applied sciences

MDPI

Article

Embedding-Driven Synthetic Malware Generation with Autoencoders and Cluster-Tangent Diffusion

Gunnika Kapoor [1,†], Sathvika Nadipalli [2,†] and Fabio Di Troia [2,*]

[1] Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX 76019, USA
[2] Department of Computer Science, San José State University, San Jose, CA 95192, USA; sathvika.nadipalli@sjsu.edu
* Correspondence: fabio.ditroia@sjsu.edu
† These authors contributed equally to this work.

Abstract

Malware has become increasingly sophisticated over the years, with zero-day attacks emerging at an alarming pace. Effective detection and analysis demand real malware samples, which are expensive and skill-dependent to extract. As a result, generating high quality synthetic samples from scarce data sets becomes a crucial method for strengthening detection software. This paper focuses on presenting generation techniques that optimize the embedding space to produce high-quality synthetic samples, even under constrained datasets. The dataset used in this paper consists of 500 Windows malware API call samples that were processed using embedding and Generative AI (Gen AI) techniques to generate synthetic malware. Two novel contributions are highlighted in this paper. (1) The integration of autoencoders with pretrained NLP models (BERT and ELMo) to enhance the quality of embeddings. Autoencoders extract features and learn patterns from the data to generate higher-quality embeddings than those generated using other techniques alone. (2) Cluster-Tangent Diffusion (CT-Diff): a novel application of manifold diffusion. Manifold diffusion improves upon diffusion and other Gen AI techniques by focusing on generating samples along the distribution of the original data using structured noise instead of standard gaussian noise. Collectively these two contributions have consistently outperformed previous techniques. Furthermore, the results demonstrate the feasibility of generating reliable fake samples even in low data scenarios.

Keywords: synthetic malware generation; diffusion; manifold diffusion; Natural Language Processing; graph embeddings; autoencoder; Artificial Intelligence; cybersecurity

Academic Editors: Ming Hour Yang, Vijayalakshmi Murugesan and Mercy Shalinie Selvaraj

Received: 3 October 2025
Revised: 27 October 2025
Accepted: 31 October 2025
Published: 5 November 2025

Citation: Kapoor, G.; Nadipalli, S.; Di Troia, F. Embedding-Driven Synthetic Malware Generation with Autoencoders and Cluster-Tangent Diffusion. *Appl. Sci.* **2025**, *15*, 11791. https://doi.org/10.3390/app152111791

1. Introduction

Malware, or malicious software, is an intrusive software intended to steal data or harm computers and computer systems [1]. Malware can be classified by type of malware or by malware family, where malware families are groups of malware that share similar attack techniques [2]. With cybercrime costs expected to reach $10.5 trillion USD annually in 2025 [3], it is critical to improve defenses against them. However, in order to perform malware detection and classification, a large amount of malware is required. To this end, generating high-quality synthetic malware can be beneficial [4].

Prior to generating synthetic malware, samples must be preprocessed and transformed into embeddings. Various methods may be used for this purpose based on the data type [5–12]. One method creates embeddings [5,6] by representing malware based on its

API calls [5,6,10]. Another captures program statement and program flow information [7] using function call and system call information derived from malware executables [7–9]. Alternatively, malware may be represented as a set of opcodes [11]. Regardless of how the malware is represented, it may either be statically or dynamically analyzed. Static analysis refers to analyzing malicious code without execution [13]. In dynamic analysis, malware is executed in a controlled environment to monitor real-time behavior [13].

Machine learning models often need to be trained on large amounts of data to generate high-quality outputs, but it can be challenging and time-consuming to collect a diverse set of real malware samples [4]. Recently, methods such as few-shot learning, transfer learning, and data augmentation, have been explored as alternative techniques [14,15]. Yet, to the best of our knowledge, no previous work has evaluated the use of autoencoders in combination with embedding models or manifold diffusion techniques to generate synthetic malware with a limited set of real malware samples.

In this paper, we dynamically analyze malware by extracting API call sequences from 7 malware families for a total dataset size of just under 500 samples. These samples are processed using various combinations of aforementioned techniques, specifically exploring three Natural Language Processing (NLP) models: FastText, ELMo, and BERT, with embeddings from ELMo and BERT being further processed using autoencoders; two graph-based techniques: Graph2Vec and Node2Vec; and three Generative AI (Gen AI) models: WGAN-GP, Diffusion, and Cluster-Tangent Diffusion (CT-Diff), where CT-Diff is a novel application of manifold diffusion to malware. Furthermore we incorporate dimension-reduction techniques through autoencoders to improve the performance of select models. The performance of these models is evaluated using multi-class classification techniques, t-SNE modeling, and cosine similarity. The primary focus of this paper is on evaluating the use of autoencoders and CT-Diff, with the remaining models serving as a benchmark for assessing the robustness of the fake malware generated through these techniques.

Our work suggests that combining autoencoder processing with pretrained NLP models, BERT and ELMo, can improve performance. Notably, on limited datasets, CT-Diff is among the models that can most accurately capture the structure of the original data distribution and generate realistic synthetic malware samples.

Our work makes the following contributions:

- Autoencoders: We generate synthetic malware samples using NLP and graph embedding models and further process embeddings outputted from those with low performance, ELMo and BERT, to improve on the embedding quality.
- CT-Diff: We present an application of manifold diffusion to malware, which we denote as CT-Diff. To the best of our knowledge, this paper is the first application of manifold diffusion techniques in the malware domain.
- Synthetic malware generation under data scarcity: We utilize a dataset of under 500 Windows malware samples and show that through strategic use of embedding and generative techniques, high-quality synthetic malware can be generated. The success of our approach suggests that these techniques are capable of capturing deep structural patterns of malware. This raises the possibility that these techniques could be used to effectively simulate behavioral patterns found in zero-day attacks.

The rest of the paper is structured as follows: Section 2 discusses related papers and background for our work. In Section 3, we describe the models used and how the NLP embeddings, Graph embeddings, and synthetic malware were generated. In Section 4, we evaluate the models and provide results of this evaluation. In Sections 5 and 6, we conclude with a discussion of limitations and future work.

2. Related Work

Generative AI (Gen AI) has emerged as a powerful tool for augmenting malware detection, particularly in scenarios where data scarcity limits the effectiveness of traditional learning approaches. Prior research has explored Gen AI through the lens of Natural Language Processing (NLP), leveraging pretrained language models to enhance feature extraction and representation. In parallel, graph embedding techniques have gained traction for modeling structural relationships within malware behavior, offering complementary perspectives on data generation and classification. This section reviews relevant literature in both domains, highlighting how Gen AI has been integrated with NLP and graph-based methods to improve the quality and utility of synthetic malware samples. A quick overview is provided in Table 1.

Table 1. Literature Review and Identified Gaps.

Study (Author, Year, Title)	Topic	Findings/Gap
Kale et al. [16], Malware classification with Word2Vec, HMM2Vec, BERT, and ELMo	Opcode embeddings for classification	Strong accuracy with BERT on opcode data; lacks generative modeling or data-scarcity augmentation.
Aggarwal & Di Troia [17], Malware Classification Using Dynamically Extracted API Call Embeddings	Dynamic API sequence embeddings	Effective for classification; no synthetic data generation or manifold-guided diffusion.
Maniriho et al. [18], API-MalDetect: Automated malware detection framework for Windows	Deep learning-based detection of malware attacks	Robust detection pipeline; focuses only on classification, not sample generation.
Amer & Zelinka [19], Contextual understanding of API call sequence	Dynamic API analysis	Captures temporal dependencies; lacks embedding optimization or synthetic augmentation.
Qiao et al. [20], MLP + Word2Vec for IoT malware classification	Word2Vec for IoT malware	Strong accuracy with static embeddings; no autoencoder or generative refinement.
Yesir & Soğukpınar [21], Malware detection and classification using FastText and BERT	NLP-based embeddings	High detection accuracy; lacks diffusion or manifold-aware synthesis.
Tran & Di Troia [22], Word Embeddings for Fake Malware Generation	Embedding-driven malware generation	Demonstrated feasibility of fake sample generation; not manifold- or cluster-conditioned.
Bao et al. [4], Generating Synthetic Malware Samples using Generative AI against Zero-day Attacks	Use of Word2Vec, GAN, WGAN-GP, and Diffusion to generate synthetic malware	Diffusion improved F1 scores; limited to opcode data; lacks embedding optimization and structure awareness.
Mollah et al. [23], Control Flow Graphs, Node2Vec, and GCN Integration	Graph embeddings for malware detection	Captured graph structure well; no generative malware synthesis.
McLaren et al. [24], GAN-based malware generation using behavioural graphs	Graph-based GAN generation	Effective structure modeling; GAN prone to mode collapse; lacks manifold-guided refinement.
Wesego [25], Graph Representation Learning with Diffusion Generative Models	Diffusion for graph embeddings	Highlights diffusion's promise; not applied to malware or data-scarce environments.
Lee & Choi [26], Local Manifold Approximation and Projection (LoMAP)	Manifold-aware diffusion	Improves generation fidelity via local manifolds; not used for malware embeddings.
Chung et al. [27], Improving diffusion models for inverse problems using manifold constraints	Manifold-regularized diffusion	Improves reconstruction quality; no evaluation on malware embeddings.
He et al. [28], Manifold Preserving Guided Diffusion	Structure-preserving diffusion	Preserves data geometry; lacks application to malware or low-data scenarios.

2.1. Natural Language Processing (NLP) and Generative AI (Gen AI)

In order to use malware for classification, detection, and generation tasks, it must be converted into a machine-understandable representation. This is achieved using embeddings. Natural Language Processing (NLP) techniques are commonly employed to transform raw text data into high-quality word embeddings.

Four examples of NLP models are Word2Vec [29], FastText [30], BERT [31], and ELMo [32]. All four models have previously been used to generate embeddings for malware represented as opcode sequences [16], API call sequences [17], and binary executable files [20].

Kale et al. [16] and Aggarwal et al. [17] explore embedding and classifying malware represented as opcode sequences [16] and API calls [17] using NLP models such as Word2Vec, BERT, and ELMo [16,17]. While BERT performed best on the opcode data [16], Word2Vec performed best on the API call data [17]. Qiao et al. [20] similarly achieved high classification accuracy when using Word2Vec to classify malware binary executables both for traditional malware and for IoT malware. Finally, Feng et al. [33] and Yesir et al. [21] show that FastText has potential for use in classifying API sequences, as it achieved a high accuracy and detection rate [33].

Generative AI models are those used for data generation, such as text generation, audio generation, or image generation [34,35]. In the malware domain, these models are often used to generate synthetic samples that closely resemble real samples [13] as a way to compensate for data scarcity [35], to serve as a form of data augmentation [35], or to identify trends to predict future threats [36]. This is essential, as generating malware is time-consuming, skill-based [36], and cost intensive.

When Gen AI is used in conjunction with NLP models, high-quality malware embeddings may be generated. For example, Tran et al. [22] use BERT and WGAN-GP to generate realistic opcode samples from seven malware families, achieving realistic malware generation. Bao et al. [4] similarly explore malware generation, using Word2Vec to generate opcode embeddings and three Gen AI models, GAN, WGAN-GP, and Diffusion, to generate synthetic samples. Overall, augmenting the original malware set with synthetic samples led to improved F1 scores for malware classification, with the Diffusion model performing the best.

2.2. Graph Embedding Techniques and Generative AI (Gen AI)

Graph embedding techniques have been applied for API call graph feature extraction in malware classification. Mollah et al. [23] utilized both Graph2Vec [37] and Node2Vec [38] vectorization algorithms, and found that Node2Vec yielded the best performance in capturing the structural relationships of malware through Control Flow Graphs [39].

Diffusion [40] has been shown to have strong generative capabilities largely attributed to its state-of-the-art noising and denoising architecture.

Wesego [25] trained a Graph Convolutional Network (GCN) [41] to obtain graph level embeddings and found that embeddings learned from the Discrete Diffusion Model had the highest accuracy in classification tasks. This work emphasizes diffusion's strength in capturing high-quality representations and highlights its value in the domain of graph embeddings.

McLaren et al. [24] used a control flow inspired graph to represent malware API calls for GAN based malware generation. They constructed directed API call subgraphs which were vectorized and passed to a GAN. This work demonstrates the potential of graph based embeddings in effective malware generation tasks.

2.3. Structure-Aware Diffusion

In recent years Structure Aware Diffusion has gained a lot of traction. Structure Aware Diffusion refers to any diffusion model that uses the structure of the original data to steer the diffusion generation process.

This is a new method that has been tested in various fields. Strudel et al. [42] have explored this method in text generation by using prompted and unprompted models for generation. The goal is to steer the unprompted generation towards the prompted generation using the prompts as a guide. More geometry-based approaches have also been explored in the works of Lee et al. [26] and Adaloglou et al. [43]. Lee et al. created a Local Manifold Approximation and Projection (LoMAP) approach that restricts diffusion generation using PCA on nearby points to ensure the generated data stays close to the original data. Adaloglou et al. use clustering using KMeans to condition diffusion models to generate within the main clusters of the original data distribution.

2.4. Our Approach

In our work, we note the success previous approaches have found in generating synthetic malware using embedding and generation techniques and utilize a selection of them with our dataset. Specifically, we explore dynamic analysis using an API call dataset. Dynamic analysis sheds additional light on malware behavior [13,44], as it captures API call sequences from malware execution. What sets our approach apart from previous ones is our investigation of generation under data-scarce scenarios, which remains underexplored despite its real-world relevance. This offers new insights for low-resource malware research and synthetic data generation. To the best of our knowledge, this is the first study to compare optimizing embeddings through the embedding space with other NLP, graph, and Gen-AI models. Specifically, we present what we believe is the first application of structure-aware, or manifold, diffusion to malware. This is not only due to its prior success in generating high-quality samples, but also its potential to more accurately capture the structure of the original data's distribution than other generative techniques. Additionally, we use autoencoders to process the outputs of pretrained NLP models and better capture the features of the original malware data.

3. Methodology

To address the challenge of generating high-quality synthetic malware samples from limited data, our methodology integrates multiple components designed to optimize feature representation and sample generation. We begin by outlining the system architecture that orchestrates the data flow and model interactions. Preprocessing steps ensure that raw malware API call sequences are normalized and structured for downstream analysis. We then apply NLP techniques, including pretrained language models, to extract rich semantic embeddings. These embeddings are further enhanced through graph-based representations that capture structural relationships within the data. Finally, Generative AI (Gen AI) techniques are employed to synthesize realistic malware samples, leveraging both embedding quality and manifold-aware generation strategies.

3.1. System Architecture

In this paper, we present a system architecture (Figure 1) for synthetic malware generation composed of five primary steps.

- Preprocessing (Figure 1A): Windows executable files are dynamically executed to extract API calls from a total of 7 malware families (Adload, Bancos, OnlineGames, Vbinject, Vundo, WinWebSec, and Zwangi). It was sourced from work conducted by Aggarwal et al. [17].

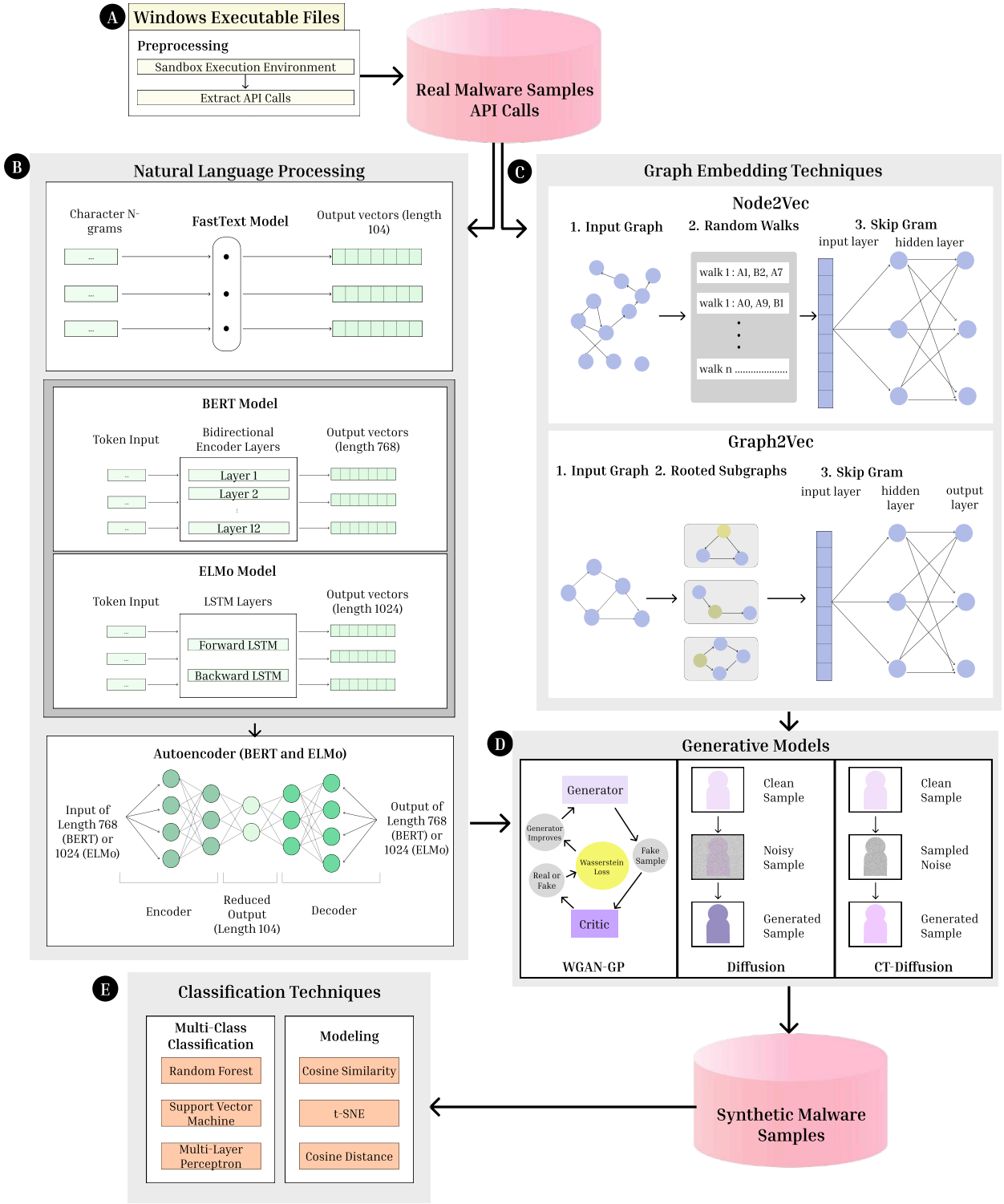

Figure 1. System Architecture. This paper creates a system composed of five primary parts: (**A**) Extracting malware API calls from 7 malware families by dynamically executing Windows malware executables, (**B**) Natural Language Processing through 3 NLP models and dimension reduction techniques to generate vector embeddings, (**C**) Graph-based processing through 2 graph embedding methods to generate vector embeddings, (**D**) Processing word and graph embeddings through 3 Generative AI Models with an end goal of synthetic malware sample generation. (**E**) The use of six evaluation metrics, both multi-class classification and modeling techniques, to evaluate the generated embeddings.

- Natural Language Processing (Figure 1B): Word embeddings were created using FastText, ELMo, and BERT. This allowed us to explore embedding generation using both pretrained and non-pretrained models. The outputs of the pretrained models (ELMo and BERT) were further processed using an autoencoder.
- Graph Embedding Creation (Figure 1C): Graph embeddings were generated using Node2Vec and Graph2Vec. This allowed us to explore embedding generation when API call data was represented using graphs that captured node-level and graph-level structural information.
- Generative AI Models (Figure 1D): WGAN-GP, Diffusion, and CT-Diff were used to process the NLP and Graph embeddings to generate synthetic malware samples.
- Evaluation Metrics (Figure 1E): Both multi-class classification and modeling techniques were used to evaluate the quality of the generated embeddings.

3.2. Preprocessing

The dataset (Table 2) used in this paper consists of a total of 7 malware families, each of which contain either 70 or 71 samples. Each sample has a varying API call sequence length, with the maximum length being 17,364 API calls. The API call dataset was sourced from work conducted by Aggarwal et al., in which they dynamically extracted API call data from Windows malware executable samples [17].

Dynamic analysis involves extracting API calls from malicious software during execution. In their work, Aggarwal et al. [17] ran a collection of Windows malware executable files in an environment consisting of Buster Sandbox Analyzer and Sandboxie sandbox environment. The outputs of the program, a set of API call log sequences, were further processed to remove non-crucial calls and any log information aside from the API call names.

Table 2. API Call Dataset.

Malware Family	Samples	Max API Seq. Length
Adload	70	198
Bancos	71	1006
OnlineGames	70	714
Vbinject	70	17,364
Vundo	71	3918
WinWebSec	70	3406
Zwangi	70	772

3.3. Natural Language Processing

In order to convert the malware API call data into word embeddings, three primary NLP models were used: FastText [30], BERT [31], and ELMo [32]. ELMo and BERT are pretrained models, and are thus pretrained on English text corpora before being finetuned on the API call data [31,32]. This process of training a model on a dataset and utilizing it on a different downstream task is also called transfer learning [45], and may be used to improve model performance and generalization for the downstream task. Embeddings from BERT and ELMo are context-dependent, which means that the same word may have different embeddings in different contexts [16]. Conversely, FastText is trained solely on the API call data [29,30]. Its embeddings are static, which means that it has a single, fixed representation regardless of the context in which it is found [16]. Once the embeddings from the models were generated, ELMo and BERT yielded lower quality results than FastText. Previous papers have utilized autoencoders as a means of extracting and learning patterns and features in data [34,46]. For this reason, we combined the embeddings from the ELMo and BERT models with the encoder portion of an autoencoder in order to generate higher-quality embeddings. Table 3 shows samples of API calls and their relative length.

Table 3. Sample function calls for each malware sample.

Sample Number	Sample Calls	Sample Length
s_0	'virtualallocex', 'getmodulehandle', 'freelibrary'	3
s_1	'VirtualAllocEx', 'QuerySystemInformation', ..., 'FreeLibrary', 'TerminateProcess'	76
s_2	'LdrFindEntryForAddress', 'GetModuleHandle', ..., 'OpenProcessToken', 'VirtualAllocEx'	174
s_3	'CreateThread', 'ResumeThread', ..., 'LdrFindEntryForAddress', 'GetModuleHandle'	17,364

3.3.1. FastText

The FastText [30] model is a non-pretrained model that creates word vector representations from text data such that similar words have embeddings that are close to one another. It represents words as a bag of character n-grams and associates a vector representation with each character n-gram. Its architecture is based on the Skip-gram model, which, given an input word, predicts the context words that surround it in a given range. Furthermore, the model considers subword information along with information about whole words, which allows the model to more effectively capture information about word structure.

The model's architecture is shown in Figure 2. The API call dataset was initially tokenized before being passed through the FastText model to create embeddings for each call. As shown in Table 2, the lengths of each sample varied. To establish a standard vector size, the sample embeddings were averaged and combined to create a multi-dimensional array where each row represents one sample within a malware family.

We generated embeddings with three potential vector dimensions (88, 104, and 120) to determine the optimal vector length of these generated embeddings and evaluated them using a Random-Forest multi-class classifier. As shown in Table 4, the performance difference between 88 and 104 dimensions was marginal. We opted for 104 dimensions, since previous work [4] with similar models identified 104 dimensions as the optimal vector length.

Table 4. FastText Dimension Experimentation.

Dimension	F1 Score
88	0.97115
104	0.97022
120	0.96108

3.3.2. BERT

BERT [31] stands for Bidirectional Encoder Representations from Transformers. During the pretraining phase, it is trained on an English text corpus to create bidirectional representations of input data. In this way, it considers context before and after a target word and uses random masking to predict masked words based on this context. This pretrained model is then fine-tuned on other text corpora for downstream tasks.

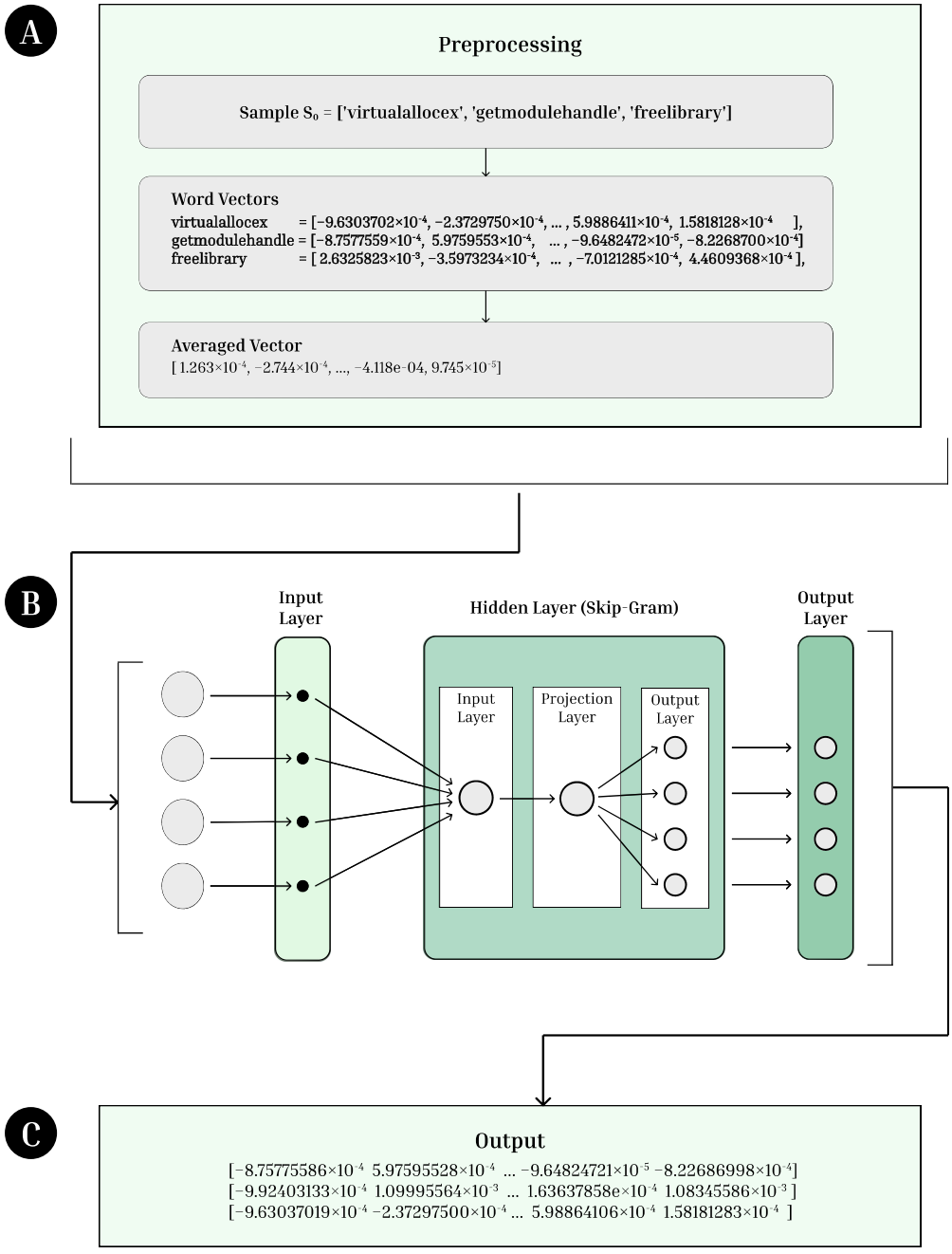

Figure 2. FastText Architecture. (**A**) API call embeddings are preprocessed and turned into word vectors. (**B**) The vectors are passed into the FastText model. After the input layer, they are processed through a hidden layer using Skip-Gram followed by an output layer. (**C**) Vectors of length 104 are outputted from the model.

The model's architecture is shown in Figure 3. We used the base BERT model in our work and fine-tuned it on the malware API call dataset. This model has a maximum input length of 512, but samples in the API dataset were significantly longer, with the maximum length of a given API call sequence being 17,364 (Table 2). So, after passing an API call sequence to our BERT model, embeddings were made by chunking the full sample sequences into chunks of length 512 and averaging them to yield one embedding per sample, of length 768. As with FastText, the sample embeddings were then concatenated to yield a multi-dimensional array with one row for each sample.

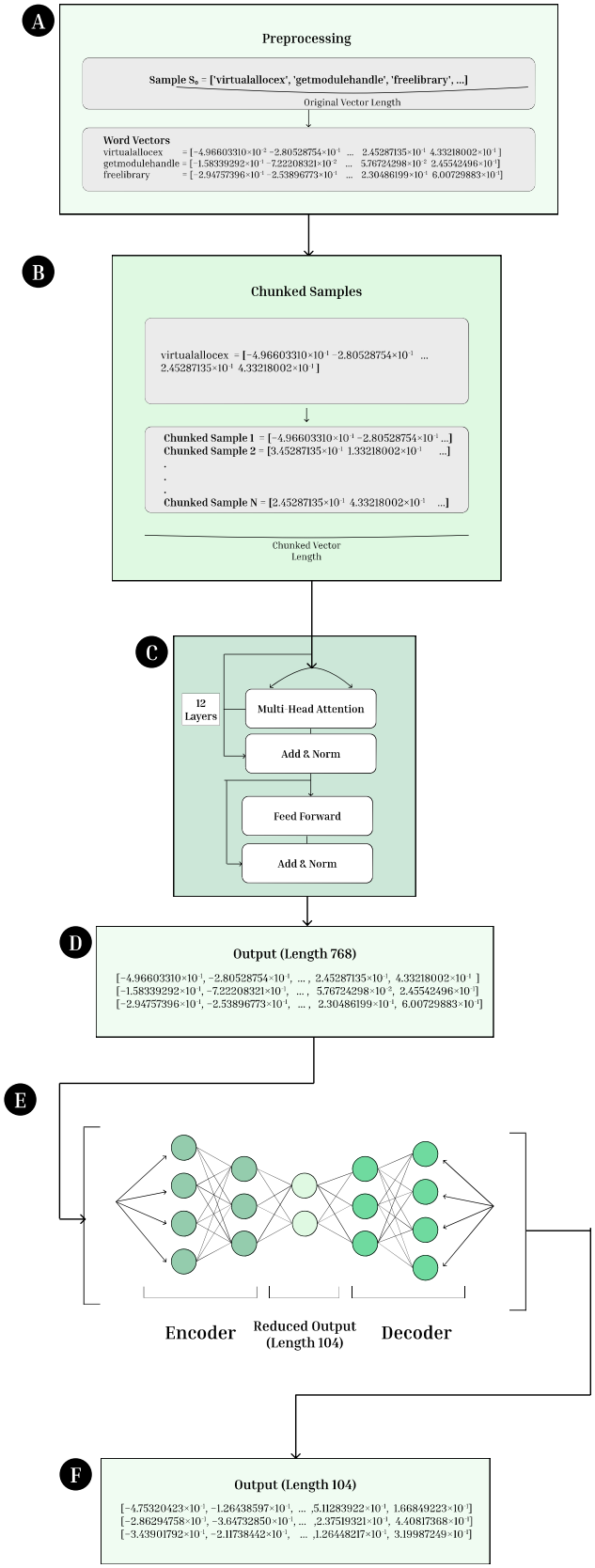

Figure 3. BERT Architecture (**A**) API call embeddings are preprocessed and turned into word vectors. (**B**) The vectors are chunked into chunks of length 512 due to limits on the length of vector that can be passed into the BERT model. (**C**) The vectors are passed into the BERT model, composed of attention layers, normalization layers, and feed forward layers. (**D**) Vectors of length 768 are outputted from the model. (**E**) The vectors are processed using an autoencoder to reduce the dimension of the output. (**F**) Final vectors of length 104 are outputted from the autoencoder.

BERT outputs vectors of length 768, and these embeddings can be visualized through t-SNE modeling. t-SNE [47] is used to visualize high-dimensional data by reducing it to lower dimensions while preserving data structure. Data that is similar in higher dimensions will cluster closely in lower dimensions to accurately represent the original data.

When we visualize the 768-dimensional vectors with t-SNE (Figure 4A), there is minimal clustering seen. This indicates a low embedding quality. So, we combined the embeddings with an autoencoder (Table 5).

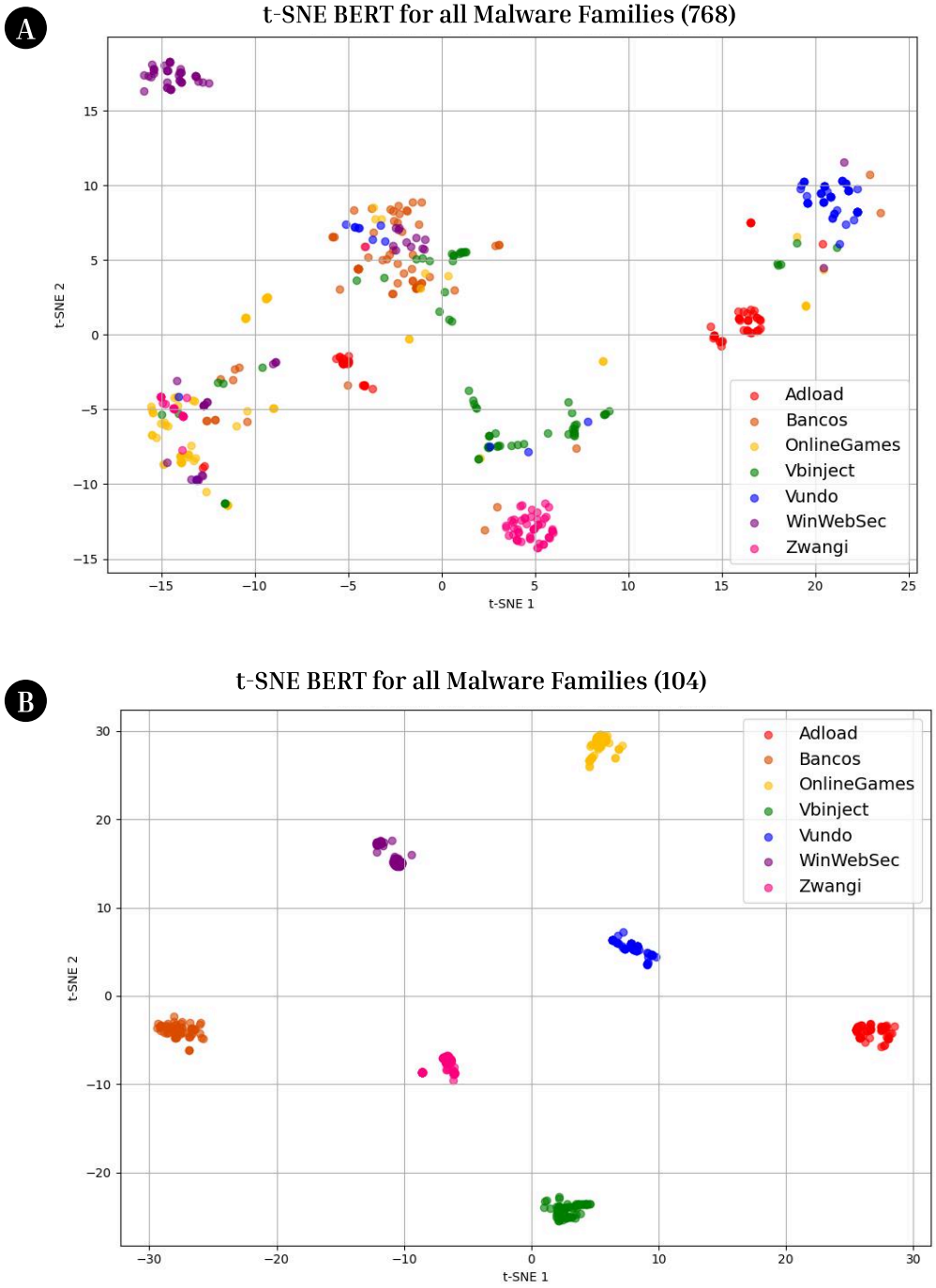

Figure 4. tSNE Plot of BERT Embeddings with Varying Vector Length. (**A**) tSNE plot with BERT embeddings of length 768. The clustering is minimal, indicating that they are not distinct by family. (**B**) tSNE plot with BERT embeddings of length 104. Distinct clusters are formed, indicating higher-quality and improved representation of the API call data.

We generated output embeddings with a length of 104, as previous work by Bao et al. [4] showed that an embedding length of 104 could yield high-quality synthetic malware samples when passed through a generative model. This also allowed us to compare the performance of the synthetic malware generated by the BERT model with that of the malware generated by FastText purely based on the way in which the model generated the embeddings and without the influence of factors such as vector length. This choice of embedding length was confirmed using a Random-Forest multi-class classifier. Results are shown in Table 6. When classified using a multiclass classifier, vectors of length 104 had an F1 score of 1.0 compared to an F1 score of 0.90632 for vectors of length 768.

Table 5. BERT Autoencoder Hyperparameters for Embedding Dimensionality Reduction.

Parameter	Value
Encoder Architecture	
Dense Layer 1	256 units, ReLU activation
Dense Layer 2	128 units, ReLU activation
Bottleneck Layer	104 units, Linear activation
Decoder Architecture	
Dense Layer 1	128 units, ReLU activation
Dense Layer 2	256 units, ReLU activation
Output Layer	768 units, Linear activation
Training Configuration	
Optimizer	Adam
Loss Function	Mean Squared Error (MSE)
Batch Size	8
Epochs	200
Validation Split	0.2
Early Stopping	Patience = 10 (restore best weights)
Train/Validation/Test Split	80%/10%/10%

Table 6. BERT Dimension Experimentation.

Dimension	F1 Score
768	0.90632
104	1.00000

Additionally, once the reduced embeddings were plotted using t-SNE (Figure 4B), distinct clusters are formed. This indicates improved representation of the API call data. Therefore, we chose to use the 104-dimensional vectors in the remainder of our experiments.

3.3.3. ELMo

Like BERT, ELMo [32] is a pretrained model that was trained on English text corpora to generate high-quality learned embeddings. It uses a bidirectional Long Short Term Memory (LSTM) to capture both the context and syntax of word meaning. In the forward pass, it considers context for a given word by looking at the tokens that come before it. In the backward pass, it considers context by looking at those that come after it.

The model's architecture is shown in Figure 5. We pass the API call input to the ELMo model for an output of one embedding per sample. Once sample-wide embeddings are created, they are then averaged and concatenated to yield a multi-dimensional array of sample embeddings. The output vector is of length 1024.

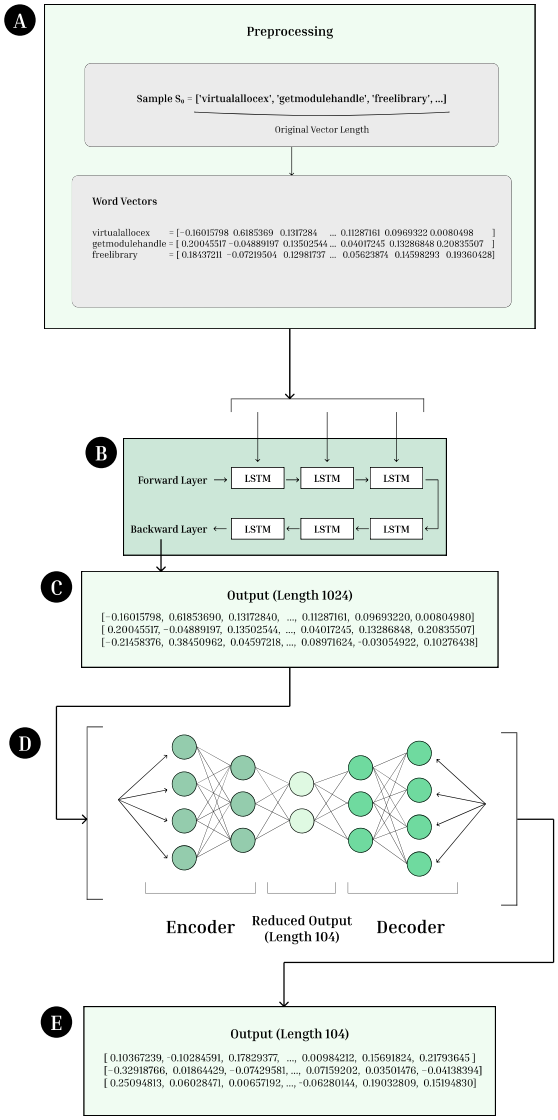

Figure 5. ELMo Architecture. (**A**) API call embeddings are preprocessed and turned into word vectors. (**B**) The vectors are passed into the ELMo model, composed of forward and backward Long Short Term Memory (LSTM) layers. (**C**) Vectors of length 1024 are outputted from the model. (**D**) The vectors are processed using an autoencoder to reduce the dimension of the output. (**E**) Final vectors of length 104 are outputted from the autoencoder.

As with BERT, the original 1024-dimensional ELMo embeddings were visualized through t-SNE and the embeddings showed minimal clustering. So, we processed them using another autoencoder model to output final embeddings of length 104. t-SNE results for these reduced embeddings showed formation of distinct clusters. Additionally, when the embeddings of length 104 and 1024 were classified using a a Random-Forest multi-class classifier, results showed that a vector length of 104 had a higher F1 than that achieved using vectors of length 1024 (Table 7). Consequently, we proceeded with the 104-dimensional embeddings.

Table 7. ELMo Dimension Experimentation.

Dimension	F1 Score
1024	0.90718
104	1.00000

3.4. Graph Embedding Creation

Before generating graph embeddings, we first convert the raw malware API call sequences into structured graph representations. To standardize the inputs, whitespace is removed from each API call and the call is mapped to a unique node index reflecting its temporal position in the sequence. Using these indices, directed edges are created between the consecutive API calls to model the flow of execution. Once the graph is encoded in the GraphML format, index prefixes are removed to maintain a standardized naming convention throughout files and families.

To convert the GraphML graphs into graph embeddings, two popular unsupervised graph embedding methods are used: Node2Vec and Graph2Vec [48]. These techniques vary in structural focus, with Node2Vec [38] creating embeddings for individual nodes and Graph2Vec [37] creating embeddings for entire graphs. Combining both Node2Vec and Graph2Vec embeddings provides a holistic representation of graph-structured data. Node2Vec focuses on the high-level understanding of each family's properties. In contrast, Graph2Vec extracts entire graph embeddings, which provide insights into how each graph connects to the family. Together, these complementary views highlight the relative importance of local versus global graph topology.

3.4.1. Node2Vec

Node2Vec [38] is a graph embedding technique that employs random walks to explore the neighborhood structure of each node (see Figure 6) These walks are treated similarly to sentences in Word2Vec [29], where they receive a learned vector representation through the Skip-Gram model. Nodes that have similar structures are mapped closely in the latent space.

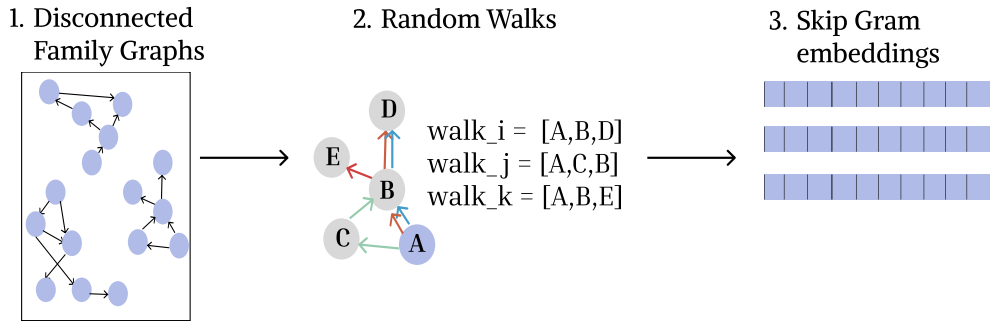

Figure 6. Node2Vec Embedding Procedure. (1) the disconnected family wide graph structure of a node2vec embedding (2) the various random walks possible from point A, and the skip gram model vectorization (3) the generated Skip Gram embeddings.

Node2Vec's embeddings are driven by two key principles: homophily, which captures shallow connections in a graph, and structural equivalence, which captures deeper connections in the graph. The trade-off between the two modes is controlled by the hyperparameters p and q, which bias the direction of the random walks.

Our objective is to understand inter-family relationships and how individual samples contribute to their families. To achieve this, we constructed a disconnected directed graph for each malware family, where each component corresponds to the behavior graph of a single malware sample.

The disconnected graph embeds all the samples within a family into a single shared latent space. This alignment accentuates the semantic features by preserving consistent structural patterns across the family. Components remain disconnected to preserve the structural independence of individual samples.

We use biased random walks with parameters (p = 2.0, q = 1.0) to capture both local and global structure. We selected the parameters p and q to induce moderately exploratory

random walks. Specifically, a higher p value discourages frequent backtracking, enabling the walks to traverse beyond immediate neighborhoods and capture broader structural relations, while a lower q value promotes homophilic exploration by favoring transitions within locally dense regions of the graph. After applying the Skip-Gram model to the walks, fixed-length graph-level embeddings are generated by applying mean pooling over the node embeddings.

3.4.2. Graph2Vec

Graph2Vec [37] is an embedding technique that creates graph-wide embeddings using a strategy inspired by Doc2Vec [49] (see Figure 7). Each graph is a single "document," and its rooted subgraphs represent the words. A rooted subgraph is a small portion of a graph centered at a specific node. Graph2Vec processes a corpus of graphs and builds a vocabulary of rooted subgraph labels, which are extracted using a Weisfeiler-Lehman (WL) relabeling strategy.

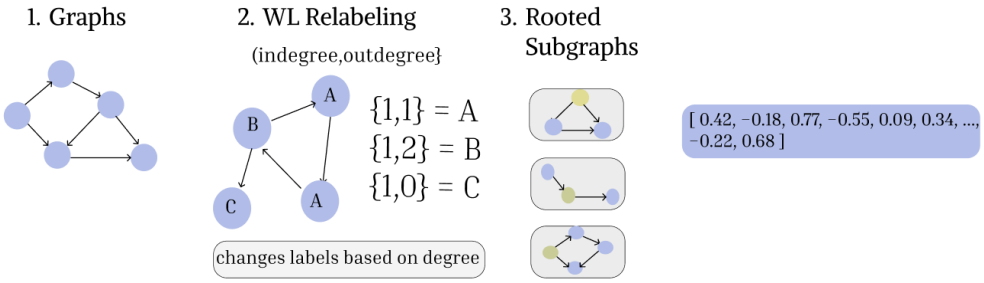

Figure 7. Graph2Vec Embedding Procedure. (1) input graphs passed into graph2vec (2) WL relabing algorithm creates new labels for the graphs (3) Rooted subgraphs extracted after relabling (4) rooted subgraphs used as words of document and embedded using Skip Gram model.

WL relabeling creates initial labels for each node based on its degree and iteratively changes the labeling of each node with respect to its neighbors. Each node's new label is derived by compressing its current label with neighbors' labels. The labels produced in the final iteration represent the tokenized rooted subgraphs. These subgraphs are treated like words, and the embeddings for each graph are learned by a Skip-Gram model that predicts tokenized subgraphs within it.

In our implementation, we trained a separate Graph2Vec [37] for each malware family, using all the samples in that family as a corpus. This approach emphasizes the variations among families, enabling generative models to learn family-specific patterns without interference from unrelated samples.

For data preprocessing, we parsed the GraphML files per family and mapped each node to a unified integer index. These standardized graphs were fed into Graph2Vec to generate fixed length graph-level embeddings of length 120. An embedding dimension of 120 was selected as it yielded the highest F1 score in our evaluation, as presented in Table 8.

Table 8. Graph Embedding F1 Scores by Embedding Dimension

Embedding Dimension	Graph Embedding F1 Score
88	0.96
104	0.96
120	0.98

3.5. Generative AI

Generative AI (Gen AI) models generate synthetic data that mimics the distribution of real data [13]. In this paper, we use three Gen AI models, WGAN-GP, Diffusion, and CT-

Diff, to generate synthetic malware from both NLP and graph-based embeddings. While both WGAN-GP and Diffusion have previously demonstrated success in synthetic malware generation, our work is the first of our knowledge to apply manifold diffusion to synthetic malware generation using a model we term CT-Diff.

3.5.1. Key Terminology

The Manifold Hypothesis states that high-dimensional data is concentrated around low-dimensional manifolds embedded in a high-dimensional space [50].

Related to this idea is the Curse of Dimensionality. As a dataset increases in dimension, the distance between data points increases as well, increasing the sparsity of the data.

3.5.2. WGAN-GP

WGAN-GP [51] models are composed of a generator and a critic. The generator inputs random noise values and outputs synthetic samples that mimic real data. The critic then determines whether the generated samples are real or synthetic, and provides updated weights to the generator, allowing it to generate new samples that more closely resemble real data (Figure 8). In this way, the generator generates increasingly realistic samples, and the critic improves at distinguishing between real and synthetic samples. WGAN-GP ensures high critic performance through a loss function called the Wasserstein Loss [51]. Furthermore, it enforces the Lipschitz constraint [51] by penalizing the norm of the gradient of the critic.

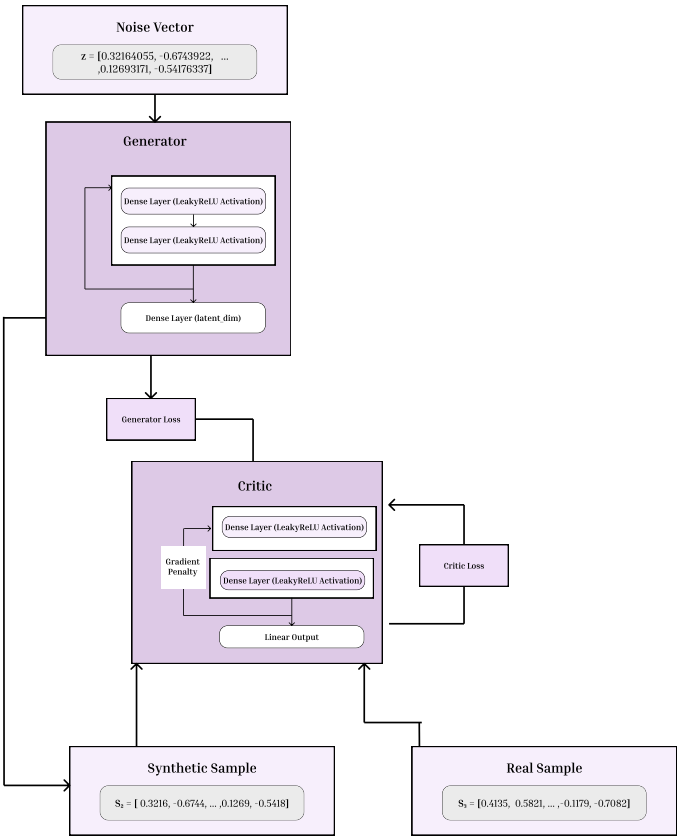

Figure 8. WGAN-GP Model Architecture The architecture begins with a random noise vector sampled from a latent distribution, which is transformed by the generator through successive dense layers to produce synthetic samples. These generated samples, along with real samples from the dataset, are evaluated by the critic network. The critic computes a Wasserstein distance–based loss and applies a gradient penalty to maintain Lipschitz continuity. The generator is updated to produce samples that the critic cannot distinguish from real data, progressively improving the realism of the synthetic embeddings.

Our WGAN-GP model (Table 9) contains three dense layers with LeakyReLU activation in both the generator and the critic. Our model was trained for 2000 epochs as our convergence plots showed no significant change in the behavior of the generator or critic beyond that point.

Table 9. WGAN-GP hyperparameters.

Parameter	WGAN-GP Value
Generator	
Activation	LeakyReLU
Hidden Layer 1	128 units
Hidden Layer 2	64 units
Hidden Layer 3	32 units
Batch Normalization	Enabled (momentum = 0.8)
Discriminator/Critic	
Activation	LeakyReLU ($\alpha = 0.2$)
Hidden Layer 1	128 units
Hidden Layer 2	64 units
Gradient Penalty (λ)	10
Training Configuration	
Optimizer	Adam
Learning Rate	0.0001
Betas	(0.5, 0.9)
Batch Size	64
Training Epochs	2000
Early Stopping	Patience = 100 epochs, $\Delta = 0.001$
Train/Validation/Test Split	80%/10%/10%

3.5.3. Diffusion

Diffusion [40] is a Gen AI model that generates synthetic data in two primary steps: forward diffusion and reverse diffusion (Figure 9). During forward diffusion, Gaussian noise is added to a sample until it is fully corrupted. During reverse diffusion, the noise is incrementally removed to yield a high-quality sample. The noising and denoising processes are stochastic Markov processes, which means that noising is done randomly and each step of the noising process solely depends on the step directly before it. We will further elaborate on this process next.

In forward diffusion, clean data is converted into noise using a Markov Chain model. A transition function is repeatedly applied to convert the image from its original distribution into a normal distribution over a series of timesteps according to the equation below [40].

$$q(X_t \mid X_{t-1}) := \mathcal{N}(X_t; \sqrt{1-\beta_t}\, X_{t-1}, \beta_t \mathbf{I}) \tag{1}$$

In the reverse process, noise is gradually removed from the noised sample, again using Markov Chains and normal gaussian probabilities. The reverse process follows the equation below [40].

$$p_\theta(x_{t-1} \mid x_t) := \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \Sigma_\theta(x_t, t)) \tag{2}$$

During the noising process, the function add gaussian noise to the sample through a normal distribution centered at 0 over a series of timesteps [40]. Given that the noise being added is a normal distribution and the original sample has a more complex distribution, when the noising process is reversed, it often results in unrealistic data trajectories [26] and samples being generated that lie outside of the data manifold [27]. This is visualized

in Figure 10B. For this reason, and due to the curse of dimensionality, samples generated through Diffusion may not align well with the original data passed into it. This is especially the case for our dataset, since each malware family solely contains 70 samples.

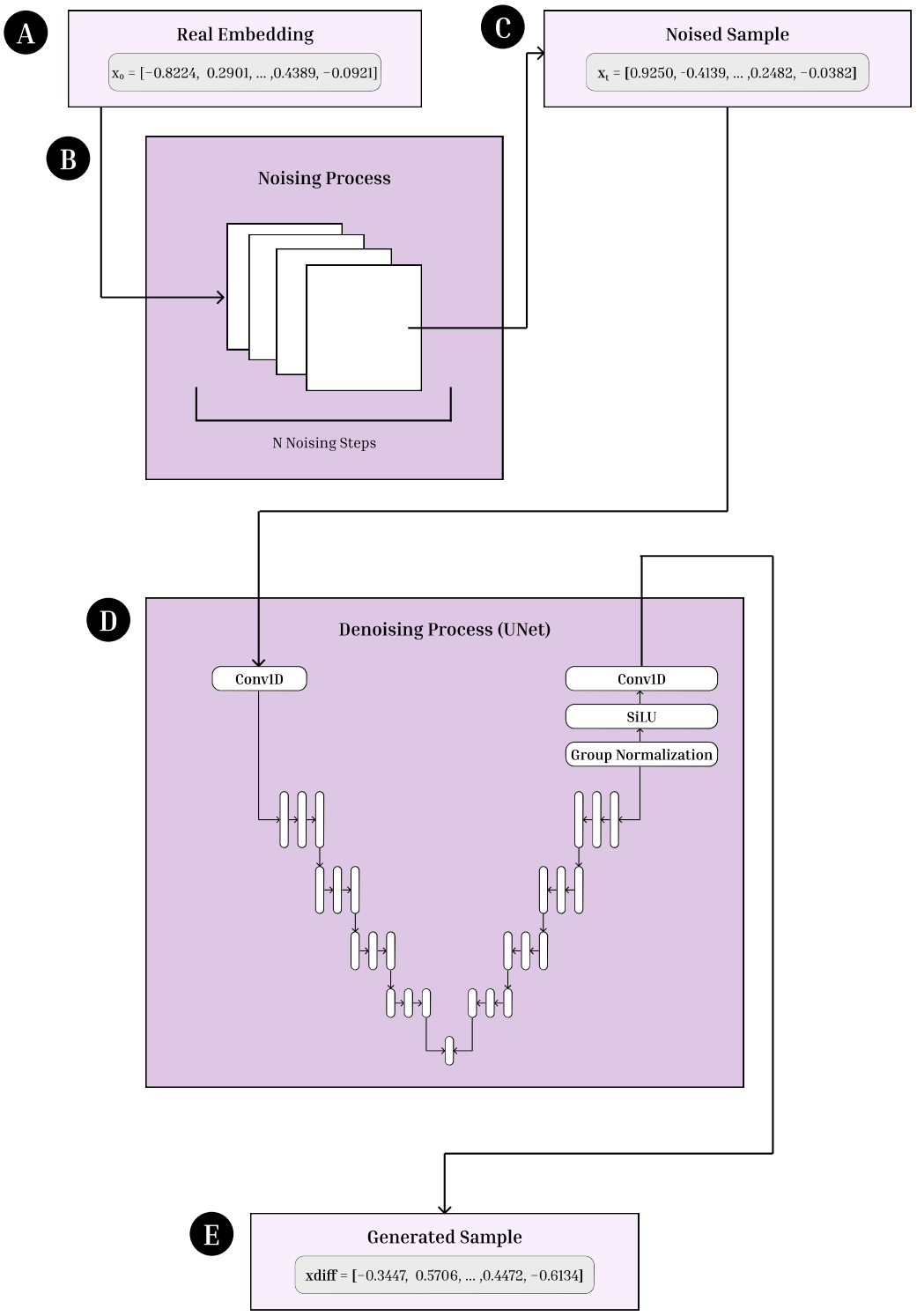

Figure 9. Diffusion Architecture. (**A**) A real embedding is passed into the (**B**) forward noising process, in which Gaussian noise is added to input samples to corrupt them. (**C**) A noised sample is outputted from this noising process and passed into a (**D**) reverse noising process, in which noise is gradually removed to reveal synthetic malware samples that closely resemble real malware samples. A U-Net is used to remove this noise. (**E**) Final vectors of length 104 are outputted from the Diffusion model.

We ran the diffusion model for a total of 10,000 noising steps and generated samples every 1000 noising steps. These samples were then saved to enable later evaluation and selection of the best-performing checkpoints for chosen metrics. The hyperparamters used in this research are summarized in Table 10.

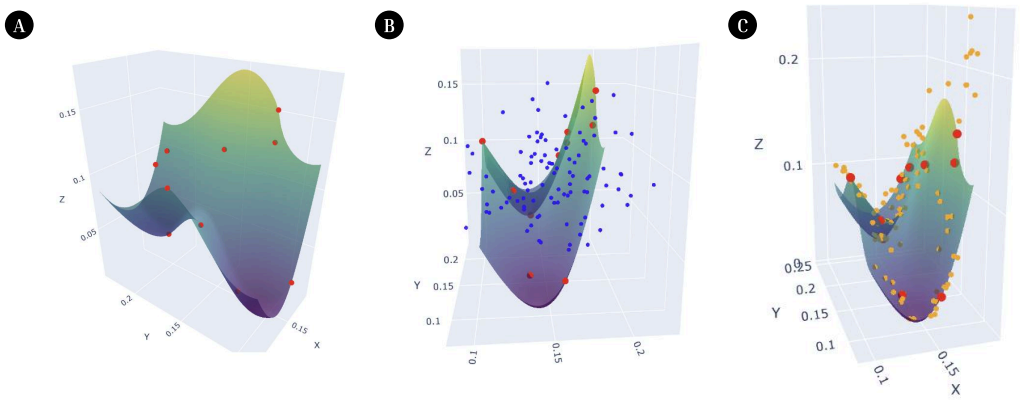

Figure 10. Manifolds Plots. This plot visualizes a selection of points from (**A**) Node2Vec output embeddings, (**B**) Diffusion noise, and (**C**) CT-Diff noise. This figure is a largely simplified version of the original data. For illustration purposes, it only visualizes 3 of the 120 original dimensions of the Node2Vec embeddings and 10 of its original data points. (**A**) The first three dimensions of 10 Node2Vec output embeddings are visualized as red points plotted in 3D space and a manifold is created to connect them. The manifold is visualized as the green and blue curved plane. (**B**) Diffusion noises samples using a normal Gaussian distribution. This is simulated through the blue dots. This noise largely falls outside of the manifold because the original data does not follow a normal distribution. (**C**) CT-Diff samples noise from the original distribution. Its noise is visualized using orange points, and they fall on or close to the original data's manifold.

Table 10. Diffusion hyperparameters.

Parameter	Details
Forward Diffusion	
Time Schedule	Cosine ($s = 0.008$), 1000 timesteps
Channels	32
Channel Multipliers	(1, 2, 4, 8)
Reverse Diffusion	
Layers	4 encoding, 4 decoding
Base Width	64
Attention Heads/Dim Head	4/32
Activation	Sigmoid Linear Unit (SiLU)
Normalization	RMSNorm
Training Configuration	
Optimizer	Adam
Learning Rate	8×10^{-4}
Betas	(0.9, 0.99)
EMA Decay	0.995
Gradient Accumulation	2 steps
Batch Size	64
Total Steps	8000
Train/Validation/Test Split	80%/10%/10%

3.5.4. Cluster-Tangent Diffusion (CT-Diff)

Cluster-Tangent Diffusion (CT-Diff) is a guided diffusion technique that generates synthetic samples that better preserve the structure of the original data distribution compared to Diffusion. It employs principles of manifold diffusion, which means that unlike standard Diffusion models that sample noise from an isotropic Gaussian distribution, it samples

structured noise concentrated around the manifold of the data (Figure 10C), allowing the generation process to better reflect the underlying data distribution. This results in more diverse and structurally aware samples that more accurately capture the true data distribution. Previous papers have employed various additional techniques to ensure that generated samples follow closely with the manifold, such as placing a penalty on the gradient [27]. In our model, we utilize a hybrid loss that ensures that the generated samples are aligned to the real samples by minimizing Euclidean and cosine distance. Additionally, to address underrepresented regions we add a Kernel Density Estimate(KDE) weight to our loss. This results in richer, structurally consistent synthetic samples. For a visualization of the system architecture refer to Figure 11. The hyperparamters used in this research are summarized in Table 11.

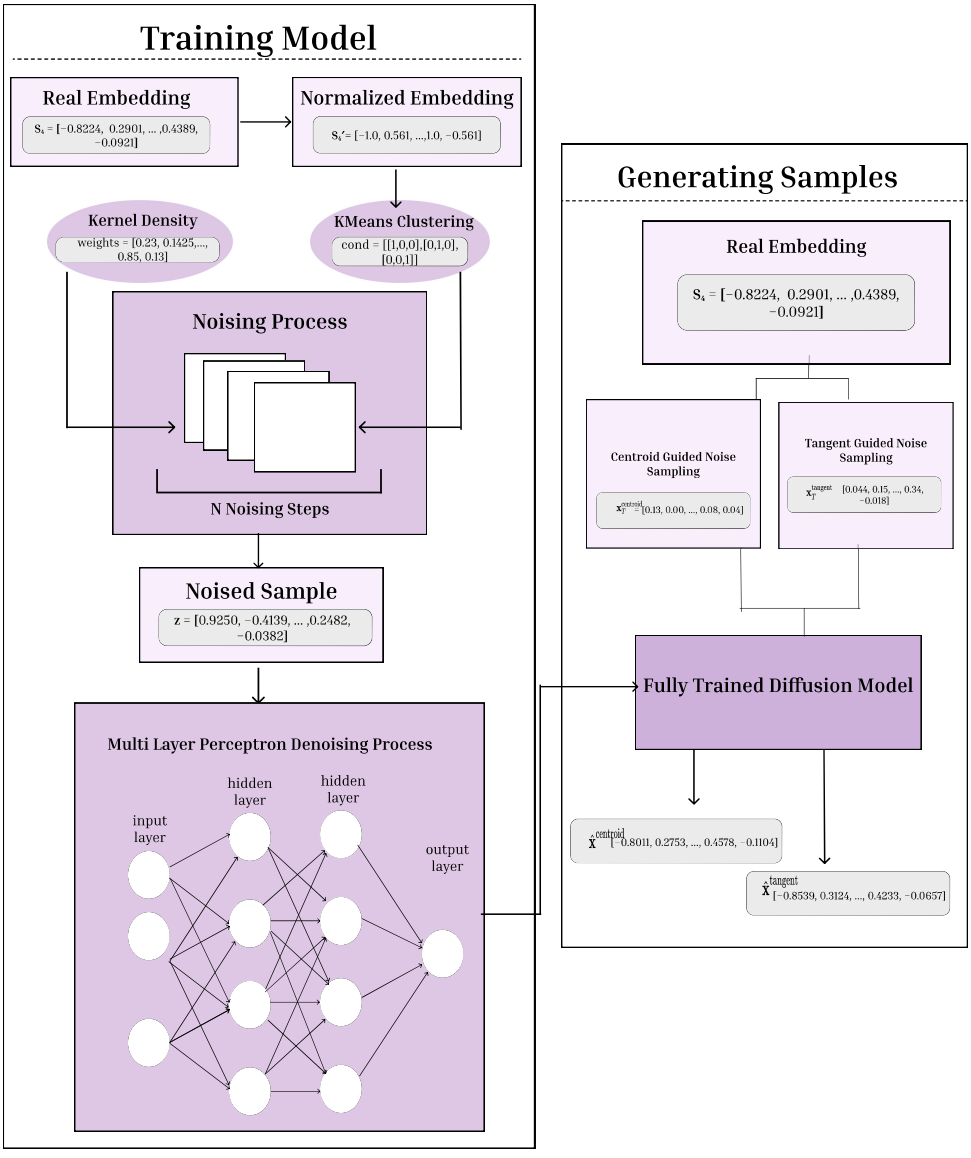

Figure 11. Cluster-Tangent Model Architecture. This diagram details the sampling process in both Cluster Guided and Tangent Guided Diffusion methods. The training segment builds the full trained diffusion model by applying noise to normalized embeddings and learning to denoise them with denoising MLP. KMeans clustering and KDE guide the noising process. In the generation phase synthetic embeddings are created by sampling noise from either centroid guided or tangent guided noise, then denoised using the trained diffusion model.

Forward Process

The forward noising steps follow the original Denoising Diffusion Probabilistic Models [40] paper. At each time step $t \in [0, T]$ noise is added to the clean sample (x_0) to produce a noisy version (x_t). Following the work of Nichol et al. [52] a cosine noise schedule was chosen for its demonstrated stability and efficiency in diffusion tasks.

Table 11. CT-Diff hyperparameters.

Parameter	Details
Forward Diffusion	
Time Schedule	Cosine ($s = 0.008$), 1000 timesteps
Reverse Diffusion (MLP Model)	
Hidden Layers	3 fully connected layers
Hidden Size	256 units per layer
Activation	ReLU
Training Configuration	
Optimizer	Adam
Learning Rate	1×10^{-3}
Epochs	1000
Loss Function	MSE + $0.5(1 - \text{CosSim})$
Train/Validation/Test Split	80%/10%/10%
Density Estimation (KDE)	
Kernel Function	Gaussian
Bandwidth	0.5
Tangent-Guided Sampling	
k (Nearest Neighbors)	10 (for local PCA)
Tangent Noise Scale	0.2 along local tangent direction
Drift Noise Mix	0.85 tangent + 0.15 orthogonal
Center Pull Factor	0.05 toward mean prediction

MLP Denoising

The main diffusion model in this architecture is a multi-layer perceptron (MLP) with ReLU activation. The decision to use a lightweight MLP over the traditional convolutional U-Net was guided by the nature of our data. We have a small non-image, tabular dataset, and prior works such as TabDDPM [53] have established that lightweight MLP denoising is well suited for small datasets. The MLP receives a noisy sample (x_t), the timestep (t), and the cluster condition (*cond*) and outputs a prediction of what the original sample (x_0) was.

Kernel Density Estimation (KDE)

To prevent mode collapse and encourage data diversity we leverage KDE in the loss. KDE approximates the Probability Density Function at point x_i.

The KDE estimate at any given point x_i is:

$$p(x_i) = \frac{1}{Nh^d} \sum_{j=1}^{N} K\left(\frac{\|x_i - x_j\|}{h}\right) \tag{3}$$

where K is a Gaussian kernel, d is the dimensionality of the data, and N is the number of samples.

We normalize the density scores and invert them to prioritize underrepresented areas:

$$\tilde{p}(x_i) = \frac{p(x_i)}{\max_j p(x_j)} \tag{4}$$

Loss Function

To guide the denoising process we employ a hybrid loss consisting of Mean Squared Error (MSE) and cosine similarity:

- MSE: measures the distance between the predicted and real sample:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(\hat{x}_{0,i} - x_{0,i})^2 \tag{5}$$

- Cosine Similarity: ensures directional consistency between predicted sample and the original sample by minimizing the angle between the original and synthetic sample.

$$\mathcal{L}_{\cos} = 1 - \cos(\hat{x}_0, x_0) \tag{6}$$

- Final loss: combination of MSE, cosine loss, and density-aware weighting.

$$\mathcal{L} = \tilde{p}(x_i) \cdot (\mathcal{L}_{\text{MSE}} + \mathcal{L}_{\cos}) \tag{7}$$

Sampling: Cluster and Tangent Initialization

Our data distributions follow two primary paradigms: cluster distributions and tangent distributions. For this reason, we employ two methods to generate samples.

- Cluster Guided Initialization: Cluster Guided Initialization is employed if the original t-SNE plots show tight clusters for each family.

  Initialize x_T near a KMeans cluster centroid:

$$x_T^{\text{centroid}} = \mu_c + \alpha \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I) \tag{8}$$

- μ_c: Centroid of cluster c, obtained from KMeans
- α: Noise scaling factor (controls spread around centroid)
- $\epsilon \sim \mathcal{N}(0, I)$: Standard isotropic Gaussian noise

- Tangent-Guided Initialization: Tangent-Guided Initialization is employed when the original t-SNE lacks well-formed clusters.

Initialize x_T near a local neighborhood midpoint, with noise along the PCA estimated tangent:

$$x_T^{\text{tangent}} = m_i + \beta \cdot z \cdot t_i, \quad z \sim \mathcal{N}(0, 1) \tag{9}$$

- m_i: Local midpoint for sample i, mean of its k-nearest neighbors
- t_i: local tangent direction at sample i, estimated via PCA on its neighborhood
- β: Tangent noise scaling factor
- $z \sim \mathcal{N}(0, 1)$: Scalar Gaussian noise

4. Evaluation

In order to evaluate the performance of the synthetic malware generated by the Gen AI models, we used a series of metrics, including multi-class classification, t-SNE modeling, cosine similarity, and cosine distance. Our multi-class classification models include Random Forest (RF), Support Vector Machine (SVM), and Multi-Layer Perceptron (MLP). Random Forest [54,55] is a classification technique that relies on the creation of trees. Random samples are selected from the original dataset with replacement, and classification trees are grown from the sampled data. SVM [16] instead classifies data by finding the optimal boundary, known as a hyperplane, that separates the data point classes. The larger the distance between the hyperplane and the data points, the better the classification. Finally,

MLP [4,56] is a fully-connected model that consists of an input layer, a series of hidden layers, and an output layer that process the input data to output a final prediction.

t-SNE [47] is a modeling technique that visualizes high-dimensional data in a lower number of dimensions while preserving data structure. It clusters similar points in this lower representation such that the clusters are representative of the original representations.

Cosine similarity [4,57] measures the similarity between two vector embeddings by calculating the angle between them. A high cosine similarity (close to 1) indicates that the vectors are similar, while a low cosine similarity (close to 0) indicates that they are dissimilar. We aim for a high cosine similarity between the real and generated embeddings.

Cosine distance measures how distinct two vectors are from one another and may be calculated as [58]

$$\text{Cosine Distance} = 1 - \text{Cosine Similarity} \tag{10}$$

A low cosine distance (close to 0) indicates that the vectors are identical, while a high cosine distance (close to 1) indicates that the vectors are distinct [59]. When measuring the cosine distance between the real and generated embeddings, we hope to generate synthetic embeddings that point in a similar direction to the real embeddings but that are not identical.

When performing multiclass classification with the generative models, we tested a variety of train and test splits between the real and fake data. We report on three of them, as detailed below:

- Train: 1.0 real, 0.0 fake | est: 0.0 real, 1.0 fake—This split measures how well the model can classify fake data when only trained on real data. A high F1 score indicates that the fake data is realistic.
- Train: 0.8 real, 0.2 fake | Test: 0.2 real, 0.8 fake—This split tests the model's performance when real data is augmented with fake data.
- Train: 0.0 real, 1.0 fake | Test: 1.0 real, 0.0 fake—This split was chosen as it measures how well the model can classify real data when only trained on fake data. If the F1 results from this metric are similar to those from when we train on all real data and test on all fake data, then results are consistent. A high F1 score indicates that the fake data is realistic.

While multi-class classification results will be shown for all embedding types, the results for the remaining metrics will only be shown for select, high-performing models.

4.1. Autoencoder

We evaluated the use of an autoencoder in the synthetic malware generation process by comparing the malware generated from embeddings from BERT and ELMo with those generated by FastText. We generated this malware using three generative models, WGAN-GP, Diffusion, and CT-Diff, and will be discussing results from all three in this section.

Multiclass classification of WGAN-GP is reported in Table 12. ELMo has the highest F1 score among the NLP embedding techniques, although BERT comes in second as a close match. Similarly, for Diffusion (Table 13), while performance varied based on the train-test split, one of the highest performing NLP models was BERT. Finally, as with WGAN-GP, for CT-Diff, ELMo was the highest performing NLP model (Table 14). In all three cases, the augmented pretrained models achieved the highest F1 scores. This suggests that processing the outputs of pretrained NLP models using an autoencoder can capture more informative structural patterns and can improve generative performance in data scarce scenarios.

Table 12. WGAN-GP Multiclass Classification F1 Scores.

Embedding Type	RF	SVM	MLP
Train: 100% Real—Test: 100% Fake			
FastText	0.6199	1.0000	1.0000
ELMo	0.7983	1.0000	1.0000
BERT	0.6592	1.0000	1.0000
Node2Vec	0.7858	1.0000	1.0000
Graph2Vec	0.5448	0.7764	0.7736
Train: 80% Real, 20% Fake—Test: 20% Real, 80% Fake			
FastText	0.8100	1.0000	1.0000
ELMo	0.6417	1.0000	1.0000
BERT	0.8069	1.0000	1.0000
Node2Vec	0.7831	1.0000	1.0000
Graph2Vec	0.6682	1.0000	1.0000
Train: 100% Fake—Test: 100% Real			
FastText	0.6628	0.9818	0.9837
ELMo	0.6788	0.9980	1.0000
BERT	0.6109	1.0000	1.0000
Node2Vec	0.7666	1.0000	1.0000
Graph2Vec	0.4224	0.6965	0.7435

Table 13. Diffusion Multiclass Classification F1 Scores.

Embedding Type	RF	SVM	MLP
Train: 100% Real—Test: 100% Fake			
FastText	0.5040	0.9571	0.9595
ELMo	0.4531	0.6730	0.7378
BERT	0.5600	0.9335	0.9877
Node2Vec	0.6929	0.9812	0.9676
Graph2Vec	0.0357	0.9382	0.9837
Train: 80% Real, 20% Fake—Test: 20% Real, 80% Fake			
FastText	0.6068	0.9980	1.000
ELMo	0.7966	1.000	1.000
BERT	0.7638	1.000	1.000
Node2Vec	0.6920	1.000	1.000
Graph2Vec	0.5724	0.9980	1.000
Train: 100% Fake—Test: 100% Real			
FastText	0.7280	0.8658	0.9649
ELMo	0.8068	0.2115	0.9979
BERT	0.8080	0.0354	1.000
Node2Vec	0.7805	0.8526	0.9788
Graph2Vec	0.5451	0.9192	0.9772

Cosine similarity results for ELMo indicate that the generated samples are relatively similar to the real embeddings, with ELMo achieving a maximum cosine similarity of 0.68 for Diffusion (Figure 12) and a maximum cosine similarity of 0.98 for CT-Diff (Figure 13). Since CT-Diff had a higher score than Diffusion, synthetic malware generated using the model is more realistic than those generated using Diffusion. This can be explained by the t-SNE results for ELMo (Figure 14). In this plot, Diffusion is highlighted as purple dots, CT-Diff is visualized as red dots, and the original embeddings are shown as blue dots. The red dots cluster close to the original blue dots, while the purple dots do not. This tight

clustering indicates that CT-Diff is more effectively able to capture the structure of the real data than Diffusion. Regardless, as combining the augmented ELMo model with a strategic choice of generative model allows for the generation of synthetic malware with a high degree of similarity to the original data, autoencoders are an effective method of improving the quality of embeddings generated from pretrained NLP models.

Table 14. CT-Diff Multiclass Classification F1 Scores.

Embedding Type	RF	SVM	MLP
Train: 100% Real—Test: 100% Fake			
FastText	0.8025	1.0000	1.0000
ELMo	0.8270	1.0000	1.0000
BERT	0.6851	1.0000	1.0000
Node2Vec	0.8095	1.0000	1.0000
Graph2Vec	0.8137	1.0000	1.0000
Train: 80% Real, 20% Fake—Test: 20% Real, 80% Fake			
FastText	0.8095	1.0000	1.0000
ELMo	0.6287	1.0000	1.0000
BERT	0.7098	1.0000	1.0000
Node2Vec	0.8071	1.0000	1.0000
Graph2Vec	0.8117	1.0000	1.0000
Train: 100% Fake—Test: 100% Real			
FastText	0.6126	0.9959	0.9838
ELMo	0.7458	1.0000	1.0000
BERT	0.6229	1.0000	1.0000
Node2Vec	0.5540	1.0000	1.0000
Graph2Vec	0.8079	1.0000	1.0000

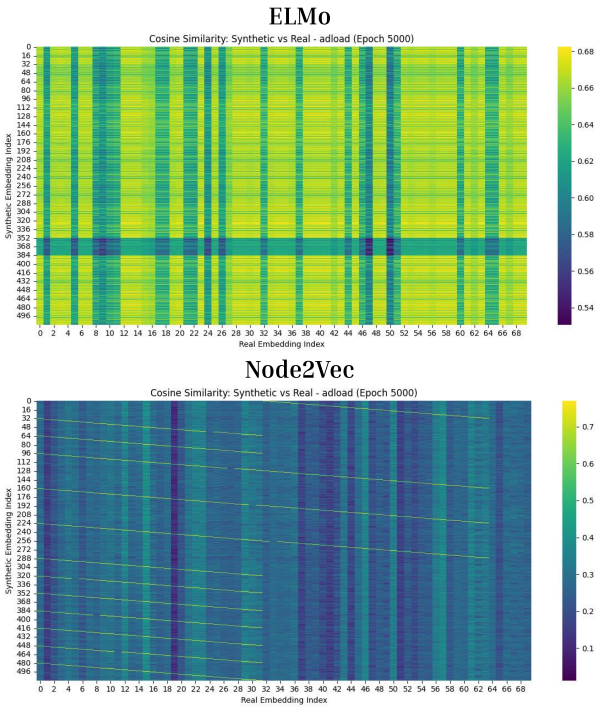

Figure 12. Cosine Similarity Results for Diffusion The cosine similarity results are shown for Epoch 5000 for the Adload malware family for ELMo and Node2Vec. ELMo achieved a maximum cosine similarity of 0.68, while Node2Vec achieved a maximum cosine similarity of around 0.7. This indicates that using graph embedding techniques with Diffusion yields synthetic malware that more closely aligns with the original data.

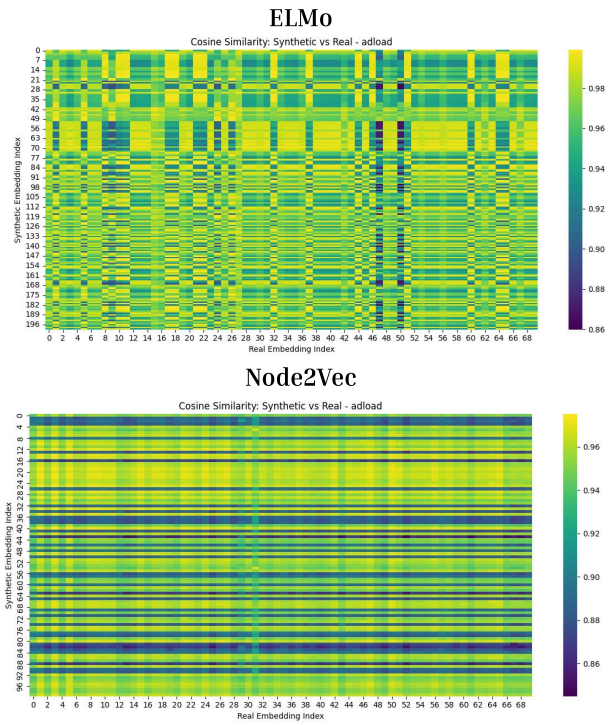

Figure 13. Cosine Similarity Results for CT-Diff. This figure presents cosine similarity for CT-generated ELMo and Node2Vec embeddings. The light colors in the heatmap indicate consistently high (0.98) similarity across most samples , with occasional dark streaks where similarity dips to 0.86. These results indicate strong generated samples.

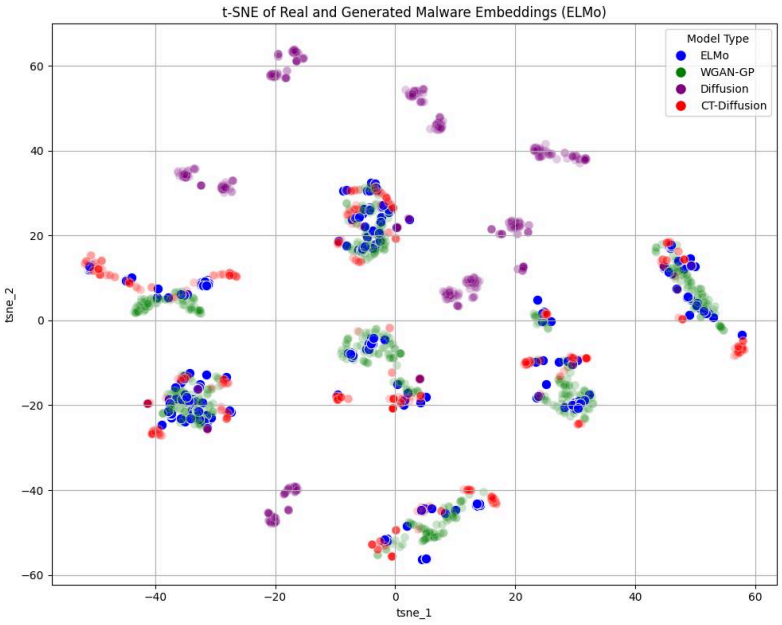

Figure 14. t-SNE Results for ELMo. This figure displays the t-SNE results with the original ELMo embeddings and the synthetic malware generated by WGAN-GP, Diffusion, and CT-Diff. The CT-Diff embeddings closely follow the original ELMo embeddings, with WGAN-GP performing similarly. Diffusion clusters, but its embeddings do not align well with those of the ELMo embeddings.

4.2. CT-Diff Evaluation

To evaluate the performance of CT-Diff we used the same metrics from the autoencoder. Table 14 reports the F1 scores across various training and testing splits in multiclass classification. CT-Diff notably improves the F1 scores of embedding types that previously

underperformed with standard Diffusion specifically, FastText, and Graph2Vec. For the remaining embedding types, CT-Diff either maintains or marginally improves the F1 scores.

These trends are visually reinforced by the t-SNE plots in Figures 14 and 15, which show that the generated samples closely follow the data distribution of real embeddings. Specifically in Figure 15 the Node2Vec t-SNE we can see the "anchoring" of CT-Diff's generations (red circles). Diffusion (purple circles) generations cluster off in space somewhere, but CT-Diff is able to center the generations at the center of the data distribution. This same principle is shown in Figure 14 the ELMo t-SNE. The CT-Diff generations very closely follow the shape of the original embeddings.

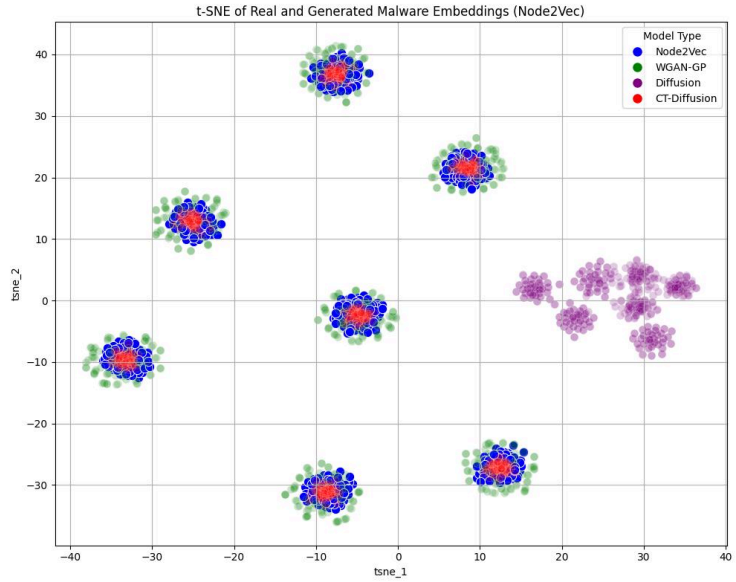

Figure 15. t-SNE Results for Node2Vec. This figure displays the t-SNE results with the original Node2Vec embeddings and the synthetic malware generated by WGAN-GP, Diffusion, and CT-Diff. The Diffusion syntheticsamples form strong clusters but are not close to the original data. The CT-Diff clusters depict how CT anchors generation to the create stronger generations. WGAN-GP performs similarly.

This demonstrates that the manifold sampling technique used in CT-Diff improves diffusion's ability to generate coherent samples even under data scarcity. The cosine similarity results in Figure 13 all exceed 0.95, indicating CT-Diff's strong capabilities in preserving intra-family relationships during generation.

Given that CT-Diff performed at a higher level than the other generative techniques, it was important to verify that generated embeddings were distinct from one another and from the original embeddings. Accordingly, we ran cosine distance (Figure 16) on the real and synthetic embeddings for all embedding types. As with the other metrics, we report on the results for ELMo and Node2Vec. The cosine distance values varied by malware family, with mean cosine distance for ELMo ranging from 0.0330 for Adload to 0.3115 for WinWebSec. Node2Vec's mean cosine distance conversely ranged from 0.3595 for Adload to 0.3658 for OnlineGames. These cosine distance values are relatively low, which indicates that the generated embeddings are similar to the real ones. However, no exact matches nor matches within a tolerance of e^{-6} were found for any malware family in any of the original embedding types. This means that the fake embeddings generated using CT-Diff are highly similar to the real malware embeddings but are distinct. Node2Vec's cosine distance is also in a much narrower range than those of ELMo, so graph embedding techniques lead to more consistent synthetic malware samples, regardless of malware family.

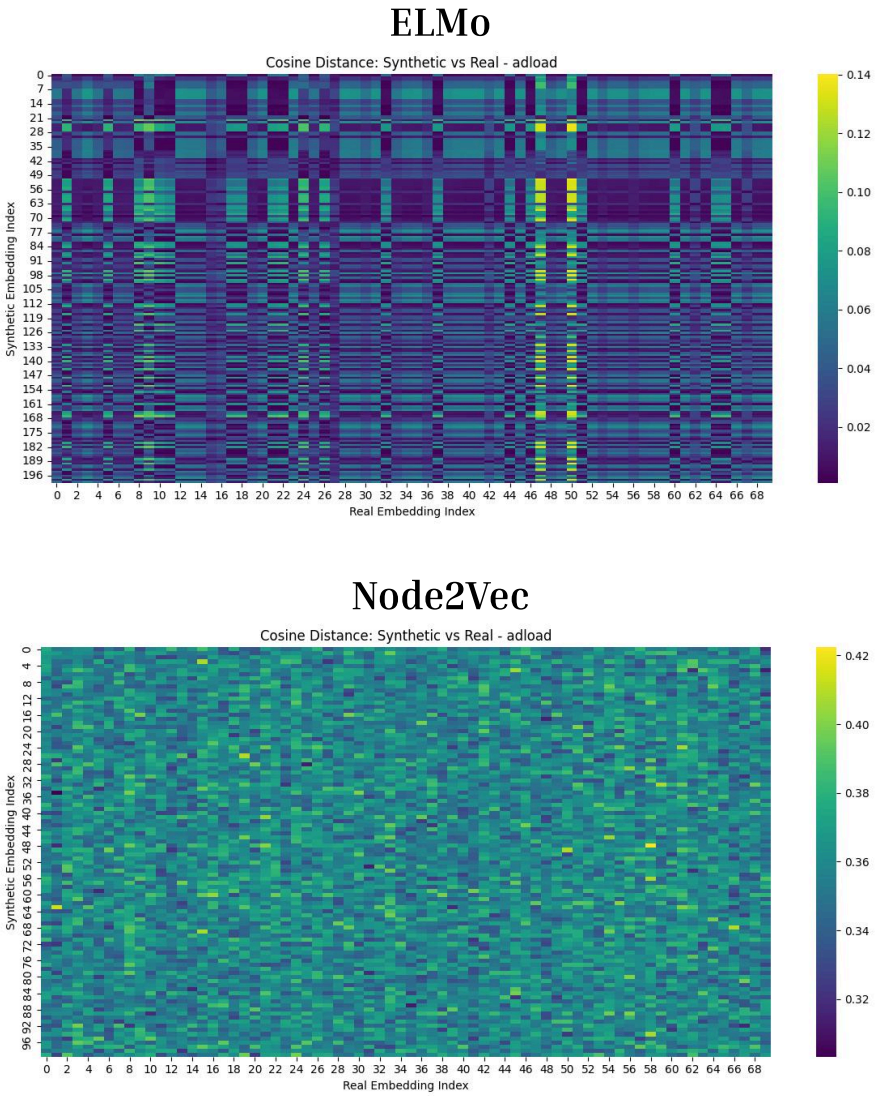

Figure 16. Cosine Distance Results for CT-Diff This figure shows the results of running Cosine Distance on the synthetic malware samples generated by CT-Diff for ELMo and Node2Vec. For Adload, ELMo achieves a mean cosine distance of 0.0330, while Node2Vec achieves a mean cosine distance of 0.3595. This indicates that generated samples are similar to the original embeddings.

5. Discussion

The results of our evaluation indicate that combining an autoencoder with pretrained NLP embedding techniques (BERT and ELMo) substantially improves both the embedding quality and generative capacity of these representations. Unlike previous applications of BERT and ELMo to malware domains, which often relied on directly applying the pretrained models to domains outside of the corpora or extensive fine tuning of models, the autoencoder is able to adapt the embeddings to the context more efficiently. This provides richer embeddings tailored to the malware domain and the generation of synthetic samples that are both more realistic and more diverse than those produced in prior studies using BERT and ELMo. Furthermore, our experiments demonstrate that CT-Diff is the most effective generative model. Its advantage over Vanilla Diffusion lies in constraining the generation process over the tangent directions of the original data manifold. By operating in this lower-dimensional space, CT-Diff captures essential features more effectively and produces samples that are more representative. This manifold alignment also mitigates the impact of the curse of dimensionality as the model focuses only on the structured regions where meaningful variation occurs.

5.1. Limitations

One limitation of this work is that our original dataset was small: 70–71 samples per malware family. As such, models were unable to perform as optimally as they may have with a larger dataset. The study conducted by Bao et al. [4], for example, utilized 40,000 samples and 25 malware families, and thus achieved F1 scores of close to 1 for classification with GAN, WGAN-GP, and Diffusion. Using a larger dataset of API call samples may have yielded higher F1 scores in this case as well. However, given that some of our models were still able to achieve high performance on this dataset, some generative models, namely CT-Diff, may still be able to capture the structure and distribution of input data when presented with scarce datasets.

Additionally, our work generates embeddings and synthetic malware samples using malicious API call samples alone, rather than comparing them with benign samples. This follows the setup in Bao et al.'s [4] work. Including benign samples as a benchmark would provide a deeper understanding into the ways in which benign API call samples differ from malicious counterparts and provide an additional way to asses the quality of the synthetic malware samples. Additionally, it would enable a more robust assessment of separability in mixed operational environments and help mitigate potential overfitting to malware-specific characteristics. This makes it a promising direction for future research.

This study specifically targets the challenge of data scarcity, where the availability of malware samples is limited (as in zero-day attacks). While the approach is optimized for low-data scenarios, future work should investigate the scalability of the model on larger and more diverse malware datasets to evaluate its robustness.

Finally, it is important to note that while high cosine similarity between real and synthetic samples indicates effective structural preservation, excessive similarity may lead to retraining bias, where models reproduce existing data patterns rather than improving generalization. In future work, it would be valuable to explore the balance between similarity and diversity, as well as to incorporate additional evaluation metrics beyond cosine distance to ensure that synthetic samples introduce meaningful variation rather than duplicating real data characteristics.

5.2. Future Work

In this paper, we focused our exploration on the use of encoders and manifold diffusion. Another area that showed promise was graph-based embedding techniques. When combined with CT-Diff and other generative models, Node2Vec and Graph2Vec achieved results similar to our augmented ELMo and BERT models. One area of future exploration would be to study ways to further augment these graph-based embedding techniques to optimize their embeddings and generate high-quality malware samples. Furthermore, future work could not only use graph-based representations of malware, but also use graph-based malware generation techniques, such as GraphGAN. By utilizing generative models that not only take graph embeddings as input but also generate synthetic malware represented as graphs, we would be able to consider whether generating malware with a graph structure more accurately preserves its features than non-graph-based generation techniques.

This line of research may aid in potentially generating approximations of zero-day attack patterns. Given that zero-day attacks are inherently rare and datasets representing these patterns are underrepresented, the ability to generate and simulate behaviors from scarce data is a promising direction. To evaluate this potential, future work may focus on fine-tuning the models and assessing the extent to which the synthetic samples generalize the patterns of real zero-day attacks.

Finally, while synthetic malware generation can be beneficial for improving malware detection, it can also have dangerous consequences if used for unethical uses. Existing real

malware samples could be replicated on a large scale by malicious parties with minimal effort for use in coordinating successful attacks on important systems. Additionally, models used to develop the malware may not align with principles of transparency, impartiality, and human oversight, leading to inaccurate or inappropriate outputs [60]. Future work may further explore potential risks associated with our generation pipeline.

6. Conclusions

In this work, we explored ways to generate high-quality synthetic malware by optimizing embeddings both during the embedding and generation processes. We evaluated whether combining an autoencoder with pretrained NLP models (BERT and ELMo) would yield optimal results across different generative settings, by comparing the results with other NLP and graph embedding models. Furthermore, we assessed the strength of using a manifold diffusion approach to generate fake malware samples through our Cluster-Tangent Diffusion (CT-Diff) model, and compared it to two generative frameworks: WGAN-GP and Diffusion.

Our results find that the autoencoder and NLP combination enhances both embedding quality and generative capacity, outperforming previous applications of pretrained NLP techniques in malware analysis. Moreover CT-Diff outperformed Vanilla Diffusion in classification accuracy.

Beyond the methodological advancements, this work has practical applications for cybersecurity practitioners and malware analysts. The synthetic samples generated through CT-Diff can be used to augment scarce or imbalanced datasets, improving the robustness of malware classifiers, and potentially simulate zero-day attack behaviors. This work provides a safe method to gather data to train and test robust malware detection systems.

Author Contributions: Conceptualization, F.D.T.; methodology, F.D.T., G.K., and S.N.; software, G.K. and S.N.; validation, F.D.T.; formal analysis, F.D.T., G.K., and S.N.; investigation, F.D.T., G.K., and S.N.; resources, F.D.T.; data curation, G.K. and S.N.; writing—original draft preparation, G.K. and S.N.; writing—review and editing, F.D.T.; visualization, G.K. and S.N.; supervision, F.D.T.; project administration, F.D.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: https://drive.google.com/file/d/1EO46egj_Rtqj44EpCpF0u7zBEwLmJ4qa/view?usp=sharing.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Namanya, A.P.; Cullen, A.; Awan, I.U.; Disso, J.P. The world of malware: An overview. In Proceedings of the 2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud), Barcelona, Spain, 6–8 August 2018; pp. 420–427.
2. Tchakounté, F.; Hayata, F. Supervised learning based detection of malware on android. In *Mobile Security and Privacy*; Elsevier: Amsterdam, The Netherlands, 2017; pp. 101–154.
3. Martins, A.M.; Moutinho, N. Stock-Term market impact of major cyber-attacks: Evidence for the ten most exposed insurance firms to cyber risk. *Financ. Res. Lett.* **2025**, *71*, 106361. [CrossRef]
4. Bao, T.; Trousil, K.; Tran, Q.D.; Di Troia, F.; Park, Y. Generating Synthetic Malware Samples using Generative AI against Zero-day Attacks. *IEEE Access* **2025**, *13*, 59725–59736. [CrossRef]
5. Yumlembam, R.; Issac, B.; Jacob, S.M.; Yang, L. Iot-based android malware detection using graph neural network with adversarial defense. *IEEE Internet Things J.* **2022**, *10*, 8432–8444. [CrossRef]
6. Pektaş, A.; Acarman, T. Deep learning for effective Android malware detection using API call graph embeddings. *Soft Comput.* **2020**, *24*, 1027–1043. [CrossRef]

7. Malhotra, V.; Potika, K.; Stamp, M. A comparison of graph neural networks for malware classification. *J. Comput. Virol. Hacking Tech.* **2024**, *20*, 53–69. [CrossRef]
8. Chen, Y.H.; Chen, J.L.; Deng, R.F. Similarity-based malware classification using graph neural networks. *Appl. Sci.* **2022**, *12*, 10837. [CrossRef]
9. Park, Y.; Reeves, D.; Mulukutla, V.; Sundaravel, B. Fast malware classification by automated behavioral graph matching. In Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research, Oak Ridge, TN, USA, 21–23 April 2010; pp. 1–4.
10. Faruki, P.; Laxmi, V.; Gaur, M.S.; Vinod, P. Mining control flow graph as api call-grams to detect portable executable malware. In Proceedings of the Fifth International Conference on Security of Information and Networks, Hamburg, Germany, 20–21 October 2012; pp. 130–137.
11. Gülmez, S.; Sogukpinar, I. Graph-based malware detection using opcode sequences. In Proceedings of the 2021 9th International Symposium on Digital Forensics and Security (ISDFS), Elazig, Turkey, 28–29 June 2021; pp. 1–5.
12. Singh, A.; Arora, R.; Pareek, H. Malware analysis using multiple API sequence mining control flow graph. *arXiv* **2017**, arXiv:1707.02691. [CrossRef]
13. Gebrehans, G.; Ilyas, N.; Eledlebi, K.; Lunardi, W.T.; Andreoni, M.; Yeun, C.Y.; Damiani, E. Generative Adversarial Networks for Dynamic Malware Behavior: A Comprehensive Review, Categorization, and Analysis. *IEEE Trans. Artif. Intell.* **2025**, *6*, 1955–1976. [CrossRef]
14. Khan, F.B.; Durad, M.H.; Khan, A.; Khan, F.A.; Chauhdary, S.H.; Alqarni, M. Detection of data scarce malware using one-shot learning with relation network. *IEEE Access* **2023**, *11*, 74438–74457. [CrossRef]
15. Bansal, M.A.; Sharma, D.R.; Kathuria, D.M. A systematic review on data scarcity problem in deep learning: Solution and applications. *ACM Comput. Surv. (Csur)* **2022**, *54*, 1–29. [CrossRef]
16. Kale, A.S.; Pandya, V.; Di Troia, F.; Stamp, M. Malware classification with word2vec, hmm2vec, bert, and elmo. *J. Comput. Virol. Hacking Tech.* **2023**, *19*, 1–16.
17. Aggarwal, S.; Di Troia, F. Malware Classification Using Dynamically Extracted API Call Embeddings. *Appl. Sci.* **2024**, *14*, 5731. [CrossRef]
18. Maniriho, P.; Mahmood, A.N.; Chowdhury, M.J.M. API-MalDetect: Automated malware detection framework for windows based on API calls and deep learning techniques. *J. Netw. Comput. Appl.* **2023**, *218*, 103704. [CrossRef]
19. Amer, E.; Zelinka, I. A dynamic Windows malware detection and prediction method based on contextual understanding of API call sequence. *Comput. Secur.* **2020**, *92*, 101760. [CrossRef]
20. Qiao, Y.; Zhang, W.; Du, X.; Guizani, M. Malware classification based on multilayer perception and Word2Vec for IoT security. *ACM Trans. Internet Technol. (TOIT)* **2021**, *22*, 1–22. [CrossRef]
21. Yesir, S.; Soğukpinar, İ. Malware detection and classification using fasttext and bert. In Proceedings of the 2021 9th International Symposium on Digital Forensics and Security (ISDFS), Elazığ, Turkey, 28–29 June 2021; pp. 1–6.
22. Tran, Q.D.; Di Troia, F. Word Embeddings for Fake Malware Generation. In Proceedings of the Silicon Valley Cybersecurity Conference, Virtually, 17–19 August 2022; pp. 22–37.
23. Mollah, M.S.H.; Marhusin, M.F.B.; Omar, S.N. A Robust Malware Detection Framework Using Control Flow Graphs, Node2Vec, and GCN Integration. In Proceedings of the 2025 International Conference on Electrical, Computer and Communication Engineering (ECCE), Chittagong, Bangladesh, 13–15 February 2025; pp. 1–6.
24. McLaren, R.A.; Babaagba, K.O.; Tan, Z. A generative adversarial network based approach to malware generation based on behavioural graphs. In Proceedings of the International Conference on Machine Learning, Optimization, and Data Science, Casdel Novo Verbadanca, Italy, 18–22 September 2022; pp. 32–46.
25. Wesego, D. Graph Representation Learning with Diffusion Generative Models. *arXiv* **2025**, arXiv:2501.13133. [CrossRef]
26. Lee, K.; Choi, J. Local Manifold Approximation and Projection for Manifold-Aware Diffusion Planning. *arXiv* **2025**, arXiv:2506.00867. [CrossRef]
27. Chung, H.; Sim, B.; Ryu, D.; Ye, J.C. Improving diffusion models for inverse problems using manifold constraints. *Adv. Neural Inf. Process. Syst.* **2022**, *35*, 25683–25696.
28. He, Y.; Murata, N.; Lai, C.H.; Takida, Y.; Uesaka, T.; Kim, D.; Liao, W.H.; Mitsufuji, Y.; Kolter, J.Z.; Salakhutdinov, R.; et al. Manifold preserving guided diffusion. *arXiv* **2023**, arXiv:2311.16424. [CrossRef]
29. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient estimation of word representations in vector space. *arXiv* **2013**, arXiv:1301.3781. [CrossRef]
30. Bojanowski, P.; Grave, E.; Joulin, A.; Mikolov, T. Enriching word vectors with subword information. *Trans. Assoc. Comput. Linguist.* **2017**, *5*, 135–146. [CrossRef]

31. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Minneapolis, MN, USA, 2–7 June 2019; Volume 1 (long and short papers), pp. 4171–4186.
32. Peters, M.E. Deep contextualized word representations. *arXiv* **2018**, arXiv:1802.05365.
33. Feng, L.; Cui, Y.; Hu, J. Detection and classification of malware based on FastText. In Proceedings of the 2020 IEEE International Conference on Artificial Intelligence and Information Systems (ICAIIS), Dalian, China, 20–22 March 2020; pp. 126–130.
34. Sarker, I.H. Generative AI and large language modeling in cybersecurity. In *AI-Driven Cybersecurity and Threat Intelligence: Cyber Automation, Intelligent Decision-Making and Explainability*; Springer: Berlin/Heidelberg, Germany, 2024; pp. 79–99.
35. Koç, C.; Özyurt, F.; Iantovics, L.B. Survey on latest advances in natural language processing applications of generative adversarial networks. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* **2025**, *15*, e70004. [CrossRef]
36. Metta, S.; Chang, I.; Parker, J.; Roman, M.P.; Ehuan, A.F. Generative AI in cybersecurity. *arXiv* **2024**, arXiv:2405.01674. [PubMed]
37. Narayanan, A.; Chandramohan, M.; Venkatesan, R.; Chen, L.; Liu, Y.; Jaiswal, S. graph2vec: Learning distributed representations of graphs. *arXiv* **2017**, arXiv:1707.05005. [CrossRef]
38. Grover, A.; Leskovec, J. node2vec: Scalable feature learning for networks. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016; pp. 855–864.
39. Gold, R. Control flowgraphs and code coverage. *Int. J. Appl. Math. Comput. Sci.* **2010**, *20*, 739–749. [CrossRef]
40. Ho, J.; Jain, A.; Abbeel, P. Denoising diffusion probabilistic models. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 6840–6851.
41. Zhang, S.; Tong, H.; Xu, J.; Maciejewski, R. Graph convolutional networks: A comprehensive review. *Comput. Soc. Netw.* **2019**, *6*, 1–23. [CrossRef]
42. Strudel, R.; Tallec, C.; Altché, F.; Du, Y.; Ganin, Y.; Mensch, A.; Grathwohl, W.; Savinov, N.; Dieleman, S.; Sifre, L.; et al. Self-conditioned embedding diffusion for text generation. *arXiv* **2022**, arXiv:2211.04236. [CrossRef]
43. Adaloglou, N.; Kaiser, T.; Michels, F.; Kollmann, M. Rethinking cluster-conditioned diffusion models for label-free image synthesis. In Proceedings of the 2025 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV), Tucson, AZ, USA, 26 Febreary–6 March 2025; pp. 3603–3613.
44. Mohamed, N. A Comprehensive Review of Natural Language Processing Techniques for Malware Detection. In Proceedings of the 2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT), Kamand, India, 24–28 June 2024; pp. 1–7.
45. Hosna, A.; Merry, E.; Gyalmo, J.; Alom, Z.; Aung, Z.; Azim, M.A. Transfer learning: A friendly introduction. *J. Big Data* **2022**, *9*, 102. [CrossRef]
46. Kim, J.Y.; Bu, S.J.; Cho, S.B. Zero-day malware detection using transferred generative adversarial networks based on deep autoencoders. *Inf. Sci.* **2018**, *460*, 83–102. [CrossRef]
47. Maaten, L.v.d.; Hinton, G. Visualizing data using t-SNE. *J. Mach. Learn. Res.* **2008**, *9*, 2579–2605.
48. Grohe, M. word2vec, node2vec, graph2vec, x2vec: Towards a theory of vector embeddings of structured data. In Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Portland, OR, USA, 14–19 June 2020; pp. 1–16.
49. Lau, J.H.; Baldwin, T. An empirical evaluation of doc2vec with practical insights into document embedding generation. *arXiv* **2016**, arXiv:1607.05368. [CrossRef]
50. Fefferman, C.; Mitter, S.; Narayanan, H. Testing the manifold hypothesis. *J. Am. Math. Soc.* **2016**, *29*, 983–1049. [CrossRef]
51. Gulrajani, I.; Ahmed, F.; Arjovsky, M.; Dumoulin, V.; Courville, A.C. Improved training of Wasserstein GANs. *Adv. Neural Inf. Process. Syst.* **2017**, *30*, 5767–5777.
52. Nichol, A.Q.; Dhariwal, P. Improved denoising diffusion probabilistic models. In Proceedings of the International Conference on Machine Learning, Virtual, 18–24 July 2021; pp. 8162–8171.
53. Kotelnikov, A.; Baranchuk, D.; Rubachev, I.; Babenko, A. Tabddpm: Modelling tabular data with diffusion models. In Proceedings of the International Conference on Machine Learning, Honolulu, HI, USA, 23–29 July 2023; pp. 17564–17579.
54. Breiman, L. Random forests. *Mach. Learn.* **2001**, *45*, 5–32. [CrossRef]
55. Liaw, A.; Wiener, M. Classification and regression by randomForest. *R News* **2002**, *2*, 18–22.
56. Yu, Y.; Zhang, Y. Multi-layer perceptron trainability explained via variability. *arXiv* **2021**, arXiv:2105.08911.
57. Steck, H.; Ekanadham, C.; Kallus, N. Is cosine-similarity of embeddings really about similarity? In Proceedings of the Companion Proceedings of the ACM Web Conference 2024, Singapore, 13–17 May 2024; pp. 887–890.
58. Maharani, D.A.; Machbub, C.; Rusmin, P.H.; Yulianti, L. Improving the capability of real-time face masked recognition using cosine distance. In Proceedings of the 2020 6th International Conference on Interactive Digital Media (ICIDM), Bandung, Indonesia, 14–15 December 2020; pp. 1–6.

59. Sitikhu, P.; Pahi, K.; Thapa, P.; Shakya, S. A Comparison of Semantic Similarity Methods for Maximum Human Interpretability. In Proceedings of the 2019 Artificial Intelligence for Transforming Business and Society (AITB), Kathmandu, Nepal, 5 November 2019.
60. Takgil, B. Examining the Ethical Risks of Generative AI in Cybersecurity: An Experimental Study on Ethical, Gray Area and Unethical Usage Scenarios. *Siber Guven. Dijital Ekon.* **2025**, *1*, 1–9.