Hybrid sequence-based Android malware category classification using NLP and deep learning with TensorFlow

Malware detection based on NLP and deep learning techniques with TensorFlow

A DISSERTATION SUBMITTED TO MANCHESTER METROPOLITAN UNIVERSITY
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

September 2023

By
Karvan Houshiar
Supervisor: Dr. Amna Eleyan
Department of Computing and Mathematics

Table of Contents

1 Abstract 3

1.1 Contributions 3

2 Declaration 4

3 Acknowledgements 5

4 Abbreviation 6

5 List of Figures 7

6 List of Tables 8

1 Introduction 9

1.1 motivation and problem definition 11

1.2 Aim and Objectives 11

1.3 Research Questions 11

1.4 Scope 11

1.5 Overview 12

2 Background and relevant works 13

2.1 Introduction to Malware Detection 13

2.1.1 What is Malware 13

2.1.2 Malware Analysis Methods 13

2.1.3 Malware Detection Methods 14

2.1.4 Machine Learning and Deep Learning Based Methods 15

2.2 Deep Learning for Natural Language Processing 15

2.2.1 Artificial Neural Networks (ANNs) 16

2.2.2 Deep Learning 17

2.2.3 Convolutional Neural Networks 17

2.2.4 Recurrent Neural Network (RNN) 18

2.2.5 GRU 18

2.3 Tokenization (N-Gram tokenizer) 19

2.4 TensorFlow and its Applications 21

2.5 Related Works 22

2.6 Summary 26

3 Method, Design and Implementation 27

3.1 Approach 27

3.2 Dataset 28

3.2.1 Data Collection and Formatting....28
3.3 The Model Implementation....29
3.3.1 The Environment Setup....29
3.3.2 Imported Libraries and Modules....30
3.3.3 Pre-trained Model....32
3.3.4 Binary Classification Model....34
3.4 Summary....38
4 Results and Analysis....39
4.1 Understanding the model Prediction With LIME....39
4.2 Evaluation Criteria....40
4.2.1 Classification Results....41
4.3 Comparison of the models....43
5 Conclusions and Future Work....45
5.1 Conclusion....45
5.2 Future Work....45
6 Appendix....46
7 References....55

1 Abstract

Cybercriminals are still working on improving both malware functionality and spread vectors. Malware is increasingly spreading through legitimate channels, such as official marketplaces and ads in popular apps. Thus, proposing efficient Android malware detection methods is curial in defeating malware. This study presents a framework that leverages static analysis and deep learning techniques to address the challenges posed by the evolving threat landscape of malware on Android devices. Additionally, this study presents an Android malware detection method that utilizes natural language processing (NLP) techniques and a hybrid of convolutional neural network (CNN) and bidirectional gated recurrent unit (BiGRU) to extract features and sequence modelling from the opcode sequences. These features are then used to train the model and automate the detection of Android malware without the need for manual feature engineering. Overall, this method provides a promising approach to detecting Android malware by focusing on the classes.dex file, which contains the critical information for app execution, this framework can provide a promising approach for detecting both existing and zero-day malware and reduces computational load and with low false positives and enhance the detection capabilities against sophisticated malware. Moreover, the proposed model incorporates an explainable component that provides insights into the features contributing to the final detection outcome. This assists security analysts in the decision-making process. The performance of this model is evaluated using a real-world data set. The evaluations show the effectiveness and flexibility the model, achieving high detection accuracies detecting both known and unseen malware.

1.1 Contributions

The key contribution of this study is the implementation of the framework proposed by (Maniriho et al., 2022), applied specifically to opcode sequence features extracted from Android applications. Additionally, the outcomes and rationale behind the results are explained using the LIME methodology.

2 DECLARATION

No part of this project has been submitted in support of an application for any other degree or qualification at this or any other institute of learning. Apart from those parts of the project containing citations to the work of others, this project is my own unaided work. This work has been carried out in accordance with the Manchester Metropolitan University research ethics procedures and has received ethical approval number 58590.

Signed: Karvan Houshiar

Date: 06/10/2023

3 ACKNOWLEDGEMENTS

I would like to thank Dr. Amna Eleyan for being the supervisor for this thesis and providing feedback on the work and writing process.

4 ABBREVIATION

AI Artificial Intelligent

ANN Artificial Neural Network

API Application Programming Interface

CNN Convolutional Neural Network

BiLSTM Bidirectional Long Short-Term Memory

CNNs Convolutional Neural Networks

DL Deep Learning

DNN Deep Neural Network

HMMs Hidden Markov Models

LSTM Long Short-Term Memory networks

ML Machine Learning

MLP Multilayer Perceptron

NLP Natural Language Processing

NN Neural Network

PCA Principal Component Analysis

RNNs Recurrent Neural Networks

RF Random Forests

SVM Support Vector Machines

5 List of Figures

Figure 1:The architecture of CBOW is composed of three layers, an input layer, a hidden layer, and an output layer. It conducts vector projection operations within the hidden layer. CBOW employs the context to predict the central word(Li et al., 2023). 16
Figure 2:The architecture of CNN(Maniriho et al., 2022). 18
Figure 3:The structure of GRU(Maniriho et al., 2022). 18
Figure 4:Structure of the APK file(Shinho Lee et al., 2019). 20
Figure 5:Dex format 20
Figure 6:A sample of extracting opcode sequences of Dex files (Shinho Lee et al., 2019). 21
Figure 7:Example of Dalvik opcode in a .smali file. 28
Figure 8: N-gram opcode sequence extraction(Yeboah and Baz Musah, 2022). 32
Figure 9: Virtualised opcode sequences in 2D 34
Figure 10:The proposed framework for android malware detection 35
Figure 11:: Explaining the predicted outcome(Ribeiro et al., 2016b). 39
Figure 12: An example of how LIME explains the classification result. 40
Figure 13:Accuracy and Val_accuracy trends over 100 epochs 41
Figure 14:Cell 1: Import Libraries: 46
Figure 15:Cell 5: Load Data and Preprocess(Sharma, 2019; Brownlee, 2020; Řehůřek, 2022): 48
Figure 16:Cell 6: Visualize Word Embeddings(Tam, 2021; Dp, 2023): 49
Figure 17:Cell 7: Train/Test Split(Sharma, 2019; scikit, 2023a): 49
Figure 18:Cell 8: Model Definition and Training(Sharma, 2019; Raihan, 2021; 'The Sequential model,' 2023): 50
Figure 19:Cell 9: Test Model(Sharma, 2019; scikit, 2023b): 51
Figure 20:Cell 11: Lime Explanation(M. T. Ribeiro, 2016; 'Explain your model predictions with LIME,' 2020): 53
Figure 21:Cell 13: Training Visualization(Sharma, 2019): 54

6 LIST OF TABLES

Table 1:The effects of epochs on accuracy....37
Table 2:The effects of Learning rate on validation losses. 37
Table 3:Detection accuracy achieved by the mode (n is the numbers of epochs) 41
Table 4:The Confusion matrix....42
Table 5:The performance metrics for the model 42
Table 6:Comparing the proposed model against previous works. 44

1 INTRODUCTION

The use of mobile devices is on the rise. As of 2023, there are 2.3 billion Android users worldwide, with Android's share of the mobile OS market worldwide at 70.8%('Android - statistics & facts,' 2023). This increase is partly due to the rapid advancement in the capabilities of smartphones and tablets and the growing prevalence of bring-your-own-device (BYOD) policies, which allow users to access corporate networks with their personal technology. The statistics below are derived from detection verdicts of Kaspersky products, received from users who have consented to provide statistical data. Among these, the largest share of attacks on mobile users can be attributed to malware, accounting for 79.76%, followed by adware at 17.86% and riskware at 2.38%(Shishkova, 2023).

The incidence and complexity of malware are steadily rising, posing a significant threat to mobile security, and particularly targeting Android smartphones. To mitigate the escalating presence of Android malware, several efficacious detection methodologies have been proposed (Muhammad, 2022). Many security researchers and cybersecurity vendors, in recent years, have concentrated on addressing the detection dilemma utilizing machine learning (ML) and Deep learning (DL) techniques and have achieved excellent results. However, they have faced challenges such as the degradation of detection rate over time due to the ever-changing data distribution in the cybersecurity landscape. This is caused by some main factors, the constant release of new versions of malware files by active adversaries, which significantly differ from previously known malware files, and the emergence of new types of benign executables produced by software companies as these new benign files that go unseen earlier may occasionally be falsely detected and can create serious consequences for users. These changes in data distribution lead to a decline in the effectiveness of these solutions in detecting malware and, maintaining a flexible architecture that allows for model updates "on the fly" between retraining and establishing effective processes for collecting and labelling new samples, enriching training datasets, and regularly retraining models are crucial for overcoming this challenges(kaspersky, 2021).

Generally, Android malware detection strategies based on ML/DL are categorized into static analysis, dynamic analysis, and hybrid analysis. Static analysis provides efficient and proactive protection by examining code, properties, and metadata. It doesn’t require the setting of execution environments and has relatively low computational overheads. This method is quick and resource-efficient as it doesn’t necessitate running the application, enabling it to detect malware before execution and prevent potential damage(Or-Meir et al., 2019). Unlike static analysis, dynamic analysis employs a behaviour-centric methodology to determine the functionality of malware. This is achieved by observing the actions undertaken by the malware after executing within a controlled, sandbox environment (Mahdavifar et al., 2020).These existing dynamic analysis tools are imperfect and that there is no single tool that can cover all aspects of malware behaviour(Or-Meir et al., 2019). Alternatively, researchers categorize ML-based Android malware detection methods into two types based on the features considered: semantic features (such as Application Programming Interface [API] Call, Intent, Hardware, and Permission) and syntax features (like the sequence of opcode). Several studies have demonstrated the efficacy of opcodes and system calls in identifying malware(Chen et al., 2018). However, numerous such methods are dependent on expert analysis to formulate the distinctive features that are conveyed to the machine learning system responsible for making the final classification decision. ML/DL models learn and make predictions or decisions based on the input data they receive and ensure the availability of accurate data, as it is the driving force behind machine learning. The data must be representative, pertinent to the current malware landscape, and properly labelled when necessary(kaspersky, 2021). Opcode has been chosen due to its consistent sequence, regardless of alterations in the Dalvik bytecode; this consistency prevails even with modifications to the Package Name, Class Name, Method Name, or the addition of one or more variables in reusable codes(Shinho Lee et al., 2019). For the dataset employed in this study, (Yeboah and Baz Musah, 2022) parse Dex files in APKs (Android Package, compressed by zip) to extract the opcode sequences. Opcode sequences are considered low-level data, as they are directly derived from the structure or content of a file without necessitating additional processing or interpretation. machine learning models have shown to be insufficient to solve real-world problems with intrinsic complexity and massive amounts of data since they depend on a manual feature extraction process. Consequently, deep learning emerges as a specialized approach within machine learning. It enables the extraction of high-level features from such low-level data, enhancing the interpretability and utility of the information. Deep learning algorithms take care of extracting abstract and flexible features automatically from raw data that help generalization in the classification.

Natural Language Processing (NLP) is a branch of artificial intelligence that involves the design and implementation of systems and algorithms able to interact through human language. Thanks to the recent advances of deep learning, NLP applications have received an unprecedented boost in performance (Lauriola et al., 2022). NLP techniques can be adapted for malware detection by treating malware-related data, such as opcode sequences, as natural language and utilized the resulting as feature vectors for malware detection. By leveraging NLP techniques, malware detection can benefit from the advancements made in the field of natural language processing, such as sequence modelling, and pattern recognition(Mehta et al., 2023).

Most existing ML/DL techniques for Android malware detection function as black boxes, processing input through a series of complex, non-interpretable operations to produce a predicted outcome. These models, while sophisticated, do not provide insights that are easily understandable by humans, obscuring which features significantly impact the final prediction. The utilization of explainable modules can aid researchers and security analysts in garnering more insight from detection models, allowing for a deeper understanding of the underlying logic behind predictions, thus enabling a more accurate assessment of the model's quality and ensuring correct decision-making.(Maniriho et al., 2022).

This study proposes an Android malware detection framework using a static feature (opcode sequences) based on the DL method. A Convolutional Neural Network (CNN) and Bidirectional Gated Recurrent Unit (CNN-BiGRU) have been applied as automatic feature extractors capable of classifying Android applications as benign or malicious. The experiments demonstrate that this model achieves good performance compared to state-of-the-art approaches.

1.1 MOTIVATION AND PROBLEM DEFINITION

The increasing threat of Android mobile malware has made its detection a crucial issue. The widespread adoption of Android mobile operating system and the availability of free third-party programs have led to a constant influx of evolving malicious software. The motivation lies in protecting user privacy and ensuring the security of mobile devices, considering the growing reliance on them.

This highlights the limitations of manually created detection rules in the face of rapidly increasing malware threats and the need for new, advanced protection technologies. Therefore, there is a need for an efficient, robust, and scalable malware recognition module as a key component of cybersecurity products. This underscores the importance of deep learning in enhancing malware detection by utilizing various types of data on the mobile devices.

1.2 AIM AND OBJECTIVES

The aim of this approach is to contribute and develop a binary classification model for detecting both known and zero-day malware attacks on android devices in Python using TensorFlow. The framework is based on natural language processing (NLP) and deep learning techniques. It aims to enhance traditional malware detection techniques by incorporating static analysis and by analysing opcode sequences, which may improve the design process of malware detectors.

The method will be implemented and trained on a publicly dataset to evaluate its performance compared to prior and state-of-arts works.

To do this, the following objectives are followed:

- Conduct a literature review on the definition and types of malware, NLP Techniques, deep learning, TensorFlow and big data processing.
- Research what state-of-the-art deep learning methods are available and what datasets and data collection methods are used in recent research.
- Implement and train the method from the public dataset available in the market to test the malware detection algorithms performance.
- Compare the algorithm with its prior ones to see if it has the best performance.

1.3 RESEARCH QUESTIONS

RQ1: Can a static-based approach, utilizing natural language processing (NLP) and deep learning, improve the detection of both known and zero-day Android malware while reducing the false positive rate?

RQ2: How can explainable modules, such as LIME, provide insights and explanations for the detection models?

1.4 SCOPE

To obtain meaningful results, the scope of this study is narrowed to focus exclusively on the assembly opcodes of APKs. A dataset(Yeboah, 2022), consisting of tokenized begins and malware opcode sequences, is employed to train the Word2Vec ML model. These sequences are then transformed into vector space and padded using Word2Vec to be compatible with the CNN-BiGRU deep learning framework for feature extraction and training in Python, utilizing TensorFlow, along with several other libraries and modules.

1.5 Overview

The subsequent sections of this study are structured as follows: The background and relevant works chapter will explore the fundamentals of malware, providing insights into basic concepts, terminologies related to malware detection, and an introduction to Natural Language Processing (NLP) and Deep Learning for NLP, with a focus on TensorFlow and its applications, and Android Opcodes Sequences, supplemented by a comprehensive literature review. The method and design section will detail the solutions proposed and the approach taken in this study, focusing on the design of the binary classification model. Findings will be explained in the results and analysis section, including understanding model predictions with LIME, evaluation criteria, classification results, and a comparison of models. The study concludes by summarizing its findings, proposing potential avenues for future research in the conclusion section, and providing the Python code for the proposed model in the appendix section.

2 BACKGROUND AND RELEVANT WORKS

The chapter initially introduces various techniques employed in the detection of malware. Subsequently, it delves into the background of Natural Language Processing (NLP) and explores its applications, as well as Android opcode sequences and their applications in terms of malware detection. Following this, a concise overview of Deep Learning Algorithms, Convolutional Neural Network (CNN), and Recurrent Neural Networks (RNNs), along with their relevant subsections, is provided. Lastly, the utilization of TensorFlow, a framework for deep learning, in the context of NLP applications, is discussed.

2.1 INTRODUCTION TO MALWARE DETECTION

Benign software and malware represent two distinct categories of software, each holding significant role in the field of cybersecurity. Benign software refers to programs that are considered safe, and do not possess any malicious intent or behaviour. Such software applications serve various purposes, including acting as productivity tools and entertainment applications, and are generally designed to enhance user experience and improve efficiency. It is crucial to emphasize the importance of including benign files into the evaluation data of malware detection tools. By doing so, the effectiveness of these tools can be assessed more accurately, providing insights into their false-positive rates and overall reliability in differentiating between Benign and malicious software(Or-Meir et al., 2019).

2.1.1 What is Malware

Malware, short for malicious software, refers to any script or binary code that performs malicious activities on a computer system without the user's consent or knowledge It can take on different formats such as executables, binary shell code, scripts, and firmware. Malware can compromise the confidentiality, integrity, or availability of a system, as well as annoy or disrupt the user. It includes various types of harmful programs like viruses, worms, Trojans, spyware, ransomware, and more. Various forms of malware include viruses, worms, Trojans, spyware, ransomware, and more. The primary objective of malware is to exploit vulnerabilities in computer systems, steal sensitive information, cause system damage, or gain unauthorized access to networks and resources(Or-Meir et al., 2019).

2.1.2 Malware Analysis Methods

Malware analysis involves two distinct approaches for examining malicious applications: static analysis and dynamic analysis, each offering its own set of benefits and drawbacks. Consequently, combining both methods is a more powerful strategy to enhance malware analysis in depth. Overall, both static and dynamic analysis methods are important in detecting and preventing malware attacks.

Static Analysis

Static malware analysis refers to the process of analysing and extracting information from malware without executing or running the code. It involves examining the code structure, file characteristics, and the patterns of the code to determine if it is malicious or benign. Static analysis techniques rely on predefined rules, such as signature-based detection, code pattern matching, or behaviour profiling, to identify known malware. It can be vulnerable to evasion techniques employed by malware authors, such as code obfuscation or encryption. The goal of static analysis can be classifying and filtering out codes that can be detected using existing techniques, reducing the workload for human analysts(Or-Meir et al., 2019).

Dynamic Analysis

Dynamic malware analysis refers to the process of analysing malicious software by executing it and observing its actions in real-time. It involves running the malware in a controlled environment to understand its behaviour and identify any malicious activities. This approach allows for the detection of malicious behaviour as it happens, providing valuable information about the impact of the malware on the hosting system (Or-Meir et al., 2019).

The main objective of dynamic malware analysis is to detect and expose the malicious activity performed by the malware, such as changes made to the registry, file system, or other system components. By executing the malware and observing its actions, analysts can gain a better understanding of its behaviour and intentions, helping in the detection, classification, and categorization of malware. Dynamic analysis is considered more robust than static analysis as it is not vulnerable to evasion techniques like code obfuscation, packing, or encryption(Or-Meir et al., 2019). However, dynamic analysis does have some limitations. It only observes the executed code, potentially missing certain portions that may not be executed under specific conditions. Additionally, it can introduce computational overhead, potentially slowing down the execution of the malware. Furthermore, dynamic analysis must be performed on the specific operating system and hardware targeted by the malware(Or-Meir et al., 2019).

2.1.3 Malware Detection Methods

Malware detection methods can be categorized into two approaches, behaviour-based and signature-based methods. Each method has its strengths and limitations, and a combination of approaches is often employed to build robust and effective malware detection systems.

Signature-Based Methods

It involves inspecting code structure, file characteristics to identify if it is malicious or benign. Static analysis relies on predefined rules, like signature-based detection, to spot known malware. However, it can be vulnerable to evasion tactics like code obfuscation. The goal is to classify files using existing techniques to reduce the workload for analysts. Signature-based detection identifies known malware and adds their signatures to a database. While it is fast and effective against older threats, it struggles with new malware and faces challenges like database size management and signature distribution(Mimura and Ito, 2022).

Behaviour-Based Method

This includes tracking activities such as file operations, registry changes, API calls, control flow graphs, and system calls. Behaviour-based detection is effective for identifying new malware variants and can be approached in two ways. The first method involves extracting behavioural characteristics from malware code, such as analysing the structural similarities between different polymorphic worm mutations through control flow graph analysis(Şahin, 2021). The second method runs malware in a sandbox environment, dynamically observing its behaviour, and using features like system calls to detect malicious activity(Preda et al., 2008). For instance, researchers have used cosine similarity to classify malware based on behaviour, achieving an accuracy of 85.8% for known malware types(Şahin, 2021). However, behaviour-based detection can be time-consuming and resource-intensive, especially when analysing a large number of files in real-time. False positives can also be a concern because benign software may show similar behaviours to malware. Despite these challenges, behaviour-based methods are valuable for detecting and categorizing malware based on its real-time actions and behaviour, making them particularly effective for identifying new and previously unknown malware threats(Mimura and Ito, 2022).

2.1.4 Machine Learning and Deep Learning Based Methods

Machine learning and deep learning methods in malware detection cover unsupervised. Unsupervised learning identifies patterns and anomalies without labelled data, helping clustering, and anomaly detection. Supervised learning utilizes labelled examples to train models that classify files based on learned patterns. Deep learning is a specialized machine learning approach capable of extracting high-level features from low-level data, making it applicable to malware detection. Deep learning models can learn complex feature hierarchies and integrate multiple detection steps into one model, enabling end-to-end training (kaspersky, 2021). Some of the popular methods cover a range of techniques, including Hidden Markov Models (HMMs), Random Forests (RF), Convolutional Neural Networks (CNNs), Support Vector Machines (SVM), Recurrent Neural Networks (RNNs), and Long Short-Term Memory (LSTM) networks. HMMs are utilized for opcode sequence analysis, RF combines with HMMs for enhanced performance, CNNs capture complex patterns, SVMs are effective in static and dynamic feature analysis, and RNNs/LSTMs excel at capturing sequential dependencies(Mehta et al., 2023).These machine learning and deep learning techniques play a crucial role in identifying and classifying malware based on patterns and features within the data, ultimately enhancing the efficiency and accuracy of malware detection systems(Mehta et al., 2023).

2.2 DEEP LEARNING FOR NATURAL LANGUAGE PROCESSING

Deep learning has revolutionized natural language processing (NLP) with extensive applications including text classification, information retrieval, named entity recognition, and more. Large neural networks outperform traditional methods by handling large training data and eliminating the need for feature engineering(Lauriola et al., 2022). Recent advancements in using deep learning for text classification have shown significant improvements in performance and accuracy. Deep learning models, such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), have been widely adopted in text classification tasks(Mehta et al., 2023).

Word embedding

word embedding refers to the process of representing textual elements, such as code snippets, malware samples, or textual descriptions related to malware behaviour, as low-dimensional vector representations. It is an important aspect of natural language processing (NLP) that helps in capturing syntactic and semantic similarities between words. By mapping text data into vector space, word embedding facilitates the development of more efficient and effective machine learning models for malware detection and classification tasks. It helps improve the accuracy and performance of malware detection systems by allowing them to learn and recognize subtle semantic and syntactic patterns indicative of malicious behaviour(Şahin, 2021). Popular methods for word embedding include Word2vec and GloVe(Zhang et al., 2021).

Word2vec

This work uses the Word2Vec model to construct word vectors. Word2Vec is a significant text representation model that has gained increased research interest, particularly in the field of malware classification and detection. In the context of malware detection, a Word2Vec model is pre-trained to vectorize the input data (opcode sequence), converting them into numeric vectors, replacing the traditional one-hot encoding approach. This approach leads to better performance and reduced memory usage in malware detection and categorization tasks(Sun et al., 2020). Word2Vec includes two contrasting models, skip-gram and CBOW. CBOW, or Continuous Bag of Words, is trained on contextual opcode sequences and yields the vector for specific unigram and bigram. The architecture of the CBOW model is illustrated in Figure 1.

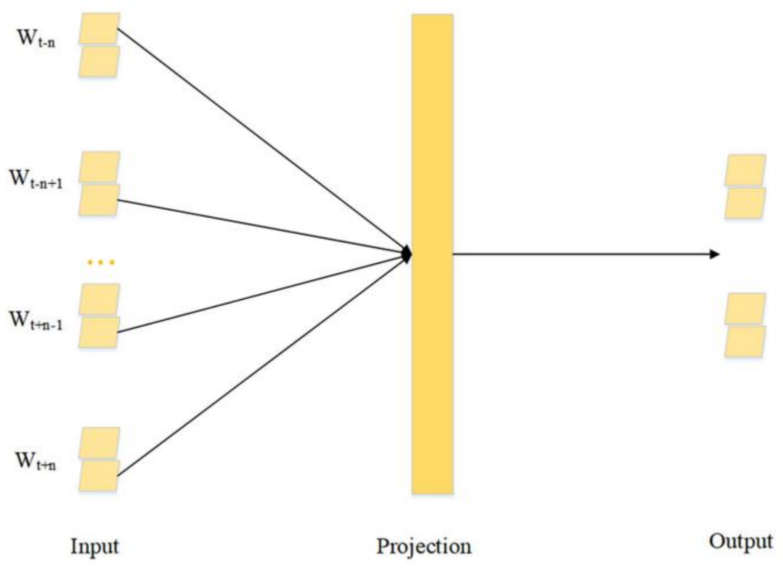


Figure 1:The architecture of CBOW is composed of three layers, an input layer, a hidden layer, and an output layer. It conducts vector projection operations within the hidden layer. CBOW employs the context to predict the central word(Li et al., 2023).

2.2.1 Artificial Neural Networks (ANNs)

Artificial Neural Networks (ANNs) are computational models inspired by the structure and function of biological neural networks, such as the brain. They are composed of interconnected nodes, known as neurons, that receive inputs, perform computations on those inputs, and produce outputs. ANNs are particularly used in the field of deep learning, a subfield of machine learning, to solve complex problems in various domains, including natural language processing (NLP) and malware detection. In ANNs, each neuron in the network receives weighted inputs from other neurons and applies a nonlinear transformation function to these inputs to generate an output. The weights on the connections between neurons are adjusted during training using algorithms such as stochastic gradient descent and backpropagation to minimize the errors or losses experienced by the network (Otter et al., 2020).

In the context of malware detection, ANNs have been used to analyse various types of data, such as byte sequences, API call sequences, network traffic, and more. Researchers have developed different architectures and techniques for training ANNs to detect and classify malware, including convolutional neural networks (CNNs), recurrent neural networks (RNNs), long short-term memory networks (LSTMs), and others(Gibert et al., 2020)

2.2.2 Deep Learning

Deep learning is a special approach within machine learning that consist of multiple layers of interconnected nodes called neurons, which perform complex mathematical computations on the input data. By iteratively adjusting the weights and biases of the neurons based on the training data, deep learning models can automatically extract and represent intricate features and relationships in the data(Alomari et al., 2023). It has been successful in various tasks such as computer vision, speech recognition, and natural language processing. Unlike other machine learning methods, deep learning focuses on inferring high-level meanings from low-level data. Deep learning models are capable of learning complex feature hierarchies and incorporating multiple steps of a detection pipeline into a single model. This makes it possible to train the model end-to-end, enabling the simultaneous learning of all components (kaspersky, 2021).

2.2.3 Convolutional Neural Networks

Convolutional Neural Network (CNN) forms a key component of the proposed model. CNN is a deep learning technique that has been widely used for processing data represented in grid patterns, such as images. In the context of malware detection, by stacking multiple convolutional layers and applying pooling operations CNN can automatically learn and extract high-level feature representations from the low-level patterns of opcode sequences and enabling effective classification(Zhang et al., 2021). CNNs consist of three fundamental components, the convolutional layer, pooling layer, and the fully connected or dense layer Figure 2. The convolutional and pooling layers focus on the extraction and selection or reduction of features, whereas the dense layer takes these features, learns from them, and eventually classifies them, determining the output, such as the category of a specific input. CNNs can have either a single or multiple convolution and pooling layers, but it's the convolution layer that stands out as its essential element. The primary aim of this layer is to find out various high-level features from the input data. This is achieved using certain adjustable parameters termed as filters or kernels. These filters produce diverse feature maps by performing convolution operations. In the convolution process, these filters traverse over sections of the input data, matching their size. This involves carrying out element-wise multiplication between the filter and every section of the input, effectively extracting features from different parts of the input data. Strides and padding are also important parameters of the convolutional layer(O'Shea and Nash, 2015). Stride dictates the step size the filter should take when moving over input data, either horizontally or vertically for 2-D data, and only horizontally for 1-D data. Padding is a technique used to maintain a feature map size close to the original input data, ensuring the edge information isn't lost as the filter progresses over the data, with zero-padding being a commonly employed method. After the convolution process, the results undergo a non-linear activation function before proceeding to the subsequent layer(Maniriho et al., 2022). The pooling layer, often referred to as the sub or down-sampling layer, takes small sections of the output feature map produced by the convolutional process and sub-samples them to produce a single, reduced output feature map. Various pooling methods exist, including max-overtime pooling, average or mean pooling, and min pooling. The primary advantage of this pooling process is that it reduces the size of the input data, leading to fewer parameters that need to be calculated in subsequent layers. This not only simplifies the network but also retains important feature representations(Yamashita et al., 2018). The final layer of a CNN is the fully connected neural network (FCNN) layer, which takes the output from the pooling layer and carries out classification or predictions. This FCNN consists of hidden layers and an output layer that undergo learning and classification processes using forward and backpropagation techniques. Therefore, a standard CNN can be conceptualized as a fusion of two primary segments, the feature extraction and selection segment and the classification segment(Yamashita et al., 2018).

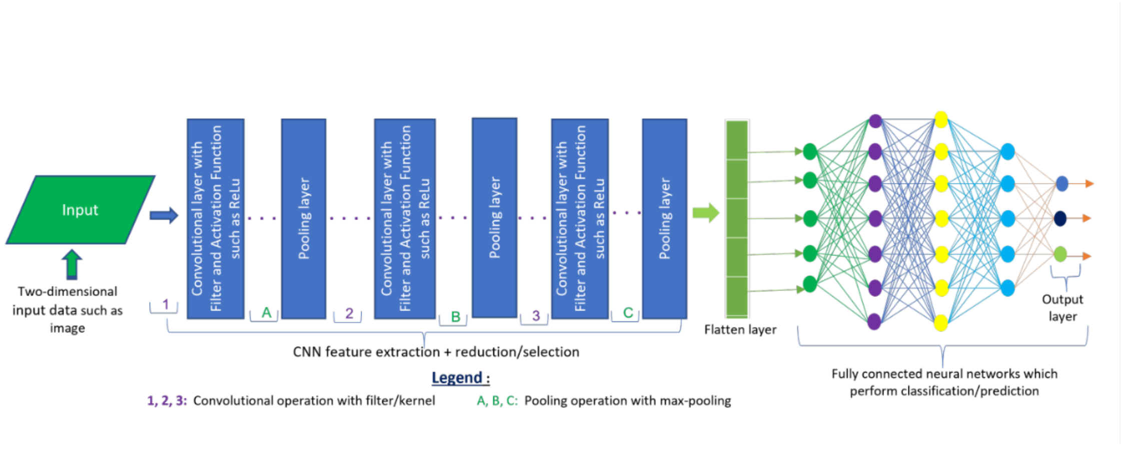

Input
Two-dimensional input data such as image
Convolutional layer with Filter and Activation Function such as ReLu
Pooling layer
Flatten layer
Output layer
CNN feature extraction + reduction/selection
Fully connected neural networks which perform classification/prediction
Legend :
1, 2, 3: Convolutional operation with filter/kernel
A, B, C: Pooling operation with max-pooling

Figure 2:The architecture of CNN(Maniriho et al., 2022).

2.2.4 Recurrent Neural Network (RNN)

Recurrent neural networks (RNNs) are a type of deep learning network architecture suitable for text classification tasks due to their ability to capture sequential dependencies in text data. Sequential data can be viewed as a representation of a program's behaviour, such as API calls, or, in this context, Android opcode sequences. This makes them well-suited for tasks like sentiment analysis and document classification. RNNs, including variants like long short-term memory (LSTM) and gated recurrent unit (GRU), are designed to capture dependencies and patterns in sequential data(Maniriho et al., 2022).

2.2.5 GRU

The GRU is a type of Recurrent Neural Network (RNN) that features a gated mechanism. It addresses the vanishing gradient issue encountered in long-term memory during backpropagation(Cho et al., 2014). The GRU has just two gates, the update and reset gates.

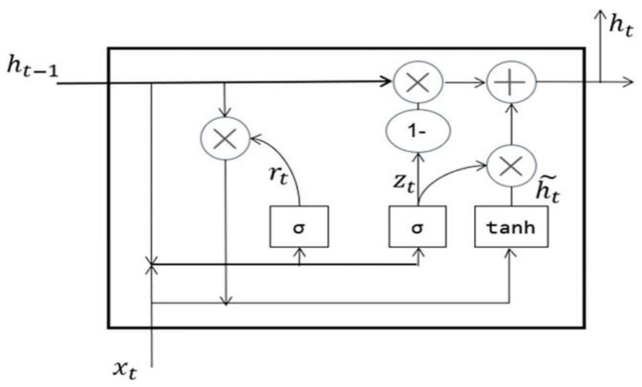


Figure 3:The structure of GRU(Maniriho et al., 2022).

This means fewer parameters to train, leading to improved training efficiency. It's commonly utilized in tasks related to natural language processing(Li et al., 2023). The structure of the GRU is shown in Figure 3.The parameters in the GRU model are updated using the following formulas.

$$Z_t = \sigma(w_z x_t + U_z h_{t-1}), \tag{1}$$

$$r_t = \sigma(w_t x_t + U_t h_{t-1}), \tag{2}$$

$$\tilde{h}_t = \tanh(w x_t + U(r_t \circ h_{t-1}), \tag{3}$$

$$h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h}_t, \tag{4}$$

In the GRU Z_t serves as the update gate while, r_t acts as the reset gate. The update gate dictates the extent to which the previous state's information is passed on to the present state. Conversely, the reset gate determines how much of the past state's data is discarded. The weights of the GRU are represented by w_z, w_t and w. Using the prior hidden state information, h_{t-1} and the present node input, x_t the GRU produces two gated states. After achieving the gated state, the reset gate formulates the forgotten state. This is then combined with x_t from the present moment and activated through the tanh nonlinear function. Ultimately, the update gate decides on and retains the present node's input(Li et al., 2023).

The GRU is calculated from the front to the back according to the natural language order, taking into account the meaning of prior words in the text and the overall context. However, it overlooks the influence of subsequent words on both the preceding words and the entire meaning. To address this, the bidirectional GRU (BiGRU) processes the text in both forward and backward directions concurrently and combines the outputs from both directions. This ensures a more complete and precise understanding of the text's semantics(Li et al., 2023).

2.3 TOKENIZATION (N-GRAM TOKENIZER)

Tokens are the units that make up a sentence. They can be sequences of characters, digits, punctuations, or combinations thereof, typically separated by spaces. Individually, tokens often convey limited meaning. While they provide the dictionary definition to the machine, they lack context and don't indicate the sentence's direction.

N-gram is a technique that indexes sentence by grouping consecutive words up to a count of N. It's commonly used to measure sentence similarity in search engines, big data, and cyber forensics. This report employs -grams to assemble a set of extracted features and measure similarity(S. Lee et al., 2019).

Below is a brief investigation of the structure of the Android file (APK) and the Dex format followed by methods that can be used to extract opcodes from Dex files and N-grams (N=1,2) to assemble a set of extracted tokens.

Android package structure

Android files use the APK extension and are structured in the ZIP file format. The primary components of an APK are AndroidManifest.xml, classes.dex, and resources.arsc. AndroidManifest.xml holds data about the app's attributes and permissions, while resources.arsc contains the app's resource information. The classes.dex, or Dex file, is vital for running the app and comprises opcode(Shinho Lee et al., 2019)**.** The structure of the APK file is depicted in Figure 4.

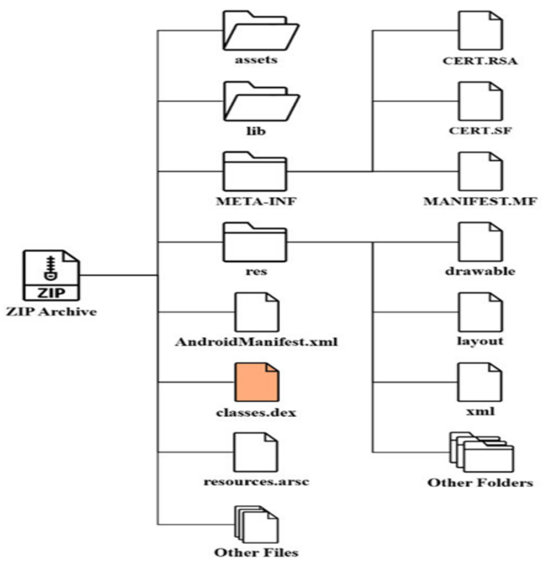


Figure 4:Structure of the APK file(Shinho Lee et al., 2019).

Dex format

The Dex files are the most important files that contains the compiled codes that run on Android. Decompiling these files (using Baksmali [1]or Androguard[2], this software are both tools related to Android application analysis, and both offer disassembly of DEX files) extract bytecode known as Smali. Figure 5 illustrates the structure of the Dex file.

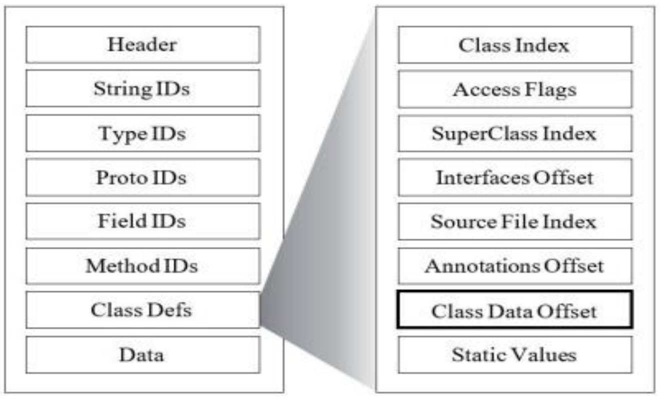


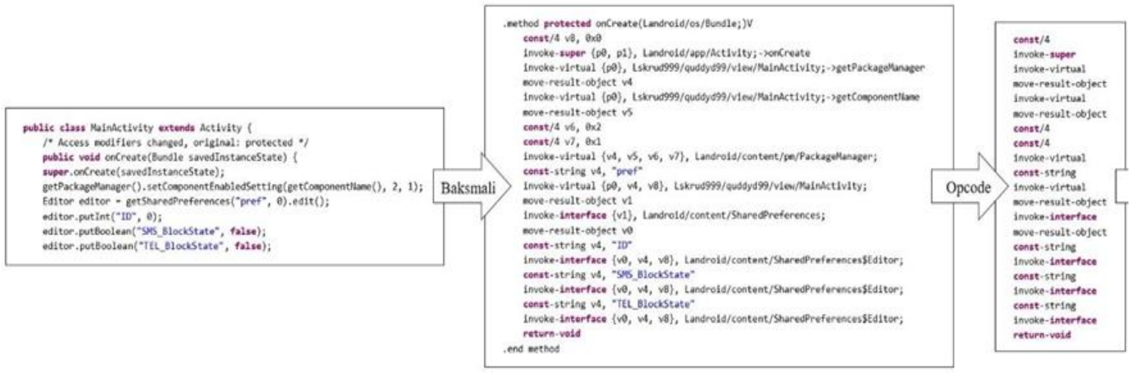
Figure 5:Dex format

[1] https://github.com/JesusFreke/smali
[2] https://github.com/androguard/androguard

The Dex file structure contains a table named "Class Defs" that defines code information for Android files by class. "Class Data Offset" represents bytecode offsets, excluding the class's attribute information(Shinho Lee et al., 2019). Specific methods can be used to parse this Class Data Offset in the Dex file, enabling the extraction of the opcode sequence.

Opcode sequence extraction

Opcode sequences are a series of assembly instructions obtained from APK files. These sequences represent the low-level operations performed by the Android operating system. Opcode sequences are used as input to the proposed model for automated Android malware detection using deep learning techniques. In(Yeboah and Baz Musah, 2022), the term "opcode sequences" refers to the raw, unstructured opcode instructions extracted using Androguard. This method enables the extraction of assembly operation codes from the classes.dex file. The length of these opcode sequences can vary among different Android applications. To standardize their sizes, unigram and bigram preprocessing techniques are employed.



```
public class MainActivity extends Activity {
    /* Access modifiers changed, original: protected */
    public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    getPackageManager().setComponentEnabledSetting(getComponentName(), 2, 1);
    Editor editor = getSharedPreferences("pref", 0).edit();
    editor.putInt("ID", 0);
    editor.putBoolean("SMS_BlockState", false);
    editor.putBoolean("TEL_BlockState", false);
```

```
.method protected onCreate(Landroid/os/Bundle;)V
    const/4 v8, 0x0
    invoke-super {p0, p1}, Landroid/app/Activity;->onCreate
    invoke-virtual {p0}, Lskrud999/quddyd99/view/MainActivity;->getPackageManager
    move-result-object v4
    invoke-virtual {p0}, Lskrud999/quddyd99/view/MainActivity;->getComponentName
    move-result-object v5
    const/4 v6, 0x2
    const/4 v7, 0x1
    invoke-virtual {v4, v5, v6, v7}, Landroid/content/pm/PackageManager;
    const-string v4, "pref"
    invoke-virtual {p0, v4, v8}, Lskrud999/quddyd99/view/MainActivity;
    move-result-object v1
    invoke-interface {v1}, Landroid/content/SharedPreferences;
    move-result-object v0
    const-string v4, "ID"
    invoke-interface {v0, v4, v8}, Landroid/content/SharedPreferences$Editor;
    const-string v4, "SMS_BlockState"
    invoke-interface {v0, v4, v8}, Landroid/content/SharedPreferences$Editor;
    const-string v4, "TEL_BlockState"
    invoke-interface {v0, v4, v8}, Landroid/content/SharedPreferences$Editor;
    return-void
.end method
```

```
const/4
invoke-super
invoke-virtual
move-result-object
invoke-virtual
move-result-object
const/4
const/4
invoke-virtual
const-string
invoke-virtual
move-result-object
invoke-interface
move-result-object
const-string
invoke-interface
const-string
invoke-interface
const-string
invoke-interface
return-void
```

Figure 6:A sample of extracting opcode sequences of Dex files (Shinho Lee et al., 2019).

Figure 6 displays the process of extracting the opcode sequences by parsing the classes.dex file. The extracted opcode sequence is bundled into a single string that stores the opcode sequence of the Dex file in list form.

2.4 TENSORFLOW AND ITS APPLICATIONS

TensorFlow is a popular and flexible software library for numerical computations, specifically designed for deep learning and neural network modelling. TensorFlow provides a comprehensive framework for building and training neural network models, including CNNs. It offers a wide range of functionalities, such as graph construction and execution tools, high-level APIs, GPU support, and distributed training(Pang et al., 2020). With TensorFlow, researchers and practitioners can easily implement and experiment with various deep learning models, including CNNs. TensorFlow's flexibility and scalability make it a valuable tool for researchers and practitioners in the field of deep learning and neural network modelling(Pang et al., 2020).

TensorFlow offers several advantages for building Convolutional Neural Networks (CNNs) compared to other deep learning frameworks(Pang et al., 2020).

- Simplified Implementation: TensorFlow greatly simplifies the implementation of CNNs by providing high-level APIs like Keras, which offer easy-to-use and consistent interfaces. These APIs allow users to stack predefined or custom layers to construct a model without the need to handle graph construction or execution explicitly. This simplifies the programming process and reduces the potential for errors.

- Fast Experimentation: In addition to high-level APIs, TensorFlow also provides low-level APIs that offer flexibility and facilitate understanding of the internals of TensorFlow. This allows for fast experimentation and reduces the delay from idea to result. Users can choose between low-level and high-level APIs depending on their specific tasks.
- GPU Acceleration: TensorFlow is optimized to run on Graphical Processing Units (GPUs) by utilizing CUDA and cuDNN. Neural network models, including CNNs, heavily involve matrix multiplication, which can be highly parallelized and benefit from GPU architecture. This optimization allows for significantly faster training times compared to running on a Central Processing Unit (CPU).
- Abundance of Models: TensorFlow provides implementations of major neural network models, including CNNs, which are available on its official website and in Google Colab. These models are accompanied by their codes, allowing users to run them or modify them for their own purposes. TensorFlow's popularity also means that many recently published neural network models are implemented in TensorFlow, with their codes being available on platforms like GitHub. This abundance of models and access to their implementations make TensorFlow a convenient choice for building CNNs.

2.5 RELATED WORKS

This paper,(Lauriola et al., 2022), focuses on the application of deep learning techniques in NLP and explores the various tasks where deep learning has shown a stronger impact. It also discusses recent advancements in NLP, particularly in word and sentence representations, and provides information on resources commonly used in NLP research. Deep learning has greatly advanced natural language processing (NLP), particularly in tasks like machine translation and named entity recognition. The introduction of large neural networks has led to improved performance compared to traditional machine learning algorithms. However, there are challenges in deep learning for NLP, such as high computational costs and lack of interpretability. Finding suitable representations for tokens, sentences, and documents is crucial, and recent advancements in word and sentence vectors, such as Word2Vec, have been made. The document also covers different NLP tasks, provides information on software and hardware used in NLP research, and highlights available datasets. Overall, it serves as a tutorial for the Machine Learning community, offering insights into NLP tasks, advancements, resources, and future directions in the field. However, the paper also highlights existing challenges in deep learning for NLP, such as computational costs, reproducibility, and lack of interpretability. Finding suitable representations for tokens, sentences, and documents is crucial for NLP systems to capture semantic relations between words and improve their overall performance. Therefore, understanding and utilizing deep learning techniques in NLP can lead to better language models, improved language understanding, and more accurate NLP applications, opening up new possibilities for human-computer interaction through natural language.

In (Otter et al., 2020) the authors discuss the use of deep learning architectures and methods in NLP and present a comprehensive summary of recent studies and contributions in the field.

The paper starts by introducing NLP and its reliance on data-driven computation, statistics, probability, and machine learning. It explains how recent advancements in computational power and the availability of large datasets have enabled the use of deep learning models, particularly artificial neural networks (ANNs), in NLP then it explores various research areas and applications of deep learning in NLP, covering linguistic processing issues such as language modelling, morphology, parsing, and semantics. Moreover, the document highlights recent advancements in NLP, such as generative pretraining (GPT) and embeddings from language models (ELMo), which have significantly improved language modelling and understanding. It also addresses the challenges faced in NLP, including the need for better machine understanding and the integration of deep networks with knowledge graphs. It also discusses practical applications of deep learning in information retrieval, information extraction, text classification, text generation, summarization, question answering, and machine translation. Despite the progress in deep learning models for NLP, challenges still exist, such as the lack of creativity and coherence in text generation tasks, the difficulty of creating reliable automatic evaluation metrics for creative tasks, and the need for clear comparisons and agreement on evaluation metrics and datasets in the NLP field.

The white paper,(kaspersky, 2021), discusses various approaches, such as Kaspersky's similarity hashing and two-stage analysis design, as well as the use of deep learning for post-execution behaviour detection. This topic is important because traditional methods of malware detection are becoming less effective against the evolving nature of cyber threats. Machine learning offers a promising solution as it can adapt and learn from new patterns and behaviours exhibited by malware. However, there are challenges in effectively utilizing machine learning, such as the need for representative and relevant data, understanding theoretical machine learning concepts, and implementing machine learning algorithms into practical cybersecurity products. The paper addresses these challenges and highlights the importance of having the right data, understanding both machine learning and cybersecurity, and ensuring multi-layered synergy in detection methods for effective and comprehensive malware detection. The study addresses several problems, questions, and hypotheses related to the use of machine learning for malware detection in the field of cybersecurity. These include, the problem of data distribution, the study recognizes that the data distribution in cybersecurity is not fixed due to active adversaries (malware writers) constantly creating new versions of malware files and the emergence of new types of benign executables, the challenge of machine learning application in cybersecurity, the study highlights specific requirements for machine learning algorithms in the context of cybersecurity, including the need for large representative datasets, interpretable models, extremely low false positive rates, and the ability to quickly adapt to malware writers' counteractions, detecting new malware in pre-execution, the study addresses the problem of detecting new malware files before they are executed. It presents Kaspersky's approach, which uses similarity hashing and a two-stage analysis design to train the model and reduce the computational load on users' systems, deep learning for post-execution behaviour detection, the study discusses the use of deep learning in detecting post-execution behaviours, especially in rare attacks where there is only one example of malware for training. It introduces the ExNet model, which distils discriminative features for various types of malware and allows efficient generalization of knowledge about single malware samples and a large collection of clean samples, log compression stage, the study talks about the log compression stage, which involves transforming logs into behaviour graphs, extracting specific subgraphs, compressing them into sparse binary vectors, and combining them into a single vector for analysis using a deep neural network. Overall, this literature review contributes to a better understanding of the challenges faced in machine learning for malware detection and highlights Kaspersky's approach, including similarity hashing, two-stage analysis, and the use of deep learning. The review also emphasizes the importance of large datasets, interpretability, low false positive rates, and adaptability in machine learning applications for cybersecurity.

The paper,(M and Sethuraman, 2023), explores a comprehensive review of deep learning-based malware detection techniques. The paper presents a detailed survey of recent deep learning-based malware detection techniques, focusing on Ransomware, Advanced Persistent Threat (APT), and other trending malwares. Additionally, the authors propose research gaps and a taxonomy, highlighting further areas for advancement. The aim of the survey is to guide researchers in building mitigation techniques for traditional and advanced malwares. The main problems addressed include the increase in security loopholes, privacy concerns, and financial losses caused by malware attacks. The researchers also aim to resolve these problems by proposing machine learning and deep learning-based malware mitigation techniques. The document discusses various machine learning and deep learning techniques proposed for malware detection. These techniques include deep Learning models used for analysing malwares based on images, machine Learning framework for detecting malwares based on the Domain Generation Algorithm (DGA), deep Neural Networks (DNNs) for feature extraction and malware classification, deep neural network models with multiple hidden layers, lightweight Convolutional Neural Network (LCNN) for multi-class classification of malware families, Advanced Persistent Threat (APT) attack detection using machine learning or deep learning algorithms, layer deep learning model for rapid detection and classification of APT attacks, decision trees and Bayesian networks for APT attack detection. These techniques aim to detect various forms of malware, including traditional malwares, IoT malwares, Ransomware, and APTs, by leveraging machine learning and deep learning approaches. The study aims to guide researchers in building mitigation techniques for both traditional and advanced malwares using machine and deep learning techniques.

In (Yeboah and Baz Musah, 2022),the author focuses proposing a framework for automating Android malware detection using a 1-dimensional convolutional neural network (1D CNN) model. The model treats Android malware detection as a time-series classification task and extracts features from raw opcode sequences using natural language processing (NLP) and deep convolutional neural networks. The model is trained on unigram and bigram opcode feature sets and combines predictions from multiple feature sets using a weighted average ensemble approach. The authors tested the model on an Android dataset and achieved high performance with a positive predictive value of 98% and sensitivity of 97%. The limitation of the proposed methodology is that it may suffer from detection downgrade over time due to evolving malware, known as concept drift. The model's features become outdated as malware evolves, and constant updates are required for maintaining its effectiveness. Overall, the study addresses problems related to labour-intensive feature engineering, performance improvement, computational efficiency, concept drift, and adversarial evasion in Android malware detection.

In (Zhang et al., 2021) the authors propose CoDroid, which utilizes both static opcode sequences and dynamic system call sequences as input. These sequences are treated as sentences in NLP and classified using a CNN-BiLSTM-Attention model. The overall architecture of CoDroid is explained, including the preprocessing steps, sequence generation, and the DL model used. The methodology is evaluated using a real-world dataset, and the results show that CoDroid outperforms other existing methods. This approach allows for a more comprehensive analysis of Android applications and captures both static and dynamic behaviours, leading to the detection of potential malicious code execution paths. This combination of features provides a more effective and flexible approach to Android malware detection. On the other hand, the limitations of CoDroid in comparison to other methods are not explicitly mentioned in the provided document snippets. The result of the evaluation of the methodology is that the proposed CoDroid hybrid detection method outperforms other existing methods in terms of detection performance. It achieves a higher F1-score, precision, recall, and accuracy compared to traditional machine learning algorithms such as K-nearest neighbour (KNN), Naive Bayes (NB), Logistic Regression (LR), and Random Forest (RF). In comparison with other related detection methods, CoDroid demonstrates superior performance, obtaining the highest accuracy score of 97.6%.

The (Ito and Mimura, 2019) involves a method for detecting unknown malware using natural language processing (NLP) techniques and ASCII strings extracted from executable files. The approach focuses on reducing uncommon words to improve the detection rate. A corpus of high-frequency words is created from the extracted strings of benign and malicious executable files. NLP techniques such as Latent Semantic Indexing (LSI) and Doc2Vec are employed to convert the words into feature vectors. A Support Vector Machine (SVM) classifier is trained using the labelled feature vectors generated from known executable files. In the test phase, strings are extracted from unknown executable files, and the words are reduced and converted into feature vectors using the language model. The classifier performs the classification of the unknown files. The proposed method achieved an F-measure of more than 0.85 in experiments using a dataset of over 23,000 malware samples and more than 16,000 benign files. The paper concludes with a discussion of related research and future tasks. The limitation of the methodology is that attackers can potentially bypass the detection by using obfuscation techniques. The effectiveness of the method against obfuscated dataset needs to be evaluated in future research. Additionally, parameter adjustments require further refinement, but the large processing time required poses challenges. The study aims to answer the question of how to effectively detect unknown malware by reducing uncommon words. The hypothesis is that by creating a corpus of high-frequency words and constructing a language model using NLP techniques, the detection rate of unknown malware can be improved.

The (Mimura and Ito, 2022) involves applying natural language processing (NLP) techniques to detect malware in a practical environment. The authors propose a fast-filtering method that combines static detection methods with NLP techniques. They use printable strings and NLP models, such as Doc2vec and LSI, to extract lexical features from both malicious and benign samples. The selected language model and trained classifier are then used to classify unknown executable files as either malicious or benign. The authors also evaluate their detection model using a dataset of over 500,000 samples obtained from multiple sources. The first limitation is related to the dataset used in the study. While the dataset used in the research consists of over 500,000 samples obtained from multiple sources, it may not represent the entire population appropriately. Additionally, as it is not feasible to use all actual samples for evaluation, the best solution is to use a large-scale dataset from multiple sources. The second limitation is the lack of detailed analysis. The study used a simple signature-based packer detector, which has approximately 30 percent false negatives. The researchers did not identify the packer names of all the samples used, so the experimental results may not be applicable to sophisticated packers that are not detected by signature-based packer detectors. This issue affects the accuracy of the experiment and further analysis is required to address these issues. The third limitation is the lack of comparison with other related studies. Due to various issues, such as the dataset or implementation, it was not feasible to provide a fair comparison with other methods. Therefore, further experiments are necessary to provide a fair comparison. The study addresses several problems and questions related to malware detection in a practical environment. Specifically, it looks at the issue of executable files being used to compromise endpoint computers, with attackers employing obfuscated malware to evade anti-virus programs. The study explores the limitations of dynamic analysis, which is time-consuming when examining all suspicious files from the Internet. The authors propose a fast-filtering method using static detection methods and natural language processing (NLP) techniques to address these challenges. The study aims to evaluate the effectiveness of using NLP techniques, specifically printable strings, in detecting malware, including new and packed malware that traditional methods struggle to detect. The study also considers the use of different evaluation metrics and the importance of time series analysis in detecting new malware.

In (Mehta et al., 2023), NLP techniques can be adapted for malware detection by treating malware-related data, such as opcode sequences, as natural language. This involves applying Natural Language Processing (NLP) techniques, like Hidden Markov Models (HMMs), on the opcode sequences to extract hidden state sequences, which can be used as feature vectors for classification. This approach is viewed as a form of feature engineering, similar to techniques used in NLP applications. By training HMMs on opcode sequences, the resulting hidden states can capture patterns and characteristics in the data, just like how word embeddings capture semantic and syntactic information in text. These hidden states are then used as feature vectors in various classifiers, such as Random Forests (RF), Support Vector Machines (SVM), and Convolutional Neural Networks (CNN). The NLP-based approach using HMMs has been found to outperform other techniques on challenging malware datasets. Therefore, adapting NLP techniques for malware detection involves applying HMMs on opcode sequences and utilizing the resulting hidden states for classification. This approach allows for the detection and analysis of malware using machine learning and deep learning techniques commonly used in NLP applications.

2.6 SUMMARY

This section provided foundational knowledge on various topics. It delved into malware detection and analysis techniques, emphasizing machine learning and deep learning methodologies for identifying malware. The interplay between deep learning and natural language processing (NLP) was explored, alongside an introduction to Android APK applications, their structural nuances, and the techniques for extracting opcode sequences from DEX files and their tokenization. The discussion further introduced TensorFlow and its myriad applications. The deep neural network algorithms, especially their operational mechanics and their role in malware detection, were elaborated, as these are pivotal for the binary classification modelling in this study. The section concluded by surveying relevant literature on state-of-art malware detection based on Natural Language Processing and Deep Learning Techniques.

3 METHOD, DESIGN AND IMPLEMENTATION

In this chapter, the presentation begins with an approach to malware detection (Maniriho et al., 2022). This is followed by an introduction to the dataset (Yeboah, 2022) and a description of the data collection methodology. The chapter also discusses the use of Word2Vec embedding to convert static and dynamic features into numeric representations. The primary focus then shifts to proposing a framework and analysing the performance of the suggested model for detecting Android malware using an NLP-based encoder, a convolutional neural network (CNN), and a bidirectional gated recurrent unit (BiGRU). All components have been implemented using TensorFlow in Collaboratory by Google, a Jupyter notebook-based runtime environment that operates entirely in the cloud.

3.1 APPROACH

Malware detection methods usually begin with the creation of a dataset and the initiation of feature extraction processes, performed through static, dynamic, or hybrid analysis. Approaches based on dynamic and hybrid analysis work by executing Android applications to examine their behaviour, while static analysis examines the content of an application's code, properties, and components without executing the Android applications.

In Android, apps are typically distributed as APK (Android Package) files, essentially archive files containing all the components needed by the app. The APK file includes the classes.dex file, crucial for static analysis. The static approach to the Android application package (APK) can extract various features used to discriminate between malware and benign samples. These features include permissions, operation codes, certificates, API function calls, call graphs, raw byte sequences, crypto API usage, and special strings. In this study, opcodes are chosen for as input for the model because their sequences remain consistent despite alterations in the Dalvik bytecode, such as changes to the Package Name, Class Name, Method Name, or additions of variables in reusable codes Figure 5. Consequently, opcodes are a robust feature for assessing the similarity of malware variants, as they stay unchanged even when other elements are modified (Shinho Lee et al., 2019).

Traditional machine learning techniques require manual feature engineering and may struggle to adapt to new variants, a process which can be time-consuming and error-prone. Deep learning is employed for malware detection because of its ability to automatically extract relevant features from raw and lengthy sequences with minimal pre-processing. It has demonstrated resilience against malware obfuscation, meaning it can generalize and detect previously unseen malware variants exhibiting similar semantic patterns.

The framework (Maniriho et al., 2022) is utilized for malware detection as it offers several advantages over traditional techniques. Firstly, the Android operating system has emerged as a prime target for cyber perpetrators, and existing detection solutions that rely on handcrafted features are labour-intensive and time-consuming. Secondly, this framework employs natural language processing (NLP) and deep learning techniques to automatically identify unique and highly relevant patterns from raw opcode sequences. NLP techniques facilitate the numerical encoding of raw opcode sequences and the capturing of semantic relationships among them. The framework amalgamates a convolutional neural network (CNN) and a bidirectional gated recurrent unit (BiGRU) to extract high-level features from raw opcode sequences and discern semantic relationships between them. The aim of this approach is to automate and achieve high performance in detecting of both known and zero-day Android malware. I plan to achieve this by converting raw opcode sequences (from Android executables) into numerical encodings. These encodings are meant to be used as input for neural network models, including convolutional neural networks (CNNs) and bidirectional gated recurrent units (BiGRUs). The intention is to capture the semantic relationships among the opcodes to better differentiate between benign and malicious software samples. In the next section, I introduce the datasets used for developing and training the framework.

3.2 DATASET

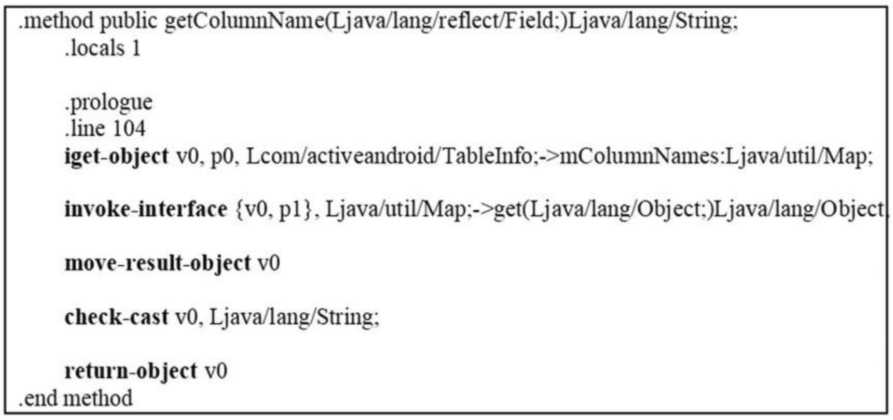
When selecting a dataset to use, updated and publicly available datasets were considered. This dataset (Yeboah, 2022) used in the study consists of Android samples, including both malware and benign samples. The purpose of this dataset is to train and evaluate the proposed model for the automated detection of Android malware. The dataset used to verify this study is publicly available and can be found in the following repository:

https://doi.org/10.6084/m9.figshare.18865805

3.2.1 Data Collection and Formatting

The dataset was collected from the Canadian Institute of Cybersecurity (CIC) Research Lab (Cybersecurity, 2020) and comprises of 4948 malware and 2477 benign Android opcode sequences with labels (0-benign, 1-malware). This dataset, assembled by CIC, draws from resources like the Contagio security blog[3], Android malware dataset (AMD)[4], MalDozer, and VirusTotal, randomly selected of 1595 SMS, 1253 adware, and 2100 malicious banking applications.

Static analysis of Android applications necessitates analysing the app's code, properties, and components without executing the app. The APK (Android Package) files include classes.dex files. Extracting these is essential for Android malware detection as they contain the compiled Java classes composing the actual executable code of an APK. By analysing the assembly operation codes within the classes.dex files, insights into the behaviour and characteristics of the Android malware can be obtained.



```
.method public getColumnName(Ljava/lang/reflect/Field;)Ljava/lang/String;
    .locals 1

    .prologue
    .line 104
    iget-object v0, p0, Lcom/activeandroid/TableInfo;->mColumnNames:Ljava/util/Map;

    invoke-interface {v0, p1}, Ljava/util/Map;->get(Ljava/lang/Object;)Ljava/lang/Object

    move-result-object v0

    check-cast v0, Ljava/lang/String;

    return-object v0
.end method
```

Figure 7:Example of Dalvik opcode in a .smali file.

[3] https://contagiodump.blogspot.com

[4] https://www.unb.ca/cic/datasets/andmal2017.html

In (Yeboah and Baz Musah, 2022), the dataset underwent re-processing steps before training the deep learning model. The study leveraged Androguard, an open-source Android reverse engineering tool, to obtain assembly operation codes from the classes.dex file. The document outlines that the opcode sequences were refined and converted into semantically unigram and bigram opcode sequences. Utilizing both unigram and bigram sequences allows the model to discern various patterns and dependencies in the opcode sequences, offering a thorough understanding of the malware samples. Word2Vec was employed to pretrain unigram and bigram opcode sequence feature sets and generate semantic-preserved vector representations. Opcode sequences were labelled as "malicious" or "benign" based on their association with Android malware or non-malicious software samples, respectively. These labelled opcode sequences served as input for training and evaluating the proposed model for Android malware detection.

3.3 THE MODEL IMPLEMENTATION

This section provides technical specifications relevant to the models in this study, detailing the training and testing environment setup, the libraries and modules used, and the intricacies of the training pipeline for both the pre-trained model and the binary classification model, based on the proposed framework. Before giving the details about the technical parts of study, we should briefly explain why we prefer to use this framework over other neural network architectures such as LSTM, MLP, BiLSTM, and GRU.

Firstly, the framework utilizes a hybrid automatic feature extraction approach by combining a convolutional neural network (CNN) and a bidirectional gated recurrent unit (BiGRU). This combination allows to capture relevant features from opcode sequences effectively. By leveraging both CNN and BiGRU, the framework can extract high-level features that are more meaningful and help in distinguishing between malware attacks and benign activities. Additionally, the framework incorporates natural language processing (NLP) techniques to produce numerical encodings of opcode sequences. This enables framework to capture semantic relationships among opcode sequences calls, which is crucial in identifying unique patterns associated with malware attacks. The NLP-based encoder constructs embedding vectors for each opcode sequences call based on their semantic relationships, improving the accuracy of the detection process.

The framework incorporates the LIME framework, offering explainable predictions crucial for security analysts to identify the main opcode sequences that contributed to the final prediction. LIME's results can assist in understanding the reasoning behind model's predictions, enabling better decision-making based on those predictions. Experimental evaluations show the framework's advantage over other neural network designs, like LSTM, MLP, BiLSTM, and GRU. It consistently surpasses these models in precision, recall, F1-score, and accuracy for malware classification.

3.3.1 The Environment Setup

Google Collaboratory, a runtime environment based on Jupyter notebook, was utilized to execute the proposed framework completely in the cloud. The framework was implemented in Python programming language version 3.10.12 using TensorFlow 2.13.0 and Keras 2.13.1 frameworks and other libraries such as Scikit-learn, NumPy, Pandas, Matplotlib and LIME. This study has used the GPU Jupyter Notebook with A100 GPU with 83.5 GB RAM.

3.3.2 Imported Libraries and Modules

In this section, snippet code provided in Figure 14,I will give a short explanation of the python libraries and their modules that are used in the development of this framework. The framework utilizes several libraries and modules in its development, which are as follows:

Pandas (pd)

The Pandas library is primarily used for loading and preprocessing the dataset which contains opcodes and labels indicating whether a sample is malware or benign.

Gensim's Word2Vec

Word2Vec from the Gensim library is utilized for training word embeddings on opcode sequences, a technique in NLP. It transforms opcode sequences into numerical vectors which are suitable for feeding into machine learning models.

Sklearn's train_test_split

This function is used to split the dataset into training and testing subsets, ensuring that the model can be evaluated on unseen data to verify its performance in detecting malware.

TensorFlow's Keras pad_sequences

This function is used to ensure that all opcode sequences are of uniform length by padding the shorter ones. Uniform length is crucial for training neural network models like CNN and BiGRU.

TensorFlow's Keras Sequential Model

This is utilized to initialize a linear stack of layers to which Conv1D, MaxPooling1D, Bidirectional GRU, and Dense layers are added for building the model structure suitable for malware detection.

TensorFlow's Keras Layers

Conv1D is used for applying convolution, a specialized kind of linear operation, over the sequences. MaxPooling1D is used for down-sampling the input representation. Both are essential for extracting local and translational invariant features suitable for sequence classification tasks like malware detection.

Bidirectional & GRU

Bidirectional is a wrapper that allows a GRU layer to process the input sequences from both forward and backward directions, which is important to capture the patterns/dependencies throughout the sequence effectively for detecting malware.

Dense

It is used for performing the final classification after feature extraction.

Numpy (np)

Numpy is used for numerical operations, like converting lists to Numpy arrays, which is essential for preparing the data to feed into the model.

Matplotlib.pyplot (plt)

This is used for visualizing the results, such as plotting the training and validation loss or accuracy curves, which aids in analysing the model's learning progress and performance in malware detection.

Sklearn's confusion_matrix

This function is used to compute the confusion matrix to evaluate the accuracy of the classification, by comparing the predicted labels with the true labels, providing insights into the model's performance in correctly detecting malware and benign samples.

PCA (Principal Component Analysis) from the sklearn

PCA is used to reduce the dimensional word vectors and can be visualized to observe the clustering of similar words. It allows me to observe whether certain malware-related opcodes are clustering together.

LIME

LIME (Local Interpretable Model-Agnostic Explanations) is a crucial library in this malware detection framework as it helps in interpreting and understanding the decisions made by the complex model involving CNN and BiGRU. It provides explanations for the predictions, which can be instrumental in understanding the behaviour of malware and benign software based on opcodes and improving model reliability and trustworthiness in malware detection.

lime_text

lime_text is a submodule of lime specifically designed for handling text data. In the context of malware detection, it is crucial for interpreting the model predictions derived from the opcode sequences representing the software, providing insights into which opcodes or sequences of opcodes are indicative of malicious or benign behaviour.

LimeTextExplainer

LimeTextExplainer is a class within the **lime_text** module responsible for generating explanations for text data. In malware detection, this is particularly important as it elucidates how different opcodes contribute to the model's prediction, allowing developers or analysts to understand why a particular software is classified as malware or benign.

3.3.3 Pre-trained Model

This study develops a pipeline designed to process and train datasets using a deep neural network, with the implementation being carried out in Python using TensorFlow, Keras frameworks, and other libraries. This pipeline is specifically purposed for handling datasets with assembly opcodes derived from DEX files, as detailed by (Yeboah and Baz Musah, 2022).

Language Training Pipeline & Tokenization

(Yeboah and Baz Musah, 2022) has used DEX files store compiled Android application code and extracting hex sequences from them. The extracted hex sequences could be tokenized using delimiter-based tokenization, given that the bytes are space-separated, thus forming the basis for opcode sequence extraction. After the extraction and tokenization of hexadecimal sequences, opcode sequences are identified.

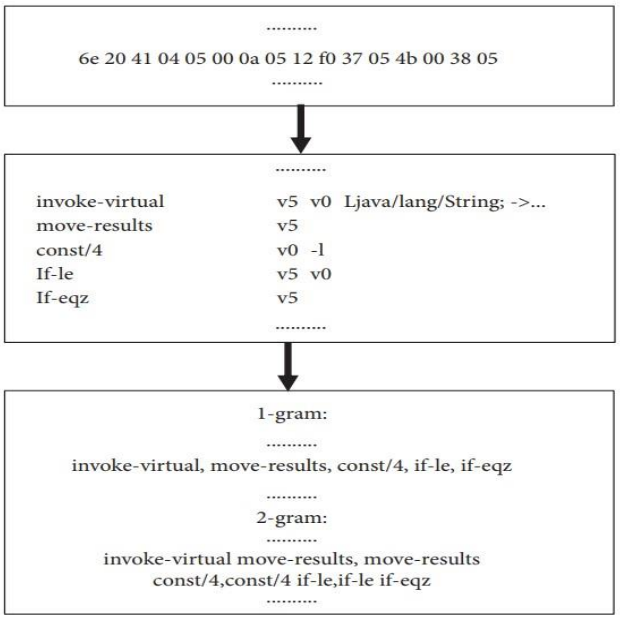

....
6e 20 41 04 05 00 0a 05 12 f0 37 05 4b 00 38 05
....
invoke-virtual v5 v0 Ljava/lang/String; ->...
move-results v5
const/4 v0 -l
If-le v5 v0
If-eqz v5
....
1-gram:
invoke-virtual, move-results, const/4, if-le, if-eqz
2-gram:
invoke-virtual move-results, move-results const/4,const/4 if-le,if-le if-eqz

Figure 8: N-gram opcode sequence extraction(Yeboah and Baz Musah, 2022).

The process involves extracting n-gram opcode sequences from the extracted assembly instruction code sequences, with n limited to 2 due to computational complexities associated with larger n-gram sizes. The opcode sequences, which vary in length from one Android APK to another, are standardized to a fixed length (L=600) to achieve uniformity. This standardization involves truncating longer sequences and padding shorter ones with zeros. Furthermore, addressing the varied frequency usage of opcodes, those that appear less than three times are represented by a singular token.

This representation and preprocessing approach are uniformly applied to both unigram and bigram opcode sequence feature sets. In subsequent sections, the study delves into the intricacies of the constructed pipeline. In (Yeboah and Baz Musah, 2022), the issue of varying opcode frequency usage was addressed. It was observed that certain opcodes like “invoke-direct,” “move-result-object,” and “new-instance” were commonly used, while others like “sget-short” and “rem-float” were infrequent. The infrequency of some opcodes raises concerns about their accurate representation when transformed into vectors. To mitigate this, opcodes appearing less than three times were represented by a unique token. This preprocessing was also applied to the bigram opcode sequence feature set, sequences were either truncated or zero-padded to a uniform length L, and rare bigram opcodes were likewise tokenized to ensure consistent embedding into the continuous vector space. This tokenization approach ensures that each opcode is treated as an individual token, allowing the Word2Vec model to learn a vector representation for each unique opcode in the training corpus(Yeboah and Baz Musah, 2022). The opcode sequences are represented as space-separated strings in a CSV file.

Training Word2Vec Model and Vectorization

To represent opcodes in contextually dense real-valued vectors, an approach proposed by (Mikolov et al., 2013), which learns words to vector embedding using machine learning and has shown promising results on natural language processing-related tasks. The referenced article presents two model architectures for word2vec embedding: the Continuous Bag-of-Words (CBOW) model, the skip-gram model. For this study, the CBOW model is chosen due to its comparative speed in training and can predicts a target word from surrounding context words.

Keras embedding layer is a component used in the development of the farmwork proposed by (Maniriho et al., 2022), which is done using the Keras API Tokenizer during the pre-processing steps in their paper and (Yeboah and Baz Musah, 2022) has mentioned that one-hot encoding, which represents each opcode with a high-dimensional sparse vector, is not suitable for this model due to computational costs and the inability to capture semantic relationships within opcode sequences. Instead, they have mentioned a more sophisticated embedding solution called word2vec to generate contextually dense real-valued vectors for the opcodes. The Word2Vec model converts opcodes sequences into vectors in a high-dimensional space. The model assigns a unique vector to each unique word in the dataset. The vectors are positioned in such a way that semantically similar words are close to each other in the vector space.

$$Opcode \dashrightarrow Word2Vec \dashrightarrow Vector$$

After training the Word2Vec model, each opcode in a sequence is replaced by its corresponding vector. The function **word2vec_model.wv[word]** retrieves the vector representation of the specified opcode from a trained Word2Vec model. **word2vec_model.wv[word]** is used for generating dense vector representations (embeddings) of words in large texts and **wv** is stand for word vectors. It's an object inside the Word2Vec model that holds the actual word embeddings (Mikolov et al., 2013).

Padding the Sequences to Uniform Length

DL neural networks require inputs in fixed-size vector spaces to facilitate efficient model training in batches('Tf.Keras.Utils.Pad_sequences,' 2023). To standardize the length of all sequences, the **pad_sequences** module from TensorFlow's Keras library is employed. Sequences that are shorter than the maximum specified length are augmented with padding elements, usually vectors of zeros. The number of zeros added is equivalent to the difference between the maximum length and the length of the sequence.

$$Number\ of\ zeros\ added = max\_length - len(sequence)$$

Parameters for Language Model

Word2Vec learns high-dimensional word representations from the opcode sequences, where words that appear in similar contexts have similar vectors. The Word2Vec model (algorithm Choice = CBOW) is trained using both unigram and bigram opcode sequences as the input, with a specified **vector_size = 64**, each word is represented as a 64-dimensional vector, **window=5,** defines the context window size, meaning five words to the left and five words to the right of a word are considered as its context, **min_count = 1**, ensures even words occurring only once are considered in training ,and **workers = 4**, 4 worker threads used to train the mode, before being saved to "word2vec.model". Each sequence in unigram and bigram opcode sequences is then converted into **vectorized_sequences**, this method is converting each word in each sequence to its corresponding vector representation learned by the Word2Vec model, resulting in a nested list where each sequence is represented as a list of vectors. This type of representation is beneficial for feeding the sequences into deep learning models, for tasks such as classification or regression. Maximum sequence length, **max_length= 1000**, is calculated from **vectorized_sequences** to equalize the length of all sequences by padding shorter ones with zeros at the end, resulting in a uniform numerical representation, **X**, with a **data type** of **float32**. Lastly, class labels are converted to a one-hot encoded format, denoted as **y**, to serve as target outputs for the model, completing the data preprocessing required for feeding into the deep neural network. The snippet code is available in Figure 15.

Visualize Word Embeddings with PCA Decomposition using Scikit-Learn

After training the Word2Vec model and vectorizing the opcode sequences, the visual representation below, Figure 9, is obtained through PCA decomposition, showcasing their range and semantic relationships. This visualization helps in understanding how Word2Vec groups the data, which can then be used for further analysis, such as determining if the data needs scaling or spotting anomalies(Padmavathi et al., 2022). The snippet code is available in Figure 16.

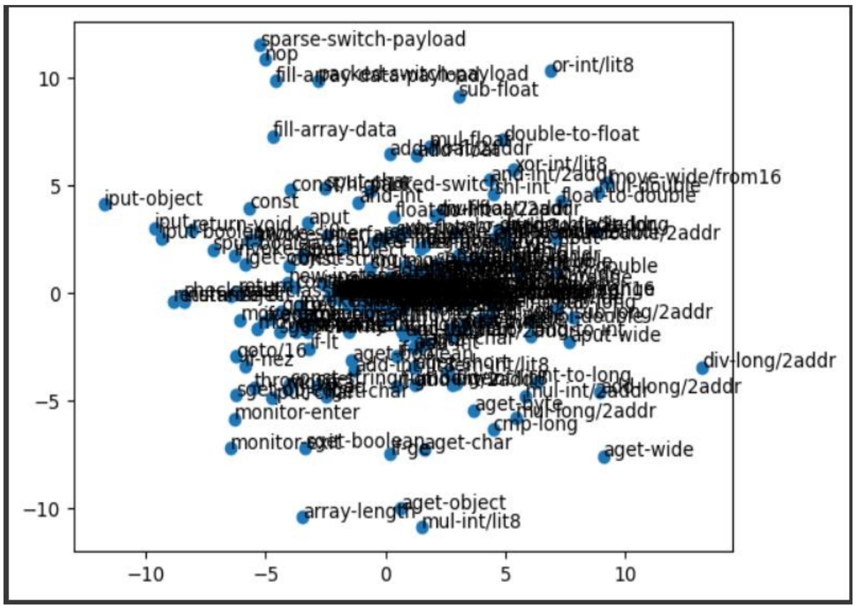


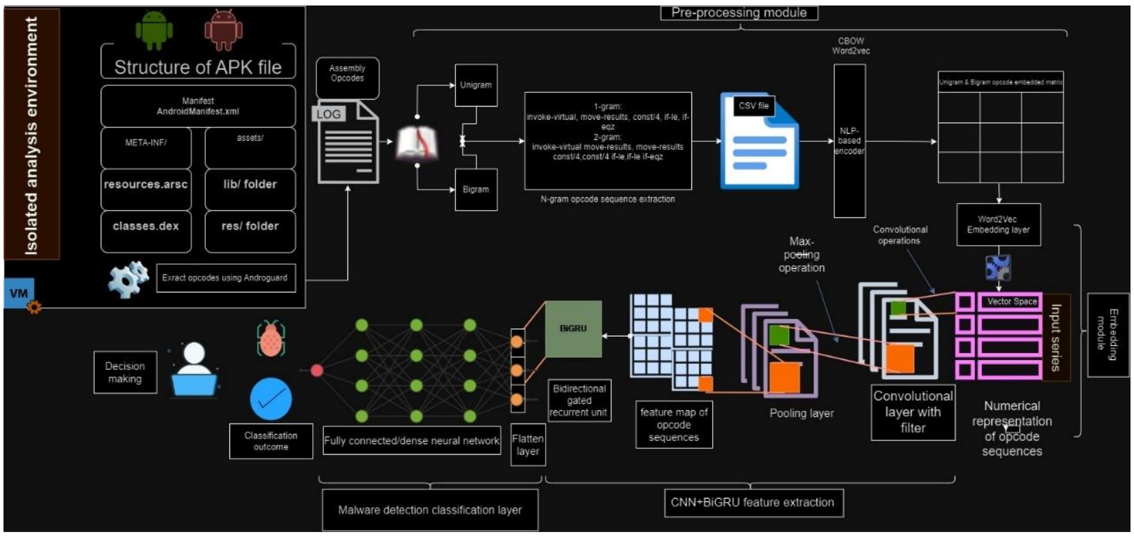
Figure 9: Virtualised opcode sequences in 2D

3.3.4 Binary Classification Model

A model has been developed based on Natural Language Processing (NLP) to categorize assembly opcodes as either benign or malicious sentences. The classification module consists of a fully connected neural network (FCNN) with hidden layers and a sigmoid activation function at the output layer. The network is trained using a binary cross-entropy loss function, and the classification outcome is computed based on the sigmoid activation of the FCNN (Maniriho et al., 2022). The experimental results presented in the document evaluate the framework's performance in binary classification tasks.

Binary Classification Model Train and Test Pipeline

Following a similar approach, I adopted in pre-trained model. I have established a pipeline for developing a binary classification model. This pipeline guides the dataset through preparation, modelling, training, and testing phases within a deep neural network. We executed the construction of this pipeline using Python 3.10.12, with the support of TensorFlow 2.13.0 open-source libraries. As detailed in the section on datasets, unigram and bigram opcode sequences were extracted from benign and malicious Dex files. This compiled dataset was then utilized to fuel a binary classification model. This dataset used to feed binary classification model based on a Natural Language Processing (NLP) and convolutional neural network (CNN)and bidirectional gated recurrent unit (CNN-BiGRU). (Yeboah and Baz Musah, 2022) labelled malicious opcode sequences rows with 1 and benign opcode sequences rows with 0 (Yeboah, 2022). Thus, I have returned the opcode sequences with their associated labels.



Isolated analysis environment
Structure of APK file
Manifest AndroidManifest.xml
META-INF/
assets/
resources.arsc
lib/ folder
classes.dex
res/ folder
Exract opcodes using Androguard
VM
Pre-processing module
Assembly Opcodes
LOG
Unigram
Bigram
1-gram: invoke-virtual, move-results, const/4, if-le, if-eqz
2-gram: invoke-virtual move-results, move-results const/4,const/4 if-le,if-le if-eqz
N-gram opcode sequence extraction
CSV file
CBOW Word2vec
NLP-based encoder
Unigram & Bigram opcode embedded matrix
Word2Vec Embedding layer
Convolutional operations
Max-pooling operation
Vector Space
Input series
Embedding module
BiGRU
Bidirectional gated recurrent unit
feature map of opcode sequences
Pooling layer
Convolutional layer with filter
Numerical representation of opcode sequences
Decision making
Classification outcome
Fully connected/dense neural network
Flatten layer
Malware detection classification layer
CNN+BiGRU feature extraction

Figure 10:The proposed framework for android malware detection

A binary classification model (Maniriho et al., 2022) is used to classify input data as either malware or benign. To construct a binary classification model, I have discussed the next essential components of the proposed framework. This binary classification model that merges a Convolutional Neural Network (CNN) with a Bidirectional Gated Recurrent Unit (BiGRU) involves utilizing the strengths of both neural network architectures to make binary decisions and this binary classification can be addressed as follows Figure 10.

CNN Feature Extractor

The CNN component of the framework is divided into a convolutional layer and a pooling layer, each performing distinctive tasks to extract high-level features of opcode sequences.

Convolutional Layer:

CNN's filters slide over the embedded opcode sequences, generating distinctive feature maps. Each filter focuses on different locations in the embedding vectors, reducing the dimensionality of features but maintaining the most relevant ones. The convolution operations are performed using a stride, and various stride values can be adapted. We have used the ReLU activation function to introduce non-linearity, enhancing the model's learning capability.

Pooling Layer:

After the convolutional layer finds all those features, the pooling layer takes over to perform dimensionality reduction and prevent overfitting. This layer uses max pooling, selecting the most important features from each feature map, capturing the essence of the processed opcode sequences. This results in a condensed feature matrix, highlighting the most relevant features of the opcode sequences, which is then passed to the next feature extractor, the BiGRU.

BiGRU feature extractor

The features developed by **CNN Feature Extractor** are quite foundational and carry lower-level semantics in a way that derived directly from the structure opcode sequences. To delve deeper into the complex detail patterns within these features, a BiGRU module is employed so that the model can be efficient in detection of new malware variants. This BiGRU module processes the sequences in both forward and backward directions. This bidirectional modelling enables the capture of significant dependencies across the opcode feature maps. The output of the BiGRU module is a set of more sophisticated features about the opcode sequences, which are then passed to a flatten layer for further processing.

Flatten layer

The flatten layer converts the output of the BiGRU, which holds the advanced features of opcode sequences, into a one-dimensional vector. This flattened single feature vector is then used as the input for the next layer, to perform the classification task for detecting malware.

Classification Module (fully connected deep neural network module)

The classification module is a crucial part of the system, employing a Fully Connected Deep Neural Network (FCDNN) to process the features derived from opcode sequences. The flattened feature vectors from the BiGRU module are evaluated through a fully connected neural network, consisting of hidden layers with ReLU activation and an output layer applying a sigmoid activation function to classify the opcode sequences into either benign or malicious. This process involves the computation of activations, within the hidden layers, using weights, W, and biases,b, from the connections of the input neurons to the hidden layer neurons. To mitigate overfitting, a dropout regularization technique is applied with a rate of 0.2, implying that 20% of the connection weights are randomly disabled at each training iteration. This forces each neuron to be more independent and generalize better on the opcode sequences. The loss or classification error is computed using binary-cross entropy, contrasting the predicted probabilities with the true classifications of the opcode sequences, and the Adaptive Moment Estimation (Adam) optimizer refines the learning weights,W, and biases,b, to minimize this error. The backpropagation method is employed to update the network learning weights, and the final classification outcomes are deduced using a sigmoid function, defining whether each executable file is benign or malicious based on the features extracted from opcode sequences.

Splitting data into training and testing sets

In this step, the dataset has been divided into two main parts: training and testing. These will be used in the subsequent training and testing processes. Based on the widely accepted convention, I have allocated 80% of the dataset for training (**X_train** and **y_train**) and 20% for testing (**X_test** and **y_test**). The **train_test_split** function from **scikit-learn** is employed to perform this split, with a **random_state** of **42** ensuring reproducibility. The shapes of the resulting subsets are then printed to verify the structure of the split data. The snippet code is available in Figure 17.

Parameters for Training and Testing Processes

The optimal parameters can be determined using a grid search method. This method specifies a grid of hyperparameters (for example, batch_size, dropout_rate, epochs, optimizer) and trains/tests a model to find the best combination to ensure accurate classification of the processed opcode sequences. However, in my case, running a grid search is computationally expensive and can take a long time, depending on the number of combinations and the amount of data available(Authorpedregosa et al., 2011). Therefore, opted to use arbitrary parameters, employing trial and error to find satisfactory results.

Epochs	Loss Values	Accuracy %
1	0.1680	0.9428
2	0.1069	0.9571
3	0.0896	0.9638
4	0.0852	0.9680
Learning Rate= 0.0001		

Table 1:The effects of epochs on accuracy.

Learning Rate per an Epoch	Loss Values	Accuracy %
0.00005	0.5928	0.7569
0.00003	0.5758	0.6627
0.00002	0.6668	0.6846

Table 2:The effects of Learning rate on validation losses.

In the model for the **Conv1D** layer with **64** filters, the **kernel_size** of **3**, the **ReLU** activation function, and an **input_shape** of **(max_length, 64)**, where `max_length` is the length of the input sequences, and `10` is 64 is the vocabulary size. Following this is the **MaxPooling1D** layer with the **pool_size** of **2**, reducing the spatial size of the representation. Then two **Bidirectional GRU layers** are employed, the first one contains **64** recurrent units and returns the full sequence, while the second one consists of **32** recurrent units and only returns the last output in the output sequence. Subsequently, the **Flatten layer** is used to flatten the input. This model also includes two **Dense layers**, paired with **Dropout** layers for regularization, the first one contains **64** neurons with a dropout rate of **0.2**, and the second one has **32** neurons, also with the **0.2** dropout rate. This model concludes with an output Dense layer with **1** neuron and a **sigmoid** activation function, suitable for binary classification. For compilation, this model utilizes **binary_crossentropy** as the **loss function**, the **Adam optimizer**, and evaluates using the **accuracy** metric. Finally, this model is trained using a **batch_size** of **64** and **10 epochs**, with **20%** of the training data reserved for testing. The snippet code is available in Figure 18.

3.4 SUMMARY

In this section, the technical specifications of a recent automated framework were presented, detailing particularly the approach, dataset collection process, dataset format, deep neural network training environment, required libraries and modules, model pipeline, and parameters used throughout the modelling process. Constructed a model pre-trained on assembly opcodes and subsequently utilized this pre-trained model to fine-tune a binary classification model to enhance detection capabilities. This framework, designed for the detection of both known and zero-day malicious opcode sequences for android devices, utilizes Word2Vec NLP-based encoder, a Convolutional Neural Network (CNN), and a Bidirectional Gated Recurrent Unit (BiGRU) to extract high-level features from android opcode sequences and a fully connected deep neural network. The implementation leveraged modules from TensorFlow and Keras libraries in the Python programming language, along with Sklearn. Lastly, the range of values tested on the necessary parameters during the training and testing processes was outlined.

4 Results and Analysis

This chapter will present the analyses and results of the training and detection performances of the binary classification model. It will begin with an interpretation of the model using LIME (Local Interpretable Model-Agnostic Explanations), to understand the basis on which the model makes predictions and classifications. Subsequently, evaluation criteria will be established to assess the model’s performance. Finally, the algorithms' performance will be compared to the findings reported in previous works.

4.1 Understanding the model Prediction With LIME

LIME (Local Interpretable Model-agnostic Explanations) is a technique to explain the predictions of any machine learning classifier. It approximates the decision boundary locally with an interpretable model and helps in explaining the model predictions(M. T. C. Ribeiro, 2016). Understanding the prediction or classification outcome of deep learning models can be very complicated, due to the numerous parameters they employ during predictions. LIME(Ribeiro et al., 2016a) has been incorporated into this proposed malware detection framework to aid in comprehending predictions. LIME serves as a free automated library capable of interpreting the predictions made by deep learning models. Specifically, in text-based classification scenarios, LIME deciphers the predicted results and uncovers the significance of the most influential words(tokens)contributing to those results. This is particularly suitable for this framework as it handles sequences of android opcode sequences symbolizing malware and benign files (Maniriho et al., 2022).

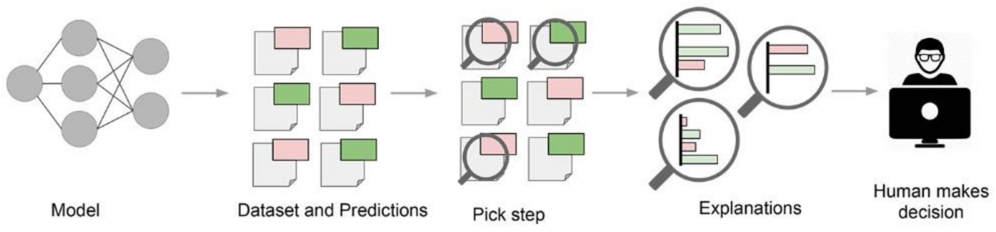
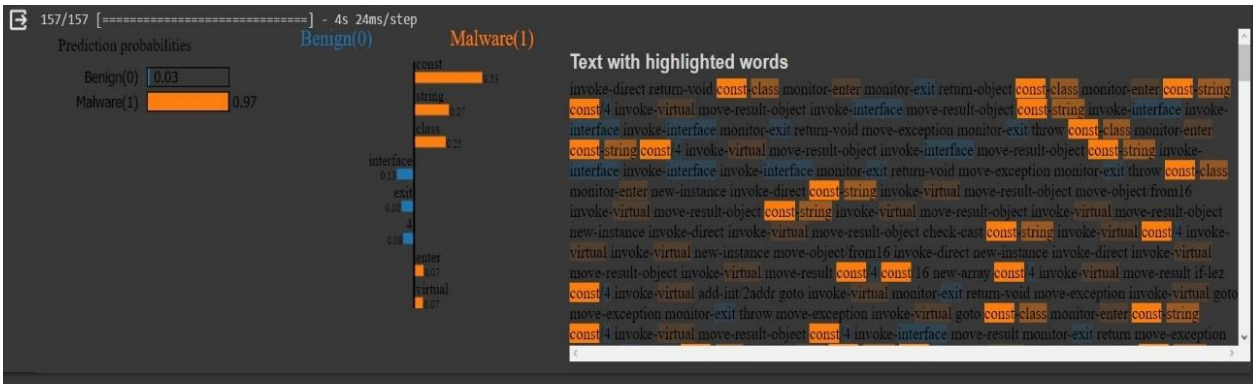


Figure 11:: Explaining the predicted outcome(Ribeiro et al., 2016b).

Figure 11 shows how LIME operates to deliver explanations for a given prediction related to opcode sequences. It explains the framework’s predictions on an individual data sample level, enabling security analysts to comprehend the framework’s predictions and base decisions on them. LIME views the deep learning model as a black box and uncovers the relationships between input and output represented by the model(Hulstaert, 2018) . The output generated by LIME includes a list of explanations, revealing the contributions of each feature to the final classification or prediction of the opcode sequences. This produces local interpretability, permitting security professionals to identify which changes in opcode features, will most affect the predicted output.

For example, in Figure 12, the opcodes **const**, **string**, **class**, **enter**, and **virtual** are depicted as the most opcode sequences contributing to the final classification of the sequences as **malicious**, while **interface**, **exit**, and **4** are opcode sequences opposing the final prediction. The output from LIME also presents another feature, it assigns weight probabilities to the correct classification of malicious opcode sequences, which are then totalled to provide an overall weight of 0.97. It's important to note that the tokens labelled under **Text with highlighted words** symbolize the original sequences of opcodes. The highlighted tokens indicate those opcode sequences which played a role in the classification of the sequence. Hence, equipped with this information, a security analyst can assess whether to trust the model's prediction or not. The snippet code is available in Figure 20.



```
157/157 [==============================] - 4s 24ms/step
```

Text with highlighted words

invoke-direct return-void const-class monitor-enter monitor-exit return-object const-class monitor-enter const-string const/4 invoke-virtual move-result-object invoke-interface move-result-object const-string invoke-interface invoke-interface invoke-interface monitor-exit return-void move-exception monitor-exit throw const-class monitor-enter const-string const/4 invoke-virtual move-result-object invoke-interface move-result-object const-string invoke-interface invoke-interface invoke-interface monitor-exit return-void move-exception monitor-exit throw const-class monitor-enter new-instance invoke-direct const-string invoke-virtual move-result-object move-object/from16 invoke-virtual move-result-object const-string invoke-virtual move-result-object invoke-virtual move-result-object new-instance invoke-direct invoke-virtual move-result-object check-cast const-string invoke-virtual const/4 invoke-virtual invoke-virtual new-instance move-object/from16 invoke-direct new-instance invoke-direct invoke-virtual move-result-object invoke-virtual move-result const/4 const/16 new-array const/4 invoke-virtual move-result if-lez const/4 invoke-virtual add-int/2addr goto invoke-virtual monitor-exit return-void move-exception invoke-virtual goto move-exception monitor-exit throw move-exception invoke-virtual goto const-class monitor-enter const-string const/4 invoke-virtual move-result-object const/4 invoke-interface move-result monitor-exit return move-exception

Figure 12: An example of how LIME explains the classification result.

4.2 Evaluation Criteria

To assess the efficiency and success of the model, evaluating its performance based on several metrics, including precision (P), recall (R), and F1-Score for each class individually (Benign, Malware) was done. These metrics, which are derived from the confusion matrix, are widely used in binary classification tasks to provide a comprehensive view of the model's performance. The **evaluate** function in Keras is used to provide the model's accuracy. Precision, recall, and F1 score were then calculated separately using the components of the confusion matrix. In addition, the performance of the model is visualized using the **Matplotlib** library to evaluate underfitting or overfitting and, if needed, to adjust the hyperparameters(Sharma, 2019).

- **Accuracy**: how many samples were correctly classified in the training dataset.
- **False positive (FP)**: how many samples were classified as malware but were not.
- **False negative** (FN): how many samples were classified as benign software but was not.
- **True positive (TP):** how many samples were classified as malware and was.
- **True negative (TN)**: how many samples were classified as benign software and was.
- **Epochs (n)**: An epoch is one complete pass through the entire training dataset.
- **Val_accuracy:** Gives insight into how well the model is likely to perform on unseen data.

For Class: Benign

$$Precision = \frac{TN}{(TN + FN)}$$

$$Recall = \frac{TN}{(TN + FP)}$$

$$F1\,Score = 2 * \frac{(Precision * Recall)}{Precision + Recall)}$$

For Class: Malware

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1\_Score = 2 * \frac{Precision \times Recall}{Precision + Recall}$$

4.2.1 Classification Results

This framework was trained and tested on opcode sequences dataset extracted from Dex files through the static analysis approach by (Cybersecurity, 2020; Yeboah, 2022). Various evaluations were carried out, and the classification results are presented in this Section. Table 3 and **Error! Reference source not found.** illustrates that the model appears to be converging as the number of epochs increase and its ability to generalize over unseen data. The optimal balance appears at 80 epochs, where the model demonstrates high reliability and robustness in classifying the data. The snippet code is available in Figure 21.

n	Accuracy (%)	Val_accuracy (%)
20	0.9897	0.9714
40	0.9895	0.9748
60	0.9918	0.9756
80	0.9914	0.9790
100	0.9905	0.9773

Table 3:Detection accuracy achieved by the mode (n is the numbers of epochs)

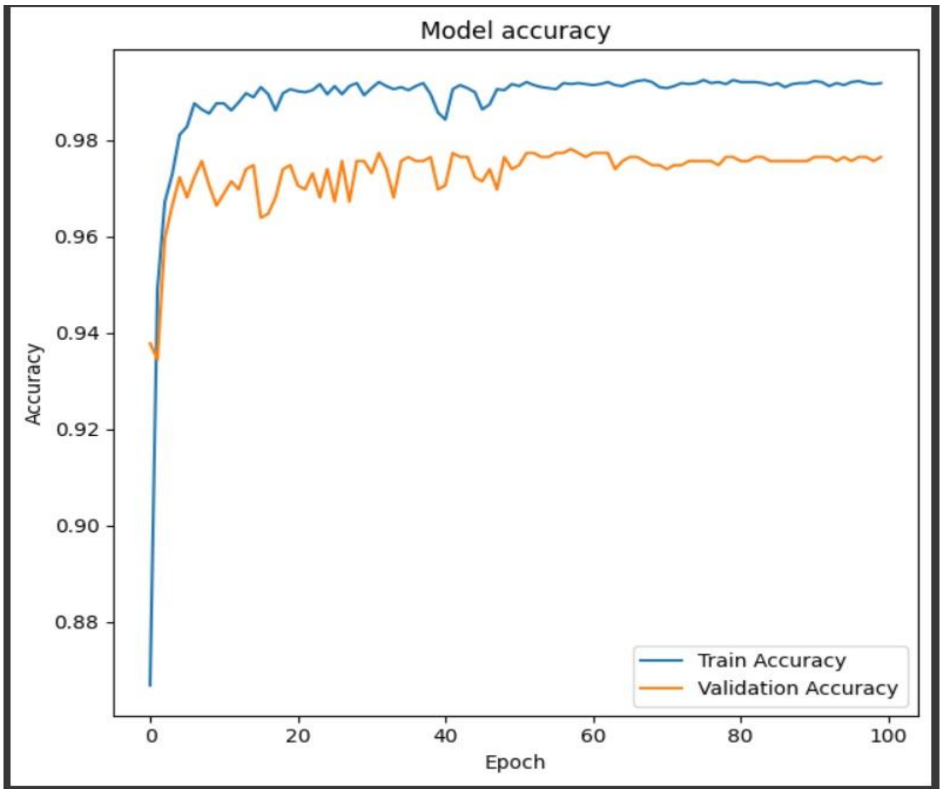


Figure 13:Accuracy and Val_accuracy trends over 100 epochs

Table 1 shows the confusion matrix which is computed using the actual labels and the predicted labels by the model. From the computed matrix, it is observed that the model accurately identified 458 benign files (True Negatives) and 979 malware files (True Positives), demonstrating good accuracy in its predictions. However, the model also provided some instances of misclassification, incorrectly labelling 27 benign files as malware (False Positives) and misidentifying 22 malware files as benign (False Negatives). This comprehensive view, provided by the confusion matrix, helps in understanding not only the instances where the model performed correctly but also where it makes mistakes in its prediction.

		Predicted Label	
		Predicted Benign	Predicted Malware
Actual Label	Actual Benign	True Negative (TN): 458	False Positive (FP): 27
	Actual Malware	False Negative (FN): 22	True Positive (TP): 979

Table 4:The Confusion matrix

Table 5 shows after running the model for 100 epochs, the performance metrics are as follows. For the Benign class, a precision of 0.95 indicates that 95% of the instances that were predicted as Benign were correctly identified, suggesting high reliability in detecting benign opcode sequences. The recall of 0.94 implies that out of all the actual Benign instances, 94% were successfully classified as such, reflecting the model's capability to capture most of the Benign instances accurately. The F1-Score, stands at 0.95, showcasing a balanced trade-off between false positives and false negatives for Benign classifications. Moving on to the Malware class, the model shows the precision of 0.97, meaning that 97% of the sequences predicted as Malware were indeed malicious. The recall for the Malware class is at 0.98, showing that the model correctly identified 98% of all actual Malware instances, demonstrating a high sensitivity to malicious opcode sequences. Lastly, the F1-Score for malware is 0.98, pointing to an excellent balance in the identification of malware opcode sequences, indicating minimal incorrect classifications. The snippet code is available in Figure 19.

n	Predicted Class	Precision	Recall	F1-Score
100	Benign	0.95	0.94	0.95
	Malware	0.97	0.98	0.98

Table 5:The performance metrics for the model

4.3 COMPARISON OF THE MODELS

Analysed approaches from other papers have addressed the same problem or used similar techniques to compare the performance of the proposed model to other detection systems.

In Table 6, the performance of the model is compared to that of (Yeboah and Baz Musah, 2022), utilizing their dataset (Yeboah, 2022) for training and testing the proposed model. Both studies employ deep learning (DL) methods. The proposed model shows considerable improvement, reaching an accuracy of 97.17%. The enhancement in accuracy is pivotal as it underscores the effectiveness of the proposed model in Android malware detection. Employing this model enabled me to achieve high positive predictive and recall (sensitivity) values of 98%. It is essential to note that this method is resilient against obfuscation compared to machine learning-based image classifications which can easily bypassed by adversarial attacks(Goodfellow et al., 2014). However , in the context of Android malware detection, the semantic nature of executable samples inhibits adversarial evasion techniques(Yeboah and Baz Musah, 2022).

In comparison to the opcode sequence-based approach model presented by (McLaughlin et al., 2017), the authors address the problems associated with traditional Android malware detection methods that rely on manual examination of behaviour and state that these methods are limited in scalability as new malware is designed to evade existing signatures. The authors propose a deep convolutional neural network system that performs classification based on the static analysis of the raw opcode sequence from a disassembled program, which achieved an accuracy of 94.8%. However, the model proposed in this report surpasses this, attaining an accuracy of 97.17%. This improvement in accuracy is crucial as it allows the model to accurately classify a substantial number of malware samples, confirming them as positive predictions and simultaneously minimizing false positives, as shown in Table 6.

The model's performance has been compared with a hybrid sequence-based Android malware detection method using natural language processing (NLP) by (Zhang et al., 2021). The authors propose CoDroid, which combined both static opcode sequences and dynamic system call sequences as input. These sequences are treated as sentences in NLP and classified using a CNN-BiLSTM-Attention model as benign or malicious. Their method outperforms other existing methods in terms of detection performance. It achieves a higher F1-score, precision, recall, and accuracy compared to traditional machine learning algorithms such as K-nearest neighbour (KNN), Naive Bayes (NB), Logistic Regression (LR), and Random Forest (RF). In comparison with other related detection methods, CoDroid demonstrates superior performance, obtaining the highest accuracy score of 97.6%. This is near the 97.17% accuracy of the model proposed in this report, with a negligible difference in accuracy. Moreover, in relation to other evaluation criteria, the model in this report demonstrated slightly better performance metrics, exhibiting a precision and recall of 0.98, Table 6. It is noteworthy that the datasets used in both this report is cleaned using the AndroGuard tool to remove replicated and invalid applications, and all applications are verified by the online malware detection engine VirusTotal.

Approach	Model	Benign	Malware	Accuracy	F1-score	Precision	Recall
opcode n-grams (n = 3)(Song, 2019)	RF	154	180	0.946	0.945	0.951	0.935
opcode-LSTM(Lu, 2019)	LSTM	123	969	0.938	0.936	0.957	0.895
(McLaughlin et al., 2017)	CNN	863	1260	0.948	0.947	0.953	0.935
CoDroid(Zhang et al., 2021)	CNN–BiLSTM–Attention	2,707	2,978	**0.976**	**0.986**	0.954	0.978
(Mahdavifar et al., 2020)	Semi-Supervised Deep Learning (Classification)	1,795	9,803	0.967	0.9784	**0.99**	0.965
(Yeboah and Baz Musah, 2022)	1D CNN (opcode n-grams (n = 1,2))	2477	4948	0.955	0.97	0.980	0.97
The proposed model	(CNN-BiGRU) (opcode n-grams (n = 1,2))	2477	4948	**0.9717**	**0.98**	0.98	**0.98**

Table 6:Comparing the proposed model against previous works.

5 CONCLUSIONS AND FUTURE WORK

5.1 CONCLUSION

This work has presented a static malware detection framework that leverages natural language processing and deep learning techniques. The dataset(Yeboah, 2022) used in this report is deemed substantial due to it is large, diverse, and realistic nature. It encompasses raw opcode sequences (Unigram and Bigram) extracted from Android executables, and the proposed model's performance was evaluated using this dataset. Additionally, Word2Vec is employed to generate semantic-preserved vector representations of the opcode sequences. The binary classification utilizes the CNN-BiGRU module to automate feature extraction and select high-relevance features, which are then fed to a fully connected neural network component for the classification of each opcode sequence. The evaluation results show high detection accuracy and better precision achieved by the model on both seen and unseen data. This provides the confidence that this framework can successfully identify newly emerging malware based on opcode sequences. LIME was integrated into the model, enabling the framework to interpret the model's predictions. This integration could help in identifying why specific opcode sequences are classified as benign or malicious and validating the model's predictions, which is pivotal in identifying and mitigating any biases or misclassifications(Ribeiro et al., 2016a).

5.2 FUTURE WORK

Due to the continually evolving nature of malware and the ongoing development of new evasion techniques by malware creators, there are growing concerns about privacy and security. Malware creators often employ techniques such as code encryption and packing to bypass detection systems. NLP-based methods may struggle to detect such obfuscated code during execution due to these advanced evasion techniques. Categorizing Android malware is a crucial step in precisely understanding the threats to which we are vulnerable. Given these challenges, the intention is to extend this methodology to a hybrid malware category classification for Android platforms in future work. Constant updates to the dataset are essential. These updates should include the hash value for each file to prevent duplication and facilitate the addition of behavioural characteristics of newly discovered malware variants in the future. Moreover, updating the model and enhancing anomaly detection techniques will be crucial to address this issue effectively.

6 APPENDIX

The source code, the Jupyter Notebook file, along with a video demonstrating how the model works have been shared on OneDrive. Both are available via the link below.

https://stummuac-my.sharepoint.com/:f:/r/personal/22574929_stu_mmu_ac_uk/Documents/MSc_Project(6G7V0007_2223_9F)?csf=1&web=1&e=RgWKkQ

Figure 14:Cell 1: Import Libraries:

```
# Imports the pandas library which provides tools for data manipulation and analysis.
import pandas as pd
# Imports the Word2Vec class from the gensim library, a model used to generate word embeddings.
from gensim.models import Word2Vec
# Imports the function to split datasets into training and testing subsets.
from sklearn.model_selection import train_test_split
#Imports the function for padding sequences to the same length.
from tensorflow.keras.preprocessing.sequence import pad_sequences
#Imports the class to create a sequential neural network.
from tensorflow.keras.models import Sequential
# Imports the numpy library for numerical operations.
import numpy as np
# Imports the matplotlib library for plotting.
import matplotlib.pyplot as plt
#Imports the function to compute the confusion matrix.
from sklearn.metrics import confusion_matrix
#Imports PCA (Principal Component Analysis) for dimensionality reduction.
from sklearn.decomposition import PCA
# Imports various layers from Keras that will be used to build the model.
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Bidirectional, GRU, Flatten, Dense, Dropout
#Imports the Adam optimizer.
from tensorflow.keras.optimizers import Adam
```

Cell 2: Install lime:

```
#Installs the lime library, which is used for explaining the predictions of black-box models.

pip install lime
```

Cell 3: Import lime Libraries:

```
#import lime and subsequent lines: Imports various modules from the lime library to explain the
#model's decisions.

import lime

from lime import lime_text

from lime.lime_text import LimeTextExplainer
```

Cell 4: Mount Google Drive:

```
#Imports tools to interact with Google Colab's environment.

from google.colab import drive

#Mounts Google Drive to the Colab environment to access files.

drive.mount('/content/drive/')
```

Figure 15:Cell 5: Load Data and Preprocess(Sharma, 2019; Brownlee, 2020; Řehůřek, 2022):

```
#Specifies the path to the CSV file containing the Android opcode sequences.

file_path = "/content/drive//MyDrive/ColabNotebooks/Android_Opcode_Sequences.csv"

data = pd.read_csv(file_path) #Uses pandas to read the CSV file into a DataFrame.

#The opcodes are space-separated in the CSV file and Split them into lists.

#Convert opcode sequences, which are represented as space-separated strings in CSV file, into lists of individual opcodes.

data['opcodes'] = data['opcodes'].apply(lambda x: x.split())

# Separate features and labels

opcode_sequences = data['opcodes'].tolist()

y = data['labels'].values

# I have trained a Word2Vec model on the opcode sequences.

# Converts the opcode sequences into vectors using the Word2Vec model.

# Train Word2Vec model

word2vec_model = Word2Vec(sentences=opcode_sequences, vector_size=10, window=5, min_count=1, workers=4,sg=0)

#Save the model

word2vec_model.save("word2vec.model")

#  Converts the opcode sequences into vectors using the trained Word2Vec model.

vectorized_sequences = [[word2vec_model.wv[word] for word in sequence] for sequence in opcode_sequences]

# Find the maximum sequence length

max_length = max(len(sequence) for sequence in vectorized_sequences)

# Pads the vectorized opcode sequences to ensure they all have the same length.

X = pad_sequences(vectorized_sequences, maxlen=max_length, dtype='float32', padding='post')
```

Figure 16:Cell 6: Visualize Word Embeddings(Tam, 2021; Dp, 2023):

```
#Visualize the word vectors using PCA technique to see if similar words are clustered
words = list(word2vec_model.wv.index_to_key)
vectors = [word2vec_model.wv[word] for word in words]
# PCA to reduce the dimensions of the word embeddings.
pca = PCA(n_components=2)
result = pca.fit_transform(vectors)
# Plots the words in the 2D PCA space to visualize the embeddings.
plt.scatter(result[:, 0], result[:, 1])
for i, word in enumerate(words):
   plt.annotate(word, xy=(result[i, 0], result[i, 1]))
plt.show()
```

Figure 17:Cell 7: Train/Test Split(Sharma, 2019; scikit, 2023a):

```
# Splits the data into training and testing sets.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Displays the shapes of the training and testing sets.
print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape:", y_test.shape)
```

Figure 18:Cell 8: Model Definition and Training(Sharma, 2019; Raihan, 2021; 'The Sequential model,' 2023):

```
# Defines the neural network architecture using the Sequential API.
model = Sequential([
    # Convolutional Layer
    Conv1D(64, kernel_size=3, activation='relu', input_shape=(max_length, 10)),# 64 is the vector_size
in Word2Vec model
    MaxPooling1D(pool_size=2),
    # Bi-directional GRU Layers
    Bidirectional(GRU(64, return_sequences=True)),
    Bidirectional(GRU(32)),
    # Flatten Layer
    Flatten(),
    # Fully Connected Neural Network Module
    Dense(64, activation='relu'),
    Dropout(0.2),  # Dropout Regularization
    Dense(32, activation='relu'),
    Dropout(0.2),  # Dropout Regularization
    # Output Layer
    Dense(1, activation='sigmoid')  # Sigmoid Activation for Binary Classification
])
# Compile the model(Specifies the loss function, optimizer, and metrics).
model.compile(loss='binary_crossentropy', optimizer=Adam(), metrics=['accuracy'])
# Trains the model on the training data.
batch_size = 64
epochs = 100
history = model.fit(X_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.2)
```

Figure 19:Cell 9: Test Model(Sharma, 2019; scikit, 2023b):

```
#Evaluates the model's accuracy on the test data.

_, accuracy = model.evaluate(X_test, y_test)

print('Accuracy: %.2f' % (accuracy*100))

# Gets predictions for the test set.

test_predictions = (model.predict(X_test)> 0.5).astype(int)

test_predictions.shape

y_test_labels = y_test.flatten()

test_predictions_labels = test_predictions.flatten()


# Compute the confusion matrix to see true positives, true negatives, false positives, and false negatives.

conf_matrix = confusion_matrix(y_test_labels, test_predictions_labels)

true_negative, false_positive, false_negative, true_positive = conf_matrix.ravel()

# Print the confusion matrix

print("Confusion Matrix:")

print(conf_matrix)

# Print individual components of the confusion matrix

#how many samples were classified as benign software and was.

print("True Negative (TN):", true_negative)

#how many samples were classified as malware but were not.

print("False Positive (FP):", false_positive)

#how many samples were classified as benign software but was not.

print("False Negative (FN):", false_negative)

#how many samples were classified as malware and was.

print("True Positive (TP):", true_positive)
```

Cell 10: Display Misclassified Examples:

```
# Iterates through test predictions and prints instances where the prediction doesn't match the
#actual label.

for i in range(1486):

   if (test_predictions[i]) != (y_test[i]):

     print('%d (expected %d)' % ((test_predictions[i]),(y_test[i])))
```

Figure 20:Cell 11: Lime Explanation(M. T. Ribeiro, 2016; 'Explain your model predictions with LIME,' 2020):

```
# Initializes an explainer to understand the model's decisions.
#Create a LimeTextExplainer
explainer = lime_text.LimeTextExplainer(class_names=['Benign(0)', 'Malware(1)'])
#  idx is the index of the instance in the dataset you want to explain
idx = 5433 # the index of the instance you want to explain
instance = ' '.join(opcode_sequences[idx])  # Converting list of opcodes to space-separated string
# Defines a function to make predictions using the model.
def predictor(texts):
    # Initialize list to hold sequences
    sequences = []
    # Convert texts to sequences of word vectors
    for text in texts:
        sequence = []
        words = text.split()  # Split text into words
        for word in words:
            # Check if the word is in the model's vocabulary
            if word in word2vec_model.wv.index_to_key:
                # Append the word vector to the sequence
                sequence.append(word2vec_model.wv[word])
        # Append the sequence to the list of sequences
        sequences.append(sequence)
    # Pad sequences
    padded_sequences = pad_sequences(sequences, maxlen=max_length, dtype='float32',
padding='post')
    # Uses Lime to explain a particular instance's prediction.
    predictions = model.predict(padded_sequences)
    # Return stacked predictions
    return np.hstack((1 - predictions, predictions))
exp = explainer.explain_instance(instance, predictor, num_features=8)
# Show explanation in the notebook
exp.show_in_notebook(text=True)
```

Figure 21:Cell 13: Training Visualization(Sharma, 2019):

```
# Plots the training accuracy and validation accuracy over epochs.

# history is the return value from model.fit(...)

plt.figure(figsize=(12, 6))

# Subplot for accuracy

plt.subplot(1, 2, 1)

plt.plot(history.history['accuracy'], label='Train Accuracy')

plt.plot(history.history['val_accuracy'], label='Validation Accuracy')

plt.title('Model accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend()

plt.tight_layout()

plt.show()
```

7 References

Alomari, E. S., Nuiaa, R. R., Alyasseri, Z. A. A., Mohammed, H. J., Sani, N. S., Esa, M. I. and Musawi, B. A. (2023) 'Malware Detection Using Deep Learning and Correlation-Based Feature Selection.' *Symmetry*, 15(1) p. 123.

'Android - statistics & facts.' (2023), 2023,

Authorpedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., et al. (2011) 'Machine Learning in Python.' 12, 2011, pp. 2825-2830.

Brownlee, J. (2020) 'How to Develop Word Embeddings in Python with Gensim.' 2020,

Chen, S., Xue, M., Fan, L., Hao, S., Xu, L., Zhu, H. and Li, B. (2018) 'Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach.' *computers & security*, 73 pp. 326-344.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H. and Bengio, Y. (2014) 'Learning phrase representations using RNN encoder-decoder for statistical machine translation.' *arXiv preprint arXiv:1406.1078*,

Cybersecurity, C. I. f. (2020) 'Datasets.' 2020,

Dp, W. b. (2023) 'Visualize Word Embeddings with PCA Decomposition using Scikit-Learn Pipelines.' 2023/7/8,

'Explain your model predictions with LIME.' (2020), 2020/2/13,

Goodfellow, I. J., Shlens, J. and Szegedy, C. (2014) 'Explaining and harnessing adversarial examples.' *arXiv preprint arXiv:1412.6572*,

Hulstaert, L. (2018) 'Understanding model predictions with LIME.' 2018/7/11,

Ito, R. and Mimura, M. (2019) *Detecting Unknown Malware from ASCII Strings with Natural Language Processing Techniques.* 1-2 Aug. 2019.

kaspersky. (2021) 'Machine Learning for Malware Detection.' 2021,

Lauriola, I., Lavelli, A. and Aiolli, F. (2022) 'An introduction to Deep Learning in Natural Language Processing: Models, techniques, and tools.' *Neurocomputing*, 470, 2022/01/22/, pp. 443-456.

Lee, S., Jung, W., Kim, S. and Kim, E. T. (2019) *Android Malware Similarity Clustering using Method based Opcode Sequence and Jaccard Index.* 16-18 Oct. 2019.

Lee, S., Jung, W., Kim, S., Lee, J. and Kim, J.-S. (2019) 'Dexofuzzy: Android malware similarity clustering method using opcode sequence.' 2019,

Li, X., Zhang, Y., Jin, J., Sun, F., Li, N. and Liang, S. (2023) 'A model of integrating convolution and BiGRU dual-channel mechanism for Chinese medical text classifications.' *Plos one*, 18(3) p. e0282824.

Lu, R. (2019) 'Malware detection with lstm using opcode language.' *arXiv preprint arXiv:1906.04593*,

M, G. and Sethuraman, S. C. (2023) 'A comprehensive survey on deep learning based malware detection techniques.' *Computer Science Review*, 47, 2023/02/01/, p. 100529.

Mahdavifar, S., Kadir, A. F. A., Fatemi, R., Alhadidi, D. and Ghorbani, A. A. (2020) *Dynamic Android Malware Category Classification using Semi-Supervised Deep Learning.* 17-22 Aug. 2020.

Maniriho, P., Mahmood, A. N. and Chowdhury, M. J. M. (2022) 'MalDetConv: Automated Behaviour-based Malware Detection Framework Based on Natural Language Processing and Deep Learning Techniques.' *arXiv preprint arXiv:2209.03547*,

McLaughlin, N., Doupé, A., Ahn, G., Martinez-del-Rincon, J., Kang, B., Yerima, S., Miller, P., Sezer, S., et al. (2017) *Deep Android Malware Detection.*

Mehta, R., Jurečková, O. and Stamp, M. (2023) 'A Natural Language Processing Approach to Malware Classification.' *arXiv preprint arXiv:2307.11032*,

Mikolov, T., Chen, K., Corrado, G. and Dean, J. (2013) 'Efficient estimation of word representations in vector space.' *arXiv preprint arXiv:1301.3781*,

Mimura, M. and Ito, R. (2022) 'Applying NLP techniques to malware detection in a practical environment.' *International Journal of Information Security*, 21(2), 2022/04/01, pp. 279-291.

Muhammad, A. (2022) '- Analyzing and comparing the effectiveness of various machine learning algorithms for Android malware detection.'

O'Shea, K. and Nash, R. (2015) 'An introduction to convolutional neural networks.' *arXiv preprint arXiv:1511.08458*, Or-Meir, O., Nissim, N., Elovici, Y. and Rokach, L. (2019) 'Dynamic Malware Analysis in the Modern Era—A State of the Art Survey.' *ACM Comput. Surv.*, 52(5) p. Article 88.

Otter, D. W., Medina, J. R. and Kalita, J. K. (2020) 'A survey of the usages of deep learning for natural language processing.' *IEEE transactions on neural networks and learning systems*, 32(2) pp. 604-624.

Padmavathi, G., Shanmugapriya, D. and Roshni, A. (2022) 'Evaluation of Principal Component Analysis Variants to Assess Their Suitability for Mobile Malware Detection.' *In* Fausto Pedro García, M. (ed.) *Advances in Principal Component Analysis*. Rijeka: IntechOpen, p. Ch. 3. [Accessed on 2023-10-05] https://doi.org/10.5772/intechopen.105418

Pang, B., Nijkamp, E. and Wu, Y. N. (2020) 'Deep Learning With TensorFlow: A Review.' *Journal of Educational and Behavioral Statistics*, 45(2) pp. 227-248.

Preda, M. D., Christodorescu, M., Jha, S. and Debray, S. (2008) 'A semantics-based approach to malware detection.' *ACM Trans. Program. Lang. Syst.*, 30(5) p. Article 25.

Raihan, D. (2021) 'Deep learning techniques for text classification.' 2021/4/12,

Řehůřek, R. (2022) 'Gensim: topic modelling for humans.' 2022,

Ribeiro, M. T. (2016) 'Lime package — lime 0.1 documentation.' 2016,

Ribeiro, M. T., Singh, S. and Guestrin, C. (2016a) *" Why should i trust you?" Explaining the predictions of any classifier.*

Ribeiro, M. T., Singh, S. and Guestrin, C. (2016b) 'Local interpretable model-agnostic explanations (LIME): An introduction.' 2016/8/12,

Ribeiro, M. T. C. (2016) 'lime: Lime: Explaining the predictions of any machine learning classifier.' [Accessed on 2023/9/28].

Şahin, N. (2021) *Malware detection using transformers-based model GPT-2.* Middle East Technical University.

scikit, l. (2023a) 'Sklearn.Model_selection.Train_test_split.' 2023,

scikit, l. (2023b) 'Sklearn.Metrics.Confusion_matrix.' 2023,

'The Sequential model.' (2023), 2023,

Sharma, V. (2019) 'Deep learning model data visualization using Matplotlib.' 2019,

Shishkova, T. (2023) 'The mobile malware threat landscape in 2022.' 2023/2/27,

Song, W. (2019) 'AndroidMalware-ngram-RF: 基于机器学习的 android 恶意代码检测，n-gram opcode + RandomForest.' [Accessed on 2023/9/29].

Sun, J., Luo, X., Gao, H., Wang, W., Gao, Y. and Yang, X. (2020) 'Categorizing malware via A Word2Vec-based temporal convolutional network scheme.' *Journal of Cloud Computing*, 9 pp. 1-14.

Tam, A. (2021) 'Principal Component Analysis for Visualization.' 2021,

'Tf.Keras.Utils.Pad_sequences.' (2023), 2023,

Yamashita, R., Nishio, M., Do, R. K. G. and Togashi, K. (2018) 'Convolutional neural networks: an overview and application in radiology.' *Insights into Imaging*, 9(4), 2018/08/01, pp. 611-629.

Yeboah, P. N. (2022) Android_Opcode_Sequences.csv. (2022/1/22): figshare.

Yeboah, P. N. and Baz Musah, H. B. (2022) 'NLP Technique for Malware Detection Using 1D CNN Fusion Model.' *Security and Communication Networks*, 2022, 2022/06/10, p. 2957203.

Zhang, N., Xue, J., Ma, Y., Zhang, R., Liang, T. and Tan, Y.-a. (2021) 'Hybrid sequence-based Android malware detection using natural language processing.' *International Journal of Intelligent Systems*, 36(10) pp. 5770-5784.

Ethics number: 58590