PROVCREATOR: Synthesizing Complex Heterogeneous Graphs with Node and Edge Attributes

Tianhao Wang*, Simon Klancher*, Kunal Mukherjee*, Josh Wiedemeier*,
Feng Chen*, Murat Kantarcioglu†, and Kangkook Jee*
*The University of Texas at Dallas, Richardson, Texas, USA
{Tianhao.Wang, Simon.Klancher, Kunal.Mukherjee}@utdallas.edu
{Josh.Wiedemeier, Feng.Chen, Kangkook.Jee}@utdallas.edu
†Virginia Tech, Blacksburg, Virginia, USA
Murat.Kantarcioglu@vt.edu

***Abstract*—The rise of graph-structured data has driven interest in graph learning and synthetic data generation. While successful in text and image domains, synthetic graph generation remains challenging — especially for real-world graphs with complex, heterogeneous schemas. Existing research has focused mostly on homogeneous structures with simple attributes, limiting their usefulness and relevance for application domains requiring semantic fidelity.**

In this research, we introduce PROVCREATOR, a synthetic graph framework designed for complex heterogeneous graphs with high-dimensional node and edge attributes. PROVCREATOR formulates graph synthesis as a sequence generation task, enabling the use of transformer-based large language models. It features a versatile graph-to-sequence encoder-decoder that *(1)* losslessly encodes graph structure and attributes, *(2)* efficiently compresses large graphs for contextual modeling, and *(3)* supports end-to-end, learnable graph generation.

To validate our research, we evaluate PROVCREATOR on two challenging domains: *system provenance graphs in cybersecurity* and *knowledge graphs from IntelliGraph Benchmark Dataset*. In both cases, PROVCREATOR captures intricate dependencies between structure and semantics, enabling the generation of realistic and privacy-aware synthetic datasets.

***Index Terms*—graph generation, heterogeneous graphs, synthetic data, system provenance**

I. INTRODUCTION

Recent advancements in Machine Learning (ML) and Artificial Intelligence (AI) research have increasingly focused on solving problems set in graph-structured datasets. The rich expressivity and flexibility of graph-structured data has attracted substantial effort towards developing graph learning and processing techniques. Real-world problems are naturally expressible as graphs that capture relationships between entities: social graphs capture interpersonal relationships, knowledge graphs encode the relationships between concepts, causal graphs capture dependencies among events, *etc.* [1]

The complexity of graph schemas ranges all from simple classical graphs with pure nodes and undirected edges all the way to modern heterogeneous graphs where every node and edge contains arbitrary attributes of varying data types. The unique combination of practicality and elegant theoretical backing makes graph-based ML an attractive research area.

However, some graph datasets are restricted in scope or expensive to collect, which limits the data available to train advanced graph models. Following the success of supplementing training data with *synthetic data* in the text and image domains [2, 3], the generation of usable *synthetic graph data* has emerged as a prominent research topic alongside the rapid advancement of generative models [4–7]. This field aims to address long-standing challenges stemming from the scarcity of high-quality datasets and the limitations in their secure and safe sharing processes. Synthetic data generation offers a promising research avenue, with the potential to enhance various aspects of ML and AI by enabling more robust, generalizable, and up-to-date analyses.

The expressivity of graph data entails substantial complexity, with each problem domain encoding unique soft and hard semantic constraints on what graphs are plausible and valid. A recently developing line of research in *synthetically generating graph-structured datasets* has begun to chip away at this complexity by considering constrained classes of graphs. While many approaches have been proposed in the literature, most studies have focused on well-defined graphs with simple schema, typically involving nodes with only a single categorical or numerical attribute. These efforts have emphasized scalability, aiming to support very large graphs where the semantics of the graph are primarily determined by the existence of edges, rather than by detailed attributes.

Modern graph applications in many domains have embraced complex heterogeneous graph schemas, including dozens of numeric, categorical, *and textual* attributes for each node and edge. For example, system provenance graphs — widely used for cybersecurity applications — include nodes that represent computer processes, and include semantically critical attributes such as the command that launched the process, the executable name, the PID, file location, the process owner, *etc.* Further, provenance graphs include nodes for other system entities (files and network sockets) that have their own long lists of complex

attributes, whose plausibility is intrinsically tied to the graph structure.

Existing synthetic graph generation research that focuses on simple structure and basic categorical and numeric attribute sets fails to adequately serve the needs of these emerging applications that deeply rely on the expressivity of complex graph schemas.

In this research, we propose PROVCREATOR, a novel framework for generating synthetic graphs with a specific focus on supporting datasets with highly complex heterogeneous schemas. The central insight of PROVCREATOR is that the graph generation problem can be cast as a sequence generation problem, for which there are ample techniques available in the literature. PROVCREATOR is constructed by wrapping a sequence generation model in a graph-to-sequence encoder and sequence-to-graph decoder. For the inner sequence generation task, PROVCREATOR leverages recent advances in transformer-based Large Language Models (LLMs). Specifically, PROVCREATOR's graph-to-sequence encoder/decoder has three design goals: *(1)* lossless preservation and recovery of both the graph's structure and arbitrary serializable node and edge attributes; *(2)* compact encoding of large graphs to efficiently utilize the sequence generation model's context window; and *(3)* learnable single-pass generation of serialized graphs to maximize the effectiveness of the sequence generation model.

Leveraging the graph-to-sequence encoder, PROVCREATOR first translates training graphs into compact sequence representations, capturing both graph structure and complex node and edge attributes. Then, these serialized graphs are used to fine-tune transformer-based LLMs. Finally, PROVCREATOR generates synthetic graphs by using the fine-tuned LLM to generate serialized graph sequences which are expanded into full graphs by the sequence-to-graph decoder.

In summary, this work makes the following contributions:

- **A unified framework for synthesizing complex heterogeneous graphs.** PROVCREATOR introduces a generic framework to generate synthetic graph datasets with rich heterogeneity, including diverse node and edge types with high-dimensional attributes.
- **Joint modeling of structure and semantics.** PROVCREATOR captures intricate dependencies between graph topology and attribute information by interleaving structural relationships with node and edge attributes during both encoding and generation phases.
- **A versatile graph-to-sequence encoder.** At the heart of PROVCREATOR is a novel graph encoder that faithfully preserves structural and semantic information in a compact sequence format. This design enables seamless integration with transformer-based large language models and generalizes to a wide range of downstream tasks beyond synthetic graph generation.

To demonstrate the effectiveness and the general applicability of PROVCREATOR, we evaluate it with datasets from two distinct domains. Firstly, we apply PROVCREATOR approach to system provenance graphs from the cybersecurity domain, where the lack of up-to-date, high-quality public datasets has posed significant research limitations. Secondly, we evaluate PROVCREATOR on knowledge graphs derived from the IntelliGraph Benchmark Dataset [8], which feature highly heterogeneous schemas and free-form attribute annotations.

To benefit the research community, we plan to publicly release our synthetic graph generation framework, as well as the datasets generated in this study after applying privacy-enhancing techniques to ensure data confidentiality and privacy.

II. BACKGROUND

A. Synthetic Data Generation.

Synthetic data generation involves creating artificial data that closely resembles real data in its statistical properties and complexity without collecting additional real data. These approaches aim to address challenges presented by imbalanced datasets by supplementing underrepresented classes with synthetic examples. Recent advancements in data analysis techniques and data processing have led to numerous research proposals for data synthesis across various domains [9–15]. Existing synthetic graph generation methods primarily focus on single categorical attributes during graph structure generation (e.g., molecular types for nodes and bond types for edges). However, capturing more complex attributes requires additional mechanisms to model their interdependencies. Most prior work addresses this by generating vector representations of node features [16, 17]. We argue and empirically demonstrate that node attributes and graph structure should be generated jointly, as they are often statistically interdependent. To this end, PROVCREATOR formulates both as a unified optimization problem. Departing from traditional approaches in graph generation, we leverage recent advances in generative transformers for sequences, adapting them to the graph domain through carefully designed serialization and deserialization strategies. By generating graphs as constrained sequences, PROVCREATOR enables unprecedented flexibility in modeling complex, interdependent node and edge attributes alongside the graph structure.

B. System Provenance Graphs

System provenance tracks fine-grained system data (`syscall` events), from large enterprise and industrial systems. It traces bidirectional information flow and control dependencies starting from a Point-Of-Interest (POI) event, enabling forensic analysis and advanced security defenses [18–21]. By examining system-call logs [22–24], system provenance graphs capture relationships (*i.e.,* `READ`, `WRITE`, `CREATE` and `EXECUTE`) among major resources (*i.e.,* processes, files and network sockets). Nodes, representing system resources, are annotated with attributes such as executable names, filenames, and IP addresses, making these graphs invaluable tools for forensic analysis to discover points of entry, track infection propagation, and assess the scale of damage. Formally, a provenance graph is a connected set of timestamped edges $e = (u, v, r)$, where $u, v \in \{processes \cup files \cup sockets\}$ and v is causally dependent on u (*e.g.,* a file u is written by a process v), and r is the relationship between the nodes.

Although synthetic graph generation research has attracted significant attention across various research domains, security datasets, particularly system provenance graphs, have seen limited progress. Synthetic data is valuable in the system provenance domain because real `syscall` traces are costly to collect and difficult to share due to privacy concerns, but are required in large quantities by ML models. Creech and Hu [25] and Haider et al. [26] proposed initial prototype implementations for synthesizing attack behaviors by generating sequences of system calls, but these implementations lacked consideration of causal dependencies among system resources (*e.g.,* processes, files, and network sockets). The operating system (OS) used in Creech and Hu [25] is decade old Ubuntu 11.04 which has reached end of life support and it contains no Windows traces. Furthermore, Creech and Hu [25] rejected traces over 3 kB thus impacting the completeness of the traces collected. By directly generating provenance graphs, PROVCREATOR synthesizes data in an OS-agnostic way with structural consideration of causal dependencies, and can generate large graphs by combining related subgraphs. Notably, edges in provenance graphs are naturally ordered by the time at which `syscall` events occur, which synergizes with PROVCREATOR's sequential generation approach. While synthetic data generation has been explored in other security research domains [11], we are unaware of any previous research focusing on synthetic data generation approaches specifically for system provenance graph datasets.

C. Provenance-based Machine Learning Research

Initially proposed to automate forensic investigations, system provenance has become a vital foundation for ML-based security detectors [21, 27–32], propelled by rapid advances in data analysis techniques. Security researchers have expanded the scope of system provenance studies to construct ML models that counter attacks where adversaries craft unique, unexposed attack vectors [33] that can only be identified during runtime. While the full provenance graph even for one host machine can be intractably large, comprising millions of edges, analysis can be performed on tractable subgraphs extracted by tracing causal dependencies to and from a POI event. Depending on the objective of the analysis task, various query conditions can determine effective POI events, with the system provenance graph serving as the primary target for analysis.

III. GRAPH SYNTHESIS METHODOLOGY

PROVCREATOR focuses on two key aspects of graph generation: the generation of the graph structure and the prediction of node attributes within the generated graph. Previous research has primarily faced graph structure and attribute generation as separate problems, without attempting structural and attribute generation into a single generative process. Our work is the first to integrate both approaches targeting graphs with highly complex node and edge attributes in the cybersecurity domain. System provenance graphs provide a uniquely challenging scenario with heterogenous graph structure and varied node attributes such as file paths, process command line arguments, and IP addresses. These attributes are not only complex but also interdependent with the graph structure, making an ideal use case for a method that generates both jointly.

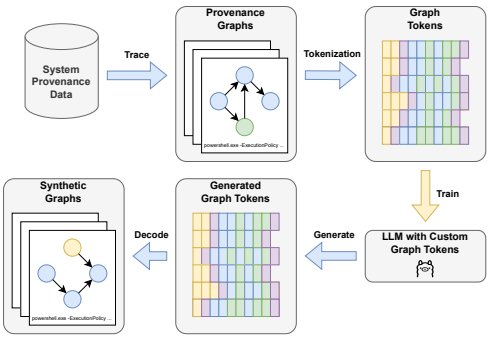


Fig. 1. PROVCREATOR's synthetic graph generation pipeline.

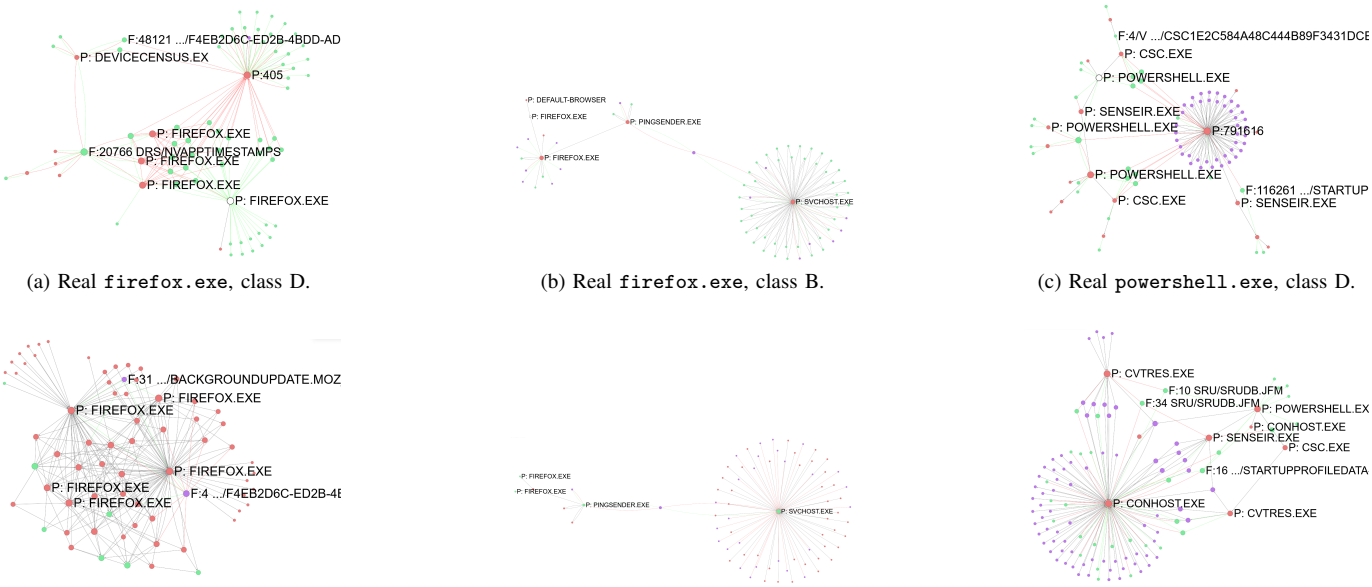
PROVCREATOR models graph generation as a sequence generation task, where each graph is represented as a sequence of tokens. This enables the use of an LLM as the backbone of our architecture, allowing us to leverage the power of pretrained models for increased performance and faster training time than training a model from scratch. We use LLama3-3.2-3B as the base model for PROVCREATOR. Additionally, we propose an efficient graph representation which addresses the challenges of generating long sequences of tokens while ensuring the validity of the generated graph structure. Utilizing the LLM backbone allows us to use prompts to condition the graph generation process, enabling the generation of graphs that are relevant to specific contexts, such as different command line arguments being passed to a process. This is particularly useful in cybersecurity applications where the context can significantly influence program behavior, thereby changing the structure and attributes of the generated graph.

The overall workflow of PROVCREATOR is shown in Figure 1. Figure 2 compares representative real and PROVCREATOR-generated provenance graphs.

A. Graph Representation

Efficiently representing a graph is crucial for the generation process. Previous works such as [6] and [34] consider representing graphs in plain-text formats. This encoding is inefficient due to the use of many tokens to represent information that is not critical to the graph structure, while also presenting challenges for reconstructing graphs from model outputs. As the graph grows large, the chance of generating syntactically invalid tokens increases, leading to difficulties in decoding the generated tokens back into a valid graph structure. This is particularly problematic for heterogeneous graphs, where different node and edge types may have distinct attributes and relationships, making it difficult to ensure that the generated tokens conform to the expected structure.

To reduce the overhead of text representation, we designed a list of special tokens for efficient graph representation, described in Table I. The graph is wrapped inside a pair of tokens <bog>, <eog> indicating the start and end of the



(a) Real `firefox.exe`, class D. (b) Real `firefox.exe`, class B. (c) Real `powershell.exe`, class D.

(d) Synthetic `firefox.exe`, class D. (e) Synthetic `firefox.exe`, class B. (f) Synthetic `powershell.exe`, class D.

Fig. 2. Real training graphs (top) and PROVCREATOR-generated graphs (bottom), aligned by program and class.

TABLE I
TOKENIZATION SCHEMA FOR GRAPH REPRESENTATION.

Token	Description
<bog>	Beginning of graph
<eog>	End of graph
<bon>	Beginning of node
<eon>	End of node
<boe>	Beginning of edge
<node$_i$>	Node i
<ntype$_j$>	Node type j
<etype$_k$>	Edge type k
<bof>	Beginning of node or edge attribute
<eof>	End of node or edge attribute

graph. We inject N node tokens to represent up to N nodes. Edges are represented as pairs of nodes. Since our dataset is heterogeneous, we also need to represent the node types and edge types. We use <ntype$_i$> and <etype$_i$> to represent the node type and edge type respectively. The node attributes are represented as a sequence of tokens, where each attribute is wrapped inside a pair of tokens <bof> and <eof>. This allows us to easily identify the start and end of each attribute. Comparing to directly representing the graph in a plain text format, such as the JSON node-link data format in python's NetworkX library, our tokenization schema only uses 35% and 22% tokens on provenance data and IntelliGraph data respectively.

To encode each distinct graph into a unique sequence of tokens, we assume that there exists a well-defined order to visit every edge in the graph. In our provenance dataset, each edge represents an action by a program with an associated timestamp, which can be used to naturally determine the order in which edges are encoded. For general graphs, we use a topological sort to determine the order of the edges. A strict ordering is important because it allows us to generate the graph in a way that respects the dependencies between nodes and edges, while reducing the ambiguity in the generated sequence of tokens. When encoding the graph, we append each edge in the graph following this pre-defined order. Whenever a new edge is added to the sequence that connects to any node not yet seen in the graph, we append the node attribute tokens immediately after.

The complete tokenization procedure is shown in Algorithm 1. It processes a heterogeneous graph $G(V, E)$ with node types T, edge types E_T, and attributes attached to nodes and edges.

Figure 3 shows an example of how a graph is represented as a sequence of tokens.

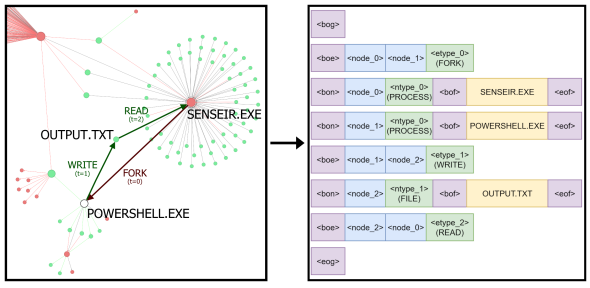


Fig. 3. Example of graph tokenization. The left side of the figure highlights a subgraph of a real provenance graph containing a `powershell.exe` process. The right side of the figure shows the corresponding tokenized sequence using our proposed schema and time-based edge ordering. File and process names are shortened.

Algorithm 1 Graph Tokenization

Input: Graph $G(V, E)$, node types T, edge types E_T
Initialize token sequence $S = [\text{<bog>}]$
Initialize node set $N = \{\}$
for each edge $(u, v) \in E$ **do**
  Append <boe> to S
  Append $\text{<node}_u\text{>}$ to S
  Append $\text{<node}_v\text{>}$ to S
  Append the edge type token for (u, v) to S
  Append <bof> to S
  for each attribute in the edge **do**
    Append the attribute token to S
  end for
  Append <eof> to S
  if u is not in N **then**
    Add u to N
    Append <bon> to S
    $\text{Append <node}_u\text{> to } S$
    $\text{Append <ntype}_{T[u]}\text{> to } S$
    for each attribute in the node **do**
      Append <bof> to S
      Append the attribute token to S
      Append <eof> to S
    end for
    Append <eon> to S
  end if
  if v is not in N **then**
    Add v to N
    Append <bon> to S
    $\text{Append <node}_v\text{> to } S$
    $\text{Append <ntype}_{T[v]}\text{> to } S$
    for each attribute in the node **do**
      Append <bof> to S
      Append the attribute token to S
      Append <eof> to S
    end for
    Append <eon> to S
  end if
end for
Append <eog> to sequence S

B. Decoding Tokens to Graph

When sampling long outputs from LLM, the model may generate invalid tokens or token sequences that do not follow the expected graph structure. To address this issue, we employ two methods to ensure the validity of the generated tokens.

Firstly, we utilize a token filtering mechanism that checks for invalid tokens or sequences during the generation process. We enforce a set of simple rules during sampling — for example, only node tokens $\text{<node}_i\text{>}$ are allowed to be followed by begin-of-node tokens <bon>, so we only sample from node tokens. This filtering step helps to eliminate any tokens that do not conform to the expected graph structure or violate the constraints of the graph representation.

Secondly, we designed anchor tokens <boe>, <bon> and <bof> to help with the parsing process. We implemented the parser with a state machine. When an invalid token is encountered, the parser attempts to recover by skipping the invalid token and continuing to parse the remaining tokens. When an anchor token is encountered it can directly trigger a state transition in the parser, allowing the parser to recover to a valid state. This approach allows us to salvage as much of the generated graph as possible, even in the presence of errors or unexpected tokens.

Combining the filtering and decoding steps, we can effectively handle the challenges associated with generating long sequences of tokens while maintaining the validity of the graph structure.

C. Efficient Training for LLM based Graph Generation

Utilizing large language model architecture for graph generation is a promising approach. However, it also presents challenges, particularly in terms of training efficiency. The training process can be computationally expensive and time-consuming, especially when dealing with large graphs. In this work, we adopt LLaMA3-3.2B as our backbone, a lightweight and efficient LLM architecture with a reasonable trade-off between performance and resource consumption. To accommodate our graph-specific tokenization, we initialize with the pretrained model weights and expand the embedding layer using mean-resizing to incorporate new special tokens. This strategy preserves the pretrained knowledge while enabling the model to learn representations for the added tokens. To further improve training efficiency, we employ mixed-precision training and LoRA-based fine-tuning, allowing us to harness the capabilities of large language models while significantly reducing computational overhead.

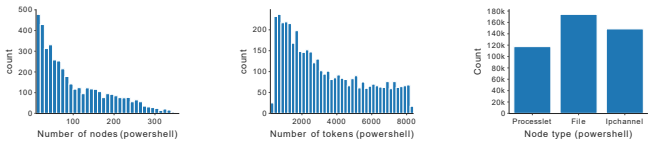
IV. Evaluation

A. Evaluation Protocol and Datasets.

We train ProvCreator to synthesize provenance graphs for `firefox.exe` and `powershell.exe` on Windows, and use the resulting graphs to augment under-represented sub-program classes in our testing corpus. These two programs are frequent targets of process-impersonation attacks by advanced cyber-threat actors [35]; moreover, their behavior is largely driven by command-line arguments, which yields clear class labels for evaluation. Our training data consists of system provenance graphs generated by forward-tracking from 50 492 instances of `powershell.exe` and 7 901 instances of `firefox.exe` representing real user and system activity collected from desktop systems actively in use by our lab.

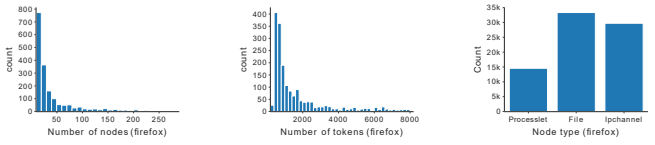
Figures 4 and 5 summarize the provenance datasets used to train the generative models. In each figure, subfigure (a) shows the node-count distribution, subfigure (b) shows token counts after tokenization, and subfigure (c) shows the node-type distribution.

We compare our model against the GDSS model [4], which is a state-of-the-art diffusion based graph generation model based on score-based generative modeling. Since the implementation



(a) (b) (c)

Fig. 4. Dataset statistics for PowerShell graphs.



(a) (b) (c)

Fig. 5. Dataset statistics for Firefox graphs.

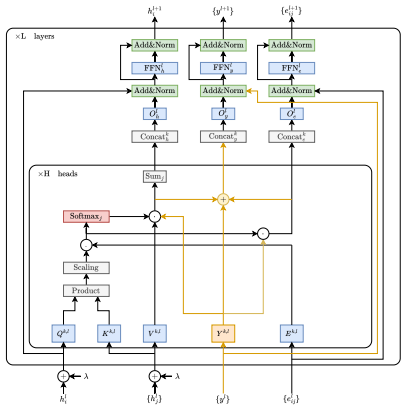


Fig. 6. GDSS structure generation model architecture ($L = 12$, $H = 8$, $\dim h = 256$, and $\dim e = \dim y = 128$). Orange arrows mark our conditional-generation components.

TABLE II
TRAINING AND MEAN PER-GRAPH GENERATION TIMES ON TWO INTEL GOLD 6226R CPUS AND EIGHT NVIDIA A6000 GPUS (4 GPUS PER TRAINING JOB, BATCH SIZE 4; 2 GPUS FOR GENERATION).

Dataset	Training Time (hours)	Generation Time (seconds)
Firefox	12	145
PowerShell	15	106

from the original paper does not scale well on very large graphs, we used a third-party graph-transformer based implementation, and we adjusted the hyperparameters to handle our dataset. The adjusted architecture and computational costs are summarized in Figure 6 and Table II.

For the structural fidelity studies (§ IV-B, § IV-D, and § IV-E), we sample 1 000 graphs for each of `firefox.exe` and `powershell.exe` from PROVCREATOR and from the score-based generator of Jo et al. [4], both trained on the same data. We then quantify the distributional distance between generated and training graphs with established graph-similarity metrics.

For downstream tasks (§ IV-F), we refer to respected works from the cybersecurity domain to guide our supervised program classification task [35]. Two identical Graph Neural Network (GNN) models are trained, one with a training set of real graphs and one with an identical number of synthetic graphs generated with PROVCREATOR. GNN models are trained separately to classify different command line arguments for both `firefox.exe` and `powershell.exe`. Both models are evaluated on real and synthetic test data.

In addition, to evaluate the general performance of PROVCREATOR, we also evaluated it on the *IntelliGraph* benchmark [8], which is a collection of synthetic and real-world graph datasets. The benchmark includes three synthetic datasets and two real-world datasets, and we compare our model against the baseline models provided in the benchmark.

TABLE III
MAXIMUM MEAN DISCREPANCY (MMD) DISTANCES OF DIFFERENT GRAPH METRICS BETWEEN REAL AND SYNTHETIC DATASETS GENERATED WITH GDSS [4] AND PROVCREATOR WHERE LOWER NUMBER IS BETTER.

	Degree ↓	Clustering ↓	Bet. Cen. ↓	Cls. Cen. ↓	Katz Cen. ↓	Spectral ↓
			`firefox.exe`			
GDSS	0.955	0.130	0.535	0.245	0.512	0.410
PROVCREATOR	**0.277**	**0.095**	**0.138**	**0.150**	**0.313**	**0.247**
			`powershell.exe`			
GDSS	1.148	0.487	0.295	1.213	1.276	1.022
PROVCREATOR	**1.109**	**0.158**	**0.147**	**0.997**	**1.251**	**0.915**

B. Structure Fidelity

Table III reports structural fidelity using multiple graph-level statistics for the real and the corresponding synthetic dataset, mirroring the evaluation protocol of Jo et al. [4]. We compute the *Maximum Mean Discrepancy* (MMD) for different structural metrics between real graphs and those synthesized by PROVCREATOR and the GDSS baseline, with both models trained independently on the `firefox.exe` and `powershell.exe` datasets. Across both programs, PROVCREATOR achieves consistently lower MMD scores than GDSS, indicating superior structural fidelity. We attribute these gains to the learning ability of the transformer, which enhances disambiguation among graph categories and enhanced generation capability due to the auto-regressive nature of PROVCREATOR.

C. Attribute Fidelity

Table IV presents the results of the attribute fidelity evaluation. Our model generates both the graph structure and attributes simultaneously, making it challenging to directly compare against ground truth attribute values. To address this, we developed the format-validation rules listed in Table V.

TABLE IV
VALID RATES FOR VARIOUS ATTRIBUTES.

Dataset	% Valid Process Names ↑	% Valid File Names ↑	% Valid IP Address and Ports ↑
Firefox	93.77	99.22	96.43
Powershell	94.83	98.70	98.92

TABLE V
VALIDATION RULES FOR GENERATED ATTRIBUTES.

Attribute	Regular Expression (Python format)
Executable Name (Windows)	`^[A-Za-z]:\\(?:[^\\/:*?\"<>|\r\n]+\\)*[^\\/:*?\"<>|\r\n]*\.exe$`
Executable Name (Linux)	`^/([^/\0]+/)*[^/\0]+$`
File Path (Windows)	`^[A-Za-z]:\\(?:[^\\/:*?\"<>|\r\n]+\\)*[^\\/:*?\"<>|\r\n]*$`
File Path (Linux)	`^/([^/\0]+/)*[^/\0]+$`
IP Address and Port	`^(?:\d{1,3}\.){3}\d{1,3}|\d{1,5}$`

TABLE VI
EVALUATION RESULTS USING THE INTELLIGRAPH BENCHMARK DATASET (ONLY THE BEST PERFORMING BASELINES ARE SHOWN).

Dataset	Model	% Valid Graphs ↑	% Novel & Vaild Graphs ↑	% Novel Graphs ↑	% Empty Graphs ↑
syn-paths	ComplEx (Sampling $P(E)$ and $P(S\|E)$)	0.71	0.71	14.27	85.73
	ComplEx (Sampling only $P(E)$)	10.10	10.10	95.58	4.42
	ProvCreator	93.14	93.14	100.0	0.0
syn-tipr	DistMult (Sampling $P(E)$ and $P(S\|E)$)	0.00	0.00	13.34	86.66
	ComplEx (Sampling only $P(E)$)	0.00	0.00	99.64	0.36
	ProvCreator	0.00	0.00	100.00	0.00
syn-types	TransE (Sampling $P(E)$ and $P(S\|E)$)	0.21	0.21	15.44	84.56
	DistMult (Sampling only $P(E)$)	1.44	1.44	96.19	4.81
	ProvCreator	86.16	86.16	100.00	0.00
wd-movies	ComplEx (Sampling $P(E)$ and $P(S\|E)$)	0.00	0.00	1.87	98.13
	ComplEx (Sampling only $P(E)$)	0.41	0.41	93.04	6.96
	ProvCreator	94.26	94.26	100.0	0.00
wd-articles	TransE (Sampling $P(E)$ and $P(S\|E)$)	0.00	0.00	4.58	95.42
	ComplEx (Sampling only $P(E)$)	0.00	0.00	100.00	0.00
	ProvCreator	0.00	0.00	100.00	0.00

* Note: PROVCREATOR is accomplishing a more difficult task while still outperforming the baselines. The baseline models does not need to generate the attributes as text, but instead sample the attributes as a set of numbers.
** Due to very strict evaluation criteria on certain datasets, both our model and the baselines may not produce a fully valid graph, leading to zero scores in some cases. We reported the scores as-is, but the actual performance of our model may be better than indicated.

D. Evaluation on IntelliGraph Benchmark Dataset

Table VI shows the results when evaluating our model on the *IntelliGraph* Datasets [8]. These datasets are made up of knowledge graphs where each node represents some entity and has a textual label, and each edge represents a relationship between nodes with a categorical relationship type. Thanapalasingam et al. [8] provide evaluation results for baseline graph generation models on each dataset, which we compare against PROVCREATOR in Table VI. These baseline methods generate node labels by sampling labels from the original dataset, whereas PROVCREATOR generates node labels as textual node attributes. We also generate the edge type as a textual attribute, falling back to the first edge type in the dataset if the generated text is invalid. Our model outperforms the provided baseline models by a large margin, indicating that our model is capable of generating diverse and realistic graphs on general graph datasets.

E. Embedding Fidelity

We evaluate the overall similarity of the synthetic graphs to the original training set by embedding the graphs into vectors, then measuring the average cosine similarity of those graph embeddings to the embeddings of the training set. Figure 7 compares graphs generated by PROVCREATOR with a baseline method. The baseline method uses real graph structures sampled from the training set, augmented with randomly sampled node attributes for parity. To embed the graph with text attributes into vectors, we sample 100 random walk paths with maximum length 10 on each graph, and log the nodes and edges visited along with the attributes to obtain a document representing the graph. Then we use doc2vec [37] to obtain the embedding. PROVCREATOR attains comparative similarity scores, demonstrating the ability to capture the relationship between the graph structure and node attributes.

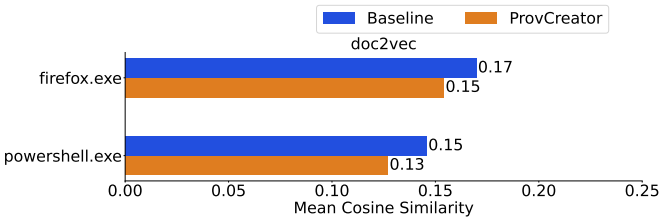


Fig. 7. Embedding cosine similarity of real and synthetic datasets generated with PROVCREATOR and a GDSS [4] baseline, using doc2vec graph embeddings [36].

TABLE VII
PROGRAM-CLASS DEFINITIONS USED IN THE DOWNSTREAM EVALUATION.

Program	Label	Command	Description
firefox.exe	A	`firefox.exe -MOZ_LOG ...`	Sets the logging level and other log-related options for specified components of Firefox.
	B	`firefox.exe -backgroundtask ...`	Invokes Firefox in a headless background task mode. This is used for periodic maintenance tasks, like checking for updates or sending telemetry, and can run even when the main Firefox browser is not open.
	C	`firefox.exe`	This starts the main Firefox web browser process.
	D	`firefox.exe -contentproc ...`	This is a subprocess of a Firefox browser instance that handles web content in a particular tab.
	E	`firefox.exe -osint ...`	This command line argument is used when Firefox is invoked by the operating system's shell to handle a search query or open a link.
powershell.exe	A	`powershell.exe -ExecutionPolicy AllSigned -NoProfile -NonInteractive -Command "& {Inline Script}"`	Executes an inline PowerShell script used by Windows Defender to verify the hash of scripts it uses.
	B	`powershell.exe -Nologo -Noninteractive -NoProfile -ExecutionPolicy Bypass; Get-DeliveryOptimizationStatus ...`	Retrieves the status of Windows Update delivery optimization jobs and filters them to identify jobs from a local cache source and meeting specific criteria including a minimum file size.
	C	`powershell.exe -ExecutionPolicy Bypass -NoProfile -Command "Add-Type 'C# Code 1'; ...`	Adds a .NET type, then calls a function from that type. This PowerShell command is found in a Windows Defender script.
	D	`powershell.exe -ExecutionPolicy Bypass -NoProfile -Command "Add-Type 'C# Code 2'; ...`	Adds a .NET type (different from class C), then calls a function from that type. This PowerShell command is found in a Windows Defender script.
	E	`powershell.exe -NoLogo -Noninteractive -NoProfile -ExecutionPolicy Bypass & 'C:\Windows\CCM\SystemTemp\GUID'`	Executes a PowerShell script in a temporary directory used by Microsoft System Center Configuration Manager (SCCM). Likely for automated management, software deployment, or configuration tasks.

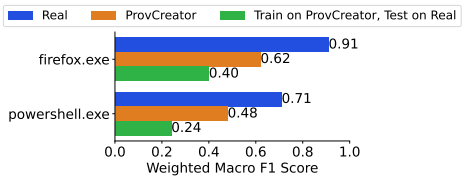


Fig. 8. Weighted macro-F1 scores for GNN-based subprogram classification using different graph types.

F. Downstream Application

To demonstrate the practical utility of PROVCREATOR, we consider security-relevant downstream tasks of program classification (Figure 8 show classification results). For both the datasets, we compare models trained and tested on real graphs, on PROVCREATOR synthetic traces, and models trained on PROVCREATOR but evaluated on real graphs. provenance domain training on ProvCreator retains 65-70% of the real-data F1, demonstrating that the synthetic graphs preserve class signal. When transferred to real traces the performance drops further, revealing a domain gap but still indicating non-trivial generalization from synthetic to real data.

1) Program Classification: We chose `firefox.exe` and `powershell.exe` because they are popular targets for impersonation by advanced cyber threats [35], and their behavior is largely determined by command-line arguments that provide clear class labels. Given a provenance graph rooted at either program, the classifier predicts the issued command from the system resources referenced in the graph. Table VII lists the complete class definitions used in our experiments.

In subplots (a,d) of Figure 9, the classifier is trained and tested on real graphs; in (b,e), it is trained and tested on PROVCREATOR synthetic graphs; and in (c,f), it is trained on PROVCREATOR graphs but evaluated on held-out real traces.

Figure 9 reveal that PROVCREATOR preserves the class-discriminative structure observed in real data. When models trained on PROVCREATOR are transferred to real traces (c,f), accuracy declines but remains substantial, demonstrating that the synthetic dataset is realistic to support cross-domain generalization and mitigate real-data scarcity.

These gains hold for both the `firefox.exe` and `powershell.exe` scenarios, underscoring PROVCREATOR's ability to generate representative synthetic graphs that bolster classification performance where training data are sparse.

V. CONCLUSION

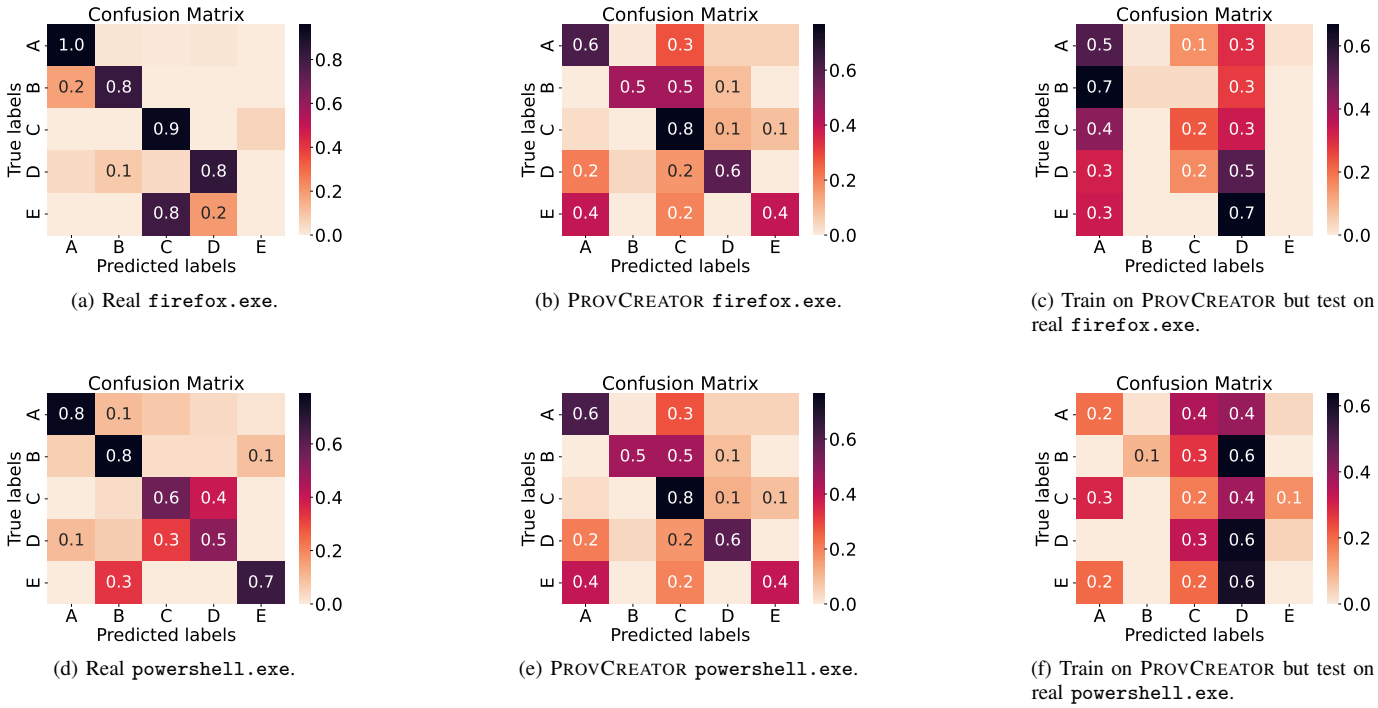
In this paper, we introduced PROVCREATOR, a novel synthetic graph generation framework designed for heterogeneous graphs with nonuniform text attributes. By jointly considering the graph structure, node attributes, and program class labels, PROVCREATOR addresses program class imbalance in system provenance datasets with synthetic graphs that contain rich textual attributes. Further, the inclusion of flexible node attribute indicators enables the generation of semantically nuanced textual attributes, with multiple attributes per node. Our evaluation on real-world programs demonstrates improved graph structural fidelity compared to prior works, as well as improved utility in security-relevant downstream tasks. PROVCREATOR marks a step towards applying synthetic data generation to support accuracy-critical applications that require rich graph attributes.

VI. REPRODUCIBILITY

All experimental code related to PROVCREATOR is available at https://anonymous.4open.science/r/provcreator-aio-4F83.

REFERENCES

[1] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.



(a) Real `firefox.exe`.

(b) PROVCREATOR `firefox.exe`.

(c) Train on PROVCREATOR but test on real `firefox.exe`.

(d) Real `powershell.exe`.

(e) PROVCREATOR `powershell.exe`.

(f) Train on PROVCREATOR but test on real `powershell.exe`.

Fig. 9. Normalized confusion matrices for subprogram classification task.

[2] X. Xu, M. Li, C. Tao, T. Shen, R. Cheng, J. Li, C. Xu, D. Tao, and T. Zhou, "A survey on knowledge distillation of large language models," 2024. [Online]. Available: https://arxiv.org/abs/2402.13116

[3] NVIDIA, :, N. Agarwal, A. Ali, M. Bala, Y. Balaji, E. Barker, T. Cai, P. Chattopadhyay, Y. Chen, Y. Cui, Y. Ding, D. Dworakowski, J. Fan, M. Fenzi, F. Ferroni, S. Fidler, D. Fox, S. Ge, Y. Ge, J. Gu, S. Gururani, E. He, J. Huang, J. Huffman, P. Jannaty, J. Jin, S. W. Kim, G. Klár, G. Lam, S. Lan, L. Leal-Taixe, A. Li, Z. Li, C.-H. Lin, T.-Y. Lin, H. Ling, M.-Y. Liu, X. Liu, A. Luo, Q. Ma, H. Mao, K. Mo, A. Mousavian, S. Nah, S. Niverty, D. Page, D. Paschalidou, Z. Patel, L. Pavao, M. Ramezanali, F. Reda, X. Ren, V. R. N. Sabavat, E. Schmerling, S. Shi, B. Stefaniak, S. Tang, L. Tchapmi, P. Tredak, W.-C. Tseng, J. Varghese, H. Wang, H. Wang, H. Wang, T.-C. Wang, F. Wei, X. Wei, J. Z. Wu, J. Xu, W. Yang, L. Yen-Chen, X. Zeng, Y. Zeng, J. Zhang, Q. Zhang, Y. Zhang, Q. Zhao, and A. Zolkowski, "Cosmos world foundation model platform for physical ai," 2025. [Online]. Available: https://arxiv.org/abs/2501.03575

[4] J. Jo, S. Lee, and S. J. Hwang, "Score-based generative modeling of graphs via the system of stochastic differential equations," in *International Conference on Machine Learning*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:246634850

[5] C. Vignac, I. Krawczuk, A. Siraudin, B. Wang, V. Cevher, and P. Frossard, "Digress: Discrete denoising diffusion for graph generation," in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=UaAD-Nu86WX

[6] Z. Chen, H. Mao, H. Li, W. Jin, H. Wen, X. Wei, S. Wang, D. Yin, W. Fan, H. Liu, and J. Tang, "Exploring the potential of large language models (llms) in learning on graphs," *ArXiv*, vol. abs/2307.03393, 2023.

[7] X. Chen, Y. Wang, J. He, Y. Du, S. Hassoun, X. Xu, and L.-P. Liu, "Graph generative pre-trained transformer," 2025. [Online]. Available: https://arxiv.org/abs/2501.01073

[8] T. Thanapalasingam, E. van Krieken, P. Bloem, and P. Groth, "Intelligraphs: Datasets for benchmarking knowledge graph generation," *arXiv preprint arXiv:2307.06698*, 2023.

[9] E. D. Cubuk, B. Zoph, D. Mané, V. Vasudevan, and Q. V. Le, "AutoAugment: Learning Augmentation Strategies From Data," 2019.

[10] M. Al Olaimat, D. Lee, Y. Kim, J. Kim, and J. Kim, "A learning-based data augmentation for network anomaly detection," in *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, 2020.

[11] C. G. Cordero, E. Vasilomanolakis, A. Wainakh, M. Mühlhäuser, and S. Nadjm-Tehrani, "On generating network traffic datasets with synthetic attacks for intrusion detection," 2021.

[12] S. Y. Feng, V. Gangal, J. Wei, S. Chandar, S. Vosoughi, T. Mitamura, and E. Hovy, "A Survey of Data Augmentation Approaches for NLP."

[13] C. Yan, Y. Yan, Z. Wan, Z. Zhang, L. Omberg, J. Guinney,

S. D. Mooney, and B. A. Malin, "A Multifaceted benchmarking of synthetic electronic health record generation models," 2022.
[14] T. Zhao, Y. Liu, L. Neves, O. Woodford, M. Jiang, and N. Shah, "Data Augmentation for Graph Neural Networks."
[15] T. Zhao, W. Jin, Y. Liu, Y. Wang, G. Liu, S. Günnemann, N. Shah, and M. Jiang, "Graph Data Augmentation for Graph Machine Learning: A Survey," 2022.
[16] M. Li, E. Kreačić, V. K. Potluru, and P. Li, "Graphmaker: Can diffusion models generate large attributed graphs?" 2024. [Online]. Available: https://arxiv.org/abs/2310.13833
[17] F. Li, X. Wang, D. Cheng, C. Chen, Y. Zhang, and X. Lin, "Efficient dynamic attributed graph generation," 2024. [Online]. Available: https://arxiv.org/abs/2412.08810
[18] S. T. King and P. M. Chen, "Backtracking intrusions," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.
[19] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal, "Towards a Timely Causality Analysis for Enterprise Security," in *Network and Distributed System Security Symposium (NDSS)*, 2018.
[20] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage." in *Network and Distributed System Security Symposium (NDSS)*, 2019.
[21] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Chen, W. Cheng, C. A. Gunter, and H. Chen, "You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis," in *Network and Distributed System Security Symposium (NDSS)*, 2020.
[22] "The linux audit framework," https://github.com/linux-audit/, 2015.
[23] "Event tracing for windows (etw) - windows drivers | microsoft docs," https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw-, 2019.
[24] B. Cantrill, "Dtrace," in *Large Installation System Administration Conference (LISA)*, 2005.
[25] G. Creech and J. Hu, "Generation of a new ids test dataset: Time to retire the kdd collection," in *2013 IEEE wireless communications and networking conference (WCNC)*, 2013.
[26] W. Haider, G. Creech, Y. Xie, and J. Hu, "Windows Based Data Sets for Evaluation of Robustness of Host Based Intrusion Detection Systems (IDS) to Zero-Day and Stealth Attacks," 2016.
[27] X. Han, T. Pasquier, A. Bates, J. Mickens, and M. Seltzer, "UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats," in *Network and Distributed System Security Symposium (NDSS)*, 2020.
[28] 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 (SEC)*, 2021.
[29] Z. Jia, Y. Xiong, Y. Nan, Y. Zhang, J. Zhao, and M. Wen, "Magic: Detecting advanced persistent threats via masked graph representation learning," in *USENIX Security Symposium (SEC)*, 2024.
[30] Z. Cheng, Q. Lv, J. Liang, Y. Wang, D. Sun, T. Pasquier, and X. Han, "Kairos: Practical Intrusion Detection and Investigation using Whole-system Provenance," in *IEEE Symposium on Security and Privacy (SP)*, 2024.
[31] K. Mukherjee, J. Wiedemeier, Q. Wang, J. Kamimura, J. J. Rhee, J. Wei, Z. Li, X. Yu, L.-A. Tang, J. Gui *et al.*, "Proviot: Detecting stealthy attacks in iot through federated edge-cloud security," in *International Conference on Applied Cryptography and Network Security*. Springer, 2024, pp. 241–268.
[32] M. U. Rehman, H. Ahmadi, and W. U. Hassan, "FLASH: A Comprehensive Approach to Intrusion Detection via Provenance Graph Representation Learning," in *IEEE Symposium on Security and Privacy (SP)*, 2024.
[33] K. Mukherjee, J. Wiedemeier, T. Wang, J. Wei, F. Chen, M. Kim, M. Kantarcioglu, and K. Jee, "Evading provenance-based ml detectors with adversarial system actions," in *USENIX Security Symposium (SEC)*, 2023.
[34] H. Wang, S. Feng, T. He, Z. Tan, X. Han, and Y. Tsvetkov, "Can language models solve graph problems in natural language?" in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: https://openreview.net/forum?id=UDqHhbqYJV
[35] F. Barr-Smith, X. Ugarte-Pedrero, M. Graziano, R. Spolaor, and I. Martinovic, "Survivalism: Systematic Analysis of Windows Malware Living-Off-The-Land," in *IEEE Symposium on Security and Privacy (SP)*, 2021.
[36] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*. PMLR, 2014, pp. 1188–1196.
[37] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," 2014.