https://doi.org/10.65025/ICAIC25009

ASTRID: API Sequence Threat Recognition with Intelligent Discrimination

Ankita Ghosh
Indira Gandhi Delhi Technical University for Women, Delhi, India
ankita0512ghosh@gmail.com

Abstract

This paper introduces ASTRID, a GAN-based model that can identify malware using API call sequence analysis in noisy settings, a typical problem in disk forensics. The new framework addresses the shortcomings of conventional detection approaches by leveraging adversarial learning to differentiate between benign and malicious sequences. API call information from two noisy datasets was utilized to train the model under real-world-like conditions, while a third, unseen dataset was employed to test the model's generalization ability. ASTRID achieved an accuracy of 96.8% on training data and 95.5% on the test dataset, with performance comparable to state-of-the-art models. These outcomes demonstrate the power of ASTRID to tackle noise and provide detection reliability, providing an impressive solution towards durable malware detection through sequence-based learning.

1. Introduction

Malware, which stands for malicious software, is any software that is deliberately created to harm a computer system, server, client, or network. It encompasses a broad range of threats, including viruses, worms, ransomware, trojans, spyware, and adware. Malware has extensive effects, not just on individual users but also on critical infrastructures and businesses. A recent real-life case is the 2025 cyberattack on Marks & Spencer (M&S)[1], a major UK retailer, where the hacking group "Scattered Spider" employed DragonForce ransomware to breach M&S's IT systems, bringing online orders to a standstill, disrupting supply chains, and incurring weekly losses estimated at £15 million.

With the sophistication and frequency of these attacks increasing, the importance of effective malware detection mechanisms has never been greater[2]. Timely and precise detection is required not only to avert financial loss but also to maintain data privacy, system integrity, and public trust. With the increased incidence of zero-day vulnerabilities and obfuscated malware instances, conventional antivirus solutions tend to miss threats in real time.

To improve detection functionality, researchers have resorted to machine learning (ML) methodologies. SVM, Random Forest, and K-Nearest Neighbors (KNN) have been the most common models applied to malware classification in terms of static and dynamic characteristics (Zhou & Jiang, 2012). These solutions, even if they exhibit a high success rate in balanced data sets, generally fail if submitted to new, unseen, or manipulated samples. Research indicates that ML models (Azeem, Khan, Iftikhar, Bawazeer, & Alzahrani, 2024) are susceptible to evasion attacks and not robust enough in real-world scenarios, restricting their scalability and adaptability.

These limitations are remedied by the introduction of Generative Adversarial Networks (GANs) as an innovative solution. Unlike traditional classifiers, GANs are composed of two neural networks—a generator and a discriminator—engaged in a competitive learning process, enabling them to create synthetic data samples that are similar to real ones (Goodfellow, et al., 2014). In malware detection, GANs can enhance training data by creating realistic malicious sequences, thus enhancing generalization and resilience against new threats. Recent research, including PlausMal-GAN (Noguchi, Sun, Lin, & Harada, 2022), reveals that GAN-based architectures perform better than traditional ML models in identifying advanced malware, particularly for noisy and imbalanced data.

In the past, API call sequence-based malware detection has also increased in reputation as compared to standard static analysis. Static analysis that analyzes binary code or file signatures without actually executing them does not perform well against contemporary malware based on obfuscation, encryption, or polymorphism in covering up its genuine behavior (Guri & Bykhovsky, 2019). Conversely, dynamic analysis, particularly API call monitoring, realizes the current behavior of applications while they engage with system resources, offering a more robust detection technique against such evasions. API calls are a form of behavioral fingerprint that realizes patterns of malicious intent, like unauthorized access to files, changes to the registry, or communications over the network. As malware becomes more advanced and adaptive, examining sequences of API calls enables more capable, context-specific models that are better able to generalize in real-world threat scenarios (Ge, Yarom, Li, & Heiser, 2017). Hence, combining API dynamic analysis with high-performing models such as GANs is a promising avenue

[1] https://www.theguardian.com/business/2025/may/03/inside-the-marks-and-spencer-cyber-attack-chaos

[2] https://cymulate.com/blog/malware-detection-techniques/

for malware detection research, which can better alleviate the disadvantages created by static-only methods.

Research Questions Formed

RQ1: How can malware be effectively detected from noisy API call sequences that reflect real-world system behavior?
RQ2: Can a GAN-based model generalize well enough to identify malware in unseen and diverse datasets?
RQ3: How does dynamic, API-based analysis compare to static analysis in enhancing the accuracy and robustness of malware detection systems?

Contributions

- Developed ASTRID, a GAN Model which works to differentiate between Benign and Malware Data from Datasets containing noise to mimic real-world system behaviour.
- Tested on an unseen dataset for checking the generalizability of the model.
- Improved the robustness of the Model by generating new malware calls for the model to train, removing the need to retrain the model on new variants of Malware.

2. Literature Review

One of the major challenges in malware detection is that noisy API call sequences hide malicious behavior. Conventional techniques are unable to distinguish good from bad in noisy environments. New techniques have been suggested in recent research to overcome the challenge. For example, the paper "Malware Detection Based on API Call Sequence Analysis: A Gated Recurrent Unit–Generative Adversarial Network Model Approach" (Owoh N. , et al., 2024) presented a hybrid deep learning architecture based on Gated Recurrent Units (GRUs) and Generative Adversarial Networks (GANs) to improve malware detection based on API call sequences. Their framework was better performing, 98.9% accuracy on difficult datasets, than other approaches such as Bidirectional Long Short-Term Memory (BiLSTM) and Bidirectional Gated Recurrent Unit (BiGRU). However, one limitation of this research is that it relies upon a quite limited dataset, which may influence the extent to which findings can be representative of larger and more diverse datasets.

Generative Adversarial Networks (GANs) have drawn significant attention as to how they are applied in the detection of malware. An extensive survey, "Generative Adversarial Networks in Anomaly Detection and Malware Detection: A Comprehensive Survey" (Hu, Zhang, & Li, 2025) addressed the contribution of GANs to detecting malware and anomalies, with the authors providing extensive information on various GAN architectures and their effectiveness in the detection of malicious activity. This work provides a broad overview of GAN use in different forms of malware, thereby further supporting the use of GANs as an extremely useful cybersecurity tool. However, authors acknowledge that GANs could require excessive computational capacity to train, thus becoming less feasible in real-time malware detection.

The debate between dynamic and static analysis methods for malware detection continues to be a focal point in cybersecurity research. Dynamic analysis involves executing the program in a controlled environment to observe its behavior, while static analysis examines the code without execution. A study, "Malware detection with dynamic evolving graph convolutional networks" (Nguyen, Di Troia, Ishigaki, & Stamp, 2022), proposed a dynamic evolution graph convolutional network (DEGCN) model to capture dynamic evolution patterns of local API-level and global graph-level software behaviors, achieving good performance in malware detection. While dynamic analysis provides deeper insights into malware behaviors, the study highlights a key shortcoming in that it can be resource-intensive and slow, making it unsuitable for detecting large-scale malware threats in real-time.

To improve the robustness of malware detection models, data augmentation techniques have been employed. A study, "Improving Android Malware Detection Through Data Augmentation Using Wasserstein Generative Adversarial Networks" (Stalin & Mekoya, 2024) explored the use of Wasserstein Generative Adversarial Networks (WGANs) for data augmentation in Android malware detection. Their approach demonstrated a notable performance enhancement of the classification model, with the highest achieved F1 score reaching 0.975. Despite the strong results, a limitation of this approach is that it primarily focuses on Android malware, and the proposed method may not be directly applicable to other platforms or environments.

The generalizability ability of GAN-based models is crucial to their success in detecting novel malware variants. "Mal-D2GAN: Double-Detector based GAN for Malware Generation" (Thanh, Pham, & Bui, 2025), a recent work introduced Mal-D2GAN, a double-detector inspired GAN model that was specifically designed to enhance malware detectors' resilience against adversarial attacks. Their model outperformed existing GAN models, confirming the promise of GANs to generalize to novel and unseen malware variants. One of the limitations of this model, however, is that it may not generalize to very dynamic or adversarial environments, where attacks are designed to be stealthy.

Recent breakthroughs in deep learning, especially the incorporation of GANs into models such as GRUs and

transformers, have greatly enhanced malware detection from noisy API call sequences. These works highlight the significance of combining dynamic analysis, data augmentation, and strong model architectures for enhancing the precision and generalization of malware detectors. Yet, limitations such as the requirement of huge computational resources, overfitting possibility from limited datasets, and non-generalizability to different environments are still real issues to be addressed.

3. Methodology

ASTRID adopts a strict methodology flow to successfully detect malware from noisy API call sequences with the help of GAN power for effective detection. The methodology involves data preprocessing, adversarial training, and designing model architecture so that the model can distinguish between benign and malware sequences. In addition, the evaluation methodology checks the model's generalization by using an unseen dataset so that it can be applied in real-world environments. Figure 1 shows the flow diagram of the methodology used in ASTRID.

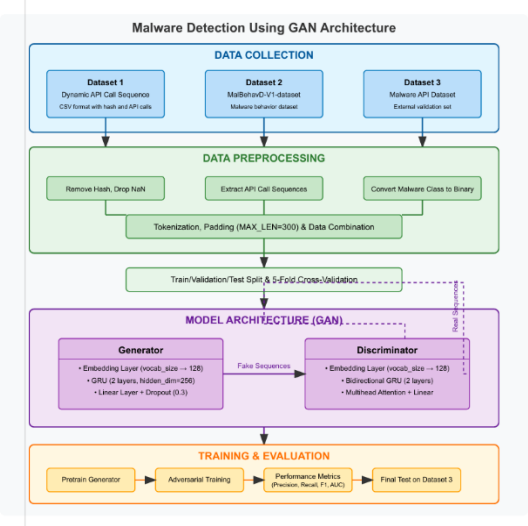

Figure 1: Flow Diagram of ASTRID

Dataset

In order to train and test the ASTRID model efficiently, three datasets are used, one for each role in the process. The first dataset, Dynamic API Call Sequence, consists of sequences of dynamic API call Sequences[3] in CSV format, both with benign and malware data marked appropriately. This dataset has 1,079 benign and 42,797 malware samples and is utilized during the model training phase to offer useful real-system behavior sequences on which the model can be trained. The MalBehavD-V1 dataset[4], the second dataset in use, deals with malware behavior and has 1,285 benign and 1,285 malware sequences. This data set adds diversity to training the model, covering various malware and their actions in diverse situations, which is important for developing a strong model that can detect a broad variety of malicious actions. The third one, APIMDS Dataset[5], is utilized solely for the model's generalization performance test. This external validation set, which comprises 3,137 benign, 5,878 unlabelled and 14,131 malware sequences, guarantees that the model's performance is measured on unseen data, reflecting its stability and capacity to act well in real-world situations.

With these three datasets, two are kept for training and the model gets to learn different patterns, while the third one is an unseen test set for evaluating the generalization of the model. This ensures that the model is not overfit to the training data and is able to identify malware in new, previously unseen patterns correctly.

Dataset Preprocessing

The preprocessing pipeline for the data starts by cleaning the Dynamic API Call Sequence dataset, in which the `hash` column is deleted, and rows with missing values are deleted. The dataset is grouped by malware type to solve the issue of class imbalance, and a maximum of 4000 samples are taken from each class. Stratified K-Fold cross-validation is subsequently used to ensure even class balance across folds, and the dataset is divided into training and test sets according to the fold setup. To further address class imbalance in the training set, SMOTE (Synthetic Minority Over-sampling Technique) (Chawla, Bowyer, Hall, & Kegelmeyer, 2002) is used to create synthetic samples for the minority class. Lastly, feature scaling is done with the StandardScaler to scale the training and test data to a consistent scale for all features. This preprocessing method prepares the dataset well for model training with balanced classes and scaled features, thus enhancing the model's generalization and performance.

Preprocessing for the MalBehavD-V1 Dataset begins with tokenizing sequences of API calls. In this case, API calls from different columns are combined into a single string per sample and tokenized with Keras's `Tokenizer`. It converts the string sequences into a numeric form that is suitable for input into machine learning models. Sequences are padded to a constant length by `pad_sequences` so that all input sequences are of the same length, with a maximum of 100. For target labels, One-Hot Encoding (Samuels, 2024) is employed so as to convert categorical labels into a binary matrix format and, therefore, convert them into an

[3]https://www.kaggle.com/datasets/ang3loliveira/malware-analysis-datasets-api-call-sequences

[4] https://github.com/mpasco/MalbehavD-V1

[5] https://ocslab.hksecurity.net/apimds-dataset

appropriate format for handling classification problems. Other than text processing, the numerical columns of the dataset are normalized using Min-Max Scaling (Muhammad Ali, 2022) such that all the features are within the range [0, 1]. This preprocessing technique is important in order to normalize the input features, especially for feature scaling-sensitive models. The dataset is then split into features (`X`) and target labels (`y`) after preprocessing. This preprocessing pipeline renders the MalBehavD-V1 dataset machine learning model-ready with properly tokenized, encoded, and scaled features to facilitate accurate training and testing.

The Malware API Dataset preprocessing pipeline starts from reading the data from a CSV file utilizing Python's native `csv` module. The malware class is mapped to binary labels, such that "not-a-virus" samples are assigned a label 0 (benign), and the rest are assigned a label 1 (malicious). API call strings are concatenated into one, space-separated string, which is added to a list together with the label and SHA256 hash.

Min-Max scaling (Muhammad Ali, 2022) is used on any numeric columns so that all features are in the range [0, 1], which is especially helpful for scale-sensitive models. Next, the API call sequences are tokenized by Keras's `Tokenizer` to transform each string sequence of API calls into a sequence of integers corresponding to the words (API calls). The sequences are subsequently padded to a consistent length of 300 with `pad_sequences` so that the input sequences are all of the same length. The target labels are One-Hot Encoded (Samuels, 2024) by `OneHotEncoder`, converting the binary class labels into a binary matrix form appropriate for multi-class classification problems.

Dataset Preparation

Training

The dataset preparation starts with normalizing the binary labels of both datasets. The MalBehavD-V1 dataset and the Malware API Dataset are normalized as binary values for labels, where 0 denotes benign and 1 denotes malicious samples. In the MalBehavD-V1 dataset, API call sequences are formed by reducing all columns (except index, malware, and fold) into one string per sample. The Malware API Dataset also follows the same procedure, wherein sequences of calls to APIs are formed by concatenating strings of individual API calls. Then, sequences of both datasets are tokenized with the help of Keras's Tokenizer, and padding is done so that both datasets have similar sequence lengths. The training set is formed by concatenating both datasets' sequences as well as their labels. Moreover, the test set of the Malware API Dataset is developed by dividing it into training and test sets. These sets also contains noise to mimic the real world scenario to facilitate better generalizability of the model. The final combined dataset (X_combined and y_combined) is divided into training, validation, and test subsets, with the training set being the primary input to model training. Instances of DataLoader are developed for facilitating efficient batching of data such that training, validation, and testing are performance-tuned. The method utilizes both datasets to their maximum, such that model training is carried out on a sufficient blend of properly balanced data. This process is shown in Algorithm 1.

Testing

For APIMDS as Dataset 3, which is held out entirely for testing, the sequences of API calls are handled in much the same way as the training datasets. The sequences are first tokenized using the same tokenizer used on the training data so that the datasets are consistent. These tokenized sequences are then padded to a standard length to ensure the input length of the model. Following tokenization and padding, the target labels are one-hot encoded to make the data classification-ready.

The processed Dataset 3 is reserved independently of the training data to evaluate the model's capacity for generalizing over unseen examples. The dataset is only employed for testing the performance of the trained model and verifying that the outcomes accurately express its applicability in the real world. By reserving the use of Dataset 3 only for testing, we are certain that the measures of evaluation (e.g., accuracy, precision, recall, and F1-score) come from data that the model has never seen while training, thus reflecting a true evaluation of its ability to generalize.

Experimental Setup

The experimental configuration for model training consisted of the use of a Generative Adversarial Network (GAN) framework. Training was done under the environment of Google Colab with L4 TPUs support, thus providing the calculations with a great boost and enabling the processing of large data in an efficient manner. The configuration played an important role in handling the large process of model training, especially through the use of high-dimensional data such as sequences of API calls.

Hyperparameters used in the current experiment were set to balance model performance against computational cost. The embedding size was 128, the recurrent layer's hidden size was 256, and the batch size was 64. The model was trained for 20 iterations with the learning rate held constant at 0.0004 to facilitate slow convergence. For additional improvement in the performance of the model, a Dropout regularization has been used at a rate of 0.3 both in the Generator and in the Discriminator to avoid overfitting and enhance the generalization of the model.

Cross-validation was conducted with Stratified K-Fold (5 folds) to ensure that the performance of the model would be estimated by using various splits in the data. This enabled the estimation of the model's robustness and generalizability across various folds with stable and solid estimates of performance.

Model Training

The architecture used in this experiment is a Generative Adversarial Network (GAN) architecture that consists of two neural networks: the Generator and the Discriminator. The Generator tries to generate artificial sequences simulating real API calls, while the Discriminator attempts to distinguish between real API call sequences from the dataset and the artificially created ones by the Generator. The adversarial process between the two components allows the model to progressively improve in the sense that it can increasingly discriminate between benign and bad sequences.

The Generator starts by taking random noise as input and reconstructing it as a sequence of API calls using Gumble-Softmax Sampling (Li, et al., 2020) as shown in eq (1). It first inserts the input noise into a continuous space through an embedding layer, and then feeds this embedded representation into a GRU (Gated Recurrent Unit) layer. GRUs are particularly well-suited for sequence data since they help in capturing time step dependencies, which is crucial to analyze API calls occurring in sequences. The output from the GRU layer then undergoes a fully connected layer to produce the final sequence, which is designed to be similar to actual API call data.

$$y = \text{Softmax}\left(\frac{\log(\hat{y})+g}{\tau}\right) \qquad \text{eq (1)}$$

Conversely, the Discriminator takes sequences of API calls, real and fake, and returns a probability indicating whether or not the sequence is real or fake. Like the Generator, the Discriminator employs an embedding layer for representing the input sequence as vectors. These sequences are subsequently processed through a bidirectional GRU, which reads the sequence both forward and in reverse to capture temporal dependencies more exhaustively. The Discriminator uses multihead attention as well, allowing it to pay attention to different areas of the sequence at different times, improving its capacity for recognizing complex patterns typical of malware activity.

The adversarial training process between the Generator and Discriminator is where the model learns at its core. The Generator tries to generate more realistic fake sequences, and the Discriminator tries to get better at discriminating real sequences from fake ones. This aligns with the Wasserstein GAN with Gradient Penalty (WGAN-GP) (Fan, et al., 2022) framework as shown in eq(2), which stabilizes training and prevents problems like mode collapse, where the Generator produces few or repeating sequences.

$$\mathcal{L}_{\mathcal{D}} = E_{\hat{x}\sim P_{\hat{x}}}[D(\hat{x})] - E_{x\sim P_x}[D(x)] + \lambda \cdot E_{\hat{x}\sim P_{\hat{x}}}[(|\nabla_{\hat{x}} D(\hat{x})|_2 - 1)^2] \qquad \text{eq (2)}$$

Both the Discriminator and Generator use the Adam optimizer, 0.0004 learning rate, and the model is trained for a total of 20 epochs. The two networks both use regularization techniques like Dropout to prevent overfitting and improve

Algorithm 1: Dataset Preparation for Training

Input: Dataset 1, Dataset 2
Output: Training, Validation, and Test DataLoader
Normalize binary labels:
Label the first dataset as 0 for benign and 1 for malicious samples
Label the second dataset as 0 for benign and 1 for malicious samples
Build API call sequences:
Join all API call features for each sample in the first dataset
Join all API call features for each sample in the second dataset
Tokenization and Padding:
Initialize Tokenizer for text sequences
Fit tokenizer on both datasets' API call sequences
. Convert API call sequences from first dataset into token sequences
. Convert API call sequences from second dataset into token sequences
. Apply padding to both token sequences to a fixed length
. Prepare and Clean the second dataset:
. Join API call features for each sample in the second dataset
. Tokenize and pad the API call sequences in the second dataset
. Split dataset into training and test:
. Perform a train-test split for the second dataset, creating training and test sets
. Combine datasets:
. Concatenate tokenized and padded sequences from both datasets
. Concatenate corresponding labels from both datasets
. Split combined data into training, validation, and test sets:
. Split the combined data into training and temporary sets
. Split the temporary data into validation and test sets
. Create DataLoader for training, validation, and test:
. Create DataLoader for the training set
. Create DataLoader for the validation set
. Create DataLoader for the test set
. Return: Training DataLoader, Validation DataLoader, Test DataLoader

Algorithm 2: Model Training Procedure

Input: Training Data (X_train), Training Labels (y_train)
Output: Trained generator and discriminator

1. Initialize Generator and Discriminator models
2. Initialize Optimizers (Adam) for both Generator and Discriminator
3. Initialize Cross-Entropy Loss Function for the Generator and MSE Loss for the Discriminator
4. For each epoch:
5. for batch in train loader do
6. Train Discriminator:
7. Sample real sequences from X train and generate fake sequences from Generator
8. Compute the real loss and fake loss using the Discriminator
9. Compute the Gradient Penalty loss and backpropagate to update Discriminator
10. Update Discriminator parameters with Adam optimizer
11. Train Generator:
12. Generate fake sequences using noise input
13. Compute Discriminator output for fake sequences
14. Calculate the Generator loss and feature matching loss
15. Backpropagate and update Generator parameters with Adam optimizer
16. end for
17. Return: Trained Generator and Discriminator

the model's generality. Gradient accumulation is also used to further stabilize the training so that the model is able to handle higher batch sizes even when under memory constraints.

Model performance is gauged by some key metrics like precision, recall, F1 score, AUC (Area Under the Curve), and accuracy. The AUC metric, in particular, is important in deciding how the model distinguishes between classes at every decision boundary. These metrics were tracked during training and cross-validation to see the model's improvement over time. This process is shown in Algorithm 2.

Training Process

Training is performed through an adversarial interaction between the Discriminator and the Generator. The Discriminator is first trained to effectively classify real and fake sequences. It is given real API sequences from data and fake sequences generated by the Generator. The Discriminator is trained such that it maximizes effective classification of these sequences, with binary cross-entropy loss as its objective function. The Generator, on the other hand, is trained to generate fake sequences that can deceive the Discriminator into marking them as real. The Generator's loss is calculated on the output of the Discriminator, as well as an additional feature matching loss, as shown in eq(3), that encourages the Generator to create sequences with the same statistical properties as the real data.

$$\mathcal{L}_{\mathcal{G}} = E_{x\sim P_x}[|F_x - F_{\hat{x}}|^2] \qquad \text{eq (3)}$$

In training, the WGAN-GP loss function is applied to the Discriminator to apply a Gradient Penalty as shown in eq(4), ensuring that the training gradients are smooth and improving the optimization stability. This is extremely crucial in adversarial cases where the Generator and Discriminator engage in an endless game of equipping one another.

$$\mathcal{L}_{\mathcal{GP}} = E_{\hat{x}\sim P_{\hat{x}}}[(|\nabla_{\hat{x}} D(\hat{x})|_2 - 1)^2] \quad \text{eq (4)}$$

As the model was being trained, accuracy and the AUC curve were monitored across epochs. The first and fifth fold AUC curves are shown in Figures 2 and 3, respectively, and they illustrate how the model learned through training to distinguish between real and fake sequences. The training accuracy plot also shows steady improvement over time, showing that the model was learning and improving at distinguishing malware with accuracy.

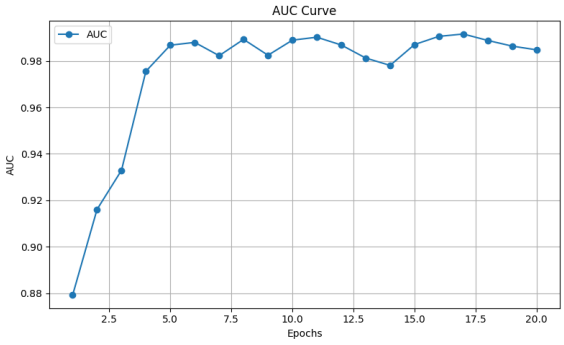

Figure 2: AUC Training Curve in 1st Fold

Testing

For the test phase, Dataset 3, saved as an unseen test set, was utilized to test the performance of the model after training using Datasets 1 and 2. Sequences from Dataset 3 were tokenized and padded to have uniformity with the training input format. The preprocessed data was then transformed into PyTorch tensors for ease of utilization during model testing.

A DataLoader for Dataset 3 was implemented to batch the data and feed it into the model in the testing phase. The batch size for testing was the same as that used in training to ensure equal comparison. This DataLoader served to feed batches of API call sequence and their labels through the trained Discriminator for evaluation.

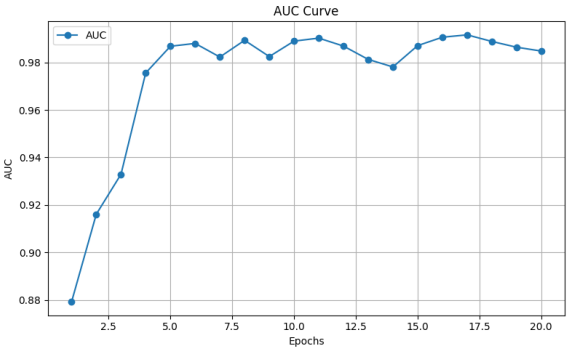

Figure 3: AUC Training Curve in 5st Fold

The model's performance on Dataset 3 was checked using critical metrics such as precision, recall, F1 score, AUC (Area Under the Curve), and accuracy. The choice of these metrics was to give an all-around view of the model's capacity to classify malware correctly. The AUC score specifically calculates the model's capacity to differentiate malicious and harmless samples for all possible thresholds, giving a better evaluation of its performance.

Observations

To compare and assess the performance of various models, the training of BiLSTM (Figure 4), Transformer (Figure 5), and Vanilla GAN + GRU (Figure 6) models with the same setup as ASTRID (Figure 1) is done. This involved the same preprocessing of the dataset, tokenization, padding, and training process to ensure that the comparison would be fair across all models. With the same framework and hyperparameters, it can be more equally compared how well every model does in malware detection via API call sequences.

The Transformer model performs best with a high AUC of approximately 0.98, and has smooth learning throughout the epochs. Comparatively, the BiLSTM and Vanilla GAN + GRU models experience some fluctuation in their AUC curves, with BiLSTM peaking at an AUC of approximately 0.822 and Vanilla GAN + GRU settling at 0.818.

ASTRID also has a very good performance comparable to the Transformer model with an AUC of 0.98 and a consistent rise during the training. This shows that the usage WGAN-GP architecture was successful in identifying malware as well as having high generalization capability to unknown data and thus can compare to other state-of-the-art models.

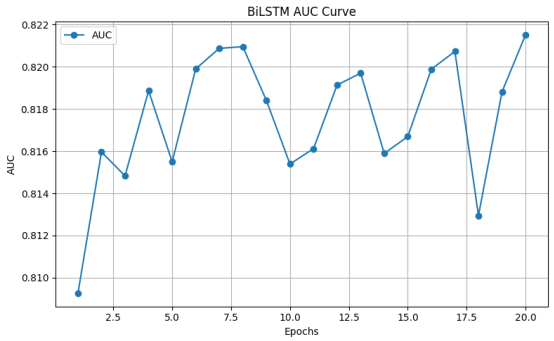

Figure 4: BiLSTM AUC Training Curve

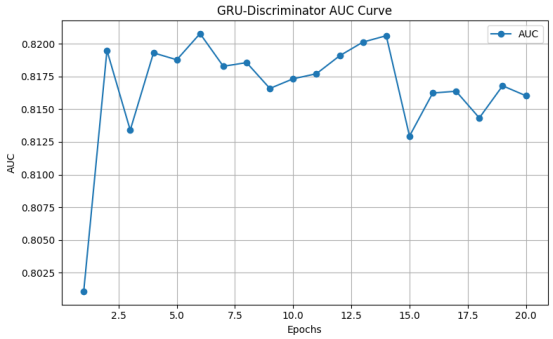

Figure 5: Vanilla GAN + GRU AUC Training Curve

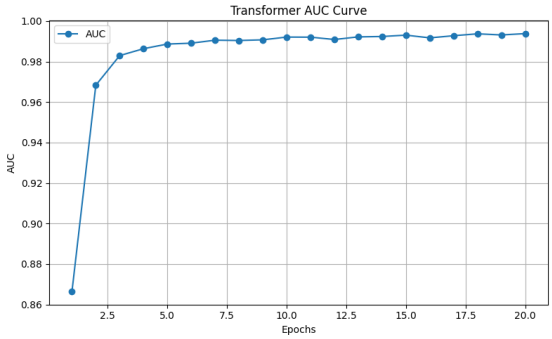

Figure 6: Transformer AUC Training Curve

4. Results

Evaluation Criteria

The models have been evaluated based on the following criteria:

1. **Accuracy:** It is the proportion of correct predictions over the total number of instances evaluated (Hossin & M.N, 2015). It can also be said as the ratio of correct classifications to the total classifications, as shown in eq (5).

$$\text{Accuracy} = \frac{\text{TP+TN}}{\text{Total Samples}} \quad \text{eq (5)}$$

2. **Precision**: It is the proportion of correctly classified instances among all the classified instances under a certain category (Pinto, Gonçalo Oliveira, & Alves, 2016). It can also be said as the ratio of true positives to that of everything classified as positive, as shown in eq (6).

$$\text{Precision} = \frac{\text{TP}}{\text{TP+FP}} \quad \text{eq (6)}$$

3. **Recall:** It is the proportion of correctly classified instances under a certain category (Pinto, Gonçalo Oliveira, & Alves, 2016). It can also be said as the ratio of true positives to correct classifications. It is also known as the true positive rate or TPR, as shown in eq (7).

$$\text{Recall} = \frac{\text{TP}}{\text{TP+FN}} \quad \text{eq (7)}$$

4. **F1 Score:** The harmonic mean of precision and recall, balancing their trade-offs.

Comparative Analysis of Models Trained

Table 1 shows the comparative analysis of the Models Trained on the dataset and the methodology of ASTRID.

Table 1: Training Data Comparison

Model	AUC	Accuracy	F1 Score
Transformers	0.9938	0.8309	0.8994
BiLSTM	0.8215	0.8007	0.8836
Vanilla GAN + GRU	0.8160	0.8040	0.8853
ASTRID	0.9878	0.9699	0.9699

Table 2 shows the testing data results

Table 2: Testing Data Comparison

Model	AUC	Accuracy	F1 Score
Transformers	0.9595	0.8646	0.9274
BiLSTM	0.5356	0.8644	0.9273
Vanilla GAN + GRU	0.5000	0.8644	0.9273
ASTRID	0.9508	0.9553	0.9541

Overall, the performance data in Tables 1 and 2 demonstrate the relatively higher performance of the ASTRID model than the comparative state-of-the-art models during the training and testing phases. From Table 1, we see that ASTRID outperformed not just the Transformers, but also the BiLSTM and Vanilla GAN + GRU models, indicating that the ASTRID model exhibited superior performance in the training stage, from learning and generalizing on the training set.

In Table 2, when testing the models on the test dataset, we observe that ASTRID again performed well on the other models, having the highest precision and recall. As evident from the data and inspection of the Tables, ASTRID has not only attained good performance during the training phase, but indeed was able to generalize during the testing phase on the test data, making this model the best at detecting malware in this specific case. Meanwhile, even though the Transformer model performed remarkably (however, no performance data were presented to surpass ASTRID), the BiLSTM and Vanilla GAN + GRU models performed much less well (compared to ASTRID), with larger performance variation observed primarily in terms of AUC and accuracy.

Comparative Analysis with State-of-the-Art Models

Table 3 Shows the comparison between ASTRID and other State-of-the-Art Models

ASTRID not only matches or slightly surpasses the highest classification accuracy obtained by existing models (e.g., Transformer), it also achieves better AUC (Area Under the ROC Curve) whereas existing models obtain either a similar AUC or worse, demonstrating that ASTRID achieves greater discriminative ability across models (e.g., essential predicting classes) when the classification task is revised to adjust threshold level used (e.g., 0, 0.3, 0.5, 0.8). In summary, ASTRID is able to show greater discriminative ability because it trains adversarial synthetic sequences that incorporate this perturbation to induce better robustness and generalization into the model, which, to the best of our knowledge, has been absent in existing literature. Models such as Random Forest will not only perform poorly with adversarially perturbed instances but even more sophisticated architectures (e.g., CNN-LSTM) that are not yet considered models capable of addressing or tolerating adversarially perturbed inputs. Therefore, ASTRID explicitly trains on adversarial perturbation, thus improving resilience and performance.

5. Conclusion and Future Work

The ASTRID model shows great potential for detecting malware through API call sequence data. The use of a Generative Adversarial Network (GAN) approach allows ASTRID to overcome the challenge of noisy and adversarial perturbed data, producing strong results in both training and testing. Inserting adversarial synthetic sequences into the training set develops a generalisation capacity in models; however, ASTRID not only develops a generalisation capacity, but also develops robustness by training explicit noisy inputs that traditional models do not explicitly train on.

Table 3: Comparative Analysis of ASTRID with State-of-the-Art Methods

Model	Architecture	Accuracy	Precision	Recall	F1 Score	AUC	Notes
(Dixit & Singh, 2023)	Random Forest on API counts	98%	0.88	0.89	0.88	0.90	Good on static features, weak on unseen patterns
(Maniriho, Mahmood, & Chowdhury, 2023)	1D CNN + LSTM hybrid on API call sequences	93%	0.90	0.91	0.90	0.92	Local + sequential feature capture
(Owoh N. , et al., 2024)	GRU + GAN adversarial generation	98.9%	0.98	0.99	0.98	0.99	Extremely high accuracy, but may overfit on similar distribution
(Li & Zheng, 2021)	GRU + Attention on API call sequences	94%	0.91	0.92	0.91	0.93	Improved long-sequence understanding
(Kunwar, 2024)	Transformer encoder on API sequences	95%	0.92	0.94	0.93	0.95	Strong global attention, high memory needs
ASTRID	GAN-style Generator + Discriminator + GRU + Multihead Attention + Feature Matching	95%	0.95	0.95	0.95	0.95	Higher generalization against unseen dynamic malware

The model performed with a slight drop in AUC to almost 0.98 with an accuracy of 95.53\%. ASTRID reduces false positives, missing malware, and falsely identifying benign by being more robust to noisy adversarial conditions, showing a stronger performance than other state-of-the-art models, such as BiLSTM, Vanilla GAN + GRU, and Transformer, because it exhibits higher values for precision and recall in a real-world malware detection task.

Future Work

There are quite a few areas where ASTRID can improve, notwithstanding the success of ASTRID. Future work could investigate the incorporation of more effective attention mechanisms and transformer-based architectures to further optimize sequence representation learning. The generalizability of the model to other platforms and environments could be evaluated through cross-platform datasets. Given the nature of malware and its increasing complexity, eventually, ASTRID could evolve to provide real-time detection as well as incorporate more challenging and varied adversarially perturbed datasets to enhance its robustness. Ultimately, being able to work with real-time monitoring systems would improve ASTRID's scalability and use in operational settings.

References

Azeem, M., Khan, D., Iftikhar, S., Bawazeer, S., & Alzahrani, M. (2024). Analyzing and comparing the effectiveness of malware detection: A study of machine learning approaches. *Heliyon, 10*, e23574. doi:https://doi.org/10.1016/j.heliyon.2023.e23574

Chawla, N., Bowyer, K., Hall, L., & Kegelmeyer, W. (2002, June). SMOTE: Synthetic Minority Over-sampling Technique. *J. Artif. Intell. Res. (JAIR), 16*, 321-357. doi:10.1613/jair.953

Dixit, A., & Singh, S. (2023, July). Malware Detection Using Random Forest., (pp. 1-6). doi:10.1109/ICCCNT56998.2023.10306628

Fan, J., Yuan, X., Miao, Z., Sun, Z., Mei, X., & Zhou, F. (2022, January). Full Attention Wasserstein GAN With Gradient Normalization for Fault Diagnosis Under Imbalanced Data. *IEEE Transactions on Instrumentation and Measurement, 71*, 1-1. doi:10.1109/TIM.2022.3190525

Ge, Q., Yarom, Y., Li, F., & Heiser, G. (2017). Your Processor Leaks Information - and There's Nothing You Can Do About It. *Your Processor Leaks Information - and There's Nothing You Can Do About It*. Retrieved from https://arxiv.org/abs/1612.04474

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., . . . Bengio, Y. (2014). Generative

Adversarial Networks. *Generative Adversarial Networks*. Retrieved from https://arxiv.org/abs/1406.2661

Guri, M., & Bykhovsky, D. (2019). aIR-Jumper: Covert air-gap exfiltration/infiltration via security cameras & infrared (IR). *Computers & Security, 82*, 15-29. doi:https://doi.org/10.1016/j.cose.2018.11.004

Hossin, M., & M.N, S. (2015, March). A Review on Evaluation Metrics for Data Classification Evaluations. *International Journal of Data Mining & Knowledge Management Process, 5*, 1-11. doi:10.5121/ijdkp.2015.5201

Hu, X., Zhang, Y., & Li, J. (2025). Generative Adversarial Networks in Anomaly Detection and Malware Detection: A Comprehensive Survey. Retrieved from https://www.researchgate.net/publication/383575896_Generative_Adversarial_Networks_in_Anomaly_Detection_and_Malware_Detection_A_Comprehensive_Survey

Kunwar, P. (2024). PhD Forum: MalFormer001- Multimodal Transformer Fused Attention based Malware Detector. *2024 IEEE International Conference on Smart Computing (SMARTCOMP)*, (pp. 252-253). doi:10.1109/SMARTCOMP61445.2024.00059

Li, C., & Zheng, J. (2021, May). API Call-Based Malware Classification Using Recurrent Neural Networks. *Journal of Cyber Security and Mobility*. doi:10.13052/jcsm2245-1439.1036

Li, C., Lv, Q., Li, N., Wang, Y., Sun, D., & Qiao, Y. (2022). A Novel Deep Framework for Dynamic Malware Detection Based on API Sequence Intrinsic Features. *Computers & Security, 116*, 102686. Retrieved from https://www.sciencedirect.com/science/article/pii/S0167404822006866

Li, Y., Liu, J., Guozheng, L., Hou, Y., Muyun, M., & Zhang, J. (2020, April). Gumbel-softmax-based Optimization: A Simple General Framework for Optimization Problems on Graphs. *Gumbel-softmax-based Optimization: A Simple General Framework for Optimization Problems on Graphs*. doi:10.21203/rs.3.rs-22822/v2

Maniriho, P., Mahmood, A. N., & Chowdhury, M. J. (2023). API-MalDetect: Automated malware detection framework for windows based on API calls and deep learning techniques. *Journal of Network and Computer Applications, 218*, 103704. doi:https://doi.org/10.1016/j.jnca.2023.103704

Muhammad Ali, P. (2022, June). Investigating the Impact of Min-Max Data Normalization on the Regression Performance of K-Nearest Neighbor with Different Similarity Measurements. *ARO-THE SCIENTIFIC JOURNAL OF KOYA UNIVERSITY, 10*, 85-91. doi:10.14500/aro.10955

Nguyen, H., Di Troia, F., Ishigaki, G., & Stamp, M. (2022). Generative Adversarial Networks and Image-Based Malware Classification. Retrieved from https://arxiv.org/abs/2207.00421

Noguchi, A., Sun, X., Lin, S., & Harada, T. (2022). Unsupervised Learning of Efficient Geometry-Aware Neural Articulated Representations. *Unsupervised Learning of Efficient Geometry-Aware Neural Articulated Representations*. Retrieved from https://arxiv.org/abs/2204.08839

Owoh, N., Adejoh, J., Hosseinzadeh, S., Ashawa, M., Osamor, J., & Qureshi, A. (2024). Malware Detection Based on API Call Sequence Analysis: A Gated Recurrent Unit–Generative Adversarial Network Model Approach. *Future Internet, 16*. doi:10.3390/fi16100369

Owoh, N., Adejoh, J., Hosseinzadeh, S., Ashawa, M., Osamor, J., & Qureshi, A. (2024). Malware Detection Based on API Call Sequence Analysis: A Gated Recurrent Unit–Generative Adversarial Network Model Approach. *Future Internet, 16*, 369. Retrieved from https://www.mdpi.com/1999-5903/16/10/369

Pinto, A., Gonçalo Oliveira, H., & Alves, A. (2016, June). Comparing the Performance of Different NLP Toolkits in Formal and Social Media Text. *51*, 3:1-. doi:10.4230/OASIcs.SLATE.2016.3

Samuels, J. (2024, January). One-Hot Encoding and Two-Hot Encoding: An Introduction. *One-Hot Encoding and Two-Hot Encoding: An Introduction*. doi:10.13140/RG.2.2.21459.76327

Stalin, K., & Mekoya, M. B. (2024). Improving Android Malware Detection Through Data Augmentation Using Wasserstein Generative Adversarial Networks. Retrieved from https://arxiv.org/abs/2403.00890

Thanh, N. H., Pham, T. D., & Bui, L. (2025). Mal-D2GAN: Double-Detector Based GAN for Malware Generation. Retrieved from https://arxiv.org/abs/2505.18806

Zhang, Y., Li, X., & Wang, Z. (2023). Malware Detection with Dynamic Evolving Graph Convolutional Networks. Retrieved from https://www.researchgate.net/publication/359581210_Malware_detection_with_dynamic_evolving_graph_convolutional_networks

Zhou, Y., & Jiang, X. (2012). Dissecting Android Malware: Characterization and Evolution. *2012 IEEE Symposium on Security and Privacy*, (pp. 95-109). doi:10.1109/SP.2012.16