Lab 2 – Model Engineering

Pierfrancesco Elia - s331497, Alessandro Meneghini - s332228, Ankesh Porwal - s328746

Overview

This lab investigates three types of deep learning architectures—Feed Forward Neural Networks (FFNN), Recurrent Neural Networks (RNN), and Graph Neural Networks (GNN)—for classifying malware based on API call sequences. The dataset consists of variable-length API sequences extracted from malware and goodware binaries, and each model requires specific preprocessing techniques. Our goal is to compare their effectiveness, computational cost, and ability to generalize.

1 Task 1: Bag-of-Words (BoW) Approach

1.1 Objective

In this task, we aim to apply classical text-based preprocessing techniques to the API call sequences and evaluate their effectiveness in malware classification. Each sample in the dataset is a sequence of API calls represented as a list of strings. We use the `CountVectorizer` method from scikit-learn to transform these sequences into fixed-length vectors.

1.2 Data Preprocessing

We first convert each sequence of API calls into a space-separated string. This allows us to treat the sequence as a "sentence" of API tokens:

`["OpenProcess", "WriteFile", "CloseHandle"]` → `"OpenProcess WriteFile CloseHandle"`

This string representation enables the use of `CountVectorizer`, which generates a fixed-length feature vector where each dimension corresponds to the frequency of a specific API call.

1.3 Vocabulary and Feature Space

Q: How many columns do you have after applying the CountVectorizer? What does that number represent?
After applying `CountVectorizer` to the training set, we obtain a total of **253 features**. This number represents the size of the vocabulary, i.e., the number of distinct API calls encountered in the training samples.
Q: What does each row represent? Can you still track the order of the processes (how they were called)?
Each row in the resulting feature matrix corresponds to a single sample and encodes the count of each API call within that sample. It is important to note that this representation **does not preserve the order** of API calls—any temporal relationships between them are lost.

1.4 Handling OOV Tokens

Q: Do you have out-of-vocabulary from the test set? If yes, how many? How does CountVectorizer handle them?
To evaluate the presence of out-of-vocabulary (OOV) tokens in the test set, we computed the set difference between the test and training vocabularies. We identified **5 OOV API calls**:

{`ObtainUserAgentString, WSASocketA, ControlService, WSARecv, NtDeleteKey`}

`CountVectorizer` automatically handles such tokens by ignoring them during transformation. Therefore, they do not negatively impact the input dimensionality or the model.

1.5 Classifier and Hyperparameter Selection

As a baseline, we chose **Logistic Regression**, a shallow linear model that is well-suited for high-dimensional sparse data like BoW representations.
We performed a grid search with 3-fold cross-validation over the following hyperparameter space:

- $C \in \{0.01, 0.1, 1, 10\}$
- solver $\in \{$`liblinear`, `lbfgs`$\}$
- max_iter $\in \{1000, 2000, 5000\}$

The best performing configuration was:

`C=10, solver='liblinear', max_iter=1000`

The final classifier was evaluated on the test set after standardizing the features with `StandardScaler`. The results are summarized below:

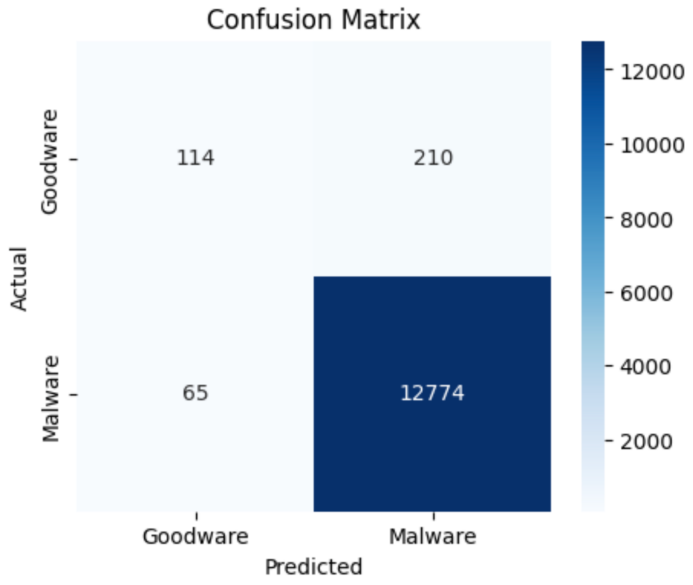
- **Test Accuracy**: 97.91%
- **Precision, Recall, F1**: Reported for each class in the classification report

```
Classification Report:

              precision    recall  f1-score   support

    Goodware       0.64      0.35      0.45       324
     Malware       0.98      0.99      0.99     12839

    accuracy                           0.98     13163
   macro avg       0.81      0.67      0.72     13163
weighted avg       0.98      0.98      0.98     13163
```




```
Test Accuracy: 97.91%
```

Figure 1: Confusion matrix for Logistic Regression classifier on test set.

The Bag-of-Words model, despite its simplicity, provides a strong baseline for this classification task. However, it has significant limitations:

- **Loss of temporal information**: It cannot distinguish between different sequences with the same API call frequencies.
- **Sparse representation**: Leads to high-dimensional inputs, which can be inefficient for neural networks.

Nevertheless, the model achieved high accuracy, highlighting that API call frequency alone is a strong indicator of malicious behavior. Future tasks will investigate whether deep models (FFNN, RNN, GNN) can exploit sequential or structural information to further improve performance.

Task 2: Feedforward Neural Network

Objective

The goal of this task is to model the malware classification problem using a Feedforward Neural Network (FFNN). Since FFNNs require fixed-size input vectors, specific preprocessing steps are needed to adapt the variable-length API call sequences to the expected input format.

Sequence Length Statistics

Q: Do you have the same number of calls for each sample? Is the training distribution the same as the test one?
We first analyzed the number of API calls per sample:

- **Training set**: sequences range from **60 to 90** API calls
- **Test set**: sequences range from **70 to 100** API calls

This confirms that the two datasets differ in distribution, and that the test set contains generally longer sequences.

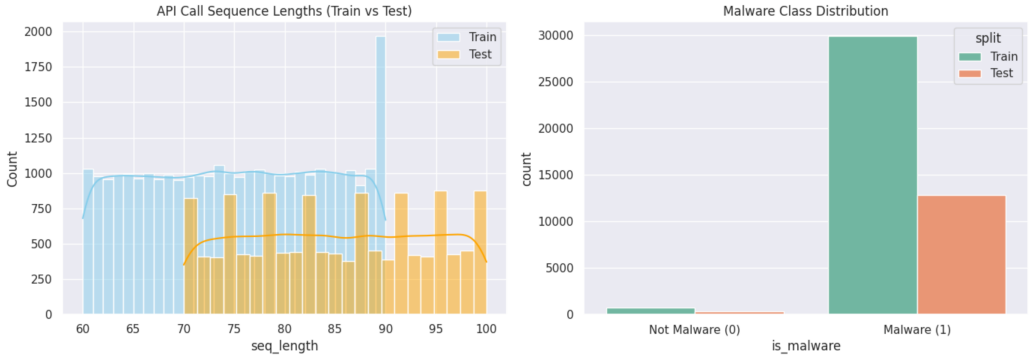


Figure 2: Histogram of API call sequence lengths (Train vs Test) and class distribution.

Handling Variable-Length Input

Q: Can a FFNN handle a variable number of elements? Why?
No. A FFNN has a fixed architecture where each input sample must match the shape expected by the input layer. The number of neurons in the first layer is fixed at model definition time, so every input sample must be of the same size.
Q: What technique is used to bring everything to a fixed size? What happens at test time?
We used a combination of:

- **Padding**: Shorter sequences were padded with zeros up to the length of the longest training sequence (90).
- **Truncation**: Longer test samples (which could be up to 100) were truncated to fit the model's input dimension.

This ensures compatibility between training and inference, although truncation may discard relevant information.

Encoding API Calls

Q: Each process is a string. How was it handled?
To represent each process (i.e., a sequence of API calls), we experimented with two encoding strategies that convert symbolic API names into numeric representations:

1. Sequential Identifiers

In this approach, each unique API call is assigned an integer ID, transforming a sequence such as:

```
["OpenProcess", "WriteFile"] → [42, 113]
```

This encoding is simple and compact but treats all API calls as independent tokens. It does not capture semantic similarities between functionally related API calls (e.g., `CreateFile` and `OpenFile` are entirely unrelated numerically).
Limitations: Lack of semantic structure may hinder generalization and increase overfitting, especially when different but similar API calls appear sparsely.

2. Learnable Embeddings

We employ a learnable embeddings layer that maps each token ID to a trainable dense vector. This allows the model to capture semantic relationships between API calls by positioning similar functions close together in the embedding space during training.
This helps capture relationships between similar actions and improves generalization to unseen patterns.
Advantages: Embeddings provide a richer representation, improving both expressiveness and robustness. This is especially helpful for sequences with rare or noisy API calls.

Hyperparameter Search Space

Q: How were hyperparameters selected? We conduct a grid search over the selected hyperparameters, incorporating an early stopping criterion with a patience of 5 epochs to prevent overfitting and improve training efficiency:

Table 1: Grid search configuration for neural model.

Hyperparameter	**Values**
Learning rate	`[0.0005, 0.005, 0.05]`
Hidden layers	`[[32, 64], [64, 32, 16], [64, 128, 128, 64]]`
Batch size	`[16, 32, 128]`
Optimizer	`SGD+Momentum(0.3)`, `SGD+Momentum(0.9)`, `AdamW`

In total, the grid search evaluates $3 \times 3 \times 3 \times 3 = \mathbf{81}$ different combinations.

Model Architecture and Training

1.5.1 Sequential Identifiers

After performing a full grid search over hyperparameter combinations, using sequential identifiers, the best configuration is:

- Hidden layers: `[64, 128, 128, 64]`
- Batch size: `32`
- Optimizer: `SGD+Momentum(0.3)`
- Learning rate: `0.005`

Test Accuracy: `98.32%`

Table 2: Classification report for the best model with sequential identifiers.

Class	**Precision**	**Recall**	**F1-score**	**Support**
0 (Not Malware)	0.87	0.37	0.52	324
1 (Malware)	0.98	1.00	0.99	12,839
Accuracy			**0.9832**	13,163
Macro avg	0.93	0.69	0.76	13,163
Weighted avg	0.98	0.98	0.98	13,163

Figure 3 shows the training and validation loss curves over the epochs, providing insight into the model's convergence behavior and overfitting.

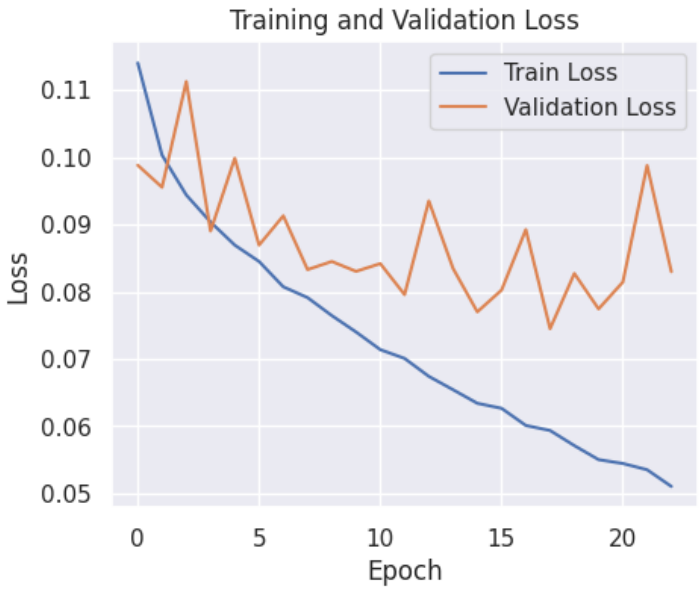


Figure 3: Training and validation loss over epochs using sequential identifiers.

1.5.2 Learnable Embeddings

After performing a full grid search over hyperparameter combinations, using learnable embedding, the best configuration is:

1.5.3 Learnable Embeddings

After performing a full grid search over hyperparameter combinations, using learnable embeddings, the best configuration is:

- Hidden layers: `[64, 128, 128, 64]`
- Batch size: `16`
- Optimizer: `AdamW`
- Learning rate: `0.0005`

Test Accuracy: `98.44%`

Table 3: Classification report for the best model with learnable embeddings.

Class	Precision	Recall	F1-score	Support
0 (Not Malware)	0.72	0.59	0.65	324
1 (Malware)	0.99	0.99	0.99	12,839
Accuracy			**0.9844**	13,163
Macro avg	0.86	0.79	0.82	13,163
Weighted avg	0.98	0.98	0.98	13,163

Figure 4 shows the training and validation loss curves over the epochs, providing insight into the model's convergence behavior and overfitting risk.

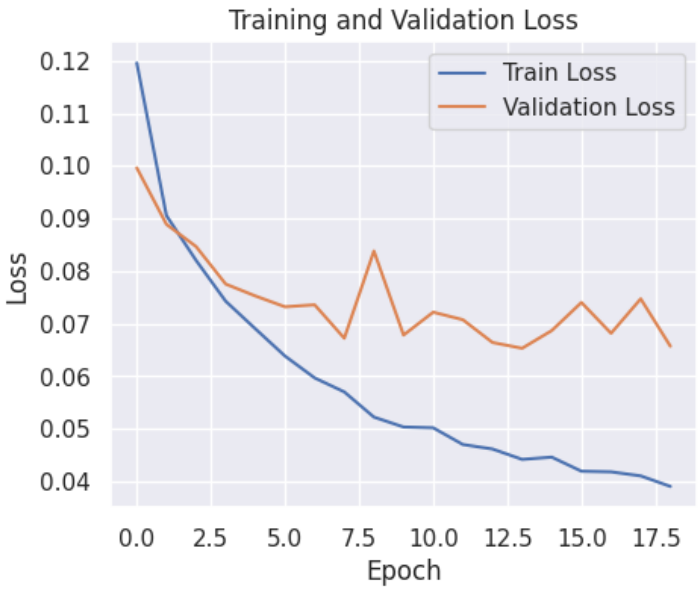


Figure 4: Training and validation loss over epochs using learnable embeddings.

Embedding vs Identifier Performance

Q: Can you obtain the same results with both approaches?
No. Although both encoding methods achieve high accuracy, they differ in training dynamics and generalization.
The model using learnable embeddings takes slightly longer to converge due to the additional parameters being optimized. However, it is more stable during training and better generalizes to unseen data. In contrast, the version using plain sequential identifiers converges faster initially but exhibits slower improvement and slightly lower performance overall.

- **Embeddings**: More robust to input variation, improved generalization on validation and test sets.
- **Sequential Identifiers**: Simpler and faster to implement, but more prone to overfitting and local optima.

Despite these differences, both models are effective. Nevertheless, due to their inability to capture temporal dependencies, FFNN-based approaches are eventually limited in modeling sequential data. This motivates the adoption of recurrent and graph-based models in subsequent tasks.

Task 3: Recurrent Neural Networks (RNN)

Objective

This task explores the application of Recurrent Neural Networks (RNNs), Bi-directional RNNs (BiRNNs), and Long Short-Term Memory networks (LSTMs) for the binary classification of software (malware vs goodware) based on sequences of API calls.

Handling Input Sequences

Q: Do you still have to pad your data? If yes, how?
Yes. While RNNs can process sequences of varying lengths during training via `pack_padded_sequence`, we still need to pad sequences in the dataset to form uniform batches. We used the `pad_sequence` utility from PyTorch, setting a special `<PAD>` token (index 0) as the padding value.
Q: Do you have to truncate the testing sequences?
No truncation was necessary. Since RNNs handle variable-length input by design (with sequence lengths passed explicitly to `pack_padded_sequence`), we preserve the full test sequence for each sample, maintaining all temporal information.

Memory and Temporal Advantages

Q: Is there any memory advantage of using RNNs vs FFNNs? Why?
Yes. RNNs process inputs sequentially and reuse hidden state memory across timesteps. This enables them to maintain a form of temporal memory, making them more efficient and expressive when dealing with sequences. Unlike FFNNs, RNNs do not require flattening or heavy padding of input data.

Model Variants and Architecture

We implemented three RNN variants using PyTorch:

1. **Simple one-directional RNN**: Processes the sequence in a left-to-right manner.
2. **Bi-directional RNN (BiRNN)**: Aggregates information from both past and future contexts.
3. **Long Short-Term Memory (LSTM)**: Incorporates forget and memory gates to better handle long-term dependencies.

All models use:

- Embedding layer of size 64
- Hidden dimension: 64
- Binary output layer with sigmoid activation
- BCE Loss and Adam optimizer (lr = 0.0005)
- Early stopping with patience = 20

Training Time Comparison

Q: Is your network as fast as the FFNN? If not, where does the overhead come from?
No. RNN-based models were significantly slower to train compared to FFNNs. The overhead comes from:

- Sequential nature of computation across timesteps (cannot parallelize easily)
- Packing and unpacking of variable-length sequences
- Backpropagation through time (BPTT) adds computational complexity

Results and Comparison

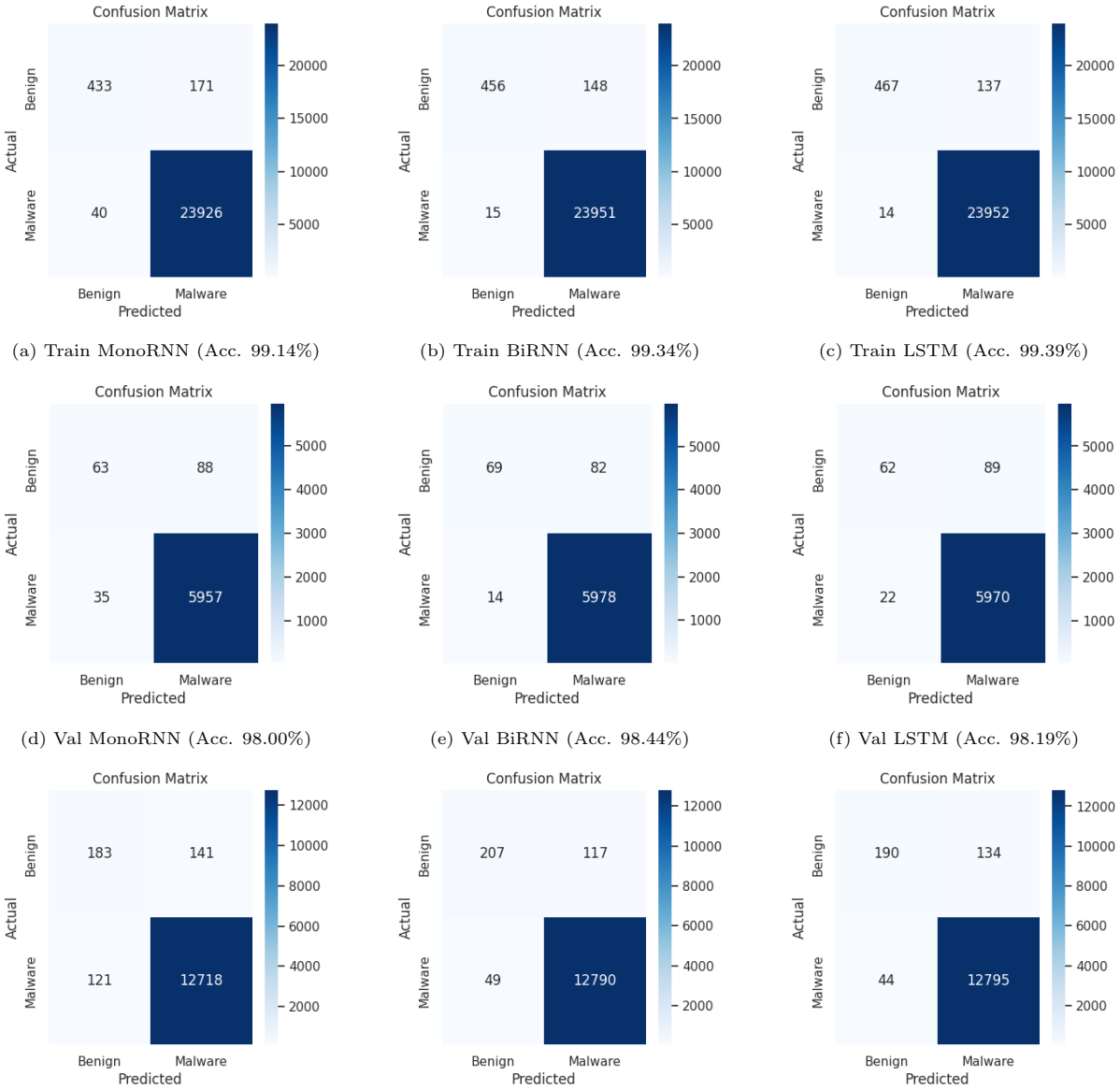
We trained all models on the same dataset and evaluated them on train, validation, and test sets.

Model	Train Acc.	Val Acc.	Test Acc.
Simple RNN	99.14%	98.00%	98.01%
Bi-directional RNN	99.34%	**98.44%**	**98.74%**
LSTM	**99.39%**	98.19%	98.65%

Table 4: Accuracy comparison of RNN-based models.

Training Behavior

Q: Can you see any differences during their training?
Yes. The BiRNN and LSTM converged faster and showed more stable validation loss, while the simple RNN exhibited greater fluctuations during training.



(a) Train MonoRNN (Acc. 99.14%) (b) Train BiRNN (Acc. 99.34%) (c) Train LSTM (Acc. 99.39%)

(d) Val MonoRNN (Acc. 98.00%) (e) Val BiRNN (Acc. 98.44%) (f) Val LSTM (Acc. 98.19%)

(g) Test MonoRNN (Acc. 98.01%) (h) Test BiRNN (Acc. 98.74%) (i) Test LSTM (Acc. 98.65%)

Figure 5: Confusion matrices for MonoRNN, BiRNN, and LSTM models across training, validation, and test sets. Accuracy is reported in parentheses.

Conclusion

RNN-based architectures outperform FFNNs in handling sequential API call data due to their ability to preserve temporal dependencies. Among them, BiRNNs and LSTMs yielded the best performance. Despite their slower training time, the memory-aware structure of these models enables better generalization and classification accuracy on variable-length data.

Task 4: Graph Neural Network (GNN)

Introduction

In this task, we explored the application of Graph Neural Networks (GNNs) to the problem of malware classification using API call sequences. Unlike Feedforward Neural Networks (FFNNs) and Recurrent Neural Networks (RNNs), GNNs provide a powerful framework for capturing the structural relationships among API calls, modeling each sample as a graph.

Q: Do you still have to pad your data?

Padding is not required in the same way as for FFNNs or RNNs. Each API call sequence is treated as an individual graph with a variable number of nodes and edges. The PyTorch Geometric framework allows us to batch graphs of different sizes without needing to pad or truncate them explicitly.
Q: Do you have to truncate the testing sequences?
No, there is no need to truncate the test sequences. Since GNNs handle sequences as graphs, they can naturally work with varying sizes, provided each graph is correctly constructed with valid edges and node features.

Why GNNs? Advantages and Trade-offs

Q: What is the advantage of modelling your problem with a GNN compared to FFNN and RNN? Are there any disadvantages?
GNNs offer key advantages:

- They model the relational structure between elements (API calls) explicitly.
- They avoid the need for strict input dimensionality like FFNNs.
- They can better capture context via message passing.

However, GNNs are more computationally expensive and require additional preprocessing. Moreover, sequential information might be harder to preserve compared to RNNs.

Graph Construction

Each sequence was converted into a graph:

- Nodes represent API calls and are one-hot encoded.
- Edges are added between consecutive API calls (linear chain).
- Each graph is assigned a binary label (malware or benign).

Training and Testing with GCN

Q: How long does it take to train and test in each configuration? How is it different from previous architectures?
A Graph Convolutional Network (GCN) with two convolutional layers followed by a linear classifier was implemented. Training was conducted on both CPU and GPU over 30 epochs. The training durations were:

- **CPU:** 256.90s
- **GPU:** 187.27s

Using a GPU yields a noticeable speed-up; however, training is still slower compared to feedforward neural networks (FFNNs). This difference is primarily due to the additional complexity of graph-based computations—specifically, the message-passing operations required at each GCN layer. These operations involve neighborhood aggregation, which is computationally heavier than standard matrix multiplications used in FFNNs.

Model Variants

We compared three architectures:

1. **GCN**: Standard Graph Convolutional Network that performs neighborhood aggregation using fixed, symmetric normalization of adjacency matrices.
2. **GraphSAGE**: Generates node embeddings by sampling and aggregating features from a fixed-size set of neighbors using a mean aggregator.
3. **GAT**: Graph Attention Network that computes attention coefficients to learn the relative importance of neighboring nodes during aggregation.

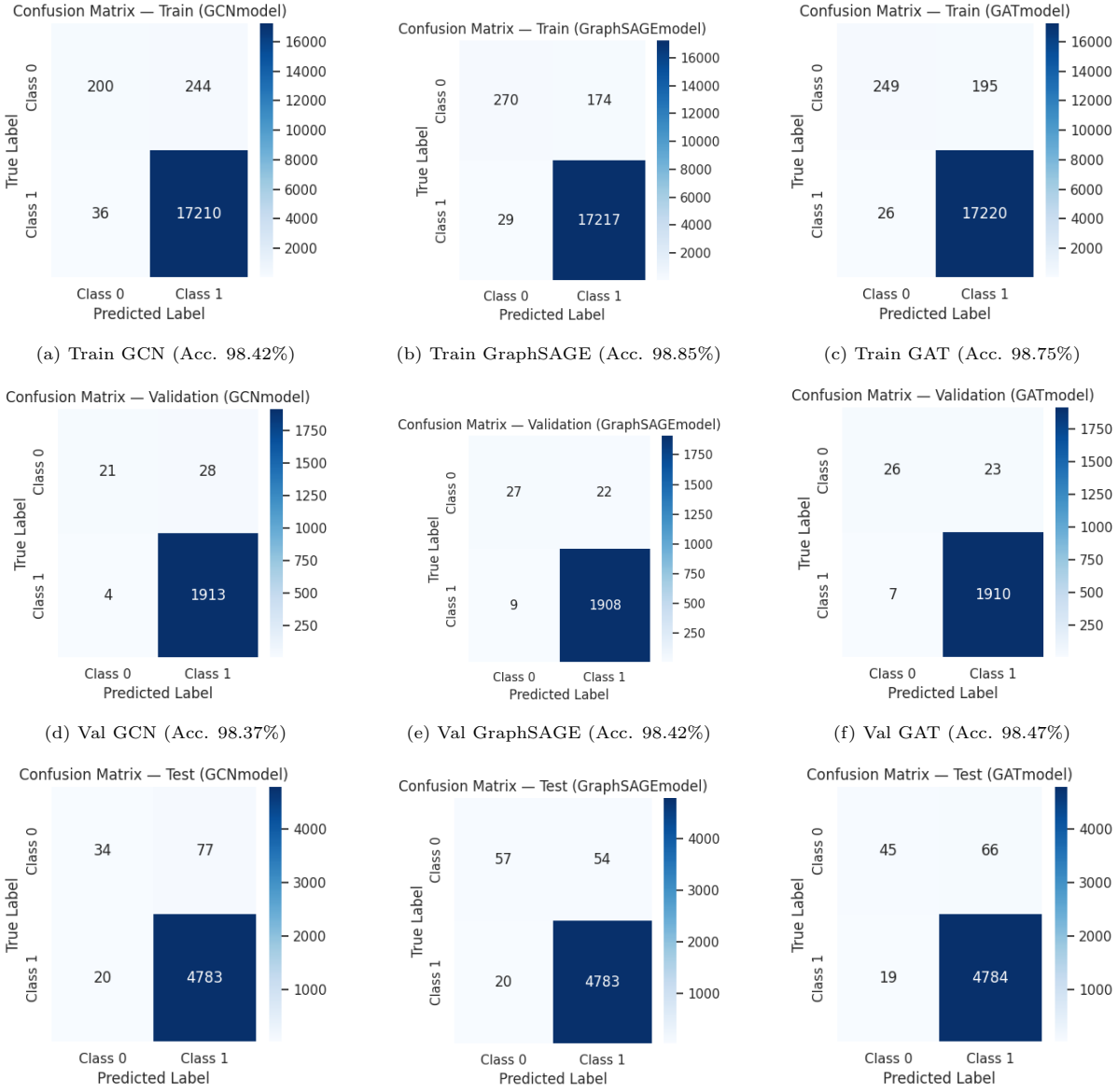
Evaluation and Results

Q: Can you see any differences in your training? Can you obtain the same performance as with the previous architectures?
All models performed well with minor differences. GAT achieved the highest validation accuracy but was the slowest to train due to attention computations.

Model	Train Acc.	Val Acc.	Test Acc.
GCN	98.42%	98.37%	98.03%
GraphSAGE	98.85%	98.42%	**98.49%**
GAT	**98.75%**	**98.47%**	98.27%

Table 5: Accuracy comparison of GNN-based models.

We plotted confusion matrices for each model. All models showed good class separation, confirming relatively low false positive and false negative rates.



(a) Train GCN (Acc. 98.42%)

(b) Train GraphSAGE (Acc. 98.85%)

(c) Train GAT (Acc. 98.75%)

(d) Val GCN (Acc. 98.37%)

(e) Val GraphSAGE (Acc. 98.42%)

(f) Val GAT (Acc. 98.47%)

(g) Test GCN (Acc. 98.03%)

(h) Test GraphSAGE (Acc. 98.49%)

(i) Test GAT (Acc. 98.27%)

Figure 6: Confusion matrices for GCN, GraphSAGE, and GAT models across training, validation, and test sets. Accuracy is reported in parentheses.

Conclusions

GNNs demonstrated good performance in malware detection by leveraging the structural nature of API calls. GAT performed slightly better at the cost of higher computational effort. Overall, GNNs offer a robust and scalable solution for graph-structured cybersecurity data.