MAGIC: Detecting Advanced Persistent Threats via Masked Graph Representation Learning

Zian Jia[1], Yun Xiong[1], Yuhong Nan[2*], Yao Zhang[1], Jinjing Zhao[3], Mi Wen[4]

[1] *Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, China*
[2] *School of Software Engineering, Sun Yat-sen University, China*
[3] *National Key Laboratory of Science and Technology on Information System Security, China*
[4] *Shanghai University of Electric Power, China*

Abstract

Advance Persistent Threats (APTs), adopted by most delicate attackers, are becoming increasing common and pose great threat to various enterprises and institutions. Data provenance analysis on provenance graphs has emerged as a common approach in APT detection. However, previous works have exhibited several shortcomings: (1) requiring attack-containing data and *a priori* knowledge of APTs, (2) failing in extracting the rich contextual information buried within provenance graphs and (3) becoming impracticable due to their prohibitive computation overhead and memory consumption.

In this paper, we introduce MAGIC, a novel and flexible self-supervised APT detection approach capable of performing multi-granularity detection under different level of supervision. MAGIC leverages masked graph representation learning to model benign system entities and behaviors, performing efficient deep feature extraction and structure abstraction on provenance graphs. By ferreting out anomalous system behaviors via outlier detection methods, MAGIC is able to perform both system entity level and batched log level APT detection. MAGIC is specially designed to handle concept drift with a model adaption mechanism and successfully applies to universal conditions and detection scenarios. We evaluate MAGIC on three widely-used datasets, including both real-world and simulated attacks. Evaluation results indicate that MAGIC achieves promising detection results in all scenarios and shows enormous advantage over state-of-the-art APT detection approaches in performance overhead.

1 Introduction

Advanced Persistent Threats (APTs) are intentional and sophisticated cyber-attacks conducted by skilled attackers and pose great threat to both enterprises and institutions [1]. Most APTs involve zero-day vulnerabilities and are especially difficult to detect due to their stealthy and changeful nature.

Recent works [2–18] on APT detection leverage data provenance to perform APT detection. Data provenance transforms audit logs into provenance graphs, which extract the rich contextual information from audit logs and provide a perfect platform for fine-grained causality analysis and APT detection. Early works [2–6] construct rules based on typical or specific APT patterns and match audit logs against those rules to detect potential APTs. Several recent works [7–9] adopt a statistical anomaly detection approach to detect APTs focusing on different provenance graph elements, e.g, system entities, interactions and communities. Most recent works [10–18], however, are deep learning-based approaches. They utilize various deep learning (DL) techniques to model APT patterns or system behaviors and perform APT detection in a classification or anomaly detection style.

While these existing approaches have demonstrated their capability to detect APTs with reasonable accuracy, they encounter various combinations of the following challenges: (1) Supervised methods suffer from lack-of-data (LOD) problem as they require *a priori* knowledge about APTs (i.e. attack patterns or attack-containing logs). In addition, these methods are particularly vulnerable when confronted with new types of APTs they are not trained to deal with. (2) Statistics-based methods only require benign data to function, but they fail to extract the deep semantics and correlation of complex benign activities buried in audit logs, resulting in high false positive rate. (3) DL-based methods, especially sequence-based and graph-based approaches, have achieved promising effectiveness at the cost of heavy computation overhead, rendering them impractical in real-life detection scenarios.

In this paper, we address the above three issues by introducing MAGIC, a novel self-supervised APT detection approach that leverages *masked graph representation learning* and simple *outlier detection* methods to identify key attack system entities from massive audit logs. MAGIC first constructs the provenance graph from audit logs in simple yet universal steps. MAGIC then employs a graph representation module that obtains embeddings by incorporating graph features and structural information in a self-supervised way. The model

*Corresponding author: Yuhong Nan

is built upon *graph masked auto-encoders* [19] under the joint supervision of both *masked feature reconstruction* and *sample-based structure reconstruction*. An unsupervised outlier detection method is employed to analyze the computed embeddings and attain the final detection result.

MAGIC is designed to be flexible and scalable. Depending on the application background, MAGIC is able to perform multi-granularity detection, i.e., detecting APT existence in batched logs or locating entity-level adversaries. Although MAGIC is designed to perform APT detection without attack-containing data, it is well-suited for semi-supervised and fully-supervised conditions. Furthermore, MAGIC also contains an optional model adaption mechanism which provides a feedback channel for its users. Such feedback is important for MAGIC to further improve its performance, combat concept drift and reduce false positives.

We implement MAGIC and evaluate its performance and overhead on three different APT attack datasets: the DARPA Transparent Computing E3 datasets [20], the StreamSpot dataset [21] and the Unicorn Wget dataset [22]. The DARPA datasets contain real-world attacks while the StreamSpot and Unicorn Wget dataset are fully simulated in controlled environments. Evaluation results show that MAGIC is able to perform entity-level APT detection with 97.26% precision and 99.91% recall as well as minimum overhead, less memory demanding and significantly faster than state-of-the-art approaches (e.g. 51 times faster than ShadeWatcher [18]).

To benefit future research and encourage further improvement on MAGIC, we make our implementation of MAGIC and our pre-processed datasets open to public[1]. In summary, this paper makes the following contributions:

- We propose MAGIC, a universal APT detection approach based on masked graph representation learning and outlier detection methods, capable of performing multi-granularity detection on massive audit logs.
- We ensure MAGIC's practicability by minimizing its computation overhead with extended graph masked auto-encoders, allowing MAGIC to complete training and detection in acceptable time even under tight conditions.
- We secure MAGIC's universality with various efforts. We leverage masked graph representation learning and outlier detection methods, enabling MAGIC to perform precise detection under different supervision levels, in different detection granularity and with audit logs from various sources.
- We evaluate MAGIC on three widely-used datasets, involving both real-world and simulated APT attacks. Evaluation results show that MAGIC detects APTs with promising results and minimum computation overhead.
- We provide an open source implementation of MAGIC to benefit future research in the community and encourage further improvement on our approach.

[1]MAGIC is available at https://github.com/FDUDSDE/MAGIC

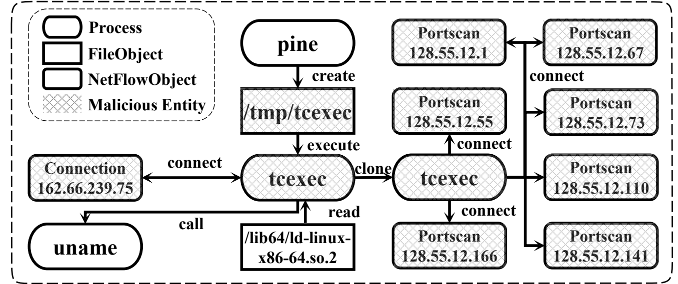


Figure 1: The provenance graph of a real-world APT attack, exploiting the Pine Backdoor vulnerability. All attack-irrelevant entities and interactions have been removed from the provenance graph.

2 Background

2.1 Motivating Example

Here we provide a detailed illustration of an APT scenario that we use throughout the paper. *Pine backdoor with Drakon Dropper* is an APT attack from the DARPA Engagement 3 Trace dataset [20]. During the attack, an attacker constructs a malicious executable (*/tmp/tcexec*) and sends it to the target host via a phishing e-mail. The user then unconsciously downloads and opens the e-mail. Contained within the e-mail is an executable designed to perform a port-scan for internal reconnaissance and establish a silent connection between the target host and the attacker. Figure 1 displays the provenance graph of our motivation example. Nodes in the graph represent system entities and arrows represent directed interactions between entities. The graph shown is a subgraph abstracted from the complete provenance graph by removing most attack-irrelevant entities and interactions. Different node shape corresponds to different type of entities. Entities covered in stripes are considered malicious ones.

2.2 Prior Research and their Limitations

Supervised Methods. For early works [2–6], special heuristic rules need to be constructed to cover all attack patterns. Many DL-based APT detection methods [11, 14–16, 18] construct provenance graphs based on both benign and attack-containing data and detect APTs in a classification style. These supervised methods can achieve almost perfect detection results on learned attack patterns but are especially vulnerable facing *concept drift* or unseen attack patterns. Moreover, for rule-based methods, the construction and maintenance of heuristic rules can be very expensive and time-consuming. And for DL-based methods, the scarcity of attack-containing data is preventing these supervised methods from being actually deployable. To address the above issue, MAGIC adopts a fully self-supervised anomaly detection style, allowing the absence of attack-containing data while effectively dealing

with unseen attack patterns.

Statistics-based Methods. Most recent statistics-based methods [7–9] detect APT signals by identifying system entities, interactions and communities based on their rarity or anomaly scores. However, the rarity of system entities may not necessarily indicate their abnormality and anomaly scores, obtained via causal analysis or label propagation, are *shallow feature extraction* on provenance graphs. To illustrate, the process *tcexec* performs multiple portscan operations on different IP addresses in our motivating example (See Figure 1), which may be considered as a normal system behavior. However, taking into consideration that process *tcexec*, derived from the external network, also reads sensitive system information (*uname*) and makes connection with public IP addresses (*162.66.239.75*), we can easily identify *tcexec* as a malicious entity. Failure to extract deep semantics and correlations between system entities often results in low detection performance and high false positive rate of statistics-based methods. MAGIC, however, employs a graph representation module to perform *deep graph feature extraction* on provenance graphs, resulting in high-quality embeddings.

DL-based Methods. Recently, DL-based APT detection methods, no matter supervised or unsupervised, are producing very promising detection results. However, in reality, hundreds of GB of audit logs are produced every day in a medium-size enterprise [23]. Consequently, DL-based methods, especially sequence-based [11, 14, 24] and graph-based [10, 12, 15–18] methods, are impracticable due to their computation overhead. For instance, ATLAS [11] takes an average 1 hour to train on 676MB of audit logs and ShadeWatcher [18] takes 1 day to train on the DARPA E3 Trace dataset with GPU available. Besides, some graph autoencoder [25–27] based methods encounter *explosive memory overhead* problem when the scale of provenance graphs expands. MAGIC avoids to be computationally demanding by introducing *graph masked auto-encoders* and completes its training on the DARPA E3 Trace dataset in mere minutes. Detailed evaluation of MAGIC's performance overhead is presented in Sec. 6.4.

End-to-end Approaches. Beyond the three major limitations discussed above, it is also worth to mention that most recent APT detection approaches [11, 17, 18] are *end-to-end* detectors and focus on one specific detection task. For instance, ATLAS [11] focused on end-to-end attack reconstruction and Unicorn [10] yields system-level alarms from streaming logs. Instead, MAGIC's approach is universal and performs multi-granularity APT detection under various detection scenarios, which can also be applied to audit logs collected from different sources.

2.3 Threat Model and Definitions

We first present the threat model we use throughout the paper and then formally define key concepts that are crucial to understanding how MAGIC performs APT detection.

Threat Model. We assume that attackers come from outside a system and target valuable information within the system. An attacker may perform sophisticated steps to achieve his goal but leaves trackable evidence in logs. The combination of the system hardware, operating system and system audit softwares is our trusted computing base. Poison attacks and evasion attacks are not considered in our threat model.

Provenance Graph. A provenance graph is a directed cyclic graph extracted from raw audit logs. Constructing a provenance graph is common practice in data provenance, as it connects system entities and presents the interaction relationships between them. A provenance graph contains nodes representing different system entities (e.g., processes, files and sockets) and edges representing interactions between system entities (e.g., execute and connect), labeled with their types. For example, */tmp/tcexec* is a *FileObject* system entity and the edge between */tmp/tcexec* and *tcexec* is an *execute* operation from a *FileObject* targeting a *Process* (See Figure 1).

Multi-granularity Detection. MAGIC is capable to perform APT detection at two-granularity: *batched log level* and *system entity level*. MAGIC's multi-granularity detection ability gives rises to a two-stage detection approach: first conduct batched log level detection on streaming batches of logs, and then perform system entity level detection on positive batches to identify detailed detection results. Applying this approach to real-world settings will effectively reduce workload, resource consumption and false positives and, in the meantime, produce detailed outcomes.

- Batched log level detection. Under this granularity of APT detection, the major task is *given batched audit logs from a consistent source, MAGIC alerts if a potential APT is detected in a batch of logs*. Similar to Unicorn [10], MAGIC does not accurately locate malicious system entities and interactions under this granularity of detection.

- System entity level detection. The detection task under this granularity of APT detection is *given audit logs from a consistent source, MAGIC is able to accurately locate malicious system entities in those audit logs*. Identification of key system entities during APTs is vital to subsequent tasks such as attack investigation and attack story recovery as it provides explicable detection results and reduces the need for domain experts as well as redundant manual efforts [11].

3 MAGIC Overview

MAGIC is a novel self-supervised APT detection approach that leverages masked graph representation learning and outlier detection methods and is capable of efficiently performing multi-granularity detection on massive audit logs. MAGIC's pipeline consists of three main components: (1) provenance graph construction, (2) a graph representation module and (3) a detection module. It also provides an optional (4)

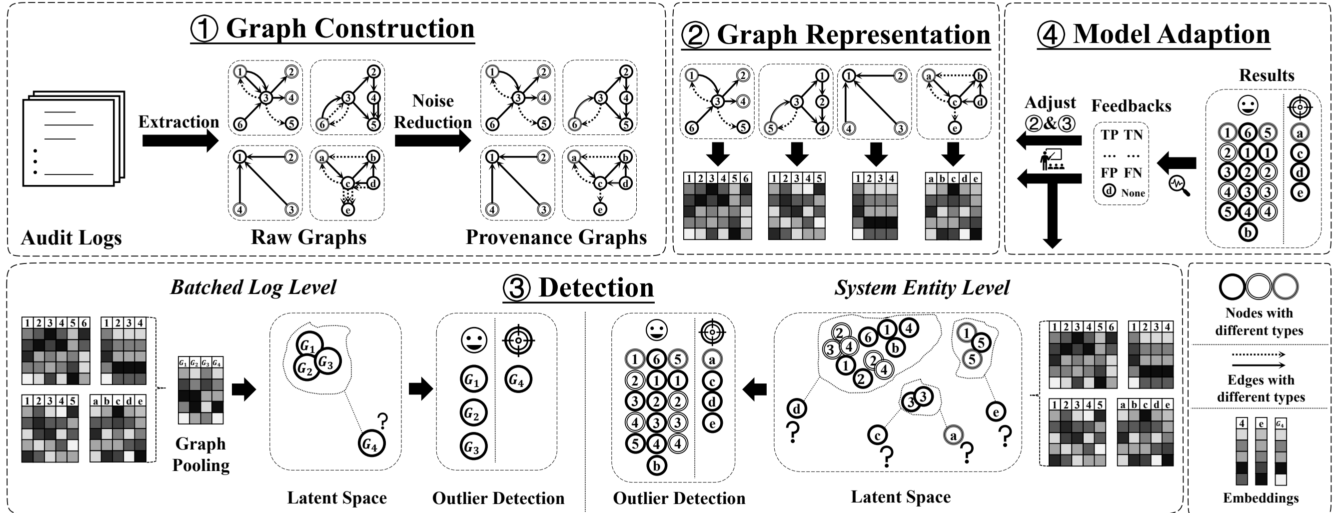


Figure 2: Overview of MAGIC's detection pipeline.

model adaption mechanism. During training, MAGIC transforms training data with (1), learns graph embedding by (2) and memorizes benign behaviors in (3). During inference, MAGIC transforms target data with (1), obtains graph embedding with the trained (2) and detects outliers through (3). Figure 2 gives an overview of the MAGIC architecture.

Streaming audit logs collected by system auditing softwares are usually stored in batches. During provenance graph construction (1), MAGIC transforms these logs into static provenance graphs. System entities and interactions between them are extracted and converted into nodes and edges respectively. Several complexity reduction techniques are utilized to remove redundant information.

The constructed provenance graphs are then fed through the graph representation module (2) to obtain output embeddings (i.e. comprehensive vector representations of objects). Built upon *graph masked auto-encoders* and integrating *sample-based structure reconstruction*, the graph representation module embeds, propagates and aggregates node and edge attributes into output embeddings, which contain both node embeddings and the system state embedding.

The graph representation module is trained with only benign audit logs to model benign system behaviors. When performing APT detection on potentially attack-containing audit logs, MAGIC utilizes *outlier detection methods* based on the output embeddings to detect outliers in system behaviors (3). Depending on the granularity of the task, different embeddings are used to complete APT detection. On *batched log level* tasks, the system state embeddings, which reflect the general behaviors of the whole system, are the detection targets. An outlier in such embeddings means its corresponding system state is unseen and potentially malicious, which reveals an APT signal in that batch. On *system entity level* tasks, the detection targets are those node embeddings, which represent the behaviors of system entities. Outliers in node embeddings indicates suspicious system entities and detects APT threats in finer granularity.

In real-world detection settings, MAGIC has two pre-designed applications. For each batch of logs collected by system auditing softwares, one can either directly utilize MAGIC's entity level detection to accurately identify malicious entities within the batch, or perform a two-stage detection, as stated in Sec. 2.3. In this case, MAGIC first scans a batch and sees if malicious signals exist in the batch (batched log level detection). If it alerts positive, MAGIC then performs entity level detection to identify malicious system entities in finer granularity. Batched log level detection is significantly less computationally demanding than entity level detection. Therefore, such a two-stage routine can help MAGIC's users to save computational resource and avoid false alarms without affecting MAGIC's detection fineness. However, if users favor fine-grain detection on all system entities, the former routine is still an accessible option.

To deal with *concept drift* and unseen attacks, an optional model adaption mechanism is employed to provide feedback channels for its users (4). Detection results checked and confirmed by security analysts are fed back to MAGIC, helping it to adapt to benign system behavior changes in a semi-supervised way. Under such conditions, MAGIC achieves even more promising detection results, which is discussed in Sec. 6.3. Furthermore, MAGIC can be easily applied to real-world online APT detection thanks to it's ability to adapt itself to *concept drift* and its minimum computation overhead.

4 Design Details

In this section, we explain in detail how MAGIC performs efficient APT detection on massive audit logs. MAGIC con-

tains four major components: a graph construction phase that builds optimised and consistent provenance graphs (Sec. 4.1), a graph representation module that produces output embeddings with maximum efficiency (Sec. 4.2), a detection module that utilizes outlier detection methods to perform APT detection (Sec. 4.3) and a model adaption mechanism to deal with *concept drift* and other high-quality feedbacks (Sec. 4.4).

4.1 Provenance Graph Construction

MAGIC first constructs a provenance graph out of raw audit logs before performing graph representation and APT detection. We follow three steps to construct a consistent and optimised provenance graph ready for graph representation.

Log Parsing. The first step is to simply parse each log entry, extract system entities and system interactions between them. Then, a prototype provenance graph can be constructed with system entities as nodes and interactions as edges. Now we extract categorical information regarding nodes and edges. For simple log format that provides entity and interaction labels, we directly utilize these labels. For some format that provides complicated attributes of those entities and interactions, we apply multi-label hashing (e.g., xxhash [28]) to transform attributes into labels. At this stage, the provenance graph is a directed multi-graph. We designed an example to demonstrate how we deal with the raw provenance graph after log parsing in Figure 3.

Initial Embedding. In this stage, we transform node and edge labels into fixed-size feature vector (i.e., initial embedding) of dimension d, where d is the hidden dimension of our graph representation module. We apply a *lookup embedding*, which establish an one-to-one mapping between node/edge labels to d-dimension feature vectors. As demonstrated in Figure 3 (I and II), process a and b share the same label, so they are mapped to the same feature vector, while a and c are embedded into different feature vectors as they have different labels. We note that the possible number of unique node/edge labels is determined by the data source (i.e., auditing log format). Therefore, the lookup embedding works under a transductive setting and do not need to learn embeddings for unseen labels.

Noise Reduction. The expected input provenance graph of our graph representation module would be simple-graphs. Thus, we need to combine multiple edges between node pairs. If multiple edges of the same label (also sharing the same initial embedding) exist between a pair of nodes, we remove redundant edges so that only one of them remains. Then we combine the remaining edges into one final edge. We note that between a pair of nodes, edges of several different labels may remain. After the combination, the initial embedding of the resulting unique edge is obtained by averaging the initial embeddings of the remaining edges. To illustrate, we show how our noise reduction combines multi-edges and how it affects the edge initial embeddings in Figure 3 (II an III). First, three *read* and two *write* interactions between a and c are merged into one for each label. Then we combine them together, forming one edge e_{ac} with initial embedding equal to the average initial embedding of the remaining edges (e'_2 and e'_5). We provide a comparison between our noise reduction steps and previous works in Appendix E.

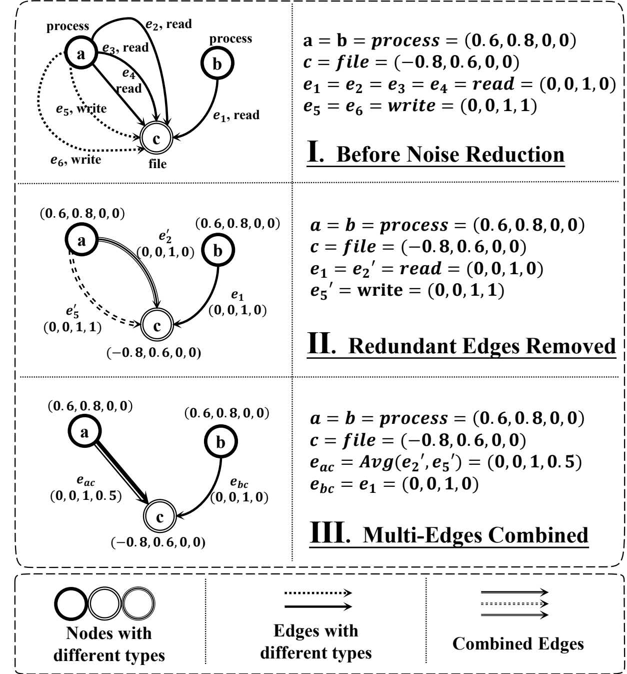


Figure 3: Example of MAGIC's graph construction steps.

After conducting the above three steps, MAGIC has finished constructing a consistent and information-preserving provenance graph ready for subsequent tasks. During provenance graph construction, little information is lost as MAGIC only damages the original semantics by generalizing detailed descriptions of system entities and interactions into labels. However, an average 79.60% of all edges are reduced on the DARPA E3 Trace dataset, saving MAGIC's training time and memory consumption.

4.2 Graph Representation Module

MAGIC employs a graph representation module to obtain high-quality embeddings from featured provenance graphs. As illustrated in Figure 4, the graph representation module consists of three phases: a masking procedure to partially hide node features (i.e. initial embeddings) for reconstruction purpose (Sec. 4.2.1), a graph encoder that produces node and system state output embeddings by propagating and aggregating graph features (Sec. 4.2.2), a graph decoder that provides supervision signals for the training of the graph representation module via masked feature reconstruction and sample-based structure reconstruction (Sec. 4.2.3). The encoder and decoder form a *graph masked auto-encoder*, which excels in

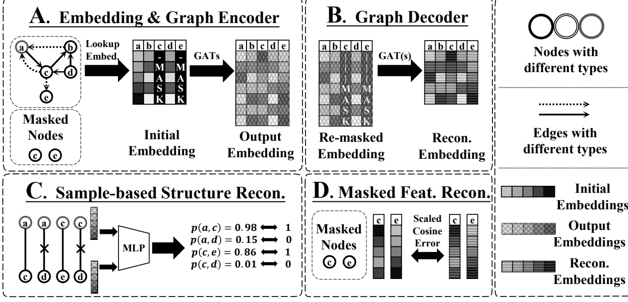


Figure 4: Graph representation module of MAGIC.

producing fast and resource-saving embeddings.

4.2.1 Feature Masking

Before training our graph representation module, we perform *masking* on nodes, so that the graph masked auto-encoder can be trained upon reconstruction of these nodes. Masked nodes are randomly chosen, covering a certain proportion of all nodes. The initial embeddings of such masked nodes are replaced with a special mask token x_{mask} to cover any original information about these nodes. Edges, however, are not masked because these edges provide precious information about relationships between system entities. In summary, given node initial embeddings x_n, we mask nodes as follows:

$$emb_n = \begin{cases} x_n, & n \notin \widetilde{N} \\ x_{mask}, & n \in \widetilde{N} \end{cases}$$

where $\widetilde{N}$ are randomly-chosen masked nodes, emb_n is the embedding of node n ready for training the graph representation module. This masking process only happens during training. During detection, we do not mask any nodes.

4.2.2 Graph Encoder

Initial embeddings obtained from the graph construction steps take only raw features into consideration. However, raw features are far from enough to model detailed behaviors of system entities. Contextual information of an entity, such as its neighborhood, its multi-hop relationships and its interaction patterns with other system entities plays an important role to obtain high-quality entity embeddings [29]. Here we employ and extend graph masked auto-encoders [19] to generate output embeddings in a self-supervised way. The graph masked auto-encoder consists of an encoder and a decoder. The encoder produces output embeddings by propagating and aggregating graph features and the decoder reconstructs graph features to provide supervision signals for training. Such encoder-decoder architecture maintains the contextual and semantic information within the generated embeddings, while its computation overhead is significantly reduced via masked learning.

The encoder of our graph representation module contains multiple stacked layers of graph attention networks (GAT) [30]. The function of a GAT layer is to generate output node embeddings according to both the features (initial embeddings) of the node itself and its neighbors. Differing from ordinary GNNs, GAT introduces an attention mechanism to measure the importance of those neighbors.

To explain in detail, one layer of GAT takes node embeddings generated by previous layers as input and propagates embeddings from source nodes to destination nodes into messages along the interactions. The message contains information about the source node and the interaction between source and destination:

$$MSG(src,dst) = W_{msg}^T(h_{src}||emb_e).$$

And the attention mechanism is employed to calculate the attention coefficients between the message source and its destination:

$$\alpha(src,dst) = LeakyReLU(W_{as}^T h_{src} + W_{am}MSG(src,dst)),$$
$$a(src,dst) = Softmax(\alpha(src,dst)).$$

Then for the destination node, the GAT aggregates messages from incoming edges to update its node embedding by computing a weighted sum of all incoming messages. The weights are exactly the attention coefficients:

$$AGG(h_{dst},h_{\mathcal{N}}) = W_{self}h_{dst} + \sum_{i\in\mathcal{N}} a(i,dst)MSG(i,dst),$$
$$h_n^l = AGG^l(h_n^{l-1}, h_{\mathcal{N}_n}^{l-1}).$$

where h_n^l is the hidden embedding of node n at l-th layer of GAT, h_n^{l-1} is that of layer $l-1$ and $\mathcal{N}_n$ is the one-hop neighborhood of n. The input of the first GAT layer are the initial node embeddings. emb_e is the initial edge embedding and remains constant throughout the graph representation module. $W_{as}, W_{am}, W_{self}, W_{msg}$ are trainable parameters. The updated node embedding forms a general abstraction of the node's one-hop interaction behavior.

Multiple layers of such GATs are stacked to obtain the final node embedding h, which is concatenated by the original node embedding and outputs of all GAT layers:

$$h_n = emb_n||h_n^1||\cdots||h_n^l.$$

where $\cdot||\cdot$ denotes the concatenate operation. The more layers of GAT stacked, the wider the neighboring range is and the farther a node's multi-hop interaction pattern its embedding is able to represent. Consequently, the graph encoder effectively incorporates node initial features and multi-hop interaction behaviors to abstract system entity behaviors into node embeddings. The graph encoder also applies an average pooling to all node embeddings to generate a comprehensive embedding of the graph itself [31], which recapitulates the overall

state of the system:

$$h_G = \frac{1}{|N|} \sum_{n_i \in N} h_{n_i}.$$

The node embeddings and system state embeddings generated by the graph encoder are considered the output of the graph representation module, which are used in subsequent tasks in different scenarios.

4.2.3 Graph Decoder

The graph encoder does not provide supervision signals that support model training. In typical graph auto-encoders [25, 27], a graph decoder is employed to decode node embeddings and supervise model training via *feature reconstruction* and *structure reconstruction*. Graph masked auto-encoders, however, abandon structure reconstruction to reduce computation overhead. Our graph decoder is a mixture of both, which integrates masked feature reconstruction and sample-based structure reconstruction to construct an objective function that optimizes the graph representation module.

Given node embeddings h_n obtained from the graph encoder, the decoder first re-masks those masked nodes and transforms them into the input of *masked feature reconstruction*:

$$h_n^* = \begin{cases} W^* h_n, & n \notin \widetilde{N} \\ W^* v_{remask}, & n \in \widetilde{N} \end{cases},$$

Subsequently, the decoder uses a similar GAT layer described above to reconstruct the initial embeddings of the masked nodes, allowing the calculation of a feature reconstruction loss:

$$x_n^* = AGG^*(h_n^*, h_{\mathcal{N}_n}^*),$$
$$L_{fr} = \frac{1}{|\widetilde{N}|} \sum_{n_i \in \widetilde{N}} \left(1 - \frac{x_{n_i}^T x_{n_i}^*}{||x_{n_i}|| \cdot ||x_{n_i}^*||}\right)^\gamma.$$

where L_{fr} is the masked feature reconstruction loss obtained by calculating a *scaled cosine loss* between initial and reconstructed embeddings of the masked nodes. This loss [19] scales dramatically between easy and difficult samples which effectively speeds up learning. The degree of such scaling is controlled by a hyper-parameter γ.

Meanwhile, sample-based structure reconstruction aims to reconstruct graph structure (i.e. predict edges between nodes). Instead of reconstructing the whole adjacency matrix, which has $O(N^2)$ complexity, sample-based structure reconstruction applies contrastive sampling on node pairs and predicts edge probabilities between such pairs. Only non-masked nodes are involved in structure reconstruction. Positive samples are constructed with all existing edges between non-masked nodes and negative samples are sampled among node pairs with no existing edges between them.

A simple two-layer MLP is used to reconstruct edges between node pairs samples, generating one probability for each sample. The reconstruction loss takes the form of a simple binary cross-entropy loss on those samples:

$$prob(n, n') = \sigma(MLP(h_n || h_{n'})),$$
$$L_{sr} = -\frac{1}{|\hat{N}|} \sum_{n \in \hat{N}} (log(1 - prob(n, n^-)) + log(prob(n, n^+))).$$

where (n, n^-) and (n, n^+) are negative and positive samples respectively and $\hat{N} = N - \widetilde{N}$ is the set of non-masked nodes. Sample-based structure reconstruction only provides supervision to the output embeddings. Instead of using dot products, we employ a MLP to calculate edge probabilities as interacting entities are not necessarily similar in behaviors. Also, we are not forcing the model to learn to predict edge probabilities. The function of such structure reconstruction is to *maximize behavioral information* contained in the abstracted node embeddings so that a simple MLP is sufficient to incorporate and interpret such information into edge probabilities.

The final objective function $L = L_{fr} + L_{sr}$ combines L_{fr} and L_{sr} and provides supervision signals to the graph representation module, enabling it to learn parameters in a self-supervised way.

4.3 Detection Module

Based on the output embeddings generated by the graph representation module, we utilize outlier detection methods to perform APT detection in an unsupervised way. As detailedly explained in previous sections, such embeddings summarize system behaviors in different granularity. The goal of our detection model is to identify malicious system entities or states given only a priori knowledge of *benign system behaviors*. Embeddings generated via graph representation learning tend to form clusters if their corresponding entities share similar interaction behaviors in the graph [19,25–27,32]. Thus, outliers in system state embeddings indicate uncommon and suspicious system behaviors. Based on such insight, we develop a special outlier detection method to perform APT detection.

During training, benign output embeddings are first abstracted from the training provenance graphs. What the detection module does at this stage is simply *memorizing* those embeddings and organize them in a *K-D Tree* [33]. After training, the detection module reveals outliers in three steps: k-nearest neighbor searching, similarity calculation and filtering. Given a target embedding, the detection module first obtains its k-nearest neighbors via K-D Tree searching. Such searching process only takes $log(N)$ time, where N is the total number of memorized training embeddings. Then, a similarity criterion is applied to evaluate the target embedding's closeness to its neighbors and compute an anomaly score. If its anomaly score yields higher than a hyper-parameter θ, the target embedding is considered an outlier and its corresponding system entity or system state is malicious. An example workflow of the detection module is formalized as follows,

using euclidean distance as the similarity criterion:

$$\mathcal{N}_x = KNN(x)$$
$$dist_x = \frac{1}{|\mathcal{N}_x|}\sum_{x_i \in \mathcal{N}_x} ||x - x_i||$$
$$score_x = \frac{dist_x}{\overline{dist}}$$
$$result_x = \begin{cases} 1, & score_x \geq \theta \\ 0, & score_x < \theta \end{cases}$$

where $\overline{dist}$ is the average distance between training embeddings and their k-nearest neighbors. When performing *batched log level detection*, the detection module memorizes benign system state embeddings that reflects system states and detects if the system state embedding of a newly-arrived provenance graph is an outlier. When performing *system entity level detection*, the detection module instead memorizes benign *node embeddings* that indicates system entity behaviors and given a newly-arrived provenance graph, it detects outliers within the embeddings of all system entities.

4.4 Model Adaption

For an APT detector to effectively function in real-world detection scenarios, concept drift must be taken into consideration. When facing benign yet previously unseen system behaviors, MAGIC produces false positive detection results, which may mislead subsequent applications (e.g. attack investigation and story recovery). Recent works address this issue by forgetting outdated data [10] or fitting their model to benign system changes via a *model adaption mechanism* [18]. MAGIC also integrates a model adaption mechanism to combat concept drift and learn from false positives identified by security analysts. Slightly different from other works that use only false positives to retrain the model, MAGIC can be retrained with all feedbacks. As discussed in previous sections, the graph representation module in MAGIC encodes system entities into embeddings in a self-supervised way, without knowing its label. Any unseen data, including those true negatives, are valuable training data for the graph representation module to enhance its representation ability on unseen system behaviors.

The detection module can only be retrained with benign feedbacks to keep up to system behavior changes. And as it memorizes more and more benign feedbacks, its detection efficiency is lowered. To address this issue, we also implement a discounting mechanism on the detection module. When the volume of memorized embeddings exceeds a certain amount, earliest embeddings are simply removed as newly-arrived embeddings are learned. We provide the model adaption mechanism as an optional solution to concept drift and unseen system behaviors. It is recommended to adapt MAGIC to system changes by feeding confirmed false positive samples to MAGIC's model adaption mechanism.

5 Implementation

We implement MAGIC with about 3,500 lines of code in Python 3.8. We develop several log parsers to cope with different format of audit logs, including StreamSpot [34], Camflow [35] and CDM [36]. Provenance graphs are constructed using the graph processing library Networkx [37] and stored in JSON format. The graph representation module is implemented via PyTorch [38] and DGL [39]. The detection module is developed with Scikit-learn [40]. For hyper-parameters of MAGIC, the scaling factor γ in the feature reconstruction loss is set to 3, the number of neighbors k is set to 10, the learning rate as 0.001 and the weight decay factor equals 5×10^{-4}. We use a 3 layer graph encoder and a mask rate of 0.5 in our experiments. The output embedding dimension d is different on two detection scenarios, batched log level detection and entity level detection. We use d equals 256 in batched log level detection and of and an we set d equals 64 in entity level detection to reduce resource consumption. The detection threshold θ is chosen by a simple linear search separately conducted on each dataset. The hyper-parameters may have other choices. We demonstrate the impact of these hyper-parameters on MAGIC later in the evaluation section. In our hyper-parameter analysis, d is chosen from {16, 32, 64, 128, 256} , l from {1, 2, 3, 4} and r from {0.3, 0.5, 0.7}. For the threshold θ, it is chosen between 1 and 10 in batched log level detection. For entity level detection, please refer to Appendix D.

6 Evaluation

We use 131GB of audit logs derived from various system auditing softwares to evaluate the effectiveness and efficiency of MAGIC. We first describe our experimental settings (Sec. 6.1), then elaborate the effectiveness of MAGIC in different scenarios (Sec. 6.2), conduct a false positive analysis and assess the usefulness of the model adaption mechanism (Sec. 6.3) and analyze the run-time performance overhead of MAGIC (Sec. 6.4). The impact of different components and hyper-parameters of MAGIC is analyzed in Sec. 6.5. In addition, a detailed case study on our motivation example is conducted in Appendix C to illustrate how MAGIC's pipeline work for APT detection. These experiments are conducted under the same device setting.

6.1 Experimental Settings

We evaluate the effectiveness of MAGIC on three public datasets: the StreamSpot dataset [21], the Unicorn Wget dataset [22] and the DARPA Engagement 3 datasets [20]. These datasets vary in volume, origin and granularity. We believe by testing MAGIC's performance on these datasets, we are able to compare MAGIC with as many *state-of-the-art*

Table 1: Datasets for batched log level detection.

Dataset	Scenario	Malicious	#Log pieces	Avg. #Entity	Avg. #Interaction	Size(GB)
StreamSpot	CNN		100	8,989	294,903	0.9
	Download		100	8,830	310,814	1.0
	Gmail		100	6,826	37,382	0.1
	VGame		100	8,636	112,958	0.4
	YouTube		100	8,292	113,229	0.3
	Attack	✓	100	8,890	28,423	0.1
Unicorn Wget	Benign		125	265,424	975,226	64.0
	Attack	✓	25	257,156	949,887	12.6

Table 2: Datasets for system entity level detection.

Dataset	Scenario	Malicious	#Node	#Edge	Size (GB)
DARPA E3 Trace	Benign		3,220,594	4,080,457	15.40
	Extension Backdoor	✓	732		
	Pine Backdoor	✓	67,345		
	Phishing Executable	✓	5		
DARPA E3 THEIA	Benign		1,598,647	2,874,821	17.91
	Attack	✓	25,319		
DARPA E3 CADETS	Benign		1,614,189	3,303,264	18.38
	Attack	✓	12,846		

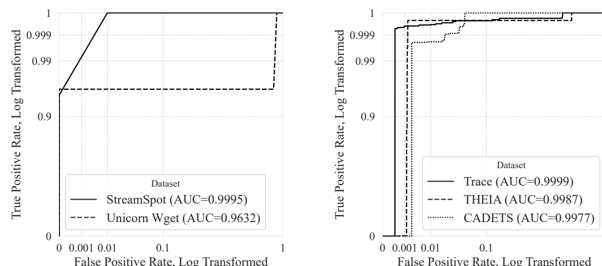
APT detection approaches as possible and explore the universality and applicability of MAGIC. we provide detailed descriptions of the three datasets as follows.

StreamSpot Dataset. The StreamSpot dataset (See Table 1) is a simulated dataset collected and made public by StreamSpot [34] using auditing system SystemTap [41]. The StreamSpot dataset contains 600 batches of audit logs monitoring system calls under 6 unique scenarios. Five of those scenarios are simulated benign user behaviors while the attack scenario simulates a drive-by-download attack. The dataset is considered a relatively small dataset and since no labels of log entries and system entities are provided, we perform *batched log level detection* on the StreamSpot dataset similar to previous works [10, 15, 17].

Unicorn Wget Dataset. The Unicorn Wget dataset (See Table 1) contains simulated attacks designed by Unicorn [10]. Specifically, it contains 150 batches of logs collected with Camflow [35], where 125 of them are benign and 25 of them contain *supply-chain attacks*. Those attacks, categorized as stealth attacks, are elaborately designed to behave similar to benign system workflows and are expected to be difficult to identify. This dataset is considered the hardest among our experimental datasets for its huge volume, complicated log format and the stealthy nature of these attacks. The same as state-of-the-art approaches, we perform *batched log level detection* on this dataset.

DARPA E3 Datasets. The DARPA Engagement 3 datasets (See Table 2), as a part of the DARPA Transparent Computing program, are collected among an enterprise network during an adversarial engagement. APT attacks exploiting different vulnerabilities [20] are conducted by the red team to exfiltrate sensitive information. Blue teams try to identify those attacks by auditing the network hosts and performing causality analysis on them. Trace, THEIA and CADETS sub-datasets are included in our evaluation. These three sub-datasets consist of a total 51.69GB of audit records, containing as many as 6,539,677 system entities and 68,127,444 interactions. Thus, we evaluate MAGIC's *system entity level detection* ability and address the overhead issue on these datasets.

For different dataset, we employ different dataset splits to evaluate the model and we use only benign samples for training. For the StreamSpot dataset, we randomly choose 400 batches out of 500 benign logs for training and the rest for testing, resulting in an balanced test set. For the Unicorn Wget dataset, 100 batches of benign logs are selected for training



(a) ROC curves on batched log level detection (b) ROC curves on system entity level detection

Figure 5: ROC curves on all datasets.

while the rest are for testing. For the DARPA E3 datasets, we use the same ground-truth labels as ThreaTrace [17] and split log entries according to their order of occurrence. The earliest 80% log entries are for training while the rest are preserved for testing. During evaluation, the average performance of MAGIC under 100 global random seeds is reported as the final result, so the experimental results may contain fractions of system entities/batches of logs.

6.2 Effectiveness

MAGIC's effectiveness of multi-granularity APT detection is evaluated on three datasets. Here we present the detection results of MAGIC on each dataset, then compare it with state-of-the-art APT detection approaches on those datasets.

Detection Result. Results show that MAGIC successfully detects APTs with high accuracy in different scenarios. We present the detection results of MAGIC on each dataset in Table 3 and their corresponding ROC curves in Figure 5.

On easy datasets such as the StreamSpot dataset, MAGIC achieves almost perfect detection results. This is because the StreamSpot dataset collects only single user activity per log batch, resulting in system behaviors that can be easily separated from each other. We further present this effect by visualizing the distribution of system state embeddings abstracted from those log batches in Figure 6. The system state embeddings are separated into 6 categories, matching the 6 scenarios involved in the dataset. Also, this indicates that MAGIC's graph representation module excels at abstracting system behaviors into such embeddings.

When dealing with the Unicorn Wget dataset, MAGIC

Table 3: MAGIC's detection results on different datasets. For batched log level detection, the detection targets are log pieces. And for system entity level detection, system entities are the targets.

Granularity	Dataset		Train Ratio	Ground Truth		#TP	#FP	#TN	#FN	Precision	Recall	FPR	F1-Score	AUC
				#Benign	#Malicious									
Batched log level	StreamSpot		80%	100	100	100.0	0.59	99.41	0.0	99.41%	100.00%	0.59%	99.71%	99.95%
	Unicorn Wget		80%	25	25	24.0	0.5	24.5	1.0	98.02%	96.00%	2.00%	96.98%	96.32%
System entity level	DARPA E3 Trace	All	80%	616,025	68,082	68,072	569	615,456	10	99.17%	99.98%	0.09%	99.57%	99.99%
		Extension Backdoor			732	727			5					
		Pine Backdoor			67,345	67,342			3					
		Phishing Executable			5	3			2					
	DARPA E3 THEIA	All	80%	319,448	25,319	25,318	456	318,992	1	98.23%	99.99%	0.14%	99.11%	99.87%
	DARPA E3 CADETS	All	80%	344,327	12,846	12,816	759	343,568	30	94.40%	99.77%	0.22%	97.01%	99.77%

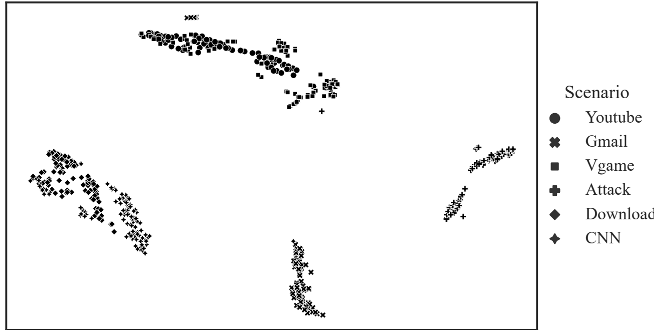


Figure 6: Latent space of system state embeddings in the StreamSpot dataset. Each point represents a log piece in the dataset, which belongs to one of six scenarios: watching YouTube, checking G-mail, playing vgame, undergoing a drive-by-download attack, downloading ordinary files and watching CNN.

Table 4: Comparison between MAGIC and state-of-the-art APT detection methods on different datasets. Within column *supervision*, B indicates benign data, A refers to attack data and SA for streaming attack data.

Dataset	Approach	Train Ratio	Supervision	Precision	F1-Score	Recall	FPR
StreamSpot	StreamSpot	80%	B	73%	81%	91%	6.6%
	Unicorn (baseline)	75%	B	95%	96%	93%	1.6%
	Prov-Gem	80%	B,A	100%	97%	94%	**0%**
	ThreaTrace	75%	B	98%	99%	99%	0.4%
	MAGIC (Ours)	80%	B	99%	**99%**	**100%**	0.6%
Unicorn Wget	Unicorn (baseline)	80%	B	86%	90%	95%	15.5%
	Prov-Gem	80%	B,A	100%	89%	80%	**0%**
	ThreaTrace	80%	B	93%	95%	**98%**	7.4%
	MAGIC (Ours)	80%	B	98%	**97%**	96%	2.0%
DARPA E3 Trace	DeepLog	N/A	B,A	41%	51%	68%	2.7%
	Log2vec (baseline)	N/A	B,A	54%	64%	78%	1.8%
	ThreaTrace	N/A	B	72%	83%	99%	1.1%
	ShadeWatcher	80%	B,SA	97%	99%	**99%**	0.3%
	MAGIC (Ours)	80%	B	**99%**	**99%**	99%	**0.1%**
DARPA E3 THEIA	DeepLog	N/A	B,A	16%	15%	14%	0.5%
	Log2vec (baseline)	N/A	B,A	62%	64%	66%	0.3%
	ThreaTrace	N/A	B	87%	93%	99%	**0.1%**
	MAGIC (Ours)	80%	B	**98%**	**99%**	**99%**	0.1%
DARPA E3 CADETS	DeepLog	N/A	B,A	23%	35%	74%	4.4%
	Log2vec (baseline)	N/A	B,A	49%	62%	85%	1.6%
	ThreaTrace	N/A	B	90%	95%	**99%**	0.2%
	MAGIC (Ours)	80%	B	**94%**	**97%**	99%	**0.2%**

yields an average 98.01% precision and 96.00% recall, significantly lower compared to that of the StreamSpot dataset. MAGIC's self-supervised style makes it difficult to distinguish between stealth attacks and benign system behaviors. However, MAGIC still successfully recovers an average 24 out of 25 log batches with only 0.5 false positives generated, better than any of the state-of-the-art detectors [10, 15, 17].

On the DARPA datasets, MAGIC achieves an average 99.91% recall and 0.15% false positive rate with only benign log entries for training. This indicates MAGIC quickly learns to model system behaviors. The test set in this scenario is unbalanced, which means the total number of ground-truth benign entities far exceeds that of malicious entities. Among 1,386,046 test entities, only 106,246 are labeled malicious. However, few false positives are generated, as MAGIC identifies malicious entities with outlier detection and anomalous entities are naturally detected as abnormalities.

Among the false negative results, we notice that most of them are malicious files and libraries involved in the attacks. This indicates that MAGIC excels at detecting malicious processes and network connections that behave differently from benign system entities. However, MAGIC finds it hard to locate passive entities such as malicious files and libraries, whose behaviors tend to be similar to benign ones. Fortunately, these intermediate files and libraries can be easily identified during attack story recovery, given the malicious processes and connections successfully detected.

MAGIC vs. *State-of-the-art*. The three datasets used to evaluate MAGIC are also used by several state-of-the-art learning-based APT detection approaches. For instance, Unicorn [10], Prov-Gem [15] and ThreaTrace [17] for Unicorn Wget and StreamSpot dataset, ThreaTrace and ShadeWatcher [18] for sub-dataset E3-Trace. Methods that require *a priori* information about APTs, such as Holmes [3], Poirot [5] and Morse [6], are not taken into consideration as MAGIC cannot be compared to them in the same detection scenario.

Comparison results between MAGIC and other state-of-the-art approaches on each dataset are presented in Table 4. Comparison between MAGIC and other unsupervised approaches (i.e. Unicorn and ThreaTrace) yields a total victory of our approach, revealing MAGIC's effectiveness in modeling and detecting outliers of benign system behaviors with no supervision from attack-containing logs.

Beyond unsupervised methods, Prov-Gem is a supervised APT detector based on GATs. However, it fails to achieve a better detection result on even the easiest StreamSpot dataset. This is mainly because simple GAT layers supervised on classification tasks are not as expressive as graph

masked auto-encoders in producing high-quality embeddings. Another APT detector mentioned, ShadeWatcher, adopts a semi-supervised detection approach. ShadeWatcher leverages TransR [42] and graph neural networks (GNNs) to detect APTs based on recommendation and is able to achieve the best recall rate on the E3-Trace sub-dataset. TransR, a self-supervised graph representation method on Knowledge Graphs, contributes most to the detection accuracy of ShadeWatcher. Unfortunately, TransR is extremely expensive in computation overhead. For example, ShadeWatcher spends as much as 12 hours training the TransR module on the E3-Trace sub-dataset. On the contrary, evaluation on the computation overhead of MAGIC(Sec. 6.4) shows that MAGIC is able to complete training on the same amount of training data 51 times faster than ShadeWatcher [18].

6.3 False Positive Analysis

For real-time applications, APT detectors must prevent false alarms at best effort, as those false alarms often tire and confuse security analysts. We evaluate MAGIC's false positive rate (FPR) on benign audit logs and investigate how our model adaption mechanism reduces those false alarms. Table 3 shows MAGIC's false positive rate on each dataset. Within each dataset, only benign logs are used for training and testing. MAGIC yields low FPR (average 0.15%) with large training data. This is because MAGIC models benign system behaviors with self-supervised embeddings, allowing it to effectively handle unseen system entities. Such a low FPR enables MAGIC's application under real-world settings. When conducting fine-grain entity-level detection only, MAGIC only yields 569 false alarms on the Trace dataset, with an average only 40 false alarms every day. Security analysts can easily handle this number of alarms and do security investigations on them. If the two-stage detection described in Sec. 3 is applied, the average number of false alarms every day can be further lowered to 24.

Our model adaption mechanism is designed to help MAGIC to learn from newly-observed unseen behaviors. We evaluate how such mechanism reduces false positives on benign audit logs in Table 5. Specifically, we test MAGIC on the Trace dataset under five different settings:

• Training on the first 80% log entries and testing on the rest 20% with no adaption, identical to our original setting.

• Training on the first 20% and testing on the last 20% with no adaption, for comparison purpose.

• Training on the first 20%, adapting on false positives generated from the following 20%, and testing on the last 20%.

• Training on the first 20%, adapting on both false positives and true negatives generated from the following 20% log entries, and testing on the last 20%.

• Training on the first 20%, adapting on both false positives and true negatives generated from the following 40% log entries, and testing on the last 20%.

Table 5: MAGIC's false positive rates on different datasets. The effect of model adaption mechanism is tested under different settings.

Dataset	Train Ratio	Adaption	Test Ratio	FPR
StreamSpot	80%	N/A	20%	0.59%
Unicorn Wget	80%	N/A	20%	2.00%
DARPA E3 Trace	80%	N/A	20%	0.089%
	20%	N/A	20%	0.426%
	20%	20% FP	20%	0.272%
	20%	20% FP & TN	20%	0.220%
	20%	40% FP & TN	20%	0.173%

Table 6: Performance overhead of MAGIC on the E3-Trace sub-dataset.

Phase	Component	Time consumption (s)		Peak Memory consumption (MB)
		with GPU	CPU only	
Graph Construction	N/A	642		2,610
Training	Graph Representation	151	685	1,564
	Detection	78		1,320
Inference	Graph Representation	5	10	2,108
	Detection	825		1,667

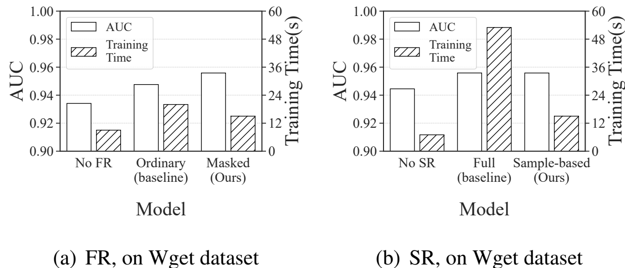
Experimental results indicate that adapting the model to benign feedbacks consistently reduces false positives. A further reduction can be achieved by feeding both false positives and true negatives to the model. This is because the graph representation module can be retrained with any data to enhance its representation ability, as described in Sec 4.4.

6.4 Performance Overhead

MAGIC is designed to perform APT detection with minimum overhead, granting it applicability under various conditions. MAGIC completes its training and inference in logarithmic time and takes up linear space. We provide a detailed analysis of its time and space complexity in Appendix B. We further test MAGIC's run-time performance on sub-dataset E3-Trace and present its time and memory consumption in Table 6.

In real-world settings, GPUs may not be available at all. We also test MAGIC's efficiency without the GPU. With only CPUs available, the training phase becomes apparently slower. The efficiency of graph construction and outlier detection phase is not affected as they are implemented to perform on CPUs. We also measure the max memory consumption during training and inference. MAGIC's low memory consumption not only prevents OOM problems on huge datasets, but also makes MAGIC approachable under tight conditions.

Evaluation results on performance overhead manifest the claim that MAGIC is advantageous over other state-of-the-art APT detectors in efficiency. For instance, ATLAS [11] takes about an hour to train its model on 676MB of audit logs and ShadeWatcher [18] takes 1 day to train on the E3-Trace sub-dataset. Compared with ShadeWatcher, MAGIC is 51 times faster in training under the same train ratio setting (i.e. 80%



(a) FR, on Wget dataset (b) SR, on Wget dataset

Figure 7: Effect of different reconstruction components on MAGIC's performance and efficiency.

log entries for training on E3-Trace sub-dataset).

These evaluation results also illustrate that MAGIC is fully practicable in different conditions. Considering the fact that sub-dataset E3-Trace is collected in 2 weeks, 1.37GB of audit logs is produced per day. This means under CPU-only conditions, MAGIC takes only 2 minutes to detect APTs from those logs and complete model adaption every day. Such promising efficiency makes MAGIC an available choice for individuals and small-size enterprises. For larger enterprises and institutions, they produces audit logs in hundreds of GBs every day [23]. In this case, efficiency of MAGIC can be ensured by training and adapting itself with GPUs and parallelizing the detection module with distributed CPU cores.

6.5 Ablation Study

In this section, we first address the effectiveness of important individual components in MAGIC's graph representation module, then carry out a hyper-parameter analysis to evaluate the sensitivity of MAGIC. Analysis on individual components and most hyper-parameters are conducted on the most difficult Wget dataset and the hyper-parameter analysis on the detection threshold θ is conducted on all datasets.

Individual Component Analysis. We study how feature reconstruction (FR) as well as structure reconstruction (SR) affect MAGIC's performance and Figure 7 presents the impact of these component to both detection result and performance overhead. Both FR and SR provide supervision for MAGIC's graph representation module. Compared with ordinary FR, Masked FR slightly boosts performance and significantly reduces training time. Sampled-based SR, however, is an effective complexity reduction component which accelerates training without losing performance, compared with full SR.

Hyper-parameter Analysis. The function of MAGIC is controlled by several hyper-parameters, including an embedding dimension d, number of GAT layers l, node mask rate r and the outlier detection threshold θ. Figure 8 illustrates how these hyper-parameters affect model performance in different situations. Hyper-parameters have little impact in most cases.

Generally speaking, relatively higher model performance is achieved with a larger embedding dimension and more GAT layers by collecting more information from more distant neighborhoods, as shown in Sub-figure 8(a) and 8(b). However, increasing d or l introduces heavier computation which leads to longer training and inference time.

The default mask rate of 0.5 yields best results. This is because MAGIC is unable to get sufficient training under a low mask rate. And under a high mask rate, node features are severely damaged which prevents MAGIC to learn node embeddings via feature reconstruction. Increasing mask rate slightly introduces more computation burden.

We further examine the anomaly scores of entities to assess the sensitive of θ. A lower θ naturally leads to a higher recall performance at the cost of more false positives and vice versa. As demonstrated in Figure 9, most malicious entities are given high anomaly scores compared with benign ones and are well-separated from them with little overlapping. The considerable spaces between benign and malicious anomaly scores support the claim that MAGIC does not depend on a precise threshold θ to perform accurate detection in practical situations. We quantify such spaces in Appendix D.

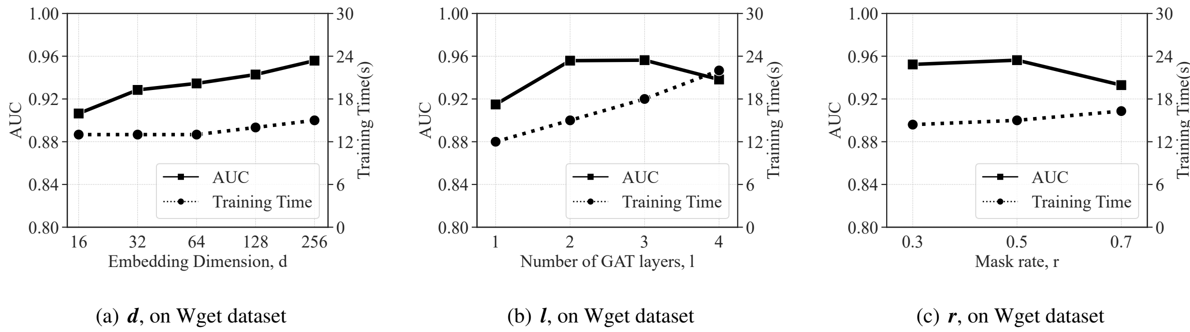
For an unsupervised detector as MAGIC, hyper-parameters are usually difficult to select. However, that is not the case for MAGIC. We provide a general guideline for hyper-parameter selection also in Appendix D.

7 Discussion and Limitations

Quality of Training Data. MAGIC models benign system behaviors and detects APTs with outlier detection. Similar to other anomaly-based APT detection approaches [7, 8, 10], we assume that all up-to-date benign system behaviors are observed during training log collection. However, if MAGIC is trained with low quality data that insufficiently covers system behaviors, many false positives can be generated. .

Outlier Detection. MAGIC implements a KNN-based outlier detection module for APT detection. While it completes training and inference in logarithmic time, its efficiency on large datasets is still unsatisfactory. To illustrate, our detection module takes 13.8 minutes to check 684,111 targets, making up 99% of total inference time. Other outlier detection methods, such as One-class SVM [43] and Isolation Forest [44], do not fit in our detection setting and cannot adapt to concept drift. Cluster-based methods and approximate KNN search may be more suitable to huge datasets. Meanwhile, KNN-based methods may be extended to GPU. We leave such improvements on our detection module to future research.

Adversarial Attacks. In Sec. 6.2, we show that MAGIC handles stealth attacks well, which avoid detection by behaving similar to the benign system. However, if attackers get to know how MAGIC works in details, they might conduct elaborately designed attacks to infiltrate our detector. We make a simple analysis on adversarial attacks and demonstrate MAGIC's robustness against them in Appendix A. As Graph-based



(a) ***d***, on Wget dataset (b) ***l***, on Wget dataset (c) ***r***, on Wget dataset

Figure 8: Effect of different hyper-parameters on MAGIC's performance and efficiency.

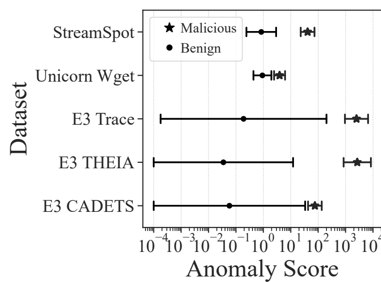


Figure 9: Anomaly scores of system entities. We exclude the highest and lowest 5% scores on each dataset.

approaches become increasingly popular and powerful in different detection applications, designing and avoiding these adversarial attacks on both the input graphs and the GNNs stands for an interesting research topic.

8 Related Work

MAGIC is mainly related to three fields of research, including the detection of APTs, graph representation learning and outlier detection methods.

APT Detection. The goal of APT detection is to detect APT signals, malicious entities and invalid interactions from audit logs. Recent works are mostly based on data provenance. As suggested by [18], provenance-based detectors can be categorized into rule-based, statistics-based and learning-based approaches. Rule-based approaches [2–6] utilize *a priori* knowledge about previous seen attacks and construct unique heuristic rules to detect them. Statistics-based approaches [7–9] construct statistics to measure the abnormality of provenance graph elements and perform anomaly detection on them. Learning-based approaches [10–18, 45] leverage deep learning to model either benign system behaviors [10, 12, 17] or attack patterns [11, 14–16, 18] and perform APT detection as classification [11, 15, 16] or anomaly detection [18, 45]. Among them, sequence-based methods [11, 14] detect APTs based on system execution/workflow patterns and graph-based methods [10, 12, 15–18, 45] model entities and interactions via GNNs and detect abnormal behaviors as APTs.

Graph Representation Learning. Embedding techniques on graphs start from the graph convolutional network (GCN) [46] and are further enhanced by the graph attention network (GAT) [30] and GraphSAGE [47]. Graph auto-encoders [19, 25–27] bring unsupervised solutions for graph representation learning. GAE, VGAE [25] and GATE [27] utilize feature reconstruction and structure reconstruction to learn output embeddings in a self-supervised way. However, they focus on link prediction and graph clustering tasks, irrelevant to our application. Recently, graph masked auto-encoders [19] leverage masked feature reconstruction and have achieved state-of-the-art performance on various applications.

Outlier Detection. Outlier detection methods view outliers (i.e. objects that may not belong to the ordinary distribution) as anomalies and aim to identify them. Traditional outlier detection methods include One-class SVMs [43], Isolation Forest [44] and Local Outlier Factor [48]. These traditional approaches are widely used in various detection scenarios, including credit card fraud detection [49] and malicious transaction detection [50]. This proves that outlier detection methods work well in anomaly detection. Meanwhile, auto-encoders themselves are effective tools for outlier detection. We explain why we do not use graph auto-encoders as anomaly detectors in Appendix F.

9 Conclusion

We have introduced MAGIC, an universally applicable APT detection approach that operates in utmost efficiency with little overhead. MAGIC leverages masked graph representation learning to model benign system behaviors from raw audit logs and performs multi-granularity APT detection via outlier detection methods. Evaluations on three widely-used datasets under various detection scenarios indicate that MAGIC achieves promising detection results with low false positive rate and minimum computation overhead.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd for their detailed and valuable comments. This work was supported by the National Natural Science Foundation of China (No. U1936213 and No. 62032025), CNKLSTISS, the Fundamental Research Funds for the Central Universities, Sun Yat-sen University (No. 22lgqb26) and Program of Shanghai Academic Research Leader (No. 21XD1421500).

References

[1] Adel Alshamrani, Sowmya Myneni, Ankur Chowdhary, et al. A survey on advanced persistent threats: Techniques, solutions, challenges, and research opportunities. *IEEE Communications Surveys & Tutorials*, 2019.

[2] Wajih Ul Hassan, Adam Bates, and Daniel Marino. Tactical provenance analysis for endpoint detection and response systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[3] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, et al. Holmes: real-time apt detection through correlation of suspicious information flows. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[4] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, et al. Sleuth: Real-time attack scenario reconstruction from cots audit data. In *USENIX Security Symposium*, 2017.

[5] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, et al. Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019.

[6] 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.

[7] Wajih Ul Hassan, Shengjian Guo, Ding Li, et al. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *network and distributed systems security symposium*, 2019.

[8] Qi Wang, Wajih Ul Hassan, Ding Li, et al. You are what you do: Hunting stealthy malware via data provenance analysis. In *NDSS*, 2020.

[9] Yushan Liu, Mu Zhang, Ding Li, et al. Towards a timely causality analysis for enterprise security. In *NDSS*, 2018.

[10] Xueyuan Han, Thomas Pasquier, Adam Bates, et al. Unicorn: Runtime provenance-based detector for advanced persistent threats. *arXiv preprint arXiv:2001.01525*, 2020.

[11] Abdulellah Alsaheel, Yuhong Nan, Shiqing Ma, et al. Atlas: A sequence-based learning approach for attack investigation. In *USENIX Security Symposium*, 2021.

[12] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, et al. Hercule: Attack story reconstruction via community discovery on correlated log graph. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, 2016.

[13] Yun Shen, Enrico Mariconti, Pierre Antoine Vervier, et al. Tiresias: Predicting security events through deep learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[14] Fucheng Liu, Yu Wen, Dongxue Zhang, et al. Log2vec: A heterogeneous graph embedding based approach for detecting cyber threats within enterprise. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019.

[15] Maya Kapoor, Joshua Melton, Michael Ridenhour, et al. Prov-gem: Automated provenance analysis framework using graph embeddings. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2021.

[16] Zitong Li, Xiang Cheng, Lixiao Sun, et al. A hierarchical approach for advanced persistent threat detection with attention-based graph neural networks. *Security and Communication Networks*, 2021.

[17] Su Wang, Zhiliang Wang, Tao Zhou, et al. Threatrace: Detecting and tracing host-based threats in node level through provenance graph learning. *IEEE Transactions on Information Forensics and Security*, 2022.

[18] Jun Zengy, Xiang Wang, Jiahao Liu, et al. Shadewatcher: Recommendation-guided cyber threat analysis using system audit records. In *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[19] Zhenyu Hou, Xiao Liu, Yuxiao Dong, et al. Graphmae: Self-supervised masked graph autoencoders. *arXiv preprint arXiv:2205.10803*, 2022.

[20] Darpa transparent computing program engagement 3 data release. `https://github.com/darpa-i2o/Transparent-Computing`. Accessed: 2022-09-20.

[21] The streamspot dataset. `https://github.com/sbustreamspot/sbustreamspot-data`. Accessed: 2022-09-17.

[22] Wget dataset. https://dataverse.harvard.edu/dataverse/unicorn-wget. Accessed: 2022-09-17.

[23] Zhang Xu, Zhenyu Wu, Zhichun Li, et al. High fidelity data reduction for big data security dependency analyses. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016.

[24] Hailun Ding, Juan Zhai, Yuhong Nan, and Shiqing Ma. Airtag: Towards automated attack investigation by unsupervised learning with log texts. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 373–390, 2023.

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

[26] Jiwoong Park, Minsik Lee, Hyung Jin Chang, et al. Symmetric graph convolutional autoencoder for unsupervised graph representation learning. In *Proceedings of the IEEE/CVF international conference on computer vision*, 2019.

[27] Amin Salehi and Hasan Davulcu. Graph attention autoencoders. *arXiv preprint arXiv:1905.10715*, 2019.

[28] Yann Collet. Xxhash: Extremely fast non-cryptographic hash algorithm. https://github.com/Cyan4973/xxHash.

[29] Keyulu Xu, Weihua Hu, Jure Leskovec, et al. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[30] Petar Veličković, Guillem Cucurull, Arantxa Casanova, et al. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[31] Chuang Liu, Yibing Zhan, Chang Li, et al. Graph pooling for graph neural networks: Progress, challenges, and opportunities. *arXiv preprint arXiv:2204.07321*, 2022.

[32] Xiao Wang, Nian Liu, Hui Han, et al. Self-supervised heterogeneous graph neural network with co-contrastive learning. In *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*, 2021.

[33] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 1975.

[34] Emaad Manzoor, Sadegh M Milajerdi, and Leman Akoglu. Fast memory-efficient anomaly detection in streaming heterogeneous graphs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[35] Thomas Pasquier, Xueyuan Han, Mark Goldstein, et al. Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing*, 2017.

[36] Joud Khoury, Timothy Upthegrove, Armando Caro, et al. An event-based data model for granular information flow tracking. In *Proceedings of the 12th USENIX Conference on Theory and Practice of Provenance*, 2020.

[37] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conference*, 2008.

[38] Adam Paszke, Sam Gross, Francisco Massa, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*. 2019.

[39] Minjie Wang, Da Zheng, Zihao Ye, et al. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

[40] F. Pedregosa, G. Varoquaux, A. Gramfort, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 2011.

[41] Vara Prasad, William Cohen, FC Eigler, et al. Locating system problems using dynamic instrumentation. In *2005 Ottawa Linux Symposium*, 2005.

[42] Yankai Lin, Zhiyuan Liu, Maosong Sun, et al. Learning entity and relation embeddings for knowledge graph completion. In *Proceedings of the AAAI conference on artificial intelligence*, 2015.

[43] Larry M Manevitz and Malik Yousef. One-class svms for document classification. *Journal of machine Learning research*, 2001.

[44] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 eighth ieee international conference on data mining*, 2008.

[45] Xueyuan Han, Xiao Yu, Thomas Pasquier, et al. Sigl: Securing software installations through deep graph learning. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[46] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[47] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 2017.

[48] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, et al. Lof: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000.

[49] Panpan Zheng, Shuhan Yuan, Xintao Wu, et al. One-class adversarial nets for fraud detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019.

[50] Zakia Ferdousi and Akira Maeda. Unsupervised outlier detection in time series data. In *22nd International Conference on Data Engineering Workshops (ICDEW'06)*, 2006.

[51] Jun Zeng, Zheng Leong Chua, Yinfang Chen, et al. Watson: Abstracting behaviors from audit logs via aggregation of contextual semantics. In *NDSS*, 2021.

Appendix

A Analysis on Adversarial Attack

Adversarial attacks against MAGIC, such as evasion attacks and poison attacks, are tricky to implement but still possible to carry out. Two types of adversarial attacks are potentially practical against MAGIC, manipulating input audit logs or exploiting model architecture. The later approach is not an option for MAGIC's attackers, as they have no access to MAGIC's inner parameters. Similar to SIGL [45], we conduct a simple experiment concerning MAGIC's robustness against graph manipulations, which shows these attacks do not affect MAGIC's detection effectiveness.

In this experiment, we expect attackers have no knowledge of MAGIC's inner parameters and cannot get any feedback from MAGIC. However, attackers can freely manipulate the malicious entities within MAGIC's input audit logs, as well as a small proportion of benign entities. Consequently, we consider four types of attacks:

Malicious Feature Evasion (MFE). Attackers have altered the features of all malicious entities in the raw audit logs trying to evade detection. This affects the node initial embeddings of the input provenance graph and forces malicious entities to mimic benign ones in node features.

Malicious Structure Evasion (MSE). Attackers have adding new edges between malicious entities and benign ones, so that malicious entities behave more normally and tend to have local structures more similar to benign entities. This affects the graph representation module and pulls the embeddings of malicious nodes towards benign ones, making them more difficult to identify.

Combined Evasion (MCE). This type of attack is a combination of the MFE and MSE and causes malicious entities to approximate benign ones in both features and structures.

Table 7: Impact of different adversarial attack strategies on MAGIC's detection effectiveness.

Attack Type	None	MFE	MSE	MCE	BFP
AUC	0.9999	0.9999	0.9999	0.9994	0.9942

Benign Feature Poison (BFP). Attackers have manipulated the features of benign entities to poison MAGIC. Attackers have injected some benign entities with similar initial features as malicious entities, trying to convince MAGIC that the malicious entities behave normally. This shifts benign entities to mimic malicious ones in node features and gradually poisons MAGIC's detection module.

We evaluate MAGIC's robustness against the above four attack strategies on the E3-Trace sub-dataset and we present the results in Table 7. Experimental results confirm that MAGIC is robust enough against adversarial attacks. MFE has almost no impact on MAGIC's effectiveness. MAGIC is slightly more vulnerable facing the structure evasion attacks, because the graph structure is critical to unsupervised graph representation learning. However, the structure attacks are still weak against MAGIC's detection effectiveness. We believe that the feature and structure reconstruction involved in our graph representation module contributes to this robustness, as it learns a model to reconstruct both node features and its neighboring structure with the information from its neighbors. Consequently, adversarial attacks involving feature and structure altering of malicious entities will fail, and with MAGIC's model itself well-protected, attackers have to either extend their effort on searching and manipulating benign entities, or find other attack strategies against MAGIC. Meanwhile, BFP poses a relatively greater threat to MAGIC's performance. However, under BFP, only 0.057 AUC reduction is achieved by manipulating the input feature of an enormous 161,029 benign entities. This level of intervention in the benign system is beyond the capability of ordinary attackers and will definitely leave observable trace. Therefore, an effective BFP attack is also very difficult to carry out.

B Time and Space Complexity of MAGIC

Given number of system entities N, number of system interactions E, number of possible node/edge labels t, graph representation dimension d, number of GAT layers l and mask rate r the graph construction steps builds a featured provenance graph in $O((N+E)*t)$ time, masked feature reconstruction is completed in $O((N+E)*d^2*l*r)$ time and sample-based structure reconstruction takes only $O(N*d)$ time. Training of the detection module takes $O(N*logN*d)$ time to build a K-D Tree and memorize benign embeddings. Detection result of a single target is obtained in $O(logN*d*k)$ time. Thus, the overall time complexity of MAGIC during training and inference is $O(N*logN*d*k+E*d^2*l*r+(N+E)*t)$.

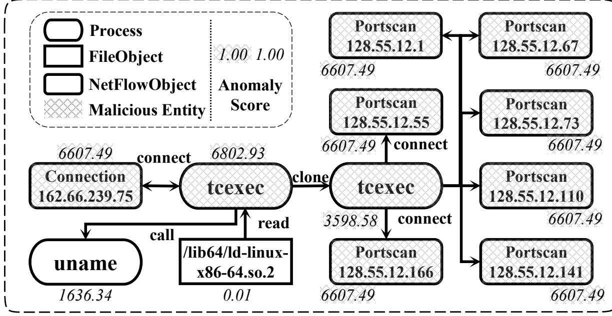


Figure 10: Another provenance graph of our motivating example (i.e. Pine Backdoor). Numbers assigned to nodes are the anomaly scores assessed by MAGIC's detection module.

MAGIC's memory consumption largely depends on d and t. The graph representation module takes up $O((N+E)*t)$ space to store a provenance graph and $O((N+E)*(t+d))$ space to generate output embeddings. The detection module takes $O(N*d)$ space to memorize benign embeddings. The overall space complexity of MAGIC is $O((N+E)*(t+d))$.

C Case Study

We use our motivating example described in Sec. 2.1 again to illustrate how MAGIC detects APTs from audit logs. Our motivating example involves an APT attack: Pine Backdoor, which implants an malicious executable to a host via phishing e-mail, aiming to perform internal reconnaissance and build a silent connection between the host and the attacker. We perform *system entity level detection* on it and obtain real detection results from MAGIC. First, MAGIC constructs a provenance graph from raw audit logs. We provide the example provenance graph in Figure 10 to illustrate. Among them, *tcexec*, *Connection-162.66.239.75* and the portscan *NetFlowObjects* which *tcexec* connects to are malicious entities while the others are benign ones. The graph representation module then obtains their embeddings via the graph masked auto-encoder. The embedding of *tcexec* is calculated by propagating and aggregating information from its multi-hop neighborhood to model its interaction behavior with other system entities. For instance, its 2-hop neighborhood is namely *Connection-162.66.239.75*, *tcexec*'s sub-process, *uname*, *ld-linux-x86-64.so.2* and other portscan *NetFlowObjects*. The detection module subsequently calculates distances between those embeddings and their k-nearest benign neighbors in the latent space and assigns anomaly scores to them. The malicious entities are given very high anomaly scores (3598.58~6802.93) that far exceed our detection threshold (3000) while others present low abnormality (0.01~1636.34). Thus, MAGIC successfully detects malicious system entities with no false alarm generated.

D Hyper-parameter Choice Guideline

The choice of detection threshold θ depends on the corresponding dataset MAGIC is operating on. However, finding a precise θ is not necessary on each individual dataset, according to Sec. 6.5. Even if there is only benign data available, users may adjust and choose the optimal θ according to the resulting false positive rate (i.e. use the first θ when false positive rate is below the desired value). Here we provide a simple analysis on how wide the threshold selection range is and what consequence selecting different θ may lead to.

Impact of Different θ Choices. Altering θ does not lead to drastic changes in detection results, especially for #FN. For example, on the E3-Trace sub-dataset, changing θ from 1000 to 5000 results in 2 more FNs and an 1% decrease in FPR. On E3-THEIA, altering θ from 100 to 500 provides 0.5% FPR decrease and does not change #FN. On E3-Cadets, setting θ as 100 instead of 10 leads to a 10% decreased FPR with only 30 new FNs generated.

Quantifing the Threshold Selection Range. By limiting FPR below 1% on entity level tasks, we get a minimum θ equals 1980 on E3-Trace, 180 on E3-THEIA dataset and 95 on E3-CADETS dataset. The resulting recalls are respectively 0.99985, 0.99996 and 0.99782. Meanwhile, the maximum θs that ensure recall > 99% are actually 6600, 1020 and 120 on the three different sub-datasets. The space between the minimum and maximum θ is big enough and the curves are flat. Consequently, MAGIC does not need a precise θ to obtain the desirable result and selecting threshold θ based on false positive rate is practical.

E Noise Reduction

Noise reduction is widely adopted by various recent works [11, 15, 17, 18] to reduce the complexity of provenance graphs and remove redundant and useless information. MAGIC applies a mild noise reduction approach as MAGIC is less sensitive to the scale of the provenance graph and more information is preserved in this way. Compared with noise reduction done in recent works [11, 18], we neither delete irrelevant nodes nor merge nodes with the same interaction behavior. This is because (1) attack-irrelevant nodes provide information for benign system behaviors and (2) multiple nodes with the same interaction behavior duplicate the information propagated to near-by nodes and impact on their embeddings.

F Auto-encoder Based Anomaly Detection

Among traditional applications of machine learning, anomaly detection via auto-encoders is common practice. Typically, auto-encoders are trained to reconstructing a target and minimize its reconstruction loss. Thus, the reconstruction loss of a newly-arrived sample indicates how similar it behaves

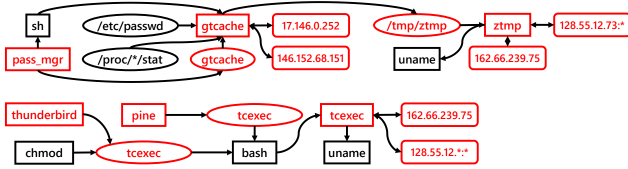


Figure 11: Attack graph of sub-dataset E3-Trace.

to training samples and samples with high reconstruction errors are detected as outliers. However, we do not apply auto-encoder-based outlier detection because of two reasons: (1) our sample-based structure reconstruction produces high-variance reconstruction loss on single sample, which prevents stable threshold-based outlier detection and (2) MAGIC performs batched log level detection by detecting outliers in system state embeddings, which do not have a reconstruction target and cannot be compared in reconstruction error.

G Entity-level Data Labeling on DARPA TC datasets

A Feasible Labeling Methodology. Mining and labeling attack-relevant entities in DARPA TC datasets can be extremely effort-consuming, given the fact that the ground truth document they provide is practically unreadable. Recently, Watson [51] have carrited out a successful attempt to label the E3-Trace sub-dataset. We are able to repeat this labeling methodology on sub-datasets E3-Trace, E3-THEIA and E3-CADETS. The following is a detailed description on this labeling process:

• Traverse all log entries in the dataset. Among those records, we extract **process**, **file** and **netflow** entities.

• Extract entity names. Entities' semantic names are stored in different fields. For instance, *Subject.properties.map.name* stores **process** names in sub-dataset E3-Trace.

• Mine key attack-relevant entities from the ground truth document. Some attacks were not well-recorded but at least one of the attacks using the same strategy is well-recorded.

• Match the names of key attack entities with all extracted entities. The matching entities are labeled as positive. Explore the neighborhood of these entities and search for other entities that are involved in the attack. Newly-identified ones are also treated as positives.

The Resulting Ground Truth. We perform such labeling steps on sub-datasets E3-Trace, E3-THEIA and E3-CADETS and obtain the following ground truth, in the form of descriptive text and attack graphs.

• E3-Trace (Figure 11). Two successful attack attempts worked on Trace: Browser Extension and Pine Backdoor. During the Browser Extension attack, the user downloaded and executed *gtcache* via browser extension *pass_mgr*. **gtcache** communicated with the attacker, scaned sensitive information and created **ztmp** to portscan host *128.55.12.73*. In the Pine Backdoor attack, the user unfortunately launched the phishing executable *tcexec*. **tcexec** connected back to the attacker and performed a wide postscan on the local network.

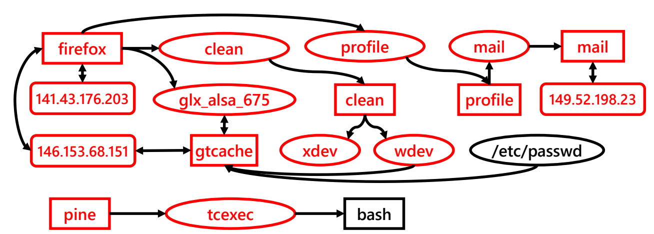


Figure 12: Attack graph of sub-dataset E3-THEIA.

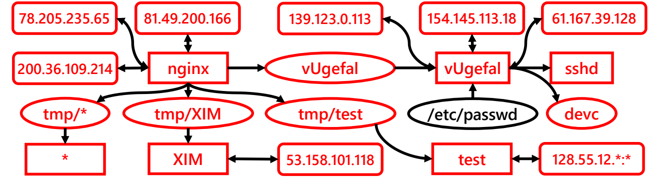


Figure 13: Attack graph of sub-dataset E3-CADETS.

• E3-THEIA (Figure 12). Three successful attack attempts worked on THEIA: Firefox Backdoor, Browser Extension and Pine Backdoor. The user was compromised by a malicious payload *clean* while browsing via *firefox*. **clean** acquired root privileges, connected back to the attacker and executed another payload *profile*. The Browser Extension attack aimed to resume the first attack by re-establishing the connection, grabbing root privileges and portscanning the local network via **gtcache** and **mail**. The Pine Backdoor attack is very similar to the one conducted on Trace but unexpectedly stopped due to missing library error.

• E3-CADETS (Figure 13). The attacker exploited the Nginx Backdoor and tried two attacks on CADETS. During the first attack, the attacker connected to a vulnerable Nginx server running on CADETS and injected process **vUgefal** with root privileges. It read sensitive information and tried to further infect the *sshd* process with malicious implants before the host crashed. The attacker then tried another attack on CADETS, resulting in a malicious process **XIM**. The attacker also created another process **test** to establish a long-lasting connection and portscan the local network.