Indian Institute of Technology Kanpur

Department of Computer Science and Engineering

Android Malware Detection and Classification

Vraj Patel - 241110080

Pavani Priya - 220415

Akshat Shrivastav - 220103

Divya Krupa - 210274

November 7, 2024

Contents

Chapter 1

Introduction

1.1 Goal

The goal of this project is to classify Android applications as either malware or benign by analyzing system call sequences.We have utilized the Maldroid 2020 dataset which includes diverse types of Android malware and benign applications. We aimed to develop a model capable of detecting between these categories. Through a process of trial and error, we formulated four distinct approaches to achieve this objective. While the first two methods did not yield optimal results, they provided significant insights and checkpoints that guided the progression of our research. Our detection models performed well, highlighting the potential of system call analysis as an effective approach for Android malware detection.

1.2 Work Division

- **Vraj Patel:** Worked on Data Preprocessing, Model Development, Report Content
- **Pavani Priya:** Worked on Data Preprocessing and Model Development
- **Akshat Shrivastav:** Worked on Report Writing and Presentation PPT
- **Divya Krupa:** Worked on Report Writing and Presentation PPT

Chapter 2

Dataset and Preprocessing

2.1 Dataset

We utilized the "**Maldroid 2020**" dataset, which contains various types of Android malware, including Adware, Benign applications, Banking Malware, SMS Malware, and Riskware. This dataset includes:

- **Raw APK Files** – Android application files that can be run in a sandbox environment for in-depth analysis.
- **JSON Files** – Static and dynamic analysis data extracted using CopperDroid, providing comprehensive information on system calls and other operational behaviors.
- **CSV File** – Pre-extracted features compiled by the dataset authors, summarizing frequency data from the JSON files.

For this project, we primarily rely on the JSON files, as they offer rich context that enables customized data extraction specific to our classification goals.

2.2 Preprocessing

While specific preprocessing steps vary for each method(each of which has been explained in the further sections),the general preprocessing was conducted using a Python script. The main goal of this process was to extract dynamic features—particularly system call sequences—and convert them into an analysis-ready format. File corruption and encoding problems were among the difficulties we faced during this stage. To ensure uniform data processing across all approaches, these problems were resolved by implementing a number of solutions into the data extraction script.

Chapter 3

Method Overview

In pursuit of an optimal approach to utilizing system call sequences for malware detection and classification, we explored four distinct methods. Although the first two methods yielded less-than-ideal results, they served as significant checkpoints in refining our approach. Each method is detailed in the following sections, providing insights into the progression of our research.

3.1 Learning RNNs Over Simple Encoding

3.1.1 Preprocessing

From each JSON file, we gathered the sequence of system calls. At first, we only kept the `sysname` attribute to keep the model simple, while other attributes were set aside for now. Each system call was seen as categorical data and sequenced using the `Id` parameter, which indicates the order of the system calls.

From the system call sequences we extracted, we identified all unique system calls and encoded them in two ways:

- **Numeric Encoding** – Each system call is given a unique integer.
- **One-Hot Encoding** – Each system call is represented as a sparse vector.

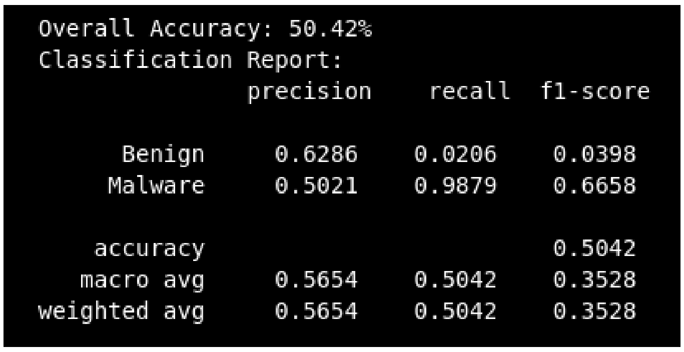
Because one-hot encoding resulted in high dimensionality and heavy memory usage, we decided to use numeric encoding for all further processing.

3.1.2 Model Training

We trained a Recurrent Neural Network (RNN) model, specifically an LSTM, on the numerically encoded syscall sequence data.

3.1.3 Evaluation

The model was evaluated using standard metrics, including accuracy, precision, recall, and F1 score. The model evaluation yielded bad results. Rendering it useless and leaving space for much improvement.



```
Overall Accuracy: 50.42%
Classification Report:
               precision    recall  f1-score

      Benign     0.6286    0.0206    0.0398
     Malware     0.5021    0.9879    0.6658

    accuracy                         0.5042
   macro avg     0.5654    0.5042    0.3528
weighted avg     0.5654    0.5042    0.3528
```

Figure 3.1: Performance of RNN with Simple encoding

3.1.4 Enhanced Feature Engineering for RNN

To boost the performance of our initial model, we added more features through further feature engineering. These features included:

- **Running Count of System Calls**: This counts how many times each system call occurs, updated as the sequence goes on.
- **Normalized Time Gap**: This measures the time difference between consecutive system calls, adjusted to highlight timing patterns.
- **Process Switch Flag**: A binary flag that shows whether a process switch happened between the current and previous system calls.

While these improvements slightly enhanced model performance, they also increased the data size, making it harder to manage with our available resources.

3.2 Graph-Based Approach

3.2.1 Preprocessing

To better represent the data, we transformed the data in a graph format, capturing syscall sequences and their relationships from each JSON file. The graph structure was created as follows:

- **Nodes**: Each node represents a system call (syscall), identified by a unique integer index for the syscall name.
- **Edges**:
  - **Sequential Edges**: These edges connect each syscall node to the next one in the sequence, maintaining the original order of the system calls.
  - **Cross-Reference Edges**: Some system calls (syscalls) refer back to earlier calls, as indicated by the "xref" field in the JSON. In this case, an edge is created that connects the current node to the node it references, providing additional context about how specific syscalls interact with each other.
- **Label**: Each graph was labeled according to its category (e.g., Adware, Benign), represented as a single integer label. This label enables the graph to be used for supervised classification.

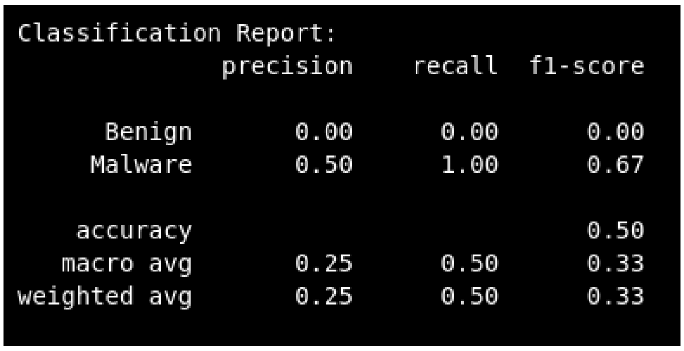
This graph structure, which consists of nodes representing syscall features, sequential and reference edges, and categorical labels, offers a thorough representation of both syscall sequences and their interconnections. The graphs were saved in `.pt` format for training the model.

3.2.2 Model Training

We trained a Graph LSTM classifier using the converted graph data. Our goal was to learn from both the sequence of system calls and the interactions that occur within the syscall patterns.

3.2.3 Evaluation

The outcomes from the graph-based model were not satisfactory, likely due to the complexity and size of the data. The model found it challenging to identify meaningful patterns, which could be attributed to the difficulties in learning from such intricate and large graph structures.



```
Classification Report:
              precision    recall  f1-score

      Benign       0.00      0.00      0.00
     Malware       0.50      1.00      0.67

    accuracy                           0.50
   macro avg       0.25      0.50      0.33
weighted avg       0.25      0.50      0.33
```

Figure 3.2: Graph LSTM evaluation

3.3 N-Gram Based Approach

To address the difficulties caused by the heavy and complex features extracted from the raw data, we aimed for a simpler yet effective method that could still capture local sequential patterns. By using n-grams, we were able to reduce data complexity while maintaining meaningful syscall sequences.

3.3.1 Preprocessing

To capture local patterns without overloading the model, we concentrated on extracting n-grams from the syscall sequences:

- **Unique Syscalls**: We first identified all unique system calls from the dataset.
- **N-Gram Construction**: We generated all possible n-grams, focusing on 2-grams for processing. This choice helped us balance the need to capture patterns while managing memory constraints.
- **Frequency Count**: For each APK, we recorded the frequency of each 2-gram as input features. We also included selected static features from the JSON data to enhance classification accuracy.

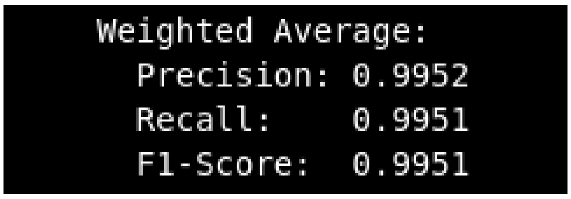
3.3.2 Model Training

We trained several models using the processed n-gram data, and we achieved the best results with **Random Forest** model. The other models we tested are:

- **Support Vector Machine (SVM)**,We applied the Support Vector Machine model, which was trained on the first 50 principal components for reducing dimensionality, helping the model to concentrate on the most relevant features.
- **Recurrent models** We also experimented with recurrent models, particularly Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks.

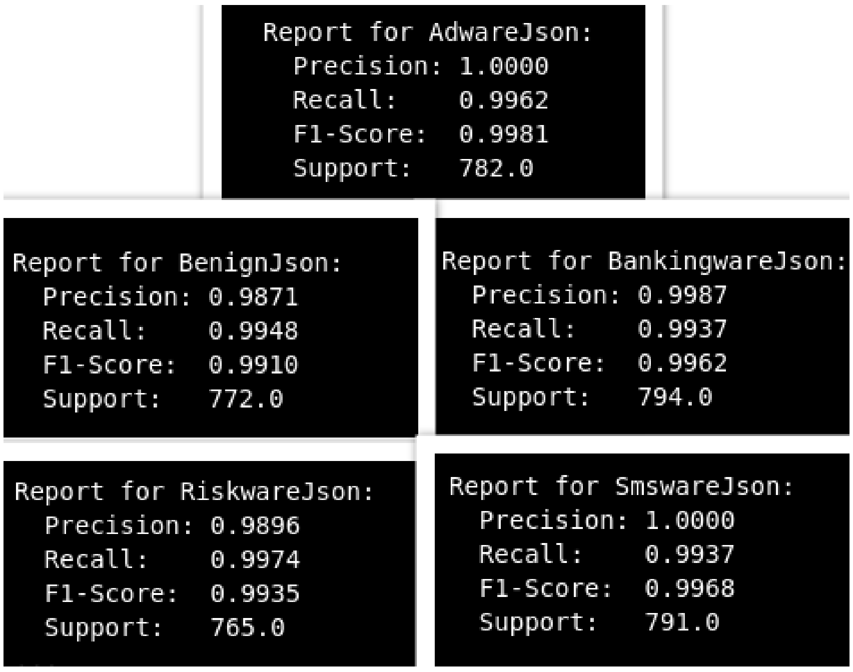
3.3.3 Evaluation

The results we got from this approach were good, giving an effective classification performance and also successful in reducing data complexity.



```
Weighted Average:
  Precision: 0.9952
  Recall:    0.9951
  F1-Score:  0.9951
```

Figure 3.3: Weighted average performance measures for random forest



```
Report for AdwareJson:
  Precision: 1.0000
  Recall:    0.9962
  F1-Score:  0.9981
  Support:   782.0
```

```
Report for BenignJson:
  Precision: 0.9871
  Recall:    0.9948
  F1-Score:  0.9910
  Support:   772.0
```

```
Report for BankingwareJson:
  Precision: 0.9987
  Recall:    0.9937
  F1-Score:  0.9962
  Support:   794.0
```

```
Report for RiskwareJson:
  Precision: 0.9896
  Recall:    0.9974
  F1-Score:  0.9935
  Support:   765.0
```

```
Report for SmswareJson:
  Precision: 1.0000
  Recall:    0.9937
  F1-Score:  0.9968
  Support:   791.0
```

Figure 3.4: Classwise performance measures for random forest

3.4 Transformer-Based Approach

For our final method, we utilized the powerful capabilities of transformers, which are good at managing the extensive and complex context found in JSON-based syscall sequences. Transformers enable high-context learning through custom embeddings, treating each system call (syscall) like a word in natural language processing. This approach helps capture both the sequence and the contextual relationships between calls.

3.4.1 Preprocessing

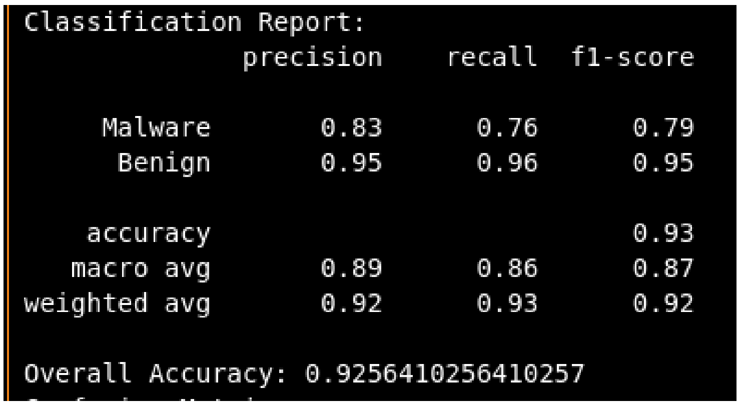
Due to computational constraints, we chose to focus solely on the `sysname` attribute from the JSON files. This allowed us to create a simplified, custom embedding for syscall sequences. Each unique syscall was treated as a word to establish a vocabulary, and the sequence from each JSON file was transformed into a sentence-like format. This setup is essential for facilitating effective learning using transformer models.

3.4.2 Model Training

Using the custom embeddings we created, we trained a transformer model on the syscall sequences. The model was set up to learn the patterns of the syscalls and to update the embeddings as it trained. This helped it capture more complex relationships within the sequences.

3.4.3 Evaluation

The early results from this approach looked promising, showing that transformers could be very effective if we trained them on the full dataset with better computing power. Additionally, this method could allow us to use fine-tuning techniques, which might help the model explain why it classifies certain sequences as malware.



```
Classification Report:
              precision    recall  f1-score

     Malware       0.83      0.76      0.79
      Benign       0.95      0.96      0.95

    accuracy                           0.93
   macro avg       0.89      0.86      0.87
weighted avg       0.92      0.93      0.92

Overall Accuracy: 0.9256410256410257
```

Figure 3.5: Performance of Transformer Model

Chapter 4

Conclusion

After training and evaluating all four models, we drew following conclusions on effectiveness of different approaches to malware classification using system call sequences.

4.1 Poor Performance

4.1.1 Simple Encoding

Application of basic numerical encoding on the system call sequence data did not provide model with the sufficient context. Result were very poor in terms of both Malware detection and classification. This method did not have depth needed to capture meaning full pattern in the raw data.

4.1.2 Graph Neural Network

Although we tried to add rich context in graph-based structures by adding additional parameters to system call sequence, they failed to produce satisfactory result. The complexity of handling large graph data set with size ranging upto hundreds of MBs posed a challenge. We suspect model struggled to detect pattern within this big data structure. Additional improvement can be made in department of data structure manufacturing and feature selection in future. We may also benefit by better computational resources since it took 14 hours to train this model for 10 epochs.

4.2 Good Performance

4.2.1 N-Grams

N-gram based approach effectively captured local sequential information in system call sequences whiles maintaining light weight data. Focusing on local patterns rather than complex, global sequences and incorporation of static features helped this model achieve outstanding results.

4.2.2 Transformer Model

The transformer-based approach, utilizing custom embeddings for system call sequences, demonstrated strong performance. While this initial model used only syscall names, incorporating additional data such as syscall parameters and process names could further improve its accuracy. Additionally, fine-tuning a pre-trained transformer model like BERT on the extracted data offers potential for developing a classifier that not only achieves higher accuracy but also provides interpretability, explaining the reasoning behind each classification.

4.3 Final Remarks

In conclusion, while simpler models struggled with the complexity of the dataset, approaches that captured local sequence patterns (N-grams) and those with powerful context modeling capabilities (Transformers) were notably successful. Future efforts could benefit from more sophisticated data integration and increased computational power to fully realize the potential of these advanced models.