FLASH: A Comprehensive Approach to Intrusion Detection via Provenance Graph Representation Learning

Mati Ur Rehman
University of Virginia

Hadi Ahmadi
Corvic Inc.

Wajih Ul Hassan
University of Virginia

Abstract*—Recently, provenance-based Intrusion Detection Systems (IDSes) have gained popularity for their potential in detecting sophisticated Advanced Persistent Threat (APT) attacks. These IDSes employ provenance graphs created from system logs to identify potentially malicious activities. Despite their potential, they face challenges in accuracy, practicality, and scalability, particularly when dealing with large provenance graphs. We present FLASH, a scalable IDS that leverages graph representation learning through Graph Neural Networks (GNNs) on data provenance graphs to overcome these limitations. FLASH employs a Word2Vec-based semantic encoder to capture essential semantic attributes (e.g., process names and file paths) and the temporal ordering of events within the provenance graph. Furthermore, FLASH incorporates a novel adaptation of a GNN-based contextual encoder to efficiently encode both local and global graph structures into expressive node embeddings. To learn benign node behaviors, we utilize a lightweight classifier that combines the GNN and Word2Vec embeddings. Recognizing the computational demands and slow processing times of GNN, particularly for large provenance graphs, we have developed an embedding recycling database to store the node embeddings generated during the training phase. During runtime, our lightweight classifier leverages the stored embeddings, obviating the need to regenerate GNN embeddings, thus facilitating real-time APT detection. Extensive evaluation of FLASH on real-world datasets demonstrates superior detection accuracy compared to existing provenance-based IDSes. The results also illustrate FLASH's scalability, robustness against mimicry attacks, and potential for accelerating the alert verification process.

1. Introduction

As cybersecurity threats evolve, Intrusion Detection Systems (IDSes) have become a critical part of enterprise security strategies, particularly in combating Advanced Persistent Threats (APTs). APTs, characterized by their stealthy and persistent nature, as seen in infamous attacks, such as Solar Winds [13] and NotPetya [10], pose severe challenges due to their potential to cause significant system damage. IDSes must detect these threats with high accuracy, maintain low false positive rates, and consume minimal resources without hindering system performance.

To enhance the ability of IDSes to combat sophisticated intrusions, the cybersecurity field has turned to data provenance. Data provenance analysis applied to system (audit) logs enables parsing host events into provenance graphs, which describe the entirety of system execution. Consequently, provenance-based IDSes [30, 51, 66, 67, 69, 71] have recently emerged as a promising approach to detect intrusions, leveraging the rich contextual information available in system logs to enhance detection performance. However, in the complex context of enterprise security, current provenance-based IDSes encounter the following main limitations.

- **Semantic Information Neglect:** Existing IDSes commonly disregard valuable semantic data, such as process names, command line arguments, file paths, and IP addresses in provenance graphs. This overlook can lead to less accurate detection.
- **Temporal & Causal Ordering Disregard:** Several IDSes overlook the importance of the temporal and causal ordering of system events. This disregard could lead to incorrect threat identification as it misses the sequence in which malicious activities occur.
- **Scalability Challenges:** Real-time detection is crucial for an effective IDS. However, several existing IDSes face scalability issues, specifically when handling large provenance graphs. This limits their real-time application, causes log congestion, and leaves systems vulnerable to ongoing attacks.
- **Coarse-grained Detection:** Many IDSes identify malicious subgraphs rather than individual malicious nodes, making alert validation and attack reconstruction both time-consuming and error-prone for security analysts. Further, this approach leaves these systems vulnerable to evasion attacks [26].

In this work, we propose FLASH, a novel IDS that leverages provenance graph representation learning to efficiently and accurately detect APT attacks. To tackle the issue of overlooking semantic information during detection, FLASH employs a Word2Vec-based embedding technique to encode various node attributes present in provenance graphs, such as process names, command line arguments, file paths, and IP addresses, into semantically-rich feature vectors. Moreover, we modified our Word2Vec technique to obtain temporally sensitive embeddings, addressing the issue of disregarding

temporal ordering among events. Our experiments demonstrate that semantically-rich feature vectors that consider temporal ordering among events more effectively distinguish between benign and malicious nodes, reducing false positive rates and enhancing overall detection performance.

To encode structural and causal information, we apply graph representation learning by leveraging Graph Neural Networks. Graph representation learning through Graph Neural Networks (GNNs) has the potential to learn expressive node embeddings that capture both local and global graph structures. However, existing GNN techniques are not scalable and notoriously slow, particularly for large graphs, rendering their application in the provenance-based IDS domain prohibitively challenging and impractical [12, 73].

The challenge of scalability in GNNs to provide real-time detection is handled by implementing two innovative techniques. First, we enhance the efficiency of graph traversal in graph representation learning by selecting only the edges that are relevant for threat identification. Second, we devised a GNN embedding database that is inspired by embedding recycling techniques previously applied to language models [59]. This database stores node embeddings during the training process and during runtime, we employ these pre-generated GNN embeddings for real-time detection, reducing real-time processing time by over 70% compared to state-of-the-art GNN-based IDS while maintaining superior detection performance.

In addressing the challenge of coarse-grained detection, FLASH conducts fine-grained anomaly detection by identifying individual malicious nodes (e.g., processes and files) using GNN instead of the entire anomalous subgraph, containing benign nodes as well. Furthermore, our approach demonstrates robustness against adversarial mimicry attacks on provenance-based IDSes, as presented by Goyal et al. [26]. After detection, FLASH aids in threat management and attack reconstruction by ranking threat alerts using FLASH-generated anomaly scores and creating Attack Evolution Graphs (AEGs). These AEGs, comprising only alert nodes and their causal links, offer a concise intrusion progression view compared to the raw provenance graph.

We evaluated the effectiveness and efficiency of FLASH through a series of experiments conducted on real-world datasets provided by DAPRA and the research community. *To the best of our knowledge, this is the first attempt to evaluate a provenance-based IDS on the DAPRA OpTC dataset, which is the largest system log dataset released by DARPA to date.* These datasets encompass a broad spectrum of attack scenarios and system behaviors.

Our results reveal that FLASH surpasses existing provenance-based IDSes in terms of detection rate. By employing our GNN embedding database, we managed to significantly reduce the time overhead for anomaly detection, demonstrating an improvement of up to three times compared to existing state-of-the-art provenance-based IDS. Furthermore, by using anomaly scores generated by FLASH for threat alerts, security analysts can effectively triage alerts and reduce their workload by up to 60%, indicating the potential of FLASH in managing threat alert fatigue [1].

Our work's main contributions are:

- We propose a provenance-based IDS, FLASH, which harnesses contextual and structural information from provenance graphs to enhance its detection capabilities.
- We introduce a two-step process to generate semantic and contextual embeddings using Word2Vec and GNN, respectively. This process is followed by real-time anomaly detection through a lightweight classifier model, ensuring system scalability and efficiency.
- We offer two schemes – selective graph traversal and embedding recycling database – that make graph representation learning practical for IDS settings.
- We conduct a comprehensive evaluation of our technique on real-world datasets. The results highlight FLASH's effectiveness in identifying malicious activities, its resilience against adversarial mimicry attacks, and its ability to accelerate the alert validation process.

Availability. FLASH is available online at https://github.com/DART-Laboratory/Flash-IDS

2. Motivation

In this section, we present a real-world attack scenario from the DAPRA OpTC [4] dataset to highlight the limitations of existing provenance-based IDSes and demonstrate the effectiveness and utility of FLASH. The experimental setup used for FLASH is described in Section 5.

Attack Scenario. In the given attack scenario, the attacker sends phishing emails to targeted victims. These emails contain malicious PowerShell Empire stagers. Upon opening the email attachment, the attacker gains access to the victim's system. The attack agent then establishes a connection to the command and control (C&C) server and covertly remains in the system for several days. The agent's objective is to examine the system configuration and search for sensitive data. To maintain stealth, the attacker carries out minimal system activity and imitates the behavior of benign system entities. Once the agent locates the desired files, it downloads a payload from the command server and exfiltrates the data to the server.

Figure 1 presents a simplified depiction of the attack scenario described above. This particular attack was executed as part of a DAPRA red team exercise during the data collection process. The red team installed a C&C agent on the system, which used `find.exe` to search for critical files and gather system information. The attack agent then downloaded a program called `fileTransfer1000.exe` via `Chrome.exe` from `news.com:8080`. This program compressed files in the documents directory and exfiltrated them to `news.com:9999`. This is a typical example of a data exfiltration attack, in which the attacker aims to steal sensitive information from the target system while remaining undetected by mimicking benign system processes.

2.1. Limitations of Existing Provenance IDSes

Existing provenance-based intrusion detection systems (IDSes) primarily function by learning benign behavior

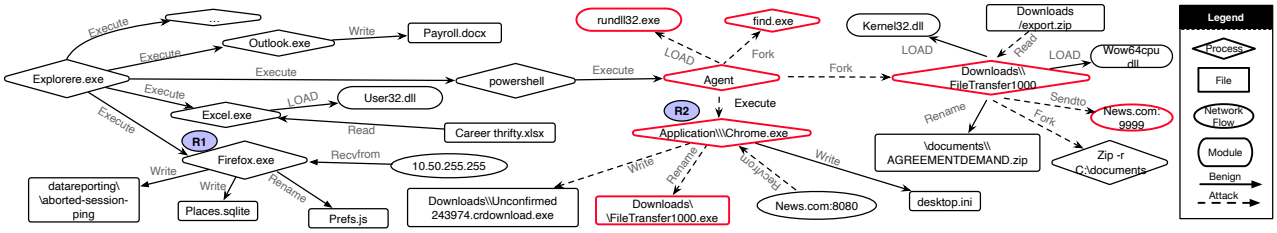


Figure 1: Example of a data exfiltration attack provenance graph from the DARPA OpTC dataset. The attack sequence involves the attacker scanning for sensitive files within the system and transmitting them to a command server. FLASH effectively identified all key attack components, represented by red nodes in the figure.

TABLE 1: Limitations of existing provenance-based intrusion detection systems. "-" denotes instances where we could not confirm the results from the corresponding paper.

	Semantic Encoding	Temporal Encoding	Scalable	Detection Granularity	Contextual Alerts	Robust Against Mimicry Attacks
FLASH	✓	✓	✓	**Node**	✓	✓
ThreaTrace [67]	X	X	X	Node	✓	✓
Unicorn [30]	X	✓	✓	Graph	X	X
ProvDetector [66]	X	X	X	Graph	X	X
StreamSpot [51]	X	X	✓	Graph	X	X
ProGrapher [69]	X	✓	✓	Graph	X	-
ShadeWatcher [71]	✓	X	X	Edge	✓	-

models from provenance graphs. Anomalies are detected whenever deviations from these established models occur. For instance, ProGrapher [69] utilizes Graph2Vec [56] and TextRCNN [44] embeddings to identify anomalies at the graph level. Similarly, StreamSpot [51] extracts graph features to construct benign models and leverages a clustering technique to discern abnormal graphs. Unicorn [30], on the other hand, employs a graph similarity matching method to locate abnormal graphs. It simplifies provenance graph features into histograms and forms fixed-size graph sketches, which it then clusters to detect anomalies. ProvDetector [66] adopts a path-based strategy for malware detection. It identifies infrequent paths in a graph, uses path embedding generation, and employs the Local Outlier Factor [9] method for anomaly detection. ThreaTrace [67] uses GNN to execute node-level anomaly detection, learning the structural information of nodes in the benign dataset and identifying anomalies based on deviations from this learned behavior. Finally, ShadeWatcher [71] uses GNN to depict a system entity's preferences towards interactive entities, suggesting adversarial interactions through edge-level anomaly detection. Despite these advancements, current provenance-based IDSes are hindered by several limitations that impact their practical applicability in real-world settings. These limitations are described below and summarized in Table 1.

Semantic Encoding. Existing provenance-based IDSes utilize ML algorithms to detect system attacks based on the distribution of system events for each node as feature vectors. However, these systems often fail to consider crucial semantic attributes, such as process names, command lines, or file paths. This oversight can lead to a high rate of false positives, as distinguishing between benign and malicious entities becomes increasingly difficult. For instance, consider the attack graph presented in Figure 1. Each node contains semantic attributes that help to establish the surrounding context. The `firefox.exe` node labeled `R1` is a benign system node, while the `Chrome.exe` node labeled `R2` is under the attacker's control. If we remove the semantic attributes from these two nodes, their activities look identical. In typical provenance graphs, such instances could be numerous. Therefore, semantic attributes are crucial for reducing false alarms and helping the detector to better understand and discriminate between malicious and benign nodes based on their activities.

Temporal Encoding. Existing provenance-based IDSes do not consider the temporal ordering of system events, focusing instead solely on event frequency per node. This approach limits the model's understanding of benign node distributions, given that several nodes can perform the same events in varying orders. This oversight results in elevated false alarms. Our experimental results, as detailed in Section 5, demonstrate a significant decrease in false positives when temporal ordering is incorporated, in comparison to a baseline that disregards temporal factors.

Scalability. Several existing provenance-based IDSes utilize GNN to extract structural information from provenance graphs, which improves detection by capturing the relationships between neighboring nodes. However, the high computational demands of GNN can hinder the scalability of the system and complicate real-time intrusion detection [73]. FLASH, on the other hand, implements scalability enhancement techniques for GNN that result in significantly faster performance (as shown in Figure 3) compared to existing systems that do not utilize these techniques.

Detection Granularity. The degree of granularity in graph-based detection plays a significant role in the effectiveness of IDSes. Many existing IDSes like StreamSpot and Unicorn use graph-level anomaly detection, which struggles to identify stealthy attacks with minimal activity. For example, in Figure 1, the attack involved just 0.02% of the total nodes, causing the graph to look largely normal. Such methods also fail to accurately pinpoint anomalous nodes, leading to lower detection rates than node-level systems, such as ThreaTrace. Systems like ProvDetector trigger alerts based on anomalous

path thresholds, providing a coarse-grained detection. Conversely, Shadewatcher detects anomalous edges, a more detailed but slower approach that restricts it to offline detection due to the sheer number of potential edges. Therefore, node-level detection strikes the optimal balance between detail and speed.

Contextual Alerts. IDSes with graph-level granularity identify potentially malicious subgraphs rather than pinpointing the exact malicious nodes (e.g., processes and files) responsible for threat alerts. As security analysts have to determine the veracity of each IDS-generated threat alert by understanding the context around the alert, this leads to a time-consuming and error-prone process of attack reconstruction and alert validation. In such cases, a security analyst must thoroughly analyze the marked subgraph to find potentially malicious nodes. On the other hand, node- and edge-level detection methods provide improved efficiency for investigations after an attack by directly pointing to the exact entities involved in the intrusion and data provenance around that entity.

Robustness Against Mimicry Attacks. Goyal et al. [26] introduced a methodology to evade IDSes that operate at the graph and path level granularity. Their approach manipulates the distributional graph encoding, modifying the node neighborhoods in the attack graph to mimic those in benign provenance graphs. When we applied the adversarial attack strategy as defined by Goyal et al. on our proposed system (as detailed in Section 5), we found our system to be resistant to such evasion attempts. This resistance affirms that node-level detection methods, such as the one used by FLASH, offer better protection against evasion attacks due to their ability to identify anomalies at a more granular level.

3. Threat Model & Assumptions

To build a comprehensive threat model, we make several assumptions about the attacker's behavior and the system's characteristics. We assume that attackers are covert and actively conceal their malicious actions by blending them with legitimate background data. This approach makes it challenging to identify malicious activity and differentiate it from benign behavior. Moreover, attackers may use zero-day exploits to target the system, implying that there are no known attack patterns available for training. This lack of prior knowledge poses a significant challenge to developing effective detection techniques.

In our threat model, we assume that attackers must leave behind identifiable patterns in the system's records to carry out malicious activities distinct from typical behavior. These patterns will differentiate the structure of the attacker's node from those of legitimate nodes with the same label. By analyzing the graph structure and the features of the nodes, we can identify unusual entities corresponding to the attacker's behavior and track them over time. Similar to other data provenance works [19, 33, 47, 49], we assume that the provenance collection system provides accurate and complete records of all activities and changes occurring in the computer system. Moreover, we assume that the integrity of the collected audit logs and the embedding recycling database is maintained all the time by using existing tamper-resistant storages [14, 57]. This ensures that the provenance graphs constructed by FLASH are reliable and can be used effectively for detecting and analyzing cybersecurity threats.

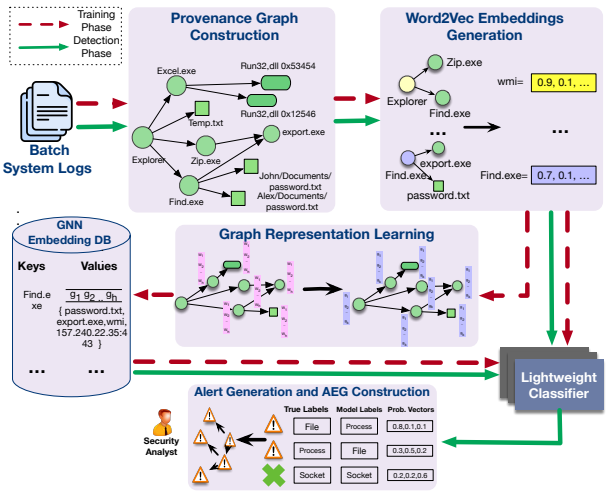


Figure 2: FLASH architecture. FLASH uses Word2Vec and GNN to learn semantic and contextual embeddings from the provenance graphs. After that, FLASH uses a downstream classifier to learn benign behaviors from those embeddings.

4. FLASH Design

FLASH, illustrated in Figure 2, is composed of five key modules: a provenance graph constructor, a Word2Vec-based semantic encoder, a GNN-based contextual encoder, an embedding database, and an anomaly detector. At a high-level, FLASH generates provenance graphs from streaming system logs, uses semantic attributes to create feature vectors for nodes, and applies GNN for context-aware encoding. These embeddings are stored in a database and later used by a lightweight classifier during threat detection.

4.1. Provenance Graph Construction

Our system begins with the transformation of system logs into provenance graphs, depicted in Figure 2. This graph construction process is rooted in existing provenance graph research [20, 32, 33, 36, 42, 48]. It involves three primary steps. First, FLASH interprets system logs such as Windows Event Logs [7] or Linux Audit Logs [8]. These logs consist of host event records like process executions, file operations, and network connections. In the next step, FLASH manages real-time log collection by processing system logs in configurable batches. The batch size K sets the number of system events per batch. FLASH then sequentially operates on each log batch and transforms it into a provenance graph for subsequent analysis.

Finally, FLASH uses the parsed data to construct the provenance graph, comprising two node types: process

nodes and object nodes. Object nodes encompass files, network flows, modules, and other system objects. Edges between these nodes bear labels specifying the event type (system calls), which demonstrate the causal relationship between connected nodes and the timestamp of the event. Further, nodes contain attributes, such as process name, command line, file path, IP address, port numbers, and module path, providing further context and details.

4.2. Semantic Attribute Encoding

The system logs contain rich attributes related to various system entities. These attributes, such as process names and file paths, first need to be encoded into vector space so that they can be utilized by the model. Some traditional methods of doing this would be one-hot encoding [11] and bag of words [2] approach. However, these methods lead to very sparse vectors which do not take into account the context around each word in the document.

To overcome these limitations, we apply the Word2Vec model [53] to translate these attributes into a dense vector space. Word2Vec, a neural network-based approach, excels at learning word vector representations. It operates on the principle of predicting a word's context with a shallow neural network. This approach yields a low-dimensional dense vector for each word, where similar words carry similar vector representations. Our model considers the following node attributes per node type: process name and command line arguments for process nodes, file path for file nodes, network IP address and port for socket nodes, and module name for module nodes. This data helps generate semantically-rich feature vectors. We form summary sentences for each node by combining semantic attributes and the types of causal events (system calls) between nodes and their 1-hop neighbors. System events are sorted by timestamps to maintain temporal ordering. Each sentence is then encoded into a fixed-length vector through a Word2Vec model trained on benign system logs.

Our method captures semantic relationships between words, generating dense embeddings that provide node features for subsequent graph representation learning. This comprehensive feature set complicates the mimicry of benign nodes by malicious ones. For successful mimicry, malicious nodes must duplicate system activity, temporal order, and semantic attributes. This improved detection ability allows our model to spot otherwise overlooked malicious activities. Algorithm 1 outlines the process for generating semantic embedding vectors.

4.2.1. Temporal Encoding. The standalone Word2Vec model has a key limitation: it does not retain the sequential order of words in a sentence. To rectify this and enhance our model's expressiveness, we devise a temporal encoding scheme that assigns individual weights to each word embedding. These weights, accumulated over a sentence, yield an embedding enriched with temporal information. We initiate this approach by arranging system events in chronological order based on timestamps, facilitating the integration of temporal ordering into our sentences. Subsequently, we incorporate positional encoding, a concept borrowed from Transformers [65], into the input embeddings to convey information about the position of each token in the sequence. This inclusion is crucial as Word2Vec lacks a built-in concept of order, so positional encoding allows the model to distinguish tokens based on their sequence position. The positional encoding vector for a token at position i is given by:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$
$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \quad (1)$$

where d represents the dimensionality of the embedding, and j is the index of the dimension within the embedding.

The positional encoding vector is then added to the Word2Vec embedding vector for each token at that position. This operation is performed element-wise, resulting in the embedding for a token at position i being:

$$Emb_i = Emb_i + PE_i \quad (2)$$

By combining the Word2Vec embeddings with positional encoding, our model can capture not only the semantic associations between words but also the order of words in the summary sentence. This enriched representation allows the GNN in the next step to better understand the context and detect malicious activities that might have previously gone undetected.

Algorithm 1: NODESENTENCEEMBEDDINGS

Inputs : Provenance Graph G;
Trained Word2Vec Model $w2v$;
Output: Array V of sentence encoding.

$D \leftarrow list([])$
/* Iterating over neighbors of the node */
foreach $N \in G$ **do**
  /* Getting the syscall performed on this node */
  A = GETACTIONPERFORMED(N)
  /* Getting node properties like process name, file path, command line, etc. */
  S = GETNODEATTRIBUTES(N)
  /* Concatenating the words into a list */
  $D.append$(A)
  $D.append$(S)
end
/* Initializing the embedding vector */
$V \leftarrow list([])$
/* Iterating over words of document D */
foreach $w \in D$ **do**
  /* Getting Word2Vec embeddings for this word */
  $E = w2v(w)$
  $V.append(E)$
end
/* Giving weight to each index of the vector V to capture the temporal order of system events */
P = GETPOSITIONALECONDINGVECTOR($len(V)$)
$V \leftarrow V + P$
/* Averaging the embeddings for all words to get one vector for the complete sentence. */
$V \leftarrow V.mean()$
return V

4.3. Provenance Graph Representation Learning

The efficient identification of stealthy attack nodes in provenance graphs necessitates the encoding of the k-hop neighborhood structure surrounding each node. The current state-of-the-art method for achieving this feat is graph representation learning via Graph Neural Network (GNN). GNN is adept at learning low-dimensional vector representations of nodes in a graph, thereby effectively capturing their structural information using the connections between nodes.

In our approach, we harness the Word2Vec embeddings acquired from the previous step as feature vectors for the nodes in our GNN. The GNN employs these features in conjunction with the graph structure to learn the structural information of each node in the provenance graph. However, employing graph representation learning can pose significant resource overhead and introduce higher latency, particularly in large graphs [73], due to the message-passing process. Message passing entails nodes exchanging information with their neighbors in the graph, which is iteratively repeated to enable nodes to receive information from increasingly distant parts of the graph. Consequently, leveraging GNN continuously during live intrusion detection would not only be resource-intensive but also lead to scalability issues, as provenance graphs in enterprise settings are immense [32, 47]. In order to tackle the aforementioned challenge and make GNN practical for provenance-based IDS, we employ two techniques, which are elaborated below.

We run GNN in an offline manner to generate contextual embeddings, which are subsequently stored for later utilization by a lightweight classifier model. We observed that a vast majority of system entities perform the same set of activities during each system execution, rendering it inefficient to run the GNN for them every time they are encountered. Our meticulous analysis of the DAPRA OpTC dataset revealed that over 80% of system entities exhibit the same neighborhood structure in both the benign and evaluation phases of system execution. By storing the GNN output for these nodes, we can avoid using the GNN for approximately 80% of the time while FLASH operates in real-time. We provide a detailed description of our embedding database in Section 4.4.

4.3.1. Selective Graph Traversal. Our GNN model takes inspiration from the GraphSage framework [28], recognized for its ability to learn node embeddings. GraphSage employs a neighborhood aggregation method, capturing localized information about nodes. However, to ensure efficient processing of the large provenance graph with relevance to threat detection, we have adapted the graph traversal algorithm of GraphSage. This adaptation gives priority to the edges that are important for threat detection and investigation, removing the necessity for complete provenance graph traversal. To accomplish this, we have devised a series of graph traversal principles. These principles guide GraphSage to selectively aggregate information from particular edges before applying GNN. We utilize the following traversal principles (TPs) to determine which nodes and edges should be included during aggregation.

- **TP1: Unique Edge Sampling:** In this traversal principle, we sample only a single edge of the same type between two nodes. Given that the provenance graph can contain multiple edges of the same type between nodes, we ensure that such edges are included only once during traversal.
- **TP2: Low-Priority Event Exclusion:** This principle involves excluding edges that represent low-priority and forensically irrelevant system events. Such events may include delete events for files that were created temporarily by a process and never interacted with other processes during system execution, as well as exit events represented as self-loops for process nodes. Previous works [34, 37, 46] have also employed similar approaches to reduce noise in system logs.
- **TP3: Execution-Specific Information Filtering:** In this principle, we include nodes and edges with the same execution-specific information only once. Many neighboring nodes in the provenance graph may only differ due to execution-specific attributes but are otherwise identical. Examples include process nodes with the same image path but differing PIDs, module nodes with the same module path but different base addresses, and network flows with the same 4-tuple but different start and end times.
- **TP4: User-Specific Attribute Handling:** Here, we treat nodes or edges differing only in user-specific attributes as the same. Certain events and nodes in the provenance graph may differ only due to user-specific information in their attributes. For example, two modules with the same module path could differ if they have different user IDs in their path. For such nodes, we ignore the user-specific attributes and select only one of them.

This set of principles allows us to reduce the computational overhead of running GNN on the complete provenance graph while still producing informative and structurally-rich vector representations for nodes. During aggregation, we generate structural embeddings for a given graph $G = (V, E)$ where V is the set of nodes and E is the set of edges. For each node $v \in V$, let X_v be the input feature vector for that node, and let $h_v^{(0)} = X_v$ be the initial embedding. At each iteration k of the algorithm, GraphSage updates the embeddings for each node $v \in V$ using the following equation:

$$h_v^{(k)} = \sigma\left(W^{(k)} \cdot \text{AGGREGATE}_k(h_u^{(k-1)} : u \in \mathcal{N}(v) \cup v)\right)$$

where $\sigma(\cdot)$ is a non-linear activation function such as ReLU, $\mathcal{N}(v)$ is the set of neighbors of node v in the graph, and $\text{AGGREGATE}_k(\cdot)$ is an aggregation function that combines the embeddings of node v and its neighbors at iteration k. The matrix $W^{(k)}$ is a learnable parameter matrix that maps the aggregated features to the new embedding space. Following this procedure, the GraphSage can encode the structural information around each node to vector space.

4.3.2. Training & Weighted Cross-Entropy Loss. We employ a semi-supervised node classification approach to

train our novel GNN model. Our model uses the node's input features and graph structure to classify its type. Our provenance graphs include several node types, such as process, file, and socket. The GNN model, trained with labeled data, learns to identify benign nodes' types. As the node types in the system logs are typically imbalanced, we use *weighted cross-entropy* as our loss function. Weighted cross-entropy loss is a variant of the cross-entropy loss function, which is used when the classes in the data are imbalanced. Weighted cross-entropy loss addresses this issue by assigning a weight to each class that reflects its importance. Typically, the weight for each class is set to be inversely proportional to the frequency of samples in that class. This approach ensures that the loss function pays greater attention to the minority classes, thereby enhancing the model's competence in correctly classifying those classes. The equation for the weighted cross-entropy loss is:

$$L(y, f(x)) = -\sum_{i=1}^{n} (w_i y_i \log(f(x)_i) + (1-y_i) \log(1-f(x)_i))$$

where y is the true label vector, $f(x)$ is the predicted label vector, n is the number of classes, w_i is the weight for the i-th class, and $\log$ is the natural logarithm. By training our GNN model with this loss function, we ensure that it can effectively learn the structural relationships between different node types in the provenance graph. Once the model is trained, the embeddings it generates can be used as input to a lightweight classifier for real-time intrusion detection. As a result, our FLASH system can efficiently identify stealthy attack nodes by leveraging the power of GNN and a lightweight classifier, without the need for continuous GNN execution during live intrusion detection.

4.4. Embedding Recycling Database

Our system employs the trained GNN model to generate structural embeddings for all nodes present in our benign dataset. For efficient retrieval and storage of these GNN output vectors during real-time threat detection, we architect a specialized key-value store. The key is designed as a Persistent Node Identifier (PNI), tied to node attributes that remain invariant across distinct system executions. These include process names, file paths, module paths, and network flow IP addresses. The corresponding value encapsulates the GNN-derived embedding, together with a set of neighbors associated with that specific node.

To derive a PNI, we utilize attribute abstraction techniques to remove user- and execution-specific information. This ensures that the stored embeddings are generalizable and adaptable to provenance graphs across various hosts within the enterprise. To facilitate our abstraction process, we group provenance node types into different abstraction schemas, which are described below.

– *User Abstraction Schema:* Implemented for process and file node types, this schema omits user IDs from process names and file paths, achieving a high degree of generalization. For instance, the file path `/Users/john/.bashrc` is abstracted to `/Users/*/.bashrc`. This abstraction mechanism allows our system to leverage the embeddings of similar behaviors across multiple users, significantly improving its ability to detect anomalous patterns.
– *Network Connection Abstraction Schema:* Applied to socket node types, this schema eliminates start and end times, which results in enhancing generalizability across different system executions, helping our system to focus on the structural and behavioral aspects of the nodes. For example, a socket node with attributes `IP:192.168.1.10, Port:22, Start: 12:00:00, End: 12:10:00` is abstracted to `IP:192.168.1.10, Port:22`, encapsulating only essential connection information.
– *Module Path Abstraction Schema:* Module nodes have a path and base address attribute. The path persists across different executions while the base address changes. This schema retains only the module paths, ensuring a consistent and generalizable representation of module nodes.

4.4.1. Harnessing Pre-computed Embeddings. The motivation behind the construction of our embedding key-value database is twofold: optimizing real-time anomaly detection and reducing computational overhead. By pre-computing and storing the GNN embeddings for nodes with a stable neighborhood structure, our approach rapidly assigns stored embeddings to nodes with matching neighborhood structures during the real-time detection phase, enhancing the performance and scalability of our detection pipeline.

In this setup, the neighborhood set serves a pivotal function. It is instrumental in determining whether a node's neighborhood structure during real-time analysis matches that observed during the benign phase. In the event of a match, the node is immediately assigned the stored embedding from the key-value store. On the other hand, if there is a discrepancy in the neighborhood structures, the model will default to real-time generated Word2Vec features for anomaly detection. This balanced strategy ensures our system's computational efficiency and adaptability to structural changes in the network.

We use the Jaccard Index to compare the stored and current neighborhood of a node before assigning it the pre-stored GNN embeddings. The Jaccard index is defined as $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$ where A and B are sets, and $|A|$ and $|B|$ denote their cardinalities. The Jaccard index measures the similarity between two sets by calculating the ratio of the size of their intersection to the size of their union. The Jaccard Index is a simple metric and easy to compute. It is commonly used with binary data where the presence or absence of specific elements is important. Furthermore, it is known to be relatively robust for imbalanced datasets, where one set is significantly larger than the other.

During the real-time detection phase of FLASH, a node must fulfill the following two conditions before it can be assigned its stored GNN embeddings:

1) The PNI for the node must be present as a key in the feature store.
2) The stored neighborhood set corresponding to this PNI in the feature store must have a Jaccard index of 1 with the current neighborhood set of the given node.

If a node satisfies the above conditions, it is assigned the stored GNN embeddings; otherwise, FLASH relies on Word2Vec features for classifying that node using our lightweight classifier.

4.5. Lightweight Classifier

In our search for the optimal lightweight classifier, we evaluated numerous candidates including XGBoost, Support Vector Machines (SVM), and Random Forest. As detailed in Section 5, XGBoost consistently outperformed other classifiers, showcasing superior speed and accuracy. Thus, we chose XGBoost for our anomaly detection task. XGBoost uses an ensemble of decision trees and advanced boosting techniques for enhanced accuracy. It predicts the probability of sample classification using the function $f(x) = \sum_{k=1}^{K} f_k(x)$, where $f_k(x) = \text{sigmoid}(\alpha_k T_k(x))$. Here, $T_k(x)$ is the prediction of the k-th tree, and α_k determines the tree's contribution.

XGBoost minimizes the regularized objective function $J = L(y, f(x)) + \Omega(f)$, using gradient boosting to iteratively add new trees to the ensemble. Each new tree aims to minimize the gradient of the loss function against the current ensemble predictions. The XGBoost model uses the concatenated Word2Vec encoding vectors and GNN embedding vectors per node. It retrieves GNN embeddings from the pre-trained key-value store, generates Word2Vec features in real-time, performs inference, and saves the output for the next pipeline stage. This robust process underpins our IDS's performance and scalability. The detailed algorithm is in Appendix D.

FLASH detects anomalous nodes by comparing predicted and actual node types. The model considers both neighborhood structure and node attributes to learn benign patterns associated with various node types. For instance, it can generalize interactions between different processes and file types, as well as their connections. Malicious entities often exhibit neighborhood structures and attributes that deviate from benign patterns. As a result, when the model encounters such outliers during runtime, they do not fit into the learned distribution of nodes, leading to misclassification.

Leveraging Word2Vec features and stored GNN embeddings, FLASH predicts system entity (node) types. Misclassified nodes from the output indicate potential threats. We introduce a threshold parameter T to control alert generation. This parameter limits classification probability for the predicted class, reflecting the model's confidence in its prediction. Higher scores imply more confidence and likely anomalies.

Algorithm 2: Attack Evolution Graph Generation

Inputs : Graph $G(V, E)$; Alerts N; Hop length h
Output: AEG Graphs List IG

$List_G \leftarrow []$
foreach *Alert* $n \in N$ **do**
  $Paths \leftarrow$ GETCAUSALPATHS(n, h)
  // List to store paths containing alert nodes
  $AttackPaths \leftarrow []$
  foreach $P \in Paths$ **do**
    $AlertNodes \leftarrow P \cap N$
    if $len(AlertNodes) > 1$ **then**
      $CompactPath \leftarrow$ KEEPALERTNODESONLY(P)
      $AttackPaths \leftarrow AttackPaths + CompactPath$
    end
  end
  $List_G$.append$(AttackPaths)$
end
$IG \leftarrow []$
foreach $Pathlist \in List_G$ **do**
  // Connect all paths originating from an alert node n to construct a graph.
  $AEGraph \leftarrow ConvertToGraph(Pathlist)$
  $IG \leftarrow IG + AEGraph$
end
return IG

4.6. Attack Evolution Graph Construction

Given that the generated alerts may be dispersed throughout a large provenance graph, understanding causal relations between nodes necessitates numerous backward and forward tracing calls for analysts to investigate alerts. Additionally, different stages of APT attacks are scattered across the provenance graph, making it challenging to identify their relationships and progression.

To tackle these challenges, we introduce a scheme to construct compact Attack Evolution Graphs (AEG) from the large provenance graph and alerts generated by FLASH. The central concept is to interlink causally related alerts within the provenance graph, consequently constructing a series of concise AEGs. These AEGs encapsulate only alert nodes and their causal links, providing a streamlined and clear view of alert node interactions. This reduction significantly simplifies the original graph, making it easier for analysts to grasp the attack progression swiftly and effectively. These condensed AEGs not only streamline the analysis of the provenance graph but also provide a powerful tool for analysts to rapidly correlate different stages of APT attacks. This correlation offers a clear view of the overall attack strategy and tactics, helping incident response.

Algorithm 2 explains the procedure for creating AEGs. The algorithm takes a provenance graph $G(V, E)$ and a set of alerts N with hop length h. It then returns a list of AEGs.

5. Evaluation

To thoroughly evaluate FLASH, we address the following research questions. All experiments were performed on a machine equipped with 8 Intel vCPUs, 80 GB RAM, an NVIDIA RTX2080 GPU, and Ubuntu 18.04.6 LTS. For our experiments, we use an event batch size of 250k and Jaccard Similarity threshold of 1.

TABLE 2: Comparison of FLASH against ThreaTrace using only the GNN as the anomaly detector and using a GNN embeddings database along with a lightweight classifier. Prec.: Precision; Rec.: Recall;

Datasets	ThreaTrace				FLASH (GNN)				FLASH (GNN + XGBoost)			
	Prec.	Rec.	F-Score	TP/ FP/ FN/ TN	Prec.	Rec.	F-Score	TP/ FP/ FN/ TN	Prec.	Rec.	F-Score	TP/ FP/ FN/ TN
Cadets (E3)	0.90	0.99	0.95	12848/ 1361/ 4/ 705,605	0.94	0.99	0.96	12851/ 818/ 1/ 706,148	0.95	0.99	0.97	12851/ 720/ 1/ 706,246
Trace (E3)	0.72	0.99	0.83	67382/ 26774/ 1/ 2,389,233	0.95	0.99	0.97	67382/ 3477/ 1/ 2,412,530	0.95	0.99	0.97	67382/ 3805/ 1/ 2,412,202
Theia (E3)	0.87	0.99	0.93	25297/ 3765/ 65/ 3,501,561	0.92	0.99	0.95	25318/ 2282/ 44/ 3,503,044	0.93	0.99	0.96	25318/ 1875/ 44/ 3,503,451
Fivedirections (E3)	0.67	0.92	0.78	389/ 188/ 36/ 569,660	0.72	0.93	0.81	395/ 150/ 30/ 569,698	0.70	0.93	0.80	395/ 170/ 30/ 569,678
OpTC (Attack 1)	0.84	0.85	0.84	53/ 10/ 9/ 552,491	0.91	0.94	0.92	58/ 6/ 4/ 552,495	0.90	0.92	0.91	57/ 6/ 5/ 552,495
OpTC (Attack 2)	0.85	0.87	0.86	358/ 64/ 52/ 553,066	0.92	0.94	0.93	387/ 32/ 23/ 553,098	0.94	0.92	0.93	378/ 22/ 32/ 553,108
OpTC (Attack 3)	0.86	0.87	0.86	155/ 25/ 23/ 181,699	0.92	0.92	0.92	163/ 15/ 15/ 181,709	0.92	0.93	0.92	165/ 15/ 13/ 181,709

- **RQ1.** How does FLASH detection accuracy compare to the existing systems?
- **RQ2.** How does FLASH's GNN optimizations enhances the performance?
- **RQ3.** How does the batch size parameter affect FLASH's performance, accuracy, and resource usage?
- **RQ4.** How robust is FLASH against mimicry attacks?
- **RQ5.** What are the results of the ablation study on various FLASH components and hyperparameters?
- **RQ6.** How effectively does FLASH assist in the alert validation process?

Implementation. Our FLASH tool is built using Python, with around 5K lines of code. We employ the Torch Geometric library for our GNN model. This model uses Sage Convolutions, a type of graph convolution layer. It combines neighboring nodes' features to create new node representations. The model also includes dropout and ReLU activation functions between the layers to improve generalization. We use the XGBoost library for our XGBoost classifier and the Gensim library for our Word2Vec model. We have written Python functions to build the provenance graphs, filter unnecessary information, and abstract node entities. This helps us preprocess the raw graph and makes it suitable for our GNN model.

Datasets. FLASH was evaluated using four open-source datasets: DAPRA E3 [3], DAPRA OpTC [5], StreamSpot [51], and Unicorn [30]. The E3 dataset[1] is from DAPRA's third red-vs-blue team exercise. It provides audit log data that show both benign and attack behaviors. We label the OpTC attack events using ground truth document from DAPRA. For E3, we utilize labels from ThreaTrace. For identifying attack graphs in StreamSpot and Unicorn datasets, we refer to their respective documentation. The DAPRA OpTC dataset presents a wide view with benign and malicious audit records from about 1000 hosts. This dataset has six days of benign activities for training, and the next three days detail malicious activities for evaluation. Each of these three days shows a different type of attack. Day one maps a PowerShell Empire staging scenario, including initial foothold, lateral movements, and privilege escalations. Day two documents data exfiltration events. The third day records malicious software upgrades.

The StreamSpot dataset includes information flow graphs from one attack and five benign scenarios. Likewise, the Unicorn dataset contains simulated APT supply-chain attacks with a mix of benign and attack system graphs. Both datasets mainly serve graph-level granularity for anomaly detection and they do not provide fine-grained data for individual node classification as benign or attack. Hence, when using these datasets, FLASH's evaluation is confined to graph-level anomaly detection. FLASH achieves this by counting the number of anomalous nodes in a graph. If the count exceeds a predetermined threshold, the graph is classified as anomalous.

Detectors for Comparison. To evaluate FLASH, we use two provenance graph-based IDSes as benchmarks: ThreaTrace [67] and Unicorn [30]. We use the same detection metrics as the respective system that we compare against. We use node-level anomaly detection for ThreaTrace comparison, while for Unicorn, we conduct graph-level anomaly detection because Unicorn is incapable of performing fine-grained anomalous node detection. FLASH identifies a graph as anomalous if the number of anomalous nodes in it surpasses a specific threshold.

To train the FLASH for node-level detection, we used audit logs from benign system activities, partitioning them into two sets: initial training and fine-tuning. After training on the initial subset, we assessed its performance on a benign validation set, with a focus on minimizing false alarms. We fine-tuned the model and detection threshold based on this evaluation. For graph-level configurations, we trained the model and assessed it on benign validation graphs, determining the threshold from the number of alerts generated during this evaluation phase. To combat overfitting, we included regularization and dropout layers during model training.

While we recognize the significance of comparisons for benchmarking, it is not feasible for us to include ShadeWatcher [71] and ProGrapher [69] in our evaluations. This is primarily due to their closed-source nature. Moreover, we could not replicate these systems accurately based on paper descriptions alone. A key component of ShadeWatcher is proprietary, confirmed through our communication with the authors. Moreover, ShadeWatcher is tailored for offline detection, taking hours to train and detect, making it unsuitable

1. Several existing IDSes have only utilized partial datasets from E3. For example, ShadeWatcher only uses the Trace dataset. This approach can lead to a sampling bias pitfall, as highlighted by Arp et al. [18].

for enterprise settings. As for ProGrapher, its limitations in enterprise environments are already summarized in Table 1. The ThreaTrace study [67] provides a comparison with ProvDetector [66] and StreamSpot [51], and exhibits superior results in detection rates and efficiency. Since FLASH outperforms ThreaTrace in our evaluations, it suggests a higher efficacy over ProvDetector and StreamSpot.

RQ1. Detection Performance

We conducted two experiments to assess FLASH's detection performance. In the first experiment detailed in this **RQ**, we used the GNN as the anomaly detector. The second experiment, detailed in **RQ2**, employed a pre-stored GNN embeddings database coupled with a lightweight downstream classifier.

Table 2 presents the performance of FLASH and ThreaTrace [67] on our evaluation datasets. FLASH consistently surpasses ThreaTrace, achieving superior precision and F-score values. In comparison to ThreaTrace, FLASH generates semantically rich vectors utilizing word2vec and system attributes. This provides valuable context for the model to establish a distinct decision boundary between normal and anomalous nodes, subsequently reducing false positives and enhancing precision and F-score. As the ThreaTrace paper lacks evaluation results for the OpTC dataset, we executed ThreaTrace on OpTC to obtain evaluation results. The findings demonstrate that FLASH significantly outperforms ThreaTrace, as OpTC attacks are more challenging to detect due to smaller, well-blended malicious activity. ThreaTrace relies solely on syscall distribution patterns as features, which is insufficient. On the other hand, the Word2Vec encoded features employed by FLASH offer more semantic information for each node, making it harder for attack nodes to conceal themselves.

At first glance, on some datasets, such as Cadets, FLASH shows incremental improvement in comparison to ThreaTrace. This is attributed to partially available semantic attributes in these datasets, which poses a challenge in effectively learning the distribution of nodes. However, we provide evidence of FLASH's effectiveness in reducing false alarms on those datasets. For instance, on the Cadets dataset, FLASH achieved a reduction of 641 false alarms, resulting in a 47% decrease in analyst workload for alert validation. Additionally, with the OpTC dataset's richness in semantic attributes, FLASH consistently outperforms ThreaTrace by a significant margin. Section 6 discusses how the real-world audit logs are semantically rich (similar to OpTC), which allows FLASH to perform better detection in real-world settings compared to ThreaTrace.

We also compare FLASH to the Unicorn system [30]. While FLASH functions as a node-level anomaly detector, Unicorn operates at a graph-level. To ensure a fair comparison between these systems, we modify FLASH to perform graph-level detection as well. Unlike ThreaTrace, Unicorn's functionality is governed by an extensive range of hyperparameters, making evaluation of other DARPA datasets a complex task due to the exhaustive tuning required. Hence, to maintain fairness, we resort to the datasets originally used by the authors of Unicorn. Table 3 presents the experimental results of the detection comparison between the two systems. While the performance of Unicorn matches that of FLASH on certain datasets like Cadets, which feature less stealthy graphs, FLASH outperforms Unicorn on more stealthy attack datasets such as Unicorn SC-1 and SC-2. This advantage is attributed to FLASH's use of Graph Representation Learning, along with temporally sensitive node embeddings, which enables it to develop a strong understanding of baseline benign behavior. As a result, FLASH can effectively flag even the most stealthy attack graphs.

TABLE 3: Comparison of FLASH and Unicorn detector.

Datasets	System	Precision	Recall	F-score
StreamSpot	Unicorn	0.95	0.97	0.96
	FLASH	1.0	0.96	0.98
Unicorn SC-1	Unicorn	0.85	0.96	0.90
	FLASH	0.92	0.96	0.94
Unicorn SC-2	Unicorn	0.75	0.80	0.78
	FLASH	0.96	0.96	0.96
Theia (E3)	Unicorn	1.0	1.0	1.0
	FLASH	1.0	1.0	1.0
Cadets (E3)	Unicorn	0.98	1.0	0.99
	FLASH	1.0	1.0	1.0

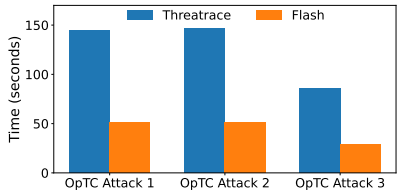


Figure 3: Inference times using one host logs from OpTC dataset. FLASH leverages embedding database to accelerate inference.

RQ2. Scalability Analysis of FLASH

Building on the evaluation from **RQ1**, we assess the scalability enhancements provided by utilizing the GNN embedding database in FLASH. These pre-stored embeddings are coupled with Word2Vec semantic encodings to serve the XGBoost classifier. When examining the E3 dataset, we identified a significant proportion of nodes lacking attribute data. For instance, among the four datasets within DARPA E3 – Cadets, Trace, FiveDirections, and Theia – there are 70%, 54%, 72%, and 76% of nodes respectively with missing attribute data. As detailed in Section 4.4, this data is vital for node and neighbor identification, as well as for assigning Persistent Node Identifier (PNI). Without these PNIs, our efficient embedding database cannot be employed. However, the OpTC dataset, providing complete node information, enables FLASH to implement the two-step pipeline. Therefore, during the inference (detection) phase, we exclusively use the embedding database for OpTC. In contrast, we generate GNN embeddings for E3 during inference.

In the training phase, FLASH's GNN model generates embedding vectors from benign data and stores these in a key-value data store for future use. During runtime, these stored embeddings are combined with Word2Vec vectors to

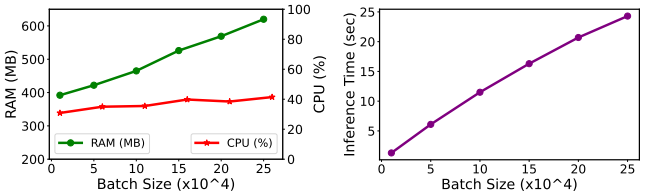
act as feature vectors for the XGBoost model, which then carries out anomaly detection. The detection results, outlined in Table 2 under the "FLASH (GNN + XGBoost)" column, indicate that the detection performance of FLASH utilizing this two-step pipeline is on par with using only GNN for anomaly detection, and considerably surpasses ThreaTrace. Furthermore, this two-step pipeline greatly accelerates processing speed compared to ThreaTrace. As depicted in Figure 3, the reuse of GNN embeddings leads to up to three times less time overhead than ThreaTrace. The reported inference time uses logs from a single host's one-day worth of system logs within the OpTC dataset. We observed a similar reduction in inference time overhead when we used logs from more hosts in our experiments. Thus, our approach can save hours of inference time in large enterprises and help prevent log congestion, ensuring timely alerts.

In our experimentation, we observed that FLASH successfully assigns pre-stored embeddings to approximately 84% of nodes on average during runtime, with the hit rate reaching up to 90%. This high rate of assignment show that many system entities exhibit repetitive behavior patterns, also shown by several studies [33, 34, 47], enabling FLASH to leverage its two-step pipeline for enhanced scalability and real-time APT detection.

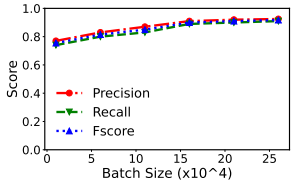
RQ3. Role of Batch Size

FLASH runtime performance is dependent on the number of logs it processes at a given time, a behavior regulated by the batch size parameter K. We conduct a comprehensive analysis to show the performance of our system under various batch sizes. We employ a sliding event window to process system logs as they are received by FLASH.

Figures 4a demonstrate the effect of increasing batch size on the runtime consumption of CPU and memory resources. Interestingly, CPU utilization remains relatively constant, while memory consumption exhibits an almost linear growth with respect to the batch size, reaching a peak of approximately 600 MB for an event window size of 250K. These observations suggest that FLASH is adept at utilizing system resources efficiently, even when processing large batches, thereby rendering it suitable for real-time IDSes. Figure 4b portrays the inference time for different batch sizes using the OpTC dataset. The event window size selection also exerts an impact on the detection performance of FLASH. Smaller window sizes hinder the complete capture of system node activities, thereby resulting in diminished detection accuracy. Figure 4c displays the improvement in various accuracy measures of FLASH as the batch size increases. Notably, the performance experiences an enhancement until a specific threshold, after which it stabilizes. From these findings, it is evident that the optimal choice of batch size relies upon the resource constraints and requirements of the enterprise environment where the FLASH system will be deployed. For systems with resource limitations, a smaller event window size may be necessary, while high-performance systems with substantial event throughput could benefit from a larger window size.



(a) RAM and CPU utilization. **(b)** Inference time.



(c) Detection performance.

Figure 4: Influence of batch size parameter K on different performance metrics of FLASH

RQ4. Robustness against Mimicry Attacks

To assess FLASH's resilience against adversarial attacks, we performed adversarial mimicry attacks on provenance-based IDSes, as detailed by Goyal et al. [26]. The susceptibility of IDSes with graph-level granularity (as described in Section 2), such as Unicorn [30], StreamSpot [51], ProvDetector [66], and SIGL [31] to adversarial attacks is a notable finding in Goyal et al.'s work. This adversarial strategy manipulates distributional graph encoding to stage an evasion attack against provenance-based IDSes, with the objective of creating misleading similarities between the node neighborhood distributions in the attack graph and those in benign provenance graphs. The goal of this strategy is to make the nodes within the attack graph have similar embeddings to nodes involved in benign activities. In pursuit of this, we integrated benign node structures from the training dataset into the attack graph. It's critical to mention that FLASH uses GNN to encode the structural information of nodes within the provenance graph. This GNN collects feature vectors from a node's parents via a message-passing mechanism to create a D-dimensional vector representation for each node based on its k-hop neighborhood.

The experimental results, visualized in Figure 5, display the number of edges added to the attack graph using benign structures (x-axis) versus the anomaly scores (minimum, average, and maximum) for all attack nodes (y-axis) with respect to the number of benign edges added. Initially, adding benign node neighborhoods decreased the anomaly scores of attack nodes, but this reduction was insufficient to ensure a successful evasion, as depicted by the threshold line in Figure 5. Subsequently, as more benign edges were introduced, the anomaly scores of attack nodes increased. This trend can be attributed to the fact that incorporating additional benign nodes around an attack node enhances its embedding, imbuing it with more benign characteristics and thereby reducing the node's anomaly score. However, the

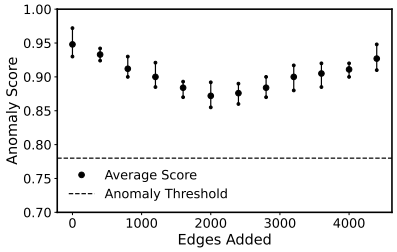


Figure 5: Adversarial mimicry attack against our system.

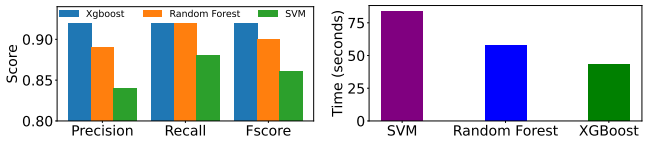
continual addition of more benign nodes raises suspicion within the model. This alteration in the node distribution, learned by the model, results in an increase in the anomaly score. This unique behavior can be linked to our model's robust semantic understanding, derived from the node-level learning task. Through this process, FLASH develops a comprehensive understanding of the characteristic neighborhood structures associated with nodes of varying semantics in the provenance graph. This understanding enables the model to accurately determine the legitimacy of causal relations between different nodes in the graph, making FLASH highly resistant to adversarial mimicry attacks. This resilience represents a significant advantage over graph-level detection approaches.

RQ5. Ablation Study

In our ablation study, we systematically assessed FLASH's performance by altering various parameter settings and components, showcasing their influence on the system's effectiveness. Key areas of our exploration include the impact of different lightweight classifiers, the use of weighted cross-entropy loss, the efficacy of GNN embeddings in contrast to Word2Vec embeddings, and the role of temporal ordering. Further aspects of our study, including hyperparameter analysis, embedding database scalability analysis, and attack case studies are discussed extensively in the Appendix (Appendices A, B, and C).

Varying Lightweight Classifiers. We tested the performance of different lightweight models, namely Random Forests and Support Vector Machines (SVM), against XGBoost using the OpTC dataset. Random Forests, ensemble learners, use multiple decision trees. SVMs are classifiers that find a hyperplane optimizing class margins. Figure 6a displays the average detection performance of these classifiers on three OpTC attacks. Figure 6b shows each model's time overhead. The SVM model was slower and less accurate than XGBoost and Random Forest. Although Random Forest performed similarly to XGBoost, XGBoost surpassed it in speed and detection accuracy. Other works [17, 62] have also shown the advantages of XGBoost over other classifiers.

Effect of Weighted Cross Entropy Loss. Weighted cross entropy addresses dataset class imbalance. We studied its impact on training GNN compared to vanilla cross entropy. Figure 7 shows the effects of these loss functions on model



(a) Detection comparison. **(b)** Time overhead comparison.

Figure 6: Detection and time comparison of different classifiers.

accuracy over epochs. With weighted cross entropy, the model achieved a 91% accuracy in just 50 epochs, while it took over 100 epochs to attain the same accuracy when using vanilla cross entropy. Consequently, weighted cross entropy facilitates the learning process of the model by ensuring equal focus on all node types.

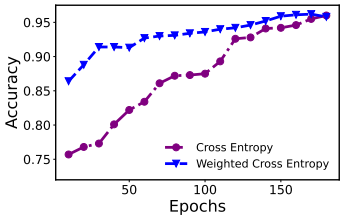


Figure 7: Effect of Weighted Cross Entropy on GNN Learning.

Efficacy of GNN Embeddings. In order to evaluate the efficacy of GNN structural embeddings, we compared FLASH's performance with Word2Vec embeddings, which were generated from 2-hop sentences employing the approach delineated in Section 4. As demonstrated in Figure 8, GNN surpasses Word2Vec embeddings, owing to its capacity to learn structural features and effectively discern pertinent patterns from extraneous noise. Conversely, Word2Vec combines all information in the vicinity, including the noise, resulting in inferior performance. This highlights the power of GNN in accurately capturing neighborhood information within the provenance graph.

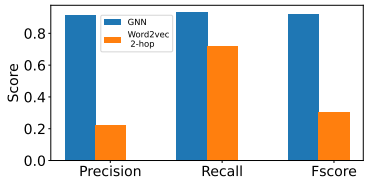


Figure 8: GNN vs. Word2Vec for capturing structural information.

Effect of Temporal Ordering. Our analysis explores the impact of temporal ordering on FLASH's performance. The featurization phase of FLASH has two main steps. First, it extracts semantic information from logs. Second, it uses the time sequence of system events, as stated in Section 4.2. We used the Trace dataset from E3 to study the effect of temporal ordering. In this experiment, we used temporally sorted system event sequences as features. We encoded these sequences into vector space with our temporal word2vec

technique. We compared our results to ThreaTrace, a method that does not consider temporal ordering. To focus on temporal ordering's effect, we excluded semantic attributes, which ThreaTrace also ignores. The results, presented in Table 4, showed a significant reduction in false alarms when considering temporal ordering. This improvement results from differentiating nodes that perform the same system events but in varying orders. Ignoring temporal ordering makes learning these nodes' distribution challenging. Incorporating it, however, allows the model to distinguish these nodes better, leading to improved detection performance.

TABLE 4: Effect of considering temporal ordering.

Temporal Order	Precision	Recall	F-Score	TP	FP
No	0.72	0.99	0.83	67382	26774
Yes	0.84	0.99	0.91	67382	12845

Effectiveness of Selective Graph Traversal. We conducted experiments to analyze how our graph traversal techniques affect the training time of GNN. We conducted tests using varying numbers of system events and trained both the GNN model with and without the selective traversal techniques for the same number of epochs. The results are illustrated in Figure 9, revealing that our traversal techniques lead up to a 43% reduction in the training time of GNN for the specified number of system events.

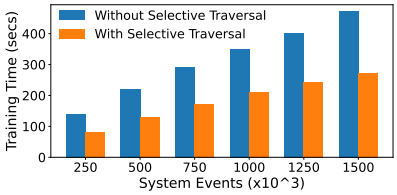


Figure 9: Effectiveness of FLASH's selective graph traversal.

RQ6. Accelerating Alert Validation

FLASH's primary design goal is threat detection, but its potential extends to facilitating faster attack investigation. We focus on two main areas: managing threat alert fatigue and generating Attack Evolution Graphs (AEGs). Alert fatigue arises when security analysts deal with excessive threat alerts, leading to overlooked threats amid false alarms. To assess if FLASH can mitigate this issue, we rank alerts based on the threat (anomaly) scores assigned by FLASH. Figure 10 presents a cumulative distribution function for ranked true and false alarms. It demonstrates that FLASH's threat scores prioritize true alerts. Furthermore, if a separation threshold is established at the score of the lowest true alert, up to 60% of false alarms can be discarded. This prioritization helps analysts focus on higher-scored alerts, lessening the chances of overlooking real threats and minimizing the attacker's exposure time. In terms of AEGs, these visual aids map an attack's evolution, assisting analysts during alert validation. As Figure 11 indicates, using AEGs reduces the number of items investigators need to review because

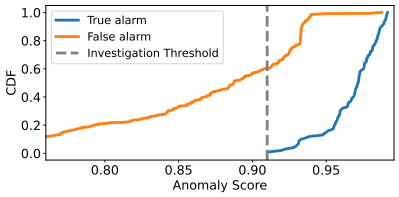
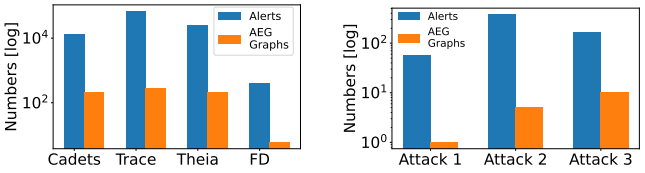


Figure 10: CDF of threat scores for false alarms and true alerts.



(a) DAPRA Engagement 3 **(b)** DAPRA OpTC

Figure 11: Number of AEGs generated from the threat alerts present in DAPRA E3 and OpTC.

there are fewer AEGs than individual alerts. This investigative approach, similar to the RapSheet system [32], allows for a more efficient, focused attack analysis. Conversely, ThreaTrace struggles with alert triaging due to the use of multiple GNN models for predictions. An alert is triggered only when all models classify a node as anomalous, resulting in each node having distinct alert scores from different models. The lack of a singular representative anomaly score for each node poses a significant challenge in the triaging process.

6. Discussion and Limitations

False positives. While FLASH displays a low false positive rate, certain scenarios might lead to false positives, especially in the presence of unobserved benign activities. This is a common challenge with anomaly-based detection systems. Regular model training and adjustment of the alert threshold parameter could alleviate this. Active learning [60], a technique that solicits analyst feedback on ambiguous classifications for model updates, could further reduce false positives. We propose a threat alert triaging method in Section 5 to alleviate the false positive issue.

Enriched system logs. FLASH relies on enriched system logs containing detailed information about system entities and their interactions. However, not all datasets in our evaluation (e.g., DAPRA E3) fully contained rich system logs to generate persistent IDs for GNN database. In such cases, we show that using only GNN-based classification, FLASH outperforms state-of-the-art provenance-based IDSes. Further, we found that almost all commercial IDSes, also known as Endpoint Detection and Response systems (EDRs), generate semantically rich system logs [6, 32, 47, 69], making our technique widely applicable.

Alert validation. While FLASH expedites alert validation, it's important to note that its primary objective is threat detection, with alert validation improvement being a secondary

benefit. Our approach comes with its inherent limitations. Alert ranking accuracy is closely tied to the precision of threat scores, while the utility of AEGs depends on the correct detection and interpretation of alert relationships - misclassifications can lead to misleading AEGs. Moreover, FLASH's performance may vary with different operational environments and datasets, requiring continuous fine-tuning. Future work could involve user studies with security analysts to enhance FLASH's effectiveness further.

Concept drift. Concept drift, where the data distribution of the underlying system evolves over time, is a potential issue. For instance, with the emergence of new system activities, the patterns learned by FLASH during training might not remain valid. This drift could lead to misclassifications, as new benign behaviors might be mistakenly identified as anomalies. One mitigation strategy for this involves periodic retraining with more recent data to update both the model and the embedding database. Due to its selective traversal, FLASH's training is efficient, enabling users to periodically retrain their models. However, this approach presents the challenge of preserving the model's ability to recognize older but still relevant attacks. Unfortunately, no public datasets currently exist to evaluate this strategy's efficacy. As such, the effective handling of concept drift within FLASH remains a challenge that warrants future research.

Adaptive adversarial attacks. We evaluated Flash's resilience against mimicry attacks in section 5 and will further discuss its vulnerabilities to other types of adversarial attacks, including gradient-based [22] and training-time poisoning attacks [40]. Gradient-based attacks typically require white-box access to the machine learning model and its parameters making them less feasible in real-world settings. In contrast, black-box attacks often use iterative, query-based methods, making these attacks less stealthy and more complex to execute. Training-time poisoning attacks compromise the model by introducing malicious data into the training set.

Various existing defenses can be integrated into Flash to improve its robustness against these attacks. For example, adversarial training [63] and gradient masking [50] are commonly used to defend against gradient-based attacks. The former involves augmenting the training dataset with adversarial examples to improve model robustness, while the latter makes it difficult for attackers to accurately compute gradients for crafting adversarial inputs. Techniques from previous work on defending against audit log tampering [14, 57] can be adapted to prevent training-time data poisoning attacks. These defenses can be readily incorporated into Flash to enhance its resistance to adaptive adversaries.

7. Related work

In Section 2.1, we described the challenges with existing provenance-based IDSes that FLASH addresses, and complement the discussion on related work here.

Specification-based IDS. Specification-based IDS methods such as Holmes [54], Rapsheet [32], and Poirot [55] detect deviations from normal behaviors using specification rules on provenance graphs. Although these approaches effectively reduce false positives, they risk being circumvented by new attacks and require continuous updates to address the evolving threat landscape. In contrast to these systems, FLASH is a more scalable and robust solution, capable of detecting previously unseen threats by leveraging rich semantic and contextual information from provenance graphs.

Embedding-based IDS. Embedding techniques are widely used for log analysis tasks, such as IDS [15, 16, 27, 52] and malware identification [21, 39, 74]. They often employ ML models, such as neural networks and n-grams to transform logs into vector forms. Examples include DeepLog using LSTM [23], ProvDetector applying Doc2Vec [45, 66], and Attack2Vec leveraging a temporal word-embedding model [61]. Euler [41] uses GNN and RNN embeddings to detect lateral movements. SIGL [31] focuses solely on detecting malicious software installations via deep learning and suffers from the limitations of graph-level granularity detection. Moreover, SIGL's evaluation is based on a small dataset of normal/malicious software installations, making it challenging to scale to large provenance graphs. DeepAid [29] uses deep neural networks to detect anomalies in network traffic. Different from these systems, FLASH is a host-based IDS blends GNN with rich semantic word embeddings from system logs and stores training-phase embeddings for real-time APT detection.

Provenance-based Investigation. Hassan et al. [34] utilized grammatical inference over provenance graphs to expedite system intrusion investigations. Pasquier et al. [58] introduced CamQuery, a real-time provenance analysis framework. Furthermore, existing systems,such as DEPCOMM [68], DEPIMPACT [24], Watson [70], NoDoze [33], Palantir [72], Deepcase [64], SAQL [25], OmegaLog [35], C2SR [43], Dossier [38], and Atlas [16], facilitate attack investigations and incident response. These existing systems are orthogonal to our research direction as they are not designed to detect APT attacks. They require initial attack symptoms from intrusion detectors to initiate investigations.

8. Conclusion

In this paper, we present FLASH, a novel host-based intrusion detection system leveraging provenance graph representation learning to address challenges in accuracy, practicality, and scalability. FLASH employs semantic and contextual encoders to capture essential graph attributes, and an embedding recycling database for real-time threat detection. Our extensive evaluations confirm FLASH's superior detection accuracy, resilience against mimicry attacks, and potential to expedite alert verification.

9. Acknowledgment

We extend our appreciation to the anonymous shepherd and reviewers for providing valuable comments on this work. We also thank Flavien Paul Moise for helping with system testing.

References

[1] "Alert Fatigue: 31.9% of IT Security Professionals Ignore Alerts," https://www.skyhighnetworks.com/cloud-security-blog/alert-fatigue-31-9-of-it-security-professionals-ignore-alerts/.

[2] "Bag of Words," https://en.wikipedia.org/wiki/Bag-of-words_model.

[3] "Darpa Engagement 3," https://github.com/darpa-i2o/Transparent-Computing/blob/master/README-E3.md.

[4] "DARPA OPTC," https://github.com/FiveDirections/OpTC-data.

[5] "DARPA TC," https://github.com/darpa-i2o/Transparent-Computing.

[6] "Endpoint Security Framework," https://developer.apple.com/documentation/endpointsecurity.

[7] "Event tracing," https://docs.microsoft.com/en-us/windows/desktop/ETW/event-tracing-portal.

[8] "The Linux audit daemon," https://linux.die.net/man/8/auditd.

[9] "Local Outlier Factor," https://towardsdatascience.com/local-outlier-factor-lof-algorithm-for-outlier-identification-8efb887d9843.

[10] "NotPetya Attack," https://en.wikipedia.org/wiki/Petya_and_NotPetya.

[11] "One Hot Encoding," https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/.

[12] "Recent Advances in Efficient and Scalable Graph Neural Networks," https://www.chaitjo.com/post/efficient-gnns/.

[13] "The SolarWinds Cyber-Attack: What You Need to Know," https://www.cisecurity.org/solarwinds/.

[14] A. Ahmad, S. Lee, and M. Peinado, "Hardlog: Practical tamper-proof system auditing using a novel audit device," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022.

[15] S. Aljawarneh, M. Aldwairi, and M. B. Yassein, "Anomaly-based intrusion detection system through feature selection analysis and building hybrid efficient model," *Journal of Computational Science*, vol. 25, 2018.

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

[17] A. Anghel, N. Papandreou, T. Parnell, A. De Palma, and H. Pozidis, "Benchmarking and optimization of gradient boosting decision tree algorithms," *arXiv preprint arXiv:1809.04559*, 2018.

[18] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck, "Dos and don'ts of machine learning in computer security," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[19] A. Bates, W. U. Hassan, K. Butler, A. Dobra, B. Reaves, P. Cable, T. Moyer, and N. Schear, "Transparent web service auditing via network provenance functions," in *International World Wide Web Conference (WWW)*, 2017.

[20] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel," in *USENIX Security Symposium*, 2015.

[21] S. S. Chakkaravarthy, D. Sangeetha, and V. Vaidehi, "A survey on malware analysis and mitigation techniques," *Computer Science Review*, vol. 32, 2019.

[22] A. Chakraborty, M. Alam, V. Dey, A. Chattopadhyay, and D. Mukhopadhyay, "A survey on adversarial attacks and defences," *CAAI Transactions on Intelligence Technology*, vol. 6, no. 1, pp. 25–45, 2021.

[23] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," in *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[24] P. Fang, P. Gao, C. Liu, E. Ayday, K. Jee, T. Wang, Y. F. Ye, Z. Liu, and X. Xiao, "Back-propagating system dependency impact for attack investigation," in *USENIX Security Symposium*, 2022.

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

[26] A. Goyal, X. Han, G. Wang, and A. Bates, "Sometimes, you aren't what you do: Mimicry attacks against provenance graph host intrusion detection systems," in *Network and distributed system security symposium*, 2023.

[27] M. Gyanchandani, J. Rana, and R. Yadav, "Taxonomy of anomaly based intrusion detection system: a review," *International Journal of Scientific and Research Publications*, vol. 2, 2012.

[28] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[29] D. Han, Z. Wang, W. Chen, Y. Zhong, S. Wang, H. Zhang, J. Yang, X. Shi, and X. Yin, "Deepaid: Interpreting and improving deep learning-based anomaly detection in security applications," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[30] X. Han, T. Pasqueir, A. Bates, J. Mickens, and M. Seltzer, "Unicorn: Runtime provenance-based detector for advanced persistent threats," in *Network and Distributed System Security (NDSS)*, 2020.

[31] X. Han, X. Yu, T. Pasquier, D. Li, J. Rhee, J. Mickens, M. Seltzer, and H. Chen, "Sigl: Securing software installations through deep graph learning," in *USENIX Security Symposium*, 2021.

[32] W. U. Hassan, A. Bates, and D. Marino, "Tactical provenance analysis for endpoint detection and response systems," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2020.

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

[34] W. U. Hassan, M. Lemay, N. Aguse, A. Bates, and T. Moyer, "Towards scalable cluster auditing through grammatical inference over provenance graphs," in *Network and Distributed System Security (NDSS)*, 2018.

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

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

[37] M. N. Hossain, J. Wang, R. Sekar, and S. D. Stoller, "Dependence-preserving data compaction for scalable forensic analysis," in *USENIX Security Symposium*, 2018.

[38] M. A. Inam, W. U. Hassan, A. Ahad, A. Bates, R. Tahir, T. Xu, and F. Zaffar, "Forensic analysis of configuration-based attacks," in *NDSS*, 2022.

[39] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for android malware detection," in *2011 seventh international conference on computational intelligence and security*. IEEE, 2011.

[40] M. Jagielski, A. Oprea, B. Biggio, C. Liu, C. Nita-Rotaru, and B. Li, "Manipulating machine learning: Poisoning attacks and countermeasures for regression learning," in *2018 IEEE symposium on security and privacy (SP)*. IEEE, 2018, pp. 19–35.

[41] I. J. King and H. H. Huang, "Euler: Detecting network lateral movement via scalable temporal link prediction," in *Network and Distributed System Security Symposium*, 2022.

[42] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W.-C. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. Ciocarlie *et al.*, "MCI: Modeling-based causality inference in audit logging for attack investigation," in *Network and Distributed System Security (NDSS)*, 2018.

[43] Y. Kwon, W. Wang, J. Jung, K. H. Lee, and R. Perdisci, "C 2 sr: Cybercrime scene reconstruction for post-mortem forensic analysis," in *NDSS*, 2021.

[44] S. Lai, L. Xu, K. Liu, and J. Zhao, "Recurrent convolutional neural networks for text classification," in *Proceedings of the AAAI confer-*

ence on artificial intelligence, vol. 29, 2015.

[45] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*. PMLR, 2014.

[46] K. H. Lee, X. Zhang, and D. Xu, "LogGC: Garbage collecting audit log," in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

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

[48] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "MPI: Multiple perspective attack investigation with semantic aware execution partitioning," in *USENIX Security Symposium*, 2017.

[49] S. Ma, X. Zhang, and D. Xu, "ProTracer: Towards practical provenance tracing by alternating between logging and tainting," in *Network and Distributed System Security (NDSS)*, 2016.

[50] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *International Conference on Learning Representations*, 2018.

[51] E. Manzoor, S. M. Milajerdi, and L. Akoglu, "Fast memory-efficient anomaly detection in streaming heterogeneous graphs," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[52] Z. K. Maseer, R. Yusof, N. Bahaman, S. A. Mostafa, and C. F. M. Foozy, "Benchmarking of machine learning for anomaly based intrusion detection systems in the cicids2017 dataset," *IEEE access*, vol. 9, 2021.

[53] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2013.

[54] S. M. Milajerdi, R. Gjomemo, B. 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 (S&P)*, 2019.

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

[56] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "graph2vec: Learning distributed representations of graphs," *arXiv preprint arXiv:1707.05005*, 2017.

[57] R. Paccagnella, P. Datta, W. U. Hassan, A. Bates, C. Fletcher, A. Miller, and D. Tian, "Custos: Practical tamper-evident auditing of operating systems using trusted execution," in *Network and distributed system security symposium*, 2020.

[58] T. Pasquier, X. Han, T. Moyer, A. Bates, O. Hermant, D. Eyers, J. Bacon, and M. Seltzer, "Runtime analysis of whole-system provenance," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[59] J. Saad-Falcon, A. Singh, L. Soldaini, M. D'Arcy, A. Cohan, and D. Downey, "Embedding recycling for language models," *arXiv preprint arXiv:2207.04993*, 2022.

[60] B. Settles, "Active learning literature survey," 2009.

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

[62] S. Si, H. Zhang, S. S. Keerthi, D. Mahajan, I. S. Dhillon, and C.-J. Hsieh, "Gradient boosted decision trees for high dimensional sparse output," in *International conference on machine learning*.

[63] F. Tramer and D. Boneh, "Adversarial training and robustness for multiple perturbations," *Advances in neural information processing systems*, vol. 32, 2019.

[64] T. van Ede, H. Aghakhani, N. Spahn, R. Bortolameotti, M. Cova, A. Continella, M. van Steen, A. Peter, C. Kruegel, and G. Vigna, "Deepcase: Semi-supervised contextual analysis of security events," in *IEEE Symposium on Security and Privacy (SP)*, 2022.

[65] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

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

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

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

[69] F. Yang, J. Xu, C. Xiong, Z. Li, and K. Zhang, "Prographer: An anomaly detection system based on provenance graph embedding," 2023.

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

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

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

[73] Y. Zhou, J. Leng, Y. Song, S. Lu, M. Wang, C. Li, M. Guo, W. Shen, Y. Li, W. Lin *et al.*, "ugrapher: High-performance graph operator computation via unified abstraction for graph neural networks," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023.

[74] M. F. Zolkipli and A. Jantan, "An approach for malware behavior identification and classification," in *2011 3rd international conference on computer research and development*, vol. 1. IEEE, 2011.

Appendix A. Impact of Hyperparameters

FLASH is influenced by four primary hyperparameters: similarity threshold, events batch size, anomaly threshold, and graph sampling rate. The impact of batch size has been examined in a previous subsection, while the characteristics and investigation of the other parameters are detailed below.

Jaccard Similarity Threshold. The hyperparameter gauges the match between stored and real-time node neighborhoods, guiding the reuse of GNN embeddings. A low threshold increases node coverage but may misrepresent neighborhoods, while a 100% threshold guarantees context accuracy. Figure 12 shows that a higher threshold raises the AUC score, reducing mislabeling of malicious nodes and enhancing anomaly detection.

Anomaly Threshold. This hyperparameter controls the number of anomalies generated by the detector, determining the trade-off between false positives and false negatives. The ROC-AUC curve is utilized to evaluate the influence of adjusting the threshold on detection performance. Figure 13 depicts the ROC curve for the OpTC dataset.

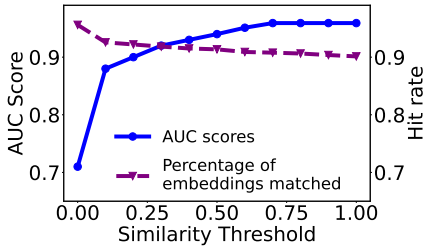


Figure 12: Jaccard similarity

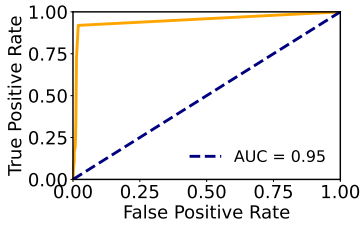


Figure 13: Classification threshold.

Host Samples. An ML system generally improves with more data, yet fewer hosts can suffice if benign patterns are consistent, as our OpTC dataset tests show. Training FLASH with data from just 10 hosts yielded high accuracy, and more hosts didn't enhance it (see Figure 14). This suggests that judicious data use can conserve resources without sacrificing accuracy.

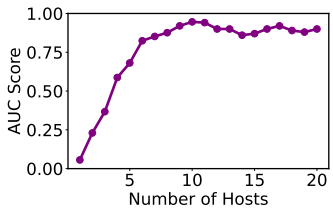
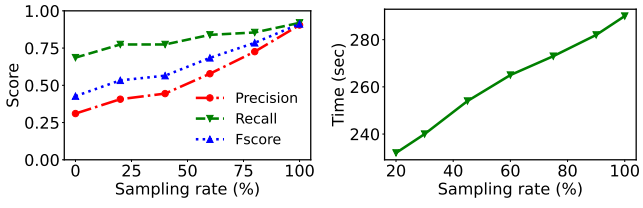


Figure 14: Host samples for training.

Graph Sampling Rate. In Graph Neural Network, the sampling rate affects neighborhood size for node features, with higher rates offering richer structural insights for GNN but increasing complexity and processing time. Our research reveals that a greater sampling rate boosts FLASH's precision due to enhanced structural embeddings, though it heightens time overhead. Figure 15a and 15b shows the effect of sampling rate on FLASH.



(a) Sampling vs. Detection **(b)** Sampling vs. Time

Figure 15: Graph sampling rate impact.

Appendix B. Growth of Embedding Database.

We conducted experiments to study the growth of embedding database over 6 days of benign logs present in the OpTC dataset. Fig 16 shows the results. Initially, we observed a steady growth rate of 3%, which gradually decreased to 1% and eventually stabilized. This intriguing behavior can be attributed to the innovative abstraction techniques we employed. These techniques effectively prevent redundant storage of identical nodes under different execution runs, leading to optimized database management.

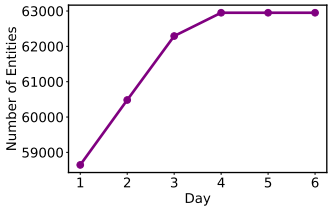


Figure 16: Growth of Embedding Database Overtime.

Appendix C. Additional Attack Case Studies

The first and third OpTC attacks are describe here while the second attack is already explained in Section 2.

OpTC Attack 1

Figure 17 illustrates the provenance graph for attack 1. The aim of this attack was to setup the attacker's foothold into the system and identify other vulnerable hosts in the network. The attack commences with the red team downloading a malicious PowerShell Empire stager stored in a batch file named `runme.bat`. This file was downloaded via `Firefox.exe`. Once downloaded, the attacker leveraged this malicious binary to perform network scanning to identify vulnerable hosts on the network. They then used `Windows Management Instrumentation` for lateral movement. FLASH was able to identify all main components of this attack. It flagged the `Firefox.exe` process used for downloading the malicious binary `runme.bat`. It also identified the binary `runme.bat` and other nodes that it interacted with, such as `wmiprvse.exe`, `lsass.exe`, and `ping.exe`.

OpTC Attack 3

Figure 18 outlines attack 3, where the red team exploited software updates to create a backdoor. They installed a vulnerable `Notepad++.exe`, which fetched a backdoor binary, `update.exe`, upon update. This binary, containing `cKfGW.exe`, connected to the attackers via `Port:8080`, facilitating network scans and information gathering. FLASH flagged the compromised `Notepad++.exe`, the binary `update.exe`, and the attacker's IP `53.192.68.50`.

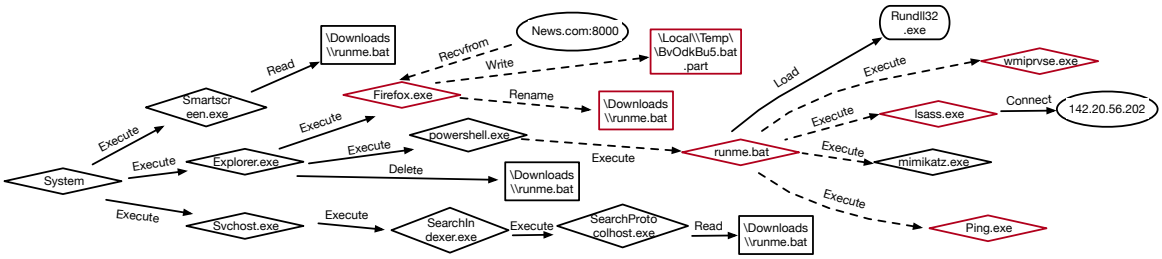


Figure 17: Attack 1 provenance graph from OpTC. This attack scenario involves PowerShell Empire staging, which includes establishing initial footholds, lateral movements, and privilege escalations The nodes in red are those detected by FLASH.

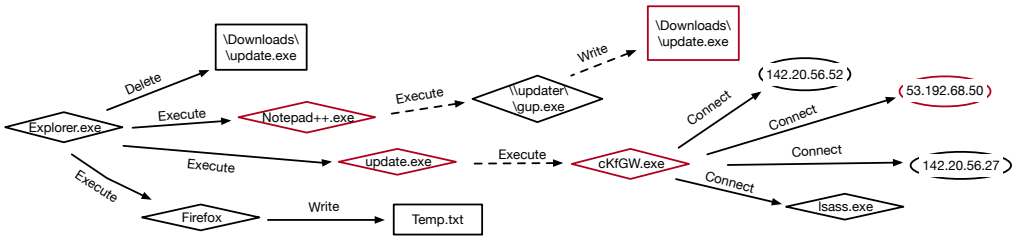


Figure 18: Attack 3 provenance graph from OpTC. The attack involves a series of malicious software updates to establish a backdoor connection to the command server.

Appendix D. Training Algorithm

Algorithm 3 shows the steps for training the GNN and XGBoost models of FLASH. The algorithm combines the Graph Neural Network and XGBoost to perform node classification on a given graph $G = (V, E)$. The algorithm starts by initializing the GNN and XGBoost models. Then, it uses our efficient graph traverser to train the GNN model using the semantic feature vectors generated using Word2Vec. Next, it generates Word2Vec and GNN embeddings for each node in the graph and concatenates them to create a joint feature vector for each node. The XGBoost model is then trained using these joint feature vectors. Additionally, the algorithm constructs a feature store containing GNN embeddings and neighbor information for each node in the graph. The trained GNN, XGBoost models, and the embedding store are returned as the output of the algorithm. This integrated approach leverages both GNN and traditional machine learning (XGBoost) to capture complex causal relationships in the given provenance graph.

Algorithm 3: TRAININGALGORITHM

Inputs : Graph $G = (V,E)$;
Output: GNN Model gnn;
XGBoost Model xgb;
Feature Store $featureMap$

$gnn \leftarrow$ INITIALIZEGRAPHSAGE ()
$xgb \leftarrow$ INITIALIZEXGBOOSTCLASSIFIER()
$traverser$ = GRAPHTRAVERSER(G)
/* Training the GNN model */
foreach *Subgraph* $(V_n,E_n) \in traverser$ **do**
  $X_n \leftarrow$ GETWORD2VECFEATURES(V_n,E_n)
  $gnn \leftarrow gnn.train(X_n,E_n)$
end
/* Training the XGBoost model. */
foreach $(V_n,E_n) \in G$ **do**
  /* Generating Word2Vec embeddings */
  $X \leftarrow$ GETWORD2VECFEATURES(V_n,E_n)
  /* Generating the GNN embeddings */
  $Y \leftarrow$ GETNODEGNNEMBEDDING(V_n,E_n)
  /* Concatenate the Word2Vec & GNN embeddings */
  $Z \leftarrow$ CONCAT(X,Y)
  $xgb \leftarrow xgb.train(Z)$
end
/* Initializing the GNN embedding database */
$featureMap \leftarrow$ INITIALIZEHASHTABLE()
foreach *Node* $N \in G$ **do**
  /* Key of the hashmap is node persistent ID and values are neighbor set & GNN embeddings */
  $Neighbors \leftarrow$ CONSTRUCTNEIGHBORSET(N)
  $ID \leftarrow$ GETPERSISTENTNODEID(N)
  $Emb \leftarrow$ GETNODEGNNEMBEDDING(N,gnn)
  $featureMap[ID] \leftarrow (Emb,Neighbors)$
end
return gnn, xgb, $featureMap$

Appendix E.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

E.1. Summary

This paper presents FLASH, an intrusion detection system (IDS) based on provenance graphs. The main contribution is to use an improved Word2Vec method to encode node attributes and temporal information and a graph neural network (GNN) to generate node embeddings. The system introduces a series of designs (including a pre-computed embedding database) to quickly generate embeddings during the testing time to improve scalability and run-time efficiency. The authors evaluated FLASH with multiple datasets and also tested the adversarial robustness.

E.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

E.3. Reasons for Acceptance

1) The paper provides a valuable step forward in an established field. It introduces a new solution to a known problem in existing provenance-based intrusion detection systems which is the detection efficiency (i.e., run-time scalability). A series of ideas are proposed and evaluated including selective graph traversal, recycling database, and embedding caching to speed up the detection efficiency.
2) The paper creates a new tool to enable future science. The proposed solution incorporates other designs such as node attribute encoding and temporal information encoding to improve the detection performance. The tools are extensively evaluated to justify the contribution of each design. The authors will open-source the developed tool to facilitate future research in this area.