations of graphs. *CoRR*, abs/1707.05005, 2017.

[45] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher Fletcher, Andrew Miller, and Dave Tian. Custos: Practical tamper-evident auditing of operating systems using trusted execution. In *Network and distributed system security symposium*, 2020.

[46] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. HERCULE: attack story reconstruction via community discovery on correlated log graph. In Stephen Schwab, William K. Robertson, and Davide Balzarotti, editors, *Annual Conference on Computer Security Applications*, pages 583–595. ACM, 2016.

[47] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: online learning of social representations. In Sofus A. Macskassy, Claudia Perlich, Jure Leskovec, Wei Wang, and Rayid Ghani, editors, *International Conference on Knowledge Discovery and Data Mining*, pages 701–710. ACM, 2014.

[48] purseclab. ATLAS. `https://github.com/purseclab/ATLAS`, 2020.

[49] Yun Shen and Gianluca Stringhini. {ATTACK2VEC}: Leveraging temporal word embeddings to understand the evolution of cyberattacks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 905–921, 2019.

[50] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *J. Mach. Learn. Res.*, 12:2539–2561, 2011.

[51] Xiaokui Shu, Frederico Araujo, Douglas L Schales, Marc Ph Stoecklin, Jiyong Jang, Heqing Huang, and Josyula R Rao. Threat intelligence computing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 1883–1898, 2018.

[52] Steve Grubb. Linux audit. `https://linux.die.net/man/8/auditd`, 2021.

[53] Zhu Sun, Jie Yang, Jie Zhang, Alessandro Bozzon, Long-Kai Huang, and Chi Xu. Recurrent knowledge graph embedding for effective recommendation. In *Proceedings of the 12th ACM conference on recommender systems*, pages 297–305, 2018.

[54] Janu Verma, Srishti Gupta, Debdoot Mukherjee, and Tanmoy Chakraborty. Heterogeneous edge embedding for friend recommendation. In *European conference on information retrieval*, pages 172–179. Springer, 2019.

[55] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 255–264, 2002.

[56] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, Carl A. Gunter, and Haifeng Chen. You are what you do: Hunting stealthy malware via data provenance analysis. In *NDSS*. The Internet Society, 2020.

[57] Yulai Xie, Dan Feng, Yuchong Hu, Yan Li, Staunton Sample, and Darrell Long. Pagoda: A hybrid approach to enable efficient real-time provenance based intrusion detection in big data environments. *IEEE Transactions on Dependable and Secure Computing*, 17(6):1283–1296, 2018.

[58] Yulai Xie, Yafeng Wu, Dan Feng, and Darrell Long. P-gaussian: provenance-based gaussian distribution for detecting intrusion behavior variants using high efficient and real time memory databases. *IEEE Transactions on Dependable and Secure Computing*, 18(6):2658–2674, 2019.

[59] Z. Xu, P. Fang, C. Liu, X. Xiao, Y. Wen, and D. Meng. Depcomm: Graph summarization on system audit logs for attack investigation. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, 2022.

[60] Pinar Yanardag and S. V. N. Vishwanathan. Deep graph kernels. In Longbing Cao, Chengqi Zhang, Thorsten Joachims, Geoffrey I. Webb, Dragos D. Margineantu, and Graham Williams, editors, *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, pages 1365–1374. ACM, 2015.

[61] Dingqi Yang, Bin Li, Laura Rettig, and Philippe Cudré-Mauroux. Histosketch: Fast similarity-preserving sketching of streaming histograms with concept drift. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 545–554. IEEE, 2017.

[62] Limin Yang, Wenbo Guo, Qingying Hao, Arridhana Ciptadi, Ali Ahmadzadeh, Xinyu Xing, and Gang Wang. {CADE}: Detecting and explaining concept drift samples for security applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2327–2344, 2021.

[63] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *Advances in neural information processing systems*, 31, 2018.

[64] J. Zeng, X. Wang, J. Liu, Y. Chen, Z. Liang, T. Chua, and Z. Chua. Shadewatcher: Recommendation-guided cyber threat analysis using system audit records. In *2022 IEEE Symposium on Security and Privacy (SP) (SP)*, 2022.

A Pseudo-code of PROGRAPHER components

The pseudo-code of snapshot generation is shown in Algorithm 1. The algorithm takes event logs, forgetting rate fr of logs and snapshot size n as input. For each event in event logs, lines 6-8 extract the node and edge information and then add nodes and edges to the created graph. When the graph size meets the predefined size n, lines 9-13 output the current graph as the first snapshot. After that, whenever the graph size reaches limit size $n \times (1+fr)$, lines 15-16 sort all nodes by timestamp values and remove $n \times fr$ older nodes with smaller timestamps, then line 17 exports the graph as a snapshot. Finally, the result of the snapshot generation algorithm is the snapshot sequence $\{\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_k\}$.

Algorithm 1 Snapshot Generation

```
Input: Event logs, forgetting rate $fr$, snapshot size $n$
Output: snapshot sequence $\{\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_k\}$
1: $S = \{\}$                                   ▷ initialize the snapshot list
2: $k = 0$                                      ▷ initialize the snapshot index
3: $first\_flag$ = True                         ▷ check first snapshot
4: $G = Graph()$                                ▷ empty graph
5: for each $event \in$ Event logs do
6:     $G.add\_node\_from(event.src)$
7:     $G.add\_node\_from(event.dest)$
8:     $G.add\_edge\_from(event)$
9:     if $(first\_flag)\&(len(G.nodes) >= n)$ then
10:        $S[k] = G$
11:        $k = k + 1$
12:        $first\_flag = False$
13:    end if
14:    if $len(G.nodes) >= n \times (1 + fr)$ then
15:        $older = sort(G.nodes, G.timestamps)[: n \times fr]$ ▷ sort by timestamp
16:        $G.remove\_nodes\_from(older)$
17:        $S[k] = G$
18:        $k = k + 1$
19:    end if
20: end for
21: return $\{\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_k\}$
```

The pseudo-code for RSG generation is shown in Algorithm 2. This algorithm takes a graph $\mathcal{G}$, the neighborhood hop d and the centric node v as input. When $d = 0$, only the node type λ of v is used to construct the RSG. When $d > 0$, the RSG $\mathcal{R}_v^d$ is constructed by examining the neighborhood recursively. Line 6 obtains RSG $\mathcal{F}_v^d$ at depth $d-1$ for all the neighborhoods of node v. Line 7 retrieves the edge types I_v^d between v and its neighborhood nodes v'. Finally in Line 8, we sort $\mathcal{F}_v^d$ and I_v^d, remove the duplicated labels (through $set(\cdot)$), concatenate them with RSG at $d-1$ hop (through $\oplus$) to obtain the final RSG for v. The output of the RSG generation function is the rooted subgraph $\mathcal{R}_v^d$ for node v with hop number d.

Algorithm 2 RSG Generation for a node

```
Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \lambda, \delta, \gamma)$, hop number $d$, centric node $v$
Output: Rooted Subgraph for a node $\mathcal{R}_v^d$.
 1: $\mathcal{R}_v^d = \{\}$
 2: if $d == 0$ then
 3:     $r$ = generate_graph($\lambda_v$)
 4:     $\mathcal{R}_v^d.append(r)$
 5: else
 6:     $\mathcal{F}_v^d = \{\mathcal{R}_u^{d-1} \mid u \in v.neighbors\}$    ▷ collect all the neighbors' RSG of node $v$
 7:     $I_v^d = \{\delta_e \mid e \in (v, v.neighbors)\}$    ▷ collet the edges' types connecting to the node $v$
 8:     $\mathcal{R}_v^d = \mathcal{R}_v^d \cup RSG\_Genetration(\mathcal{G}, (d-1), v) \oplus set(sort(\mathcal{F}_v^d)) \oplus set(sort(I_v^d))$
 9: end if
10: return $\mathcal{R}_v^d$.
```

The pseudo-code for embedding generation is shown in Algorithm 3. We learn the embeddings of all the input snapshots in a limited number of epochs. Line 1 initializes the embedding matrix E and line 3 randomly shuffles the snapshot sequence. In the following steps, line 7 extracts RSGs for each node from the corresponding snapshot, line 8 selects negative samples based on extracted RSGs, and lines 9-10 learn the embedding of the corresponding snapshot. Finally, lines 2-14 repeat this process to obtain the final embedding matrix E after a given number of epochs.

Algorithm 3 The process to generate the embeddings for all snapshots

```
Input: Snapshot Sequence $\{\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_k\}$ with corresponding initial embedding matrix $E$, learning rate $\alpha$, epoch number $epochs$, node set $\mathcal{V}$, maximum degree of root subgraph $\mathbb{D}$.
Output: the embeddings matrix $E$
 1: initialize $E$
 2: for _ in $range(epochs)$ do
 3:     $\mathcal{S}$ = randomly_shuffle($\{\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_k\}$)
 4:     for each $\mathcal{S}_i \in \mathcal{S}$ do
 5:         for each $v \in \mathcal{V}$ do
 6:             for $d = 0 \to \mathbb{D}$ do
 7:                 $\mathcal{R}_v^d = RSG\_Generation(\mathcal{S}_i, d, v)$
 8:                 $\mathcal{R}'^d_v$ = negative_sampling($\mathcal{R}_v^d$)
 9:                 $loss = -\sum_{\mathcal{R}_v^d + \mathcal{R}'^d_v} J(E)$
10:                 $E = E - \alpha \frac{\partial loss}{\partial E}$
11:             end for
12:         end for
13:     end for
14: end for
15: return $E$
```