Static Analysis for Android Malware detection using document vectors

Utkarsh Raghav

A thesis submitted to the University of Canberra
in partial fulfillment of the requirement for the degree of
Master by Research

January, 2023

FACULTY OF SCIENCE AND TECHNOLOGY

UNIVERSITY OF
CANBERRA
AUSTRALIA'S CAPITAL UNIVERSITY

Abstract

The prevalence of smart mobile devices has led to an upsurge in malware that targets mobile platforms. The dominant market player in the sector, Android OS, has been a favourite target for malicious actors. Various feature engineering techniques are used in the current machine learning and deep learning approaches for Android malware detection. In order to correctly identify dependable features, feature engineering for Android malware detection using multiple AI algorithms requires a particular level of expertise in Android malware and the platform itself. The majority of these engineered features are initially extracted by applying different static and dynamic analysis approaches. These allow researchers to obtain various types of information from Android application packages (APKs), such as required permissions, opcode sequences and control flow graphs, to name a few. This information is used (as is or in vectorised form) for training supervised learning models. Researchers have also applied Natural Language Processing techniques to the features extracted from APKs.

In order to automatically create feature vectors that can describe the data included in Android manifests and Dalvik executable files inside an APK, this study focused on developing a novel method that uses static analysis and the NLP technique of document embeddings. We designed a system that takes Android APK files as input documents and generates the feature embeddings. This system removes the need for manual identification & extraction of features. We use these embeddings to train various Android Malware detection models to experimentally evaluate the effectiveness of these automatically generated features. The experiments were done by training and evaluating 5 different supervised learning models. We did our experiments on APKs from two well-known datasets, DREBIN and AndroZoo. We trained and validated our models with 4000 files (training set). We had kept separate 700 files (test set) which were not used during training and validation. We used our trained models to predict the classes of the unseen file embeddings from the test set.

The automatically generated features allowed training of robust detection models. The Android malware detection models performed best with Android manifest file embeddings concatenated with Dalvik executable file embeddings, with some of the models achieving Precision, Recall and Accuracy values above 99% consistently during development

and over 97% against unseen file embeddings. The prediction accuracy of the detection model trained on our automatically generated features was equivalent to the accuracy achieved by one of the most cited research works known as DREBIN, which was 94%. We also provided a simple method to directly utilise the file present in Android APK to create feature embeddings without scouring through Android application files to identify reliable features. The resulting system can be further improved against new emerging threats and be better trained by just gathering more samples.

Acknowledgements

I want to start by expressing my sincere gratitude to my supervisory panel (primary supervisor Dr. Elisa Martinez-Marroquin, secondary supervisor Dr. Wanli Ma and Dr. Yohannes Kinfu) for their indispensable and continuous guidance, criticism, and support throughout the various phases of my research study.

Second, I want to express my gratitude to my close friends: Dr. Danish Ahmad, Mr. Sigit Jati, Mr. Ziqi Guo and Ms. Roji Ranjit. Throughout my entire postgraduate degree in Canberra, Australia, they offered me joy and happiness. I also want to thank my parents and let them know how grateful I am. Without their patience and support, I would not have finished my degree on time.

I would also like to thank Dr. Abu Barkat Ullah, Dr. Ram Subramanian and Mrs. Tharanga Samaranayaka Jayawardana for their thoughtful advice and help with my professional progress.

Abbreviations

ML	Machine Learning
DL	Deep Learning
doc2vec	Document Vectors (or paragraph vectors)
APK	Android Application Package
PV-DM	Paragraph Vector - Distributed Memory
PV-DBoW	Paragraph Vector - Distributed Bag of Words
Dex	Dalvik executable
TF-IDF	Term Frequency - Inverse Document Frequency
CNN	Convolutional Neural Network
SVM	Support Vector Machine
LR	Logistic Regression
kNN	k-Nearest neighbours
DT	Decision Tree

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

The Android marketplaces now have a higher prevalence of malware than they did a few years ago. As of March 2020, 400,000 samples of Android malware were being taken per month on average [1]. Since 2014, Android malware has been steadily increasing [2]. Therefore, an automated detection strategy is needed to keep up with the growing number of dangerous Android applications. One strategy for moving forward is to think about mining the static data provided in Android applications to find attributes that can automatically portray an application's behaviour. Natural Language Processing (NLP) has been applied to mine static as well as dynamic data for the task of Android malware detection by various researchers.

Malware detection is a non-traditional use case for NLP but based on the previous research results, it is safe to say that it does work very well for this task ([3], [4]). The Artificial intelligence models created by researchers consider the contents of malware as textual information. This information is mined for valuable features to train standard and deep machine learning models to differentiate between malicious and benign applications. *Doc2Vec* is one of the techniques from the field of NLP; Mikolov *et al.* designed that to find a way to conserve document-level similarity [5]. It is an unsupervised method that preserves a document's semantic information, and it does so by applying a prediction process to create n-dimensional vectors representing a document. In this study, we look into the viability of using the idea of document similarity to the challenge of identifying Android malware. The background, the research problem, aims, and objectives of the investigation, as well as its significance and limitations, were all covered in this chapter as an introduction to this research study.

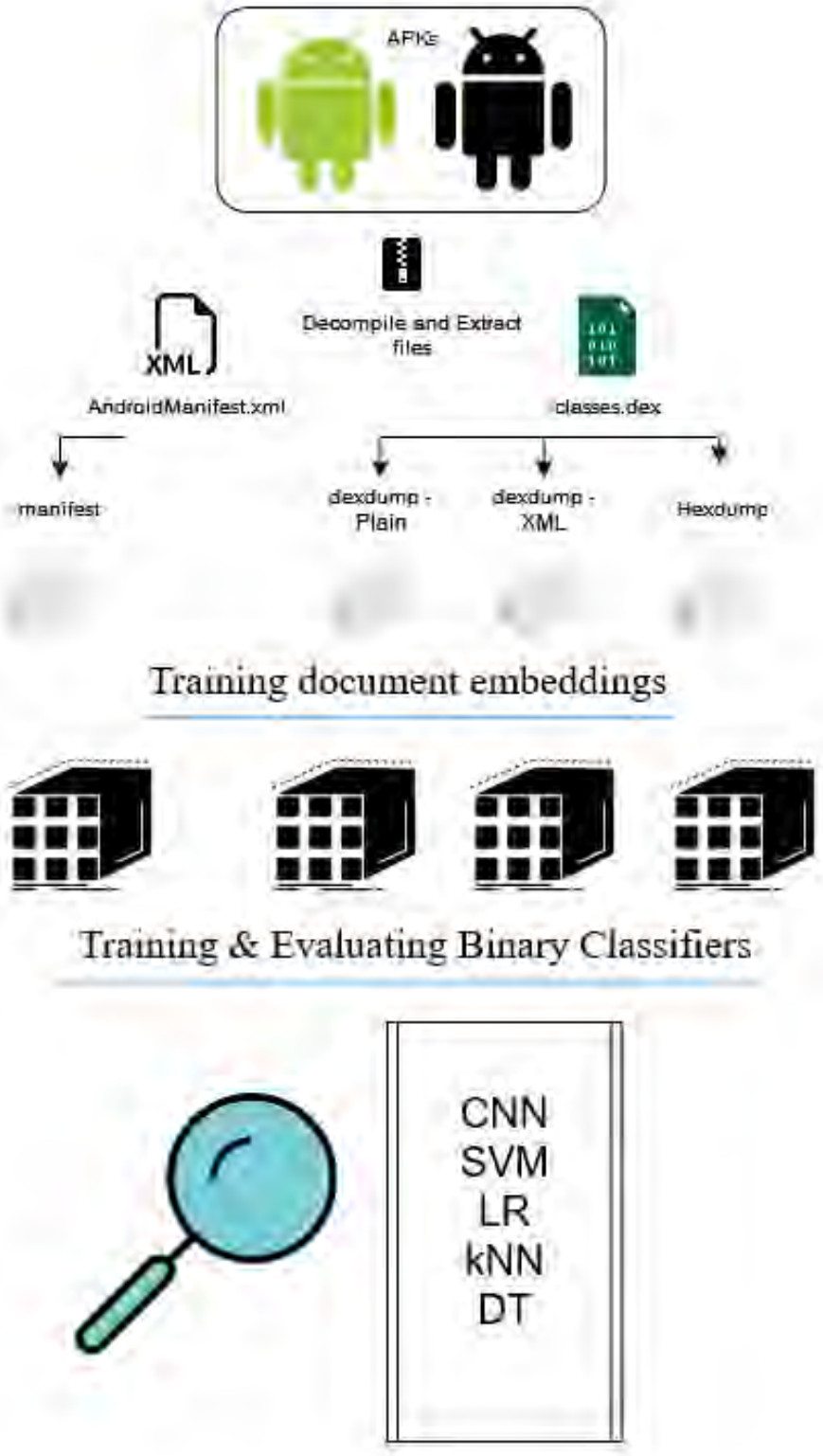


Figure 1.1: Research Overview

1.2 Study Background

An Android application package (APK) is an archive file containing the information required by the Android Operating System (OS) to successfully install and execute an application on the device. This archive file contains the compiled form of the manifest, application code and various other resources needed for an application to run. NLP techniques such as n-grams and Bag of Words have been applied to different files (or parts selected from the files) from the Android APK to generate feature vectors. The earlier research works ([6] [7] [8] [9] [3], [4], [6], [8]) have mined these application files to manually identify and extract features that can represent an application behaviour effectively and also utilised various other techniques, including NLP ones, to vectorise the said features. These features and vectors were, in turn, used for training supervised learning-based Android malware detection models similar to what can be seen from Figure 1.1, which showcases an overview of our research work. We attempted to utilise the files present in the APK, but our primary goal was to remove the need to identify the features manually as those methods are incapable of adapting to the continuously evolving malware. The Figure 1.1, from the top, shows that we first decompile the Android applications to get to *AndroidManifest.xml* and different formats of *classes.dex* files. This is followed by training of document embeddings for each of these files. Once the embeddings are ready, these are used to train commonly used ML and DL classifiers. We proposed a novel method that can automatically generate (or extract) reliable features from the Android application files. We attempted to create document embeddings-based feature vectors using *doc2vec* algorithms for the Android malware detection task.

1.3 Research Problems

As stated earlier, an Android application consists of multiple files. Out of these files *classes.dex* and *AndroidManifest.xml* file have been repeatedly used by researchers for mining features [6] [7] [8] [9]. These files contain useful information representing an application behaviour for training classification models. Different researchers, on numerous occasions, have also applied NLP techniques to extract features to train various classifiers for Android malware detection and Android malware family classification tasks ([3], [4], [6], [8]). Techniques such as the terms document matrix, Bag of Words (BoW), Term Frequency – Inverse Document Frequency (TF-IDF) and n-grams, text mining and word embeddings have been used in many research works for extracting feature vectors. These feature vectors helped create more robust classification models that help detect unknown malwares effectively. However, the NLP-based feature vectors were generated from selected areas of Android APK files which were identified by researchers based on their knowledge about Android applications.

Most of these methods, except n-grams and word embeddings, do not conserve the context in which a word (or, in our case, a piece of code) appears in the document; rather depend on the frequency of the words to create feature vectors. Although such vectorisation methods allow one to compute the feature vectors quickly, they can only capture the presence of words individually and do not capture any positional information, i.e., the order of appearance of code within the document. This type of feature vector leads to difficulty in quantifying the similarity between documents. Additionally, the current methods require the researcher (or the Machine Learning (ML) practitioner) to have some level of knowledge about the malware and the platform the malware is designed to run on to create reliable feature vectors. These vectors are created from the information within malware that a researcher deemed helpful in defining malware behaviour. This process requires manual effort and is time-consuming. We can use the document embeddings approach to automatically generate feature vectors from Android malware files which can also retain the semantic information. Our proposed method will require a minimum of pre-requisite knowledge about an Android application structure and the files present in it.

1.4 Research Objectives and Questions

Given the pre-requisite knowledge and manual identification effort required in feature generation for Android malware detection, this study aimed to investigate the idea that the feature generation process can be automated in a way that it does not compromise on Android malware detection capabilities of machine learning and deep learning models. We did so by investigating the abilities of embeddings created using *doc2vec* algorithms at the task of Android malware detection using some basic static analysis techniques on the Android applications. We used static analysis tools such as apktool [10], dexdump [11], and hexdump [12] to decompile, disassemble and get access to static information from the Android application files. Thus, the objectives of this research project were:

- To investigate the effectiveness of automatically generated features (or *doc2vec* embeddings) at the task of Android malware detection.
- To assess the performance of automatically generated features (or *doc2vec* embeddings) from different types of documents created for static analysis.
- To evaluate the detection models trained on automatically generated features by predicting classes of unseen Android malware embeddings.

Keeping the above research objectives in mind, the main research questions we attempted to answer with this study were as follows:

- Is it possible to treat dexdump of *classes.dex* or *AndroidManifest.xml* file as natural language documents?
- Is it possible to produce feature vectors that accurately depict Android malware and benign software using the *doc2vec* technique?
- How does the malware detection performance of our feature engineering approach compares to existing approaches?

1.5 Research Significance

Our world is interconnected, and our daily lives depend on mobile devices. A malicious Android application can adversely affect the device and, consequently, the life of an unsuspecting Android device user. This research work contributes to the body of knowledge on Android malware detection by attempting to use the idea of document similarity for the malware detection task. This work shall aid the antivirus industry in developing advanced early detection systems that can quickly detect mutated Android malware with similar source codes. This research work shall also interest the research community involved in the general cybersecurity of mobile devices.

As part of this research work, we investigated if the document embedding-based features that were generated automatically could conserve the behavioural characteristics of Android applications well enough to train a binary classification model. We saw that the embeddings of benign and malicious application files tend to cluster away from each other when plotted in a 3-dimensional vector space. The results from our experiments also showed that the features generated from *AndroidManifest.xml* and *classes.dex* files could be utilised as standalone as well as concatenated together to achieve an improved detection performance, but does not require one to manually identify the features.

We have achieved our research objectives and answered the research questions by conducting thorough experiments by applying **Design Science** research methodology. Our experiments were done on research artefacts that were made up of generated document embeddings. The document embeddings were generated by applying *doc2vec* algorithms on the contents of *AndroidManifest.xml* and *classes.dex* files. These artefacts were used to train and evaluate five general binary classification models. The binary classification models had good performance metrics, i.e., as good as existing methods and even better in some cases, consistently during training and evaluation against document embeddings of previously unseen Android applications. Our experiments showed that the automatically generated features could conserve an Android application's behaviour and, thus, can be used for the task of Android

malware detection.

1.6 Study Limitations

The scope of our research work was limited to investigating if the idea of automatically generated features created using *doc2vec* algorithms can be applied to the problem of Android malware detection. We attempted to understand if the document embeddings could conserve enough information from various Android files to train a reliable binary classification model. We did not verify our feature engineering approach against polymorphic or metamorphic malwares. Polymorphic and metamorphic are a type of malware that can rewrite their own source code.

Another important limitation of this study is that we did not verify that the document embeddings conserve the similarity for Android applications directly but instead use the generated features to train binary classifiers. *doc2vec* intrinsically conserves document similarity [13]. However, we also attempted to visualise the high-dimensional document embeddings by plotting them into a 3-dimensional vector space using the Isometric mapping technique to reduce their dimensionality.

1.7 Thesis Outline

The remaining sections of this thesis are organised as follows:

- **Chapter 2:** In this chapter, the existing peer-reviewed literature will be surveyed to understand better how the static analysis of Android malware is done with respect to machine learning and deep learning. This chapter also focuses on understanding how NLP has been used within the context of Android malware detection. While reading this chapter, the reader could find it useful to look over Chapter 4.
- **Chapter 3:** In this chapter, a methodological framework is presented to conduct and report the results of this study. We also justify the adoption of a design science methodology, and more details about the research design, the two datasets used, and the design cycle is presented.
- **Chapter 4:** In this chapter, a background of Android application building blocks has been provided to give a better context on what the Android package files (or APKs) are made up of and what information can be salvaged from them. We describe the process used to

decompile Android APKs and extract the features. We also present visualisations of our document embedding 'artefacts'.

- **Chapter 5:** In this chapter, experiments are conducted by training five different binary classifiers and validating them against unseen Android malware embeddings. The results of the experiments are reviewed and reported in tabular format, as well as bar charts for comparison between the results during training and evaluation. Also, a discussion of the experiment results is presented in respect to our research goals.

- **Chapter 6:** Finally, we briefly summarise the outcomes of this study and highlight our main conclusions in this chapter. We conclude by offering some thoughts and suggesting future study directions.

Chapter 2

Literature Review

2.1 Introduction

In this chapter, our goal was to understand the feature engineering methods used by various Android malware researchers by surveying the peer-reviewed literature on Android malware detection and classification-related tasks. This survey also included applications of Natural Language Processing (NLP) techniques for the Android malware detection task. We attempted to survey the literature systematically by defining the goals and the selection criteria of the literature review. This systematic review was done by defining the keywords used to search the Google Scholar database and our screening and inclusion criteria.

Our high-level goal here is to understand better how feature engineering is performed before Machine learning (ML), and Deep Learning (DL) can be applied to the problem of Android malware detection. We also identify and better understand the application of NLP techniques to a non-linguistic problem, i.e., the domain of malware detection. The critical goals of the Literature Review chapter are, therefore, as follows:

1. To perform and present a critical review of state-of-art feature engineering techniques used in Android Malware detection by various researchers.
2. To perform and present a critical review of works done in malware detection that utilised NLP techniques for feature engineering.
3. To identify research gaps present in the field of Android malware detection with machine learning.

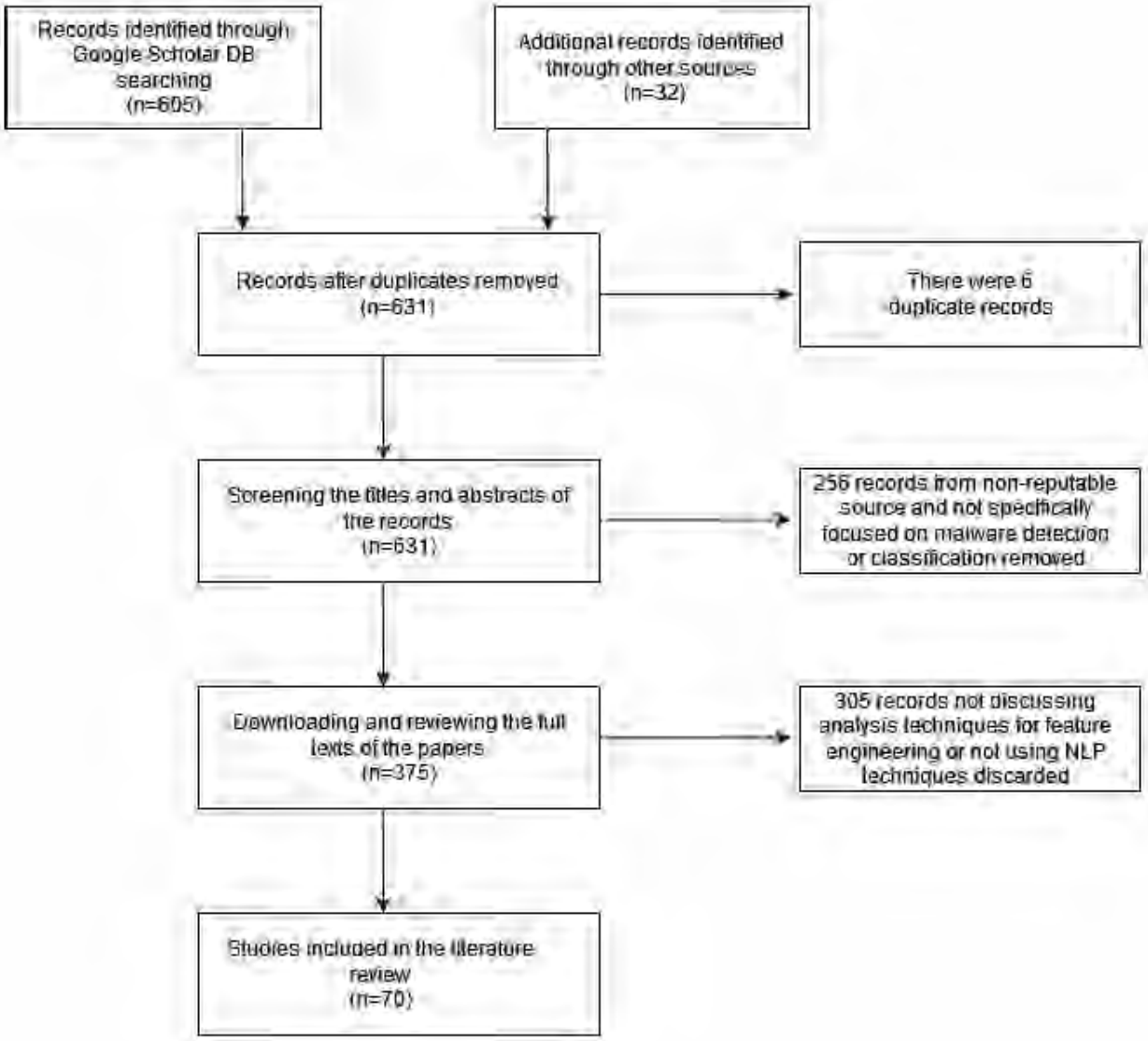


Figure 2.1: Systematic Literature Review

Literature Selection

The papers selected for this literature review were collected from the Google Scholar database using the below query targeting years between 2000 to 2021 with the help of the tool 'Publish or Perish' [14]. This tool allows downloading a list of results that appear in the Google Scholar web application. The search was performed with the following query keywords: *"android malware detection" machine learning natural language processing static analysis document embeddings* which loosely translates to: "android malware detection" **AND** (machine learning **OR** natural language processing **OR** static analysis **OR** document embeddings) in terms of a query.

With these keywords, we found a little over 600 records in the search results. Another 32 articles were selected from secondary sources, i.e., by scouring through references of works that utilised NLP to complement malware detection during the early phases of this research.

Out of the 32 articles, six were also present in the Google scholar database search. We only screened through these search results for the research works that discussed Android malware detection and classification with supervised and unsupervised learning. The literature review flow diagram presented in Figure 2.1 provides an overview of screening and inclusion criteria.

The goal was to present a critical review of the feature engineering applied to Android malware detection. Therefore, the inclusion criteria focused on including the papers discussing their feature engineering approaches in detail, and it also included the paper that utilised NLP techniques to perform malware detection. We selected some papers focused on platforms other than Android but utilised NLP techniques to generate embeddings-like features, and this was done to understand better how the researchers have approached the malware detection research domain to find NLP-based solutions. We primarily used papers written in English or had an English translation readily available. After identifying that *AndroidManifest.xml* and *classes.dex* files have been repeatedly used in many research works, we looked through older papers which have focused on mining these two files to understand better the importance of the features present in these files. Some older works from researchers cited in the latest papers have also been included. However, our primary focus was on papers from 2016 to 2021, as the most recent works give us a better understanding of the state of the art in the field of Android malware detection. The research works from non-reputable sources, not explicitly focusing on malware detection, were promptly discarded during the initial screening. Records which did not discuss their feature engineering approach or were not using any of the NLP techniques were also removed.

In the remainder of the literature review chapter, we begin by describing how Android malware detection is performed with traditional machine learning as well as deep learning. We then discuss how static and researchers have used dynamic analyses to mine Android applications for features. This is followed by an in-depth discussion on applying NLP techniques to malware detection but keeping our focus majorly on Android malware. We then bring this chapter to a close by presenting a critique of the literature, a summary of our findings, and the identified research gaps.

2.2 Android Malware Detection and Machine Learning

Using supervised learning to perform Android malware detection works in a similar way to other supervised learning tasks, i.e., by using labelled instances of features to train a classification model. The feature engineering task for Android applications becomes complex as Android APKs consist of multiple files. Additionally, two different techniques are predominantly in use for malware detection and analysis, which do not utilise machine

learning but are manual approaches used by malware researchers. These are called static and dynamic analyses, and either can be selected depending on the goal, the available resources, and time [15]. These analysis methods can either be used by themselves or together (a hybrid approach) to detect and analyse malware [16]. As the name suggests, Static analysis is a way of evaluating malware without executing it. This type of analysis is done by going through the binary code to find patterns and strings. Static analysis is less resource-intensive and easy to implement when compared to dynamic analysis but can be evaded using techniques such as code-obfuscation. On the other hand, dynamic analysis requires the malware to be executed in a secure environment (also known as a sandbox) while logging and monitoring the malware's behaviour. Dynamic analysis is, therefore, resource-intensive and costlier compared to static analysis [15]. Various researchers have used these analyses techniques to generate features for supervised learning algorithms, including [6], [7] [8], [17], [18], [19] amongst others.

The main inspiration behind the use of supervised learning techniques for the task of malware detection was the limitations in the commonly used technique of hash signature-based malware detection [20]. Hash signature-based detection is majorly used for malware detection by antivirus software packages, whereby databases of known malware hashes are used to detect malware in the user systems. Hashing algorithms like MD5, SHA256 and others help summarise the malware samples into fixed-length string signatures representing the malware. These hashing algorithms are susceptible to even the slightest changes [21]. An additional line of code or comment anywhere in the malware source code will change the signature entirely. This method is, therefore, suitable for detecting known malware. Still, it is no use for unknown malware since this requires software to be analysed by an analyst and marked as malicious. The hash of this detected malware must be added to the antivirus database. Whereas, a document embedding-based approach would be better equipped to detect malware based on the similarities that exist in malware source code.

The most critical information is available at the binary and code level when it comes to computer programs. When these code-level features are vectorised for training ML models, it has seen effective detection of malicious behaviour for even unknown malware [22]. The types of components present in the code vary depending on the application and the platform they have been designed to be run on, for instance, binary code and DLL imports of the Windows executables or Dalvik executable byte-code and permissions in the Android application packages (APKs). Albeit the hash signature-based malware detection methods seem redundant, in 2016, a group of researchers implemented fuzzy hashing techniques to generate Android APKs fingerprints [23]. This technique was combined with different n-grams-based features to capture the application's binary, structure, and semantics. They decompiled the application and applied customised n-grams techniques to the binary and assembly files. They created an APK-DNA consisting of three main application contents: meta-data, binary code, and assembly code. This method achieved a 94% F1-Score in the

peer-matching approach [23], highlighting that the implementations of NLP feature extraction techniques such as n-grams can improve malware detection capabilities of even the signature-based detection system. The application of NLP techniques to the task of Android malware detection has been further elaborated upon in a later section. The following section highlights some of the research work done in the field and is divided into implementations of feature engineering techniques based on static and dynamic analyses features. These features usually have been extracted from the Android APKs after manual identification by the researchers.

2.2.1 Static and Dynamic Analysis for feature extraction

When it comes to Android APKs, researchers have used different files present in the Android application package to extract features that can help a supervised learning model to learn how to classify malicious and healthy applications. A 2016 research paper called 'Framework for malware analysis in Android' [24] extracted the "requested permissions" present in the *AndroidManifest.xml* file, which are static features. The researchers turned them into one-hot encoded feature vectors such that the Android permissions were listed as columns. Then each application was listed as a row, with a value of 1 (if the permission existed) and 0 (if the permission did not exist) in the respective column for the individual application. They trained five different classification models, all of which gave accuracy rates of over 90%.

A research work called 'DRACO: DRoid Analyst COmbo An Android Malware Analysis Framework' [17] extracted various features from the *AndroidManifest.xml* file. These included the requested permissions and other features available in the manifest file, i.e., intents and hardware components used by the application. Researchers have also utilised Android APK as a whole and converted the binary data into greyscale images [25]. The greyscale images are then used to train Convolutional Neural Networks (CNN), which select the most relevant features and predict the application class based on them.

In the research paper 'DroidMat: Android malware detection through manifest and API calls tracing' [18] the features selected to train the classification model were related to permissions requested by the application and the hardware components used, and the activities, services, and receivers components. These were then used to trace down the related API calls present in the Java code of Android applications. These static features extracted from the *AndroidManifest.xml* were then used to train a k-Nearest Neighbours (k-NN) model. Another work that utilised manifest files and API calls for feature engineering was 'Analysis of Bayesian classification-based approaches for Android malware detection' by Suleiman Y Yerima *et al.* [26]. Taking a data-mining approach, they extracted a different set of features from the *AndroidManifest.xml* file and the disassembled dex code. Firstly, they pulled

permissions as features from the manifest file and trained a Bayesian classifier based on these, achieving an accuracy of about 90%. Secondly, they looked for API calls that could be deemed suspicious in the code. These included specific API calls such as SMS activities, accessing call logs, messages, contacts, and changing network states. They also considered looking for cryptographic libraries that are generally used for obfuscating malicious code. The above information was then used as features, and another Bayesian classifier was trained on it, achieving an accuracy of 92%. They combined the feature sets extracted above and trained a third Bayesian classifier with the highest accuracy amongst all three trained models, i.e., 93%. Another recent work on Android malware detection that utilised requested permissions was 'Permission extraction framework for Android malware detection' by Ali Ghasempour *et al.* [27]. The researchers selected the "most significant" permissions in this work using two different statistical approaches. They used their features to train two ML models, SVM and Decision Tree.

One of the most popular research works done on Android malware detection was called 'DREBIN' by researchers at the University of Göttingen [28]. DREBIN project did a comprehensive static analysis and collected as many features from an Android application as possible. Besides the features used in [18] and [26], the researchers involved in DREBIN project also checked for runtime execution of external commands and looked through the code for network addresses, URLs, and domain names. The DREBIN project analysed features from over 120,000 Android applications, out of which only 5,560 were malicious, for training their SVM-based classification model. The model was created so that it could be deployed directly onto a mobile phone running the Android operating system after training. The deployed model detected malicious applications with about 94% accuracy and took an average of 10 seconds on "five popular smartphones" even though the number of features in feature space reached over 500,000 [28].

The research work called ANASTASIA was developed using a larger dataset of multiple Android applications, including the applications from the DREBIN project, which was previously described. This research work focused on the Dalvik byte code and extracted 560 informative features from the application dex files to train nine different machine learning classifiers [29]. The design section of their work highlights what features were selected, but the information about how they were extracted has not been discussed in depth. The extracted features were from the "disassembled code" of the Android application and its components, including intents, system commands and suspicious API calls. The extracted features were refined using an Extra Trees classifier, allowing researchers to evaluate feature value and discard irrelevant features. The researchers tested the developed system on an imbalanced and balanced data sets, which contained applications taken from the DREBIN dataset, which was discussed earlier. The imbalanced dataset consisted of over 18,600 malicious applications and about 11,100 benign applications, whereas the balanced dataset consisted of 11,000 malicious

and benign applications.

In the work of Vasileios Syrris *et al.*, the researchers utilised the high dimensional feature vectors from the raw DREBIN dataset by reducing the high dimensionality [30]. This was achieved by training various machine learning and deep learning models on different features that captured the application behaviour. Their work found that an ideal number of features for Android malware detection based on the DREBIN dataset varied depending on the model they were being tested on. For instance, the Random Forest classifier produced the highest classification accuracy with just 100 odd features, whereas SVM needed about 1000 features. Their research demonstrated that more than a large dimensional feature set is needed to guarantee better performance. Their approaches achieved 99% overall accuracy which was about 5% better when compared to DREBIN's 94% accuracy [28]. They showed that a smaller number of features could perform just as well as the large number of features that were used in DREBIN.

Similar to how existing features from DREBIN project were used in the previously described work, an earlier research work called 'Detecting Android malware using Long Short-term Memory' [31] utilised the feature dataset created by Urcuqui Lopez *et al.* [24] to train different DL models. This dataset consists of the features extracted from the APKs. The researchers used the publicly available dataset to train Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTMs). The dynamic and static features collected in the research work of Urcuqui Lopez *et al.* performed better with LSTM as compared to RNN, which was part of this work. However, the results of [31] were a little better compared to the results of [24] with accuracy and precision reaching 93% whereas recall and F1-score reached 98% and 96% respectively. Although, it should be noted that this work has not been verified against unknown malwares or actual malware, and the researchers here have instead only used a pre-prepared dataset i.e., the features of malware and benignware were directly used instead of extracting them from actual APKs of DREBIN dataset. They showcased the effectiveness of LSTMs on this feature dataset without verifying their work on harvested features from real Android malwares.

Another research work that utilised DREBIN dataset was by Yi Zhang *et al.* named 'A Novel Android Malware Detection Approach Based on Convolutional Neural Network' [19]. Their work implemented CNNs on the features extracted from malware samples of the DREBIN dataset. For the benign applications, they scoured Chinese application markets from April to June 2017. The benign samples were validated against VirusTotal API. Their dataset consisted of over 5500 Android malware and over 5200 benign Android applications. In their work, they extracted five different feature sets: Permissions and Intents from *AndroidManifest.xml* files, and another two from the code disassembly, which were API calls and Strings present in the application. The last feature set was a combination of the above features turned into one-hot

encoded embeddings. They tested these features separately, and the combination/embeddings performed the best out of all the feature sets. They compared CNN model performance against k-NN, SVM and Naive Bayes models. The CNN outperformed all the others and was much faster at detection when compared to the other model's detection time.

The research work by Hongliang Liang *et al.* developed "an end-to-end model for Android malware detection" [32] as their work highlighted that most of the current Android malware detection systems rely heavily on manual efforts and domain knowledge. Their work attempts to solve this problem by implementing a dynamic analysis-based system which collects information on system calls for Android applications by running them on an Android virtual device and logging the application processes. A CNN was then trained on these system call features. The researchers tested the CNN model by using these features as one-hot encoded vectors and by generating feature embeddings, similar to a neural probabilistic model based on the work of Yoshua Bengio *et al.* [33]. The classification accuracy and F-measure score were higher when the model was trained on embeddings instead of one-hot encoded vectors. To give a fair assessment of their work against others, the researchers also applied methods used by Canfora *et al.* [6] on their dataset, which consisted of 10000 and 4231 benign and malware applications, respectively. Please note that in the work of Canfora *et al.*, the malwares were collected from the DREBIN dataset like many other research works. Another work that utilised the neural probabilistic model of Yoshua Bengio *et al.* was 'A transparent and multimodal malware detection method for Android apps' where the researchers extracted permissions, APIs and URL features from *classes.dex* and *AndroidManifest.xml* to generate word embedding based features for their detection models [34].

For detecting obfuscated malware, hybrid features, i.e., a combination of static and dynamic features, have also been used to train deep learning models for Android malware detection. One such work utilised static features extracted from resource files as well as API calls and the requested permissions. The dynamic features included the application's activity upon execution, together with network activity, access to sensitive information and geographic location, amongst others. In research work by Tianliang Lu *et al.*, the static feature were one-hot encoded and inputted into their Deep Belief Network (DBN), and on the other hand, dynamic features were converted into a vector by passing them through a recurrent neural network [35]. This vector was then input to Gate Recurrent Unit (GRU). The output of the DBN and GRU were then passed through a back-propagating neural network for classification.

Such hybrid features have been used earlier in a research work called 'DroidDetector', and the researchers also utilised static and dynamic analysis techniques to collect application features [36]. The researchers evaluated their features against a deep belief network architecture model with several layers, each layer containing several neurons. They extracted a total of 192 features from their applications, including both dynamic and static

features. From these features, 120 were taken from permissions and 59 from sensitive API calls, which were collected statically. Only the remaining 13 features were collected dynamically by monitoring the application actions. The applications were installed to run on *DroidBox* [37], which allows taint analysis and monitoring of application actions. They also used a heavily skewed dataset with about 20000 benign Android applications compared to a mere 1760 malicious Android applications. The researchers tested their features against ten different deep belief models with various combinations of several layers and neurons per layer. The features extracted from these applications performed the best when they were passed through a double-layered neural network with 150 neurons in each of the two layers.

There was one research work in which the researchers did not use machine learning or deep learning methods. Instead, they hypothesised that most malicious applications are created by very few malicious individuals and using information about the creators of applications as a feature will allow an improved detection rate. This information is found in the application certificate of an Android application, with which a developer is required to sign their application. The system the researchers designed used a three-fold approach i.e., the first part of the system would read the files present inside an Android application and extract useful information from the files, including certificates, permissions and API calls. Secondly, the system checks the certificate serial number against a pre-defined blacklist of serial numbers of known malicious actors. Finally, if the serial number is not present in the blacklist, the system then assesses other information that was extracted from the application initially. It looks for malicious commands and API calls, followed by checking for malicious permissions asked by the application [38].

Summary

During our review of static and dynamic analysis techniques for feature engineering for Android malware detection and classification, we saw that the researchers had mined static features from *AndroidManifest.xml* and *classes.dex* repeatedly. They have utilised all possible resources from the Android APK based on their domain expertise. We also see different ways where researchers have mined dynamic features using API calls, taint analysis, as well as network traffic monitoring. We also see systems that were designed using pre-defined malicious signatures that the systems look for, such as some malicious commands, meta-data or even application signing certificates. There were some other research works that are worth mentioning but could not be included in the main body of this section. One such research work looked into the 'assets' folder for APK files [39] as some malicious actors place obfuscated applications in the 'assets' folder and install them at runtime. Researchers have also utilised opcode sequences from *classes.dex* in multiple research works such as [40] and [41] to train their respective machine and deep learning classifiers. Researchers also have ensembled

permissions, features, and other such features with function call graphs to achieve better classification results [42].

From this section of our literature review, we saw that the general feature extraction and engineering techniques based on static and dynamic analyses gave good classification results but required some degree of pre-requisite knowledge about the platform and some manual effort to identify and extract these features. We now shift our focus towards the application of NLP techniques to malware detection and present a comprehensive review of the surveyed works.

2.3 NLP Techniques in Malware detection

The research works mentioned above on Android malware detection extracted features vectors as-is from the decompiled APKs. Other than the simple code-based feature vectors, research work has been done on Windows and Android malware detection using feature vectors created by various NLP techniques. This section describes the previous works that we encountered during the literature review which utilised NLP techniques to generate feature vectors for training various malware detection models using artificial intelligence. In 2010, Peter Teufl and his team [4] argued that NLP methods could be modified and applied to the area of malware detection and analysis. As part of their work, they provided a framework to adopt NLP techniques, with a slight modification, to Machine Language Processing. Although the idea seemed promising, it never caught up as one would have expected. This was probably because the NLP techniques which were being implemented to detect patterns in the text files and create feature sets were working effectively on the malware detection without the discussed modifications. NLP techniques such as n-grams, Bag of Words and others were being applied to static operation codes present in application binaries as well as to dynamic features of API calls to captured network traffic. The rest of this section, we present the research works that have utilised notable NLP techniques for the task of malware detection in general.

2.3.1 n-grams

One of the first works which implemented the NLP technique of n-grams to create Windows file signatures was done in 2009 by Igor Santos, and colleagues [3] in their paper ‘n-grams based file signatures for Malware detection’. In their research work, they used a dataset of 1000 malware and 1000 benign Windows software binaries randomly selected from a much larger dataset. These executable files were then used to create n-grams with different n values. They

successfully demonstrated that their n-gram-based features could be used to train ML models which can detect even unknown malware effectively.

Igor Santos applied n-gram technique to Windows malware detection. In a research work from 2015 n-gram techniques were applied to Android malware detection. The research work was called, 'Effectiveness of opcode n-grams for detection of multi-family Android malware'. The researchers, Gerardo Canfora and colleagues, investigated if n-gram-based feature vectors could be effectively utilised for Android malware detection problem [6]. They extracted n-grams of the opcode sequences (i.e., equivalent of machine-level instructions for Android architecture) from 5560 benign and 5560 malicious Android applications. The usage of n-grams-based features from the opcode sequences helped them achieve about 97% classification accuracy when detecting Android malware with their Random Forest classification model. ROCKY framework by Roni Mateless *et al.* decompiled and utilised Android application source code to extract "API calls, keywords and other non-obfuscated tokens" which were then categorised into three different sets based on the rules defined by them: stop-tokens, feature-tokens and long-tail tokens. After this, they extracted "generalised N-tokens" which could depict the sequences of the source code. These tokens were then vectorised using different Bag of Words representations, including TF-IDF. They then used these features to perform experiments for Android Malware detection, and family classification [43].

Similar to [6], the research work 'Deep Android Malware Detection' of 2017 used deep CNN to analyse raw opcode sequences [7]. Their model allowed the use of n-gram-like features for modelling without training or extracting n-grams from the malware samples. This method removed the need for human-engineered features for malware detection. It could be directly used to detect malware on other platforms by updating the disassembly process, depending on the platform malware was made for, and retraining the deep CNN model. This research work saw a higher accuracy on the test dataset of almost 98%, but when it was tested on an independent Android malware dataset, the accuracy fell to about 69%.

Another research work that utilised NLP and text mining techniques on Windows malware (specifically Ransomware) was this work called 'A Multi-Level Ransomware Detection Framework using Natural Language Processing and Machine Learning'. The researchers generated n-grams (and TF-IDF) feature vectors to test the effectiveness of these NLP techniques in ransomware detection [44]. They applied the NLP techniques to the corpus data extracted from Windows Dynamically Loaded Libraries (commonly known as DLLs), function calls and assembly instructions. The researchers also noted that higher N values did not necessarily guarantee a better classification model performance. The number of features also varied with the changing N value. Their results show that n-gram features alone were not very capable of representing the ransomware characteristics and did not perform as well

compared to the TF-IDF features. The performance metrics presented in the paper show that with n-gram alone, the highest accuracy achieved by their Logistic Regression was about 90%-93% with different *N* values. On the other hand, with TF-IDF version of the n-gram feature vectors reached 97%-98% of classification accuracy on their dataset.

2.3.2 Bag Of Words and TF-IDF (Term Frequency - Inverse Document Frequency)

The research works discussed in the previous section showcased that the n-grams technique can be used to improve malware detection performance. TF-IDF helps improve it further by giving weights (or importance) to different n-grams based on their frequency of appearance within the corpus. It is also evident from the work of Subash Poudyal *et al.* that the classification models can perform better when given TF-IDF weighted feature vectors [44] instead of plain n-grams feature vectors. In the research work called 'ADROIT: Android malware detection using meta-information', researchers applied a text-mining approach to the meta-information available on the application store and *AndroidManifest.xml* to generate the feature vectors. The meta-information mined included "minimum Android SDK and OpenGL versions needed to run the application, the minimum size of the screen and the supported CPU architecture types of the Android device, the developer signatures or an identification, their organisation and the country or region, and a description of application written by the developer" [8]. They created a term-document matrix after pre-processing the meta-information that was mined in the previous step. This term-document matrix was then normalised using the TF-IDF technique. They also discovered that the permissions requested in the manifest by the malicious application could sometime differ from the permissions listed on the app stores. The resulting model from these meta-information-based features had an accuracy of about 94% on their Random Forest classifier, which was still much lesser than the work in 'Effectiveness of opcode n-grams for detection of multi-family Android malware'.

In previous works, researchers have used TF-IDF on n-grams and even application metadata. However, there are other ways to detect malicious behaviours of an application statically. In one work by Shahid Alam called 'Applying Natural Language Processing for detecting malicious patterns in Android applications', the researchers created an "intermediate representation" [9] by extending on one of his previous works by the same researcher and his colleagues [45] which created human-readable interpretations of binary opcode sequences called MAIL (Malware Analysis Intermediate Language). These "intermediate representations" are used to generate control flow graphs which are later tokenised into a Bag of Words corpus to build a dictionary further. TF-IDF model is trained based on this dictionary and then later used to calculate a similarity index by utilising cosine similarity. This research

work did not utilise any supervised learning techniques in their classification process but instead relied on a similarity index by deciding upon a similarity threshold which provided the best results. Their paper states that they performed experiments with threshold values between 5 to 50% and achieved the best results by keeping 25% similarities as a threshold.

A bag of words is one of the most basic methods of feature creation from the text, and it is usually used as the first step before any other actions are performed on text in NLP-related tasks. For instance, the last work discussed in the previous section utilised a similarity index for Android malware detection. The research work applied TF-IDF on control flow graphs extracted from transitioned opcodes. Nevertheless, before TF-IDF could be applied to these control flow graphs, the graphs were broken down into a bag of words first [9]. One of the considerable research works done in the field of malware detection done by Karbab *et al.* was called *MalDy*. *MalDy* was created to work upon "behavioural analysis reports" by applying NLP techniques to them [46]. This work showcased that it was possible to use a word-based NLP model to perform the task of malware detection and even family classification for both Android and Windows malware. In their work, they generated these behavioural reports by executing the dataset malwares in a sandbox environment. The behavioural reports were then collected and converted into a Bag of Words. This was followed by n-gram feature generation from the Bag of Words of behavioural reports. These n-gram features were treated in two different ways to generate two different types of improved feature sets by using Feature Hashing and TF-IDF. These improved features were then used for the task of malware detection as well as malware family attribution by using six different Machine learning models, including k-NN, SVM, Random Forest and XGBoost. They evaluated their feature engineering approach on various datasets, which included Android malwares from DREBIN, MalGenome and MalDozer, achieving over 94% F1-score in the task of malware detection with their XGBoost model.

Bag of Words model was used in a research work by Kartik Khariwal *et al.*, which utilised only *AndroidManifest.xml* file and extracted intents and permissions from the file. The extracted permissions and intents were converted into two separate Bag of Words. Feature ranking based on *information gain* was applied to the Bag of Words features [47]. The ranked features were then used to train three different ML models, Random Forest, Naive Bayes and SVM. The best Android malware detection Accuracy they achieved with their Random Forest model was 94.7%. In the research work of Weina Niu *et al.*, NLP techniques of Bag of words as well as word embeddings were utilised. The researchers mined the application source code for "ordinary function calls" and converted them into Bag of words features. They also used "sensitive function calls" at the same time [48]. They also mined for opcode-based sequences of the above two, i.e., ordinary and sensitive function calls. They treated these as four different feature sets and trained LSTM and SVM models with them. Their different feature sets on both models gave decent results, with the highest metrics being 97% Recall, Precision and F1-score, which was from the LSTM trained on opcode sequences of ordinary functions.

Other than opcodes, behavioural reports and control flow graphs being modelled as Bag of Words features, researchers have only used requested permissions from *AndroidManifest.xml* and modelled them as Bag of Words features [49].

2.3.3 Word Vectors

This section of the literature review describes how various research works have utilised word embeddings for their feature engineering needed for the task of malware detection in general as well as on Android malware detection. The research work 'Lightweight versus obfuscation-resilient malware detection in Android applications' [50] utilised the *word2vec* feature engineering technique of NLP in one part of their work. The authors implemented two different methods to create a model that could effectively detect Android malware. The first method in this research work used *word2vec*, which was first suggested in the research work called 'Efficient Estimation of Word Representations in Vector Space' [5] to learn multidimensional word embeddings, which allowed them to store semantic information and could allow evaluation of word similarity by simple mathematical computations. The authors of [50] created word embeddings of native binaries present in APKs based on the word embeddings approach of [5] and by comparing cosine similarity, they were able to detect obfuscated malicious code as well. In the second method, the authors of [50] used the first 1000 bytes from *classes.dex* file and used them as greyscale image pixels. These greyscale pixels combined with XML features from Android Manifest helped them create a lightweight model which could be run on mobile phones directly.

Another one of the works by Karbab *et al.* was called *PetaDroid*, which used reverse engineered Dalvik executable file assembly code for feature engineering [51]. From the disassembled Dalvik files, the researchers manually extracted "Method invocation, object manipulation and field access". According to the researchers, the data they extracted from the Dalvik executables represented the fundamentals of application behaviour. The extracted data (or instructions) was vectorised with an approach which researchers called Inst2Vec. This is an adaptation of *word2vec* done on the instructions extracted from the Dalvik assembly code. These vectorised Inst2Vec features were used to train an ensemble of CNN models. The researchers also verified their model and feature performance against obfuscated Android malwares by creating obfuscated samples using *DroidChameleon* [52]. The features captured the malicious behaviour of Android applications effectively. Their model was successful in classifying even the obfuscated Android malwares. They also used these features for Android malware family attribution and achieved acceptable results. In a follow-up to their work *PetaDroid*, Elmouatez Karbab *et al.*[53] also discussed about achieving "automatic feature engineering". They also took advantage of embeddings generated using Android API calls

from the disassembled Dalvik code in their work called 'MalDozer' [54]. Both of these works are also part of their recent book on Android malware detection called 'Android Malware Detection using Machine Learning' [55].

CoDroid was a research work in which the researchers engineered their features by treating "sequences of static opcode and dynamic system call" as a natural language sentence [56]. This was done by collecting data by performing both static and dynamic analyses on the Android application. The application's *classes.dex* files were decompiled to '.smali' files as well as executed and monitored using strace [57]. These sequences were then turned into word2vec like embeddings in the first layer of their CNN-BiLSTM-Attention model. Their dataset consisted of 2978 malicious and 2707 benign Android applications. Although with their method, they achieved over 95% F1-score and Precision for both benign and malicious classes, they did not specify if their methods were verified against previously unseen Android applications. Researchers have also extracted simple API call sequences by disassembling Dalvik executables into smali files and generated *word2vec* embeddings features to create various Android malware detection models [58]. Some researchers have also utilised URLs and permissions in addition to API calls and opcodes from disassembled Android files and generated word embeddings to train their malware detection models [59].

One research work by Peng Xu *et al.*, worked on Android malware detection and family classification [60]. They achieved their goal by first using Android APK as input to generate function call graphs where functions within an application and their respective Dalvik opcode blocks are gathered. Their work used the opcode to generate the initial embeddings; this was followed by the generation of function embeddings by using the opcode embeddings. The function embeddings were then converted into a final graph embedding which could be utilised by their Multi Layer Perceptron (MLP) model. They utilised 4 significant datasets, including AndroZoo and DREBIN, to create their own slightly skewed dataset, which was made up of 90,313 benign and 45,592 malicious Android applications. In another work by Peng Xu called Android-COCO [61], the researchers used an ensemble approach where they utilised the predictions of two separate MLP classifiers. The two MLP classifiers were trained for Android malware detection on Byte code and Native code present in an application. For the byte-code sub-system, they used Program Dependence Graph (PDG). This was a complicated 4-step process, where initially the opcode embeddings are created, which are then used to create 'basic block' embeddings which represent a function node. After this, the program dependence graph is created, followed by generating its embeddings. These embeddings are then used to train a Multi-Layer perceptron. The second sub-system utilised native code for Android malware detection. This system utilised a functional call graph extracted from the Android applications. These graphs were then converted into functional call graph embeddings. These function call-graph embeddings were used to train another MLP. The output of the two sub-systems was combined through an ensemble algorithm.

One more work by Peng Xu *et al.*was *Hawkeye* which utilised *word2vec* based embeddings of control flow graph to develop a system that could perform malware detection across platforms [62]. The developed system was made up of three components. The first component was a control flow graph generator that can create both static and dynamic control flow graphs from binary files for different platforms. These control flow graphs were then used to generate embeddings and detect malware using MLP in a similar way to their previous works. Another research work co-authored by Peng Xu called *Hybroid* which utilised NLP-inspired embeddings generated from static features of opcodes, control flow graphs, and basic blocks in combination with dynamic features extracted from network traffic generated by each application [63]. Using the hybrid features, they achieved an accuracy of 97% during the Android malware detection task. *hybrid-Falcon* research work, also by Peng Xu *et al.*used similar features to *Hybroid* but handled network traffic as a sequence of 2-dimensional image [64].

A research work called 'Android malware detection via an app similarity graph' utilised *node2vec* [65] which is an algorithm motivated by *word2vec*. In this research work, the functions were extracted from the Android application source code by decompiling the Dalvik byte code. All of the defined functions were extracted and saved. Application similarity graphs (ASG) were an idea borrowed from the recommender systems. These graphs were created by processing extracted functions. ASGs are then processed by *node2vec*, after which the generated vectors were then decomposed into "lower-dimensional representations" by using Singular Value Decomposition. The low-dimensional representations were then used as features for a Random Forest classification model [66]. *Node2vec* has also been utilised by an earlier work called 'Deep learning for effective Android malware detection using API call graph embeddings', where the researchers used *Node2vec* to generate embeddings from API call graphs which are then utilised for training Deep neural networks [67]. A research work called *byte2vec* used compiled form of *AndroidManifest.xml* and *classes.dex* without any disassembly or reverse engineering [68] and utilising the concept of *mutual information*. The researchers used methods similar to the *word2vec* skip-gram algorithm with negative sampling to generate the feature vectors from the files and presented the malware detection results of their methods on DREBIN, DexShare and AMD datasets. *ByteDroid* was another such work where compiled form of *classes.dex* file was utilised to generate word embeddings for the task of Android malware detection [69].

TC-Droid, "an automatic framework for Android malware detection" created by Nan Zhang *et al.* tried to overcome the limitations of manual feature engineering [70]. The researchers utilised AndroPyTool which is used to generate analysis reports that contain information related to both dynamic and static analysis of applications. They extracted static features of "Permissions, Services, Intents and Receivers" from these reports. These features were then fed into a TextCNN model, which was a 5-layer CNN, with the first layer handling

the creation of embeddings from the static features extracted from the AndroPyTool reports. Their proposed TextCNN model had a 96.6% accuracy and F1-score, 94.6% precision and 98.4% recall against the DREBIN dataset. Another study that focused on automatic Android malware detection was 'Multi-view deep learning for zero-day Android malware detection'. This work utilised one-hot encoded opcode and API sequences as input to two different CNNs to automatically select the critical features. They also utilised the one-hot encoded vectors of permissions that were passed through a 'fully-connected neural network' [71]. Researchers have also utilised word embeddings of 'textual description', which is available with applications on application stores, in attempts to link them with permissions requested by an application from within the code and use it to depict an application behaviour [72]. In addition to the works discussed in this section on Android malware detection that utilise *word2vec* to engineer their specific features, the features used in the different works vary quite a lot. From opcodes extracted from smali files [73], dex files [74] to interprocedural control flow graphs [75] and a few others like [76], [77] have been used to generate word embeddings that can be used to train Android malware detection and classification models.

2.3.4 Document Vectors

The NLP method for creating document embeddings known as paragraph2vector (or doc2vec) is described in the research work called 'Distributed Representations of Sentences and Documents' by Tomas Mikolov and Quoc Le [13] based on Tomas Mikolov's previous work on word2vec [5] allows the creation of document embeddings for various NLP tasks, including document classification and is a significant development in the field of NLP in terms of word similarity related tasks. The paragraph2vector algorithms learn "fixed-length feature vectors from variable-length" [13] input texts, reducing the computational complexity in the process meanwhile conserving the document-level semantics. The paper on document vectors highlights two different algorithms, namely paragraph vector – distributed memory(PV-DM) and paragraph vector - distributed Bag of words(PB-DBOW), to learn document embeddings. The algorithms mentioned above use the prediction task to train the document embeddings, which are declared randomly at first and updated constantly using stochastic gradient descent by back-propagating the loss until the accuracy of the prediction task is enhanced. This technique has been used to engineer the features for Windows malware detection in some research papers but has been mostly overlooked when it comes to Android malware detection.

The research works that first proposed use of paragraph vectors for malware classification task, to best of our knowledge, are presented in the remaining section. One of them was called 'NLP-based Approaches for Malware Classification from API Sequences'. It implemented paragraph2vec on API call sequences, which were collected using dynamic

analysis of different Windows malware and benignware [78]. The authors implemented three different NLP methods, 2 of which were based on paragraph2vec-based algorithms [13], to successfully classify malware into different malware families. Another research work by researchers at the Tokyo University of Technology, 'Static Analysis with Paragraph Vector for Malware Detection' in 2017, extracted static features using paragraph2vec from DLL imports, assembly code, and hex dumps [79]. The results reiterated the effectiveness of the *paragraph2vec* algorithm on classification. The researchers tested two different classification models, linear SVM and k-Nearest Neighbours, both of which achieved an accuracy of about 99% with equivalent precision and F-score.

In 2019, Ryo Ito and Mamoru Mimura from the National Defense Academy of Japan evaluated paragraph2vec on ASCII strings extracted from Windows malware. In their research paper, 'Detecting unknown malware from ASCII strings with natural language processing techniques', they extracted ASCII strings present in the malware and benignware executable statically. These strings were then utilised to create different language models using LSI and paragraph2vec. The SVM Classifier trained on these language models had an accuracy of about 95% using paragraph2vec features [80]. During our literature review, the only paper we came across that utilised paragraph2vec or *doc2vec* on Android applications was 'Deep Neural Networks for Android Malware Detection', where researchers used *classes.dex* file as it is to generate 300-dimensional embeddings [81]. They trained four different detection models that achieved 94.4%, 95.1%, 93.7% and 95.3% accuracy, respectively.

Summary

When we reviewed the application of NLP on the malware detection task, we saw that a lot of research works have also kept their focus majorly on the features that can be either extracted statically from *AndroidManifest.xml* and *classes.dex* files in addition to dynamic features such a network traffic, API calls and call graphs generated from API calls and function calls amongst others. We saw that researchers have applied NLP techniques of n-grams, Bag of Words, TF-IDF, Word vectors and Document vectors to static analysis and behavioural reports, opcode sequences, meta-information as well as Dalvik byte-code. In this section of our literature review, we saw NLP techniques have been applied to features extracted using static and dynamic analyses alike. We also some complex ways used by researchers to extract features and then turn them into word and document embeddings which required them some pre-requisite knowledge as well as some degree of domain expertise. In the last two sections, we have tried to understand the feature generation and engineering approaches taken by the researchers working in the field of Android malware detection and how the NLP techniques have been applied to non-linguistic problems. We frequently saw research works with very high detection accuracies (up to 90%), but manual selection approaches were used to create the

features used for training the models. In the next section, we present a critique of the literature that was surveyed as part of this research work.

2.4 Critique

To present a critique of the literature, we first need a simple overview of an Android application. The Android applications are delivered as an archive packaged file, and the package consists of multiple files including *classes.dex* (compiled file Dalvik byte code), resources and the *AndroidManifest.xml*. These files contains essential information for the Android operating system to install and execute an application on a mobile device. Therefore, to evaluate APKs and classify them as malware or a benign app using supervised learning, there are multiple steps involved, including decompiling the APK and then selecting the best files for generating feature vectors. In some cases, researchers have also mined technical blogs on Android malware detection in addition to Android APK and utilised complex methods to correlate both such that malicious and benign behaviour can be defined better [82]. This literature review outlined the research done by various research teams in the past. It focused on research works surrounding Android malware detection and on feature engineering approaches taken by various different researchers. Most of the research works that were reviewed as part of our study showed that the Dalvik executable and Android Manifest files had been used numerous times over the years by multiple researchers. The results of their research work also show that these files contain much helpful information regarding Android applications' behaviour. These files have been used, and different types of details have been extracted from them by various researchers.

The review also showed that other than the Dalvik executable and Android manifest files, a few other information sources from within the Android applications have been used to generate features for the malware detection task. The most notable amongst these were developer signature certificates [38], behavioural reports [46], and meta information [8]. The features for supervised learning models have been engineered from opcode sequences, application meta-data, decompiled Dalvik executables, suspicious API calls, suspicious strings, sensitive API calls and permissions and even malicious URLs present in the source code. Extensive research has been done on graph-like features, including API call graphs, control flow graphs and function call graphs to name a few. This literature review reveals that the creation of features for malware detection generally, if not always, required manual intervention and effort to identify the most reliable features. This requires the researchers or the users to have at least preliminary knowledge of how malwares for specific platforms operate. They even need to have some degree of understanding of the platform itself to perform feature selection.

This review also showed that most of the research works are still based on old datasets, even in the latest research. DREBIN dataset being the most notable one contains applications that are a decade old, with applications dates ranging from 2010 to 2012. Android has come up with multiple versions of APIs and security measures ever since. Moreover, most of the research works encountered during the literature review process show that these have not been tested against unknown malware datasets to validate their effectiveness in real-time Android malware detection. Researchers have mined the Android APKs to gather details from different components and files present in them. Natural Language Processing techniques have also been implemented by researchers on the features extracted from the APK files. In the 2015 work [6], researchers applied n-grams to DREBIN dataset and achieved 96.8% detection Accuracy whereas the DREBIN project had achieved 93.9% [28]. DeepClassifyDroid [19], achieved 97.4% for both Accuracy and F1-score with 98.3% Recall and 96.6% Precision values. Though these methods provide satisfactory results, they require some degree of knowledge about the Android platform and applications. With the advent of automated methods and technologies, it is possible to improve this process and make it streamlined.

The NLP techniques of n-grams and TF-IDF, which were implemented in the research works mentioned in previous sections just store the information present in the documents as independent one-hot encoded vectors. This makes it difficult to find similarities between documents effectively. This weakness can also be improved upon by using document embeddings (document vectors or *doc2vec*), an improvement of the word embeddings (word vectors or word2vec) which allows the creation of vectors that can be plotted in multidimensional space storing the semantics of the pieces of code. The document vectors of similar documents are placed closer to each other in the multidimensional vector space as compared to documents with different contents, thereby increasing the efficiency of the classification tasks.

2.4.1 Research Gaps

An overall trend, post decompilation of APK files, can be seen in the literature review. The researcher looks for specific features that can be mined from within these decompiled files. The features that represent an application behaviour most effectively are selected for further improvements or training in various supervised learning models including Machine learning and Deep learning. However, it can be understood that some prior knowledge about the malware is required to create these features. One should also understand how they interact with the Operating system the malware are targetting. This process can be improved upon by trying to automatically extract the features which can be achieved by using the document embeddings technique from the NLP field. The *doc2vec* approach utilises deep learning to generate feature

vectors from input documents, paragraphs and even sentences. The results from [79] and [80], where experiments were conducted on Windows executables using *doc2vec* features, reiterate that these feature vectors can provide promising results. But the NLP methods of document classification worked simply with Windows malware executables by treating the content of executables as text (since they are generally standalone executable files) which is not simply the case with Android malware. Android applications are archive files and, therefore, need to undergo preliminary steps of unpacking and decompilation. It was also highlighted in the literature survey work called 'Deep Learning for Android Malware Defenses' [83] that, unlike image or text classification, the application of embeddings to Android APKs is complicated as the APKs are archive files made up of different types of file which are of different formats. The literature review has repeatedly shown that the Dalvik executable and Android manifests have information that can prove critical to understanding an application's behaviour and would therefore be the primary subject of this investigation. The review also reveals that there is a lack of up-to-date databases of benign and malicious Android APKs that are already benchmarked and publicly available. Various outdated datasets containing applications supporting only older versions of the Android operating system are still being used, even for recent research works.

2.5 Summary

This literature survey focused on feature engineering approaches applied for Android malware detection. From the survey, we understood that Android applications are archives made up of multiple files in different formats, making the feature engineering process very complex. Much research has gone into identifying the features that best represent an application's behaviour. Researchers have also applied NLP techniques to vectorise and further improve the quality of features they identify. The said features have been identified and extracted from static (Dalvik executable, opcode sequences, manifest file etc.) and dynamic (network traffic, API call graphs, control flow etc.) analyses of the application itself. The review also showed that for the static analysis approaches, *classes.dex* and *AndroidManifest.xml* had been repeatedly exploited to find features that can effectively represent an application behaviour. It was also noticed that there is no automated approach to finding these features. Since Android applications are complex and made up of multiple different files, finding features requires, at the very least preliminary knowledge about how the Android platform works and how Android applications function. The literature review showed that primarily *classes.dex* and *AndroidManifest.xml* had been the files of interest in many research works and the features are generally hand-crafted in nature. The complexity of finding reliable features opens an opportunity for us to design a system that takes these files of interest directly as model input and give a set of feature vectors as output. This system will remove the need to feature mining the Android APK files manually. To accomplish

this, we propose to create document embeddings-based features and investigate their abilities at the task of Android malware detection as part of this research work. We would also need to evaluate all of the "feature embeddings" on existing datasets and our dataset, which consists of applications from DREBIN and AndroZoo datasets. More details about the application datasets, how the "feature embeddings" are generated from Android APKs and their evaluation process are described in Chapters 4 and 5.

Chapter 3

Methodology & Methods

3.1 Introduction

The methodology is a framework of "principles, practices and procedures" dealing with the process of creation of new knowledge [84]. Using qualitative and quantitative research methods is quite common in studies surrounding the fields of natural sciences and social sciences alike. The goal of the studies in these fields is to better understand the world around us; thus, the qualitative and quantitative methods fit very well. However, our study aims to utilise different files present inside an Android application package (APK) and automatically create feature vectors from them. Therefore, the goal of our study is to create the feature vectors from different files and evaluate the feature vectors for the task of Android malware detection. This goal will be achieved by iteratively creating 'artefacts' from different files and training binary classifiers on them. We also want to evaluate the trained binary classifiers by using them to predict malicious and benign classes for previously unseen Android application embeddings. As this research aims to design an artefact that reduces (or removes) manual feature identification and creation effort for Android malware detection, the search for artefact would require multiple iterations to identify the 'right' artefact that can realistically achieve the aim without losing on the malware detection capabilities. Therefore, the methodology of **Design Science** is best suited for this work as this methodology outlines a framework for the creation of artefacts "that serve human purposes" [85]. This chapter describes the application of Design science methodology to our research problem.

Design science methodology in itself is an iterative process to "design and investigate artefacts" in a specific set of available knowledge, existing artefacts and their specific limitations [86]. The research problems in design science methodology are generally enhancement or refinement problems — every one of the research problems targets enhancing

the existing knowledge by creating new or better artefacts. During the research cycle, knowledge about artefact limitations is also gathered. Based on this background knowledge about the design science methodology, we can clearly state that the artefact designed as part of this research work will be a 'document embedding-based feature engineering approach that automatically selects a good set of features representing an Android application behaviour (malicious/benign) using an unsupervised shallow neural network'. Furthermore, the context of this design science research is to remove the need for manual feature identification and creation, which is generally applied to the task of Android malware detection but requires a certain degree of knowledge about the platform and the malware.

The rest of this chapter is split into two main sections. The first section is devoted to the Research Design, where we define an overall Research Problem by dividing it into a set of **Design Problem** and **Knowledge Questions**. We then present the **Knowledge Context** where the existing knowledge, i.e., *document embeddings* and its relevant algorithms (PV-DBoW and PV-DM) that are used to generate the artefacts for this research study are briefly explained. The later part of this chapter focuses on the **Design Cycle**, where we define the problem-solving methods we apply to find solutions to our design problems and answer the knowledge questions. We define the artefacts created for this research work and the methods applied to design our solution (or artefact) and validating the solution's malware detection abilities. We discuss the two Android application datasets - AndroZoo and DREBIN; that were used for this study. We also describe the methods applied to present the findings collected during the experiments performed on the files of interest, i.e., *AndroidManifest.xml* and *classes.dex* in the design cycle of this study.

3.2 Research Design

This section of the chapter describes the research problem formulation for a design science research work and the methods that are applied to solve the determined problems. One of the major problems identified during the literature survey on Android malware detection with ML and DL was the need for automated approaches to generate features. Most research works, if not all, rely on the researcher's knowledge for selecting features which improve their respective malware detection model performances. This study aims to solve that problem by creating an automated method of feature generation from Android applications for the task of Android malware detection.

The document embeddings technique will be used to extract features automatically, and binary classifiers will be trained on these features to evaluate their performance. The research philosophy of this work, therefore, is positivism. The philosophy of positivism is

based on the idea that a researcher can objectively observe 'reality' and that the 'reality' remains as is, even without a researcher's interaction. With respect to this work, the results from the binary classification models will allow for objectively verifying the quality of the automatically generated features. The quality of binary classification based on these features will be an observable reality.

The effectiveness of the features extracted automatically at the task of Android malware detection will be evaluated experimentally. The research strategy would be of iterative experiments, each of which will require evaluating the various possible feature artefacts. This will be done based on the design cycle, which is further explained later in the chapter. An analysis of the capability of different document embeddings artefacts at the task of Android malware detection will be done based on performance metrics collected during the experimentation phase. We will try visualising some of the document embeddings-based artefacts to understand our experiments' results better. The artefacts will be evaluated against the DREBIN [28], as well as AndroZoo dataset [87]. It was already noted in the literature review section that various researchers had used the DREBIN dataset repeatedly. Running and evaluating our automated feature creation methods on this dataset would allow us to have a fair comparison against other methods applied to DREBIN dataset by various researchers.

3.2.1 Research Problem

As mentioned before, design science research is an iterative process of Design and Investigation. These two segments of the research process combined together represent two different kinds of research problems which are; Design problems and knowledge questions [86] It has been put in simple terms by the author, Roel J. Wieringa, in his book *Design Science Methodology for Information Systems and Software Engineering* that the design problems demand some improvements or changes to the problem context. On the contrary, knowledge questions seek to find factual knowledge based on the existing literature available in the context of design problems. The interaction of the artefact with the design problem also answers some of the knowledge questions. Based on these insights provided by Roel J. Wieringa, this research work targets to solve one of the design problems and answer some of the knowledge questions that arise in the process of solving the design problem. These have been briefly described in the below sections.

Design Problem

The literature review suggests that the field of Android malware detection utilises features from the Android package files. These features are manually mined from the applications and then

used to create feature vectors similar to sparse one-hot encoded vectors. A certain degree of knowledge about Android malwares and the Android operating system is required to select the features that represent an application behaviour carefully. This process will be improved; therefore, the design problem is to "Perform the task of Android malware detection by using a novel feature generation process that eliminates manual selection of features".

We attempt to solve this design problem by considering the information present in an Android manifest or dex file as simply textual information. We then generate document embeddings-based artefacts from manifest and dex files. We also generate artefacts made up of embeddings of both manifest and dex files concatenated together. We use different file combinations from APKs and different *doc2vec* algorithms to understand how well different document embeddings artefacts can represent the behaviour of an Android application. The document embeddings artefacts are used to train multiple binary classifiers, and these binary classifiers are later used to predict unseen file embeddings to validate their performance.

Knowledge Questions

Based on the above Design problem, a few knowledge questions arise regarding the artefact being created and how it improves upon the problem. These knowledge questions have been listed below:

1. Is it possible to effectively utilise the Document embeddings technique to generate feature vectors representing the Android malware and benignware?
2. Which Android package files should be used for the task of feature creation?
3. How do they compare against the existing/manual methods used by the researchers?

We try to answer the first question from the above set of knowledge questions in Chapter 4. We do this by attempting to visualise the high-dimensional *doc2vec* embeddings. Although for the task of malware detection, we trained 200-dimensional vectors, we created 30-dimensional vectors for the task of visualisation. This was done to reduce complexity during dimensionality reduction as we reduced the 30-dimensional embeddings to 3-dimensions using isometric mapping. Question #2 has already been partially answered based on the findings in the literature review. We repeatedly saw that manifest and dex files contain valuable information regarding the behaviour of an application throughout the literature review. We also utilise these files to generate the Android malware feature embeddings as part of our research work. To answer the last question, we will use our high dimensional doc2vec-based features to train ML and DL classifiers on the AndroZoo dataset. We also apply our feature generation technique

to Android APKs from the DREBIN dataset to generate similar high-dimensional doc2vec-based features. The performance of the embedding artefacts on binary classification models is presented in Chapter 5

3.2.2 Knowledge Context

The existing engineering knowledge about the artefact, the scientific theories, design specifications and the factual knowledge about the shortcomings or strengths of the currently available systems (or artefacts) is what is referred to as knowledge context in the paradigm of design science research. The researchers' goal is to use this knowledge to create design solutions that improve interactions with the world or expand this knowledge by "answering knowledge questions" based on the said interactions [86]. The following paragraphs in this section briefly explain the pre-existing or *Prior Knowledge* that is utilised in artefact creation for this research work. This prior knowledge is the natural language processing (NLP) technique of 'document embeddings' (or *doc2vec*) approach. The document embeddings algorithms would be used to generate the features for the task of Android malware detection.

Document Embeddings

The main idea behind document embeddings was to solve document similarity problems [13]. The document vectors are learnt by utilising an unsupervised prediction task where the words present in a document are used to predict a document's ID or document ID along with words are used to predict other words present in the document corpus. The vectors are also referred to as paragraph vectors (*paragraph2vec*) and document vectors (*doc2vec*). The learnt document embeddings can be directly used as features for training classifier models. These embedding vectors are learnt using two algorithms, namely:

1. Distributed Memory (or PV-DM)
2. Distributed Bag of Words (or PV-DBoW)

The work on *doc2vec* was a follow up on *word2vec* [5]. Through the *word2vec* technique, the researchers gave two algorithms that generated word vectors such that the vectors of words with high similarity will appear closer to each other when their vectors are plotted in a vector space. This idea was extended by Mikolov *et al.*to documents and paragraphs. The word vectors for all the words present in the document corpus are also used for creating document vectors. A word vector is maintained for each of the words in the

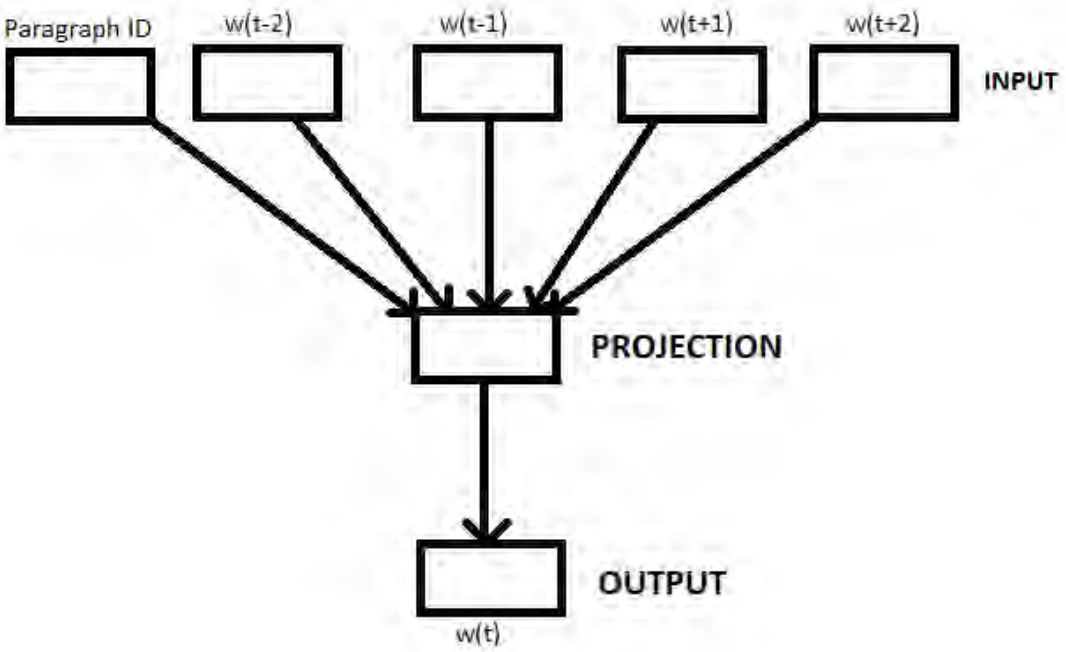


Figure 3.1: PV-DM prediction task

vocabulary. A vector for each document (similar to word vector) in the corpus is also maintained when learning document embeddings.

For PV-DM, this paragraph vector is used along with word vectors to predict different words present inside a context. This prediction task has been depicted in Figure 3.1. In this aspect, context means a set of words, i.e., w(t-2), w(t-1), w(t+1) and w(t+2) that appear together in the text. Paragraph ID is the paragraph vector for a specific document. Regarding our research, context refers to the text that appears inside a Dalvik executable file or a manifest file belonging to an Android malware or benignware.

In PV-DBoW, a paragraph vector is used to predict context words present inside this paragraph (or document) as depicted in the Figure 3.2. The vectors for each of the paragraphs are updated by backpropagating the loss with a goal of improved prediction of words, given the paragraph vector of the document. It shall also be noted that we are not interested in the prediction task here; instead, we only utilise the paragraph vectors that were trained during this prediction process.

The pre-existing knowledge of *doc2vec* was utilised for this research work. The *doc2vec* algorithm is mainly used for the tasks of document similarity. The embeddings generated by the algorithms learn about the similarity of documents in the corpus as a side effect of the prediction tasks. The interaction of this knowledge with the problem of Android malware detection leads to the creation of automatically generated vectors. We try to use the two algorithms of *doc2vec* to generate malware document embeddings which conserve the

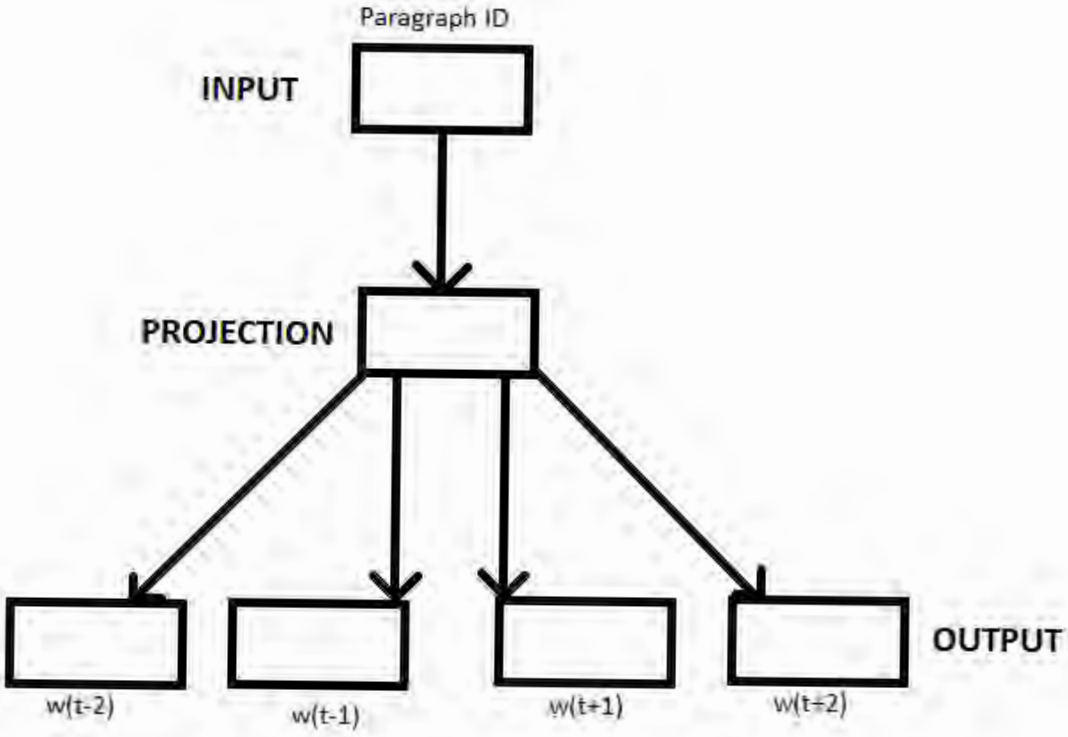


Figure 3.2: PV-DBoW prediction task

document-level semantic information. This is done with multiple files extracted from the Android applications. The classification models learn to differentiate between the embeddings of Android malware and benignware files.

The next part of this chapter discusses the 'design cycle', which is at the core of any Design Science methodology-based research work. This will be followed by a brief description of the datasets used for the research work. The rest of this chapter briefly describes the 'artefacts' that will be generated and evaluated during each iteration of the design cycle. The 'artefact' is basically document embeddings generated from the relevant files from an Android APK. We train various binary classification models on these various document embeddings to investigate the' artefacts' performance. The different classification models which would be used and how their results would be presented have also been briefly explained later in this chapter.

3.3 Design Cycle

The proposed methodology for this study takes an iterative approach which is the main characteristic of design science research methodology. We perform iterations of solution design and validations on the files of interest, as seen from Figure 3.3. In a single iteration, we design two artefacts, i.e., document embeddings-based features by applying PV-DBoW and PV-DM algorithms on the files of interest. These document embeddings features are then evaluated by their performance on different supervised deep learning and machine learning classifiers which learn to classify Android applications into malware and benignware. At the end of each iteration, the performance metrics for both the artefacts are recorded in the respective tables of classifier results.

As described earlier, research projects based on design science methodology repeatedly go over the design and investigation process. This is done to create and improve upon the desired artefact. We also iteratively generate artefacts for all the files of interest to identify the artefact that has the best performance metrics at the task of Android malware detection. The design cycle is a subset of the larger Engineering cycle [86] which includes two additional steps, i.e., steps of solution implementation and its evaluation. Design science research does not concern itself with creating an end-product for the larger market and, therefore, focuses on creating a feasible artefact during design cycle iterations. This process is thus referred to as the design cycle [86], and consists of the following three processes:

1. Problem Investigation
2. Solution Design
3. Solution Validation

The problem investigation part of the design cycle looks for the problems that can be improved upon after each iteration. The initial design problem in the case of this research work has been described earlier. As part of our research work, we solve this initial design problem and answer the knowledge questions, which were also discussed earlier in this section. Solution Design would undertake a task to utilise the document embeddings approach to create a 'solution', which would be validated by using this document embeddings-based 'solution' to train different classifiers and evaluate the classification metrics carefully. In each iteration, the solutions will be built using different file types.

Although, before the steps of design cycle can begin, the *AndroidManifest.xml* and *classes.dex* files have to be extracted from the applications present in our datasets. Once the

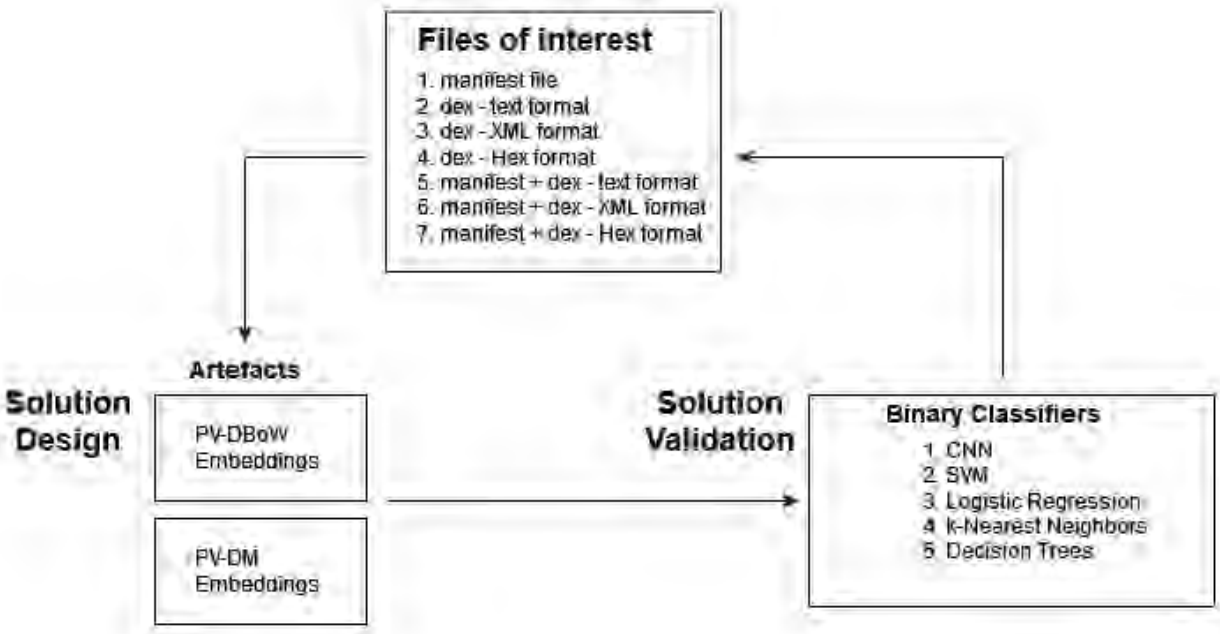


Figure 3.3: Design Cycle - Iterations

files are extracted from the Android APKs the *classes.dex* files (or Dalvik executables) would be sampled using different output formats to learn document embeddings from them. On the other hand, *AndroidManifest.xml* files (or manifests) were used as it is to create document embeddings, in their raw decompiled XML form. These file embeddings were further used to train classification models that achieve optimal classification accuracy. The file extraction and document embeddings creation process has been described in the Chapter 4. The next section of this chapter describes the datasets we have used to conduct the experiments for this study.

3.3.1 Datasets

For this research work to be conducted, a large dataset consisting of both malicious and benign Android applications was required. This dataset was, thus, collected from a secondary data source with many Android applications that have been already verified (and tagged) as benign or malicious using different anti-virus systems. An existing dataset was also required to achieve a fair comparison between previous research works and the methods suggested in this work. This requirement and the knowledge gathered from the literature review led to us shortlisting two significant datasets containing Android applications that are currently publicly available for research, namely AndroZoo [88] and DREBIN [28].

As the literature review showed, the DREBIN dataset is still being used by researchers but is an ancient dataset and contains outdated Android applications. Nevertheless, it allows us to compare our methods with published methods and their results. The APKs for this research work have, therefore, majorly been taken from the DREBIN dataset and

AndroZoo. The AndroZoo dataset was created and is still maintained by researchers at the University of Luxembourg. It is still being updated regularly with new and current applications, representing the threat the Android platform faces much better. It is a "Growing collection of Android applications collected from several sources, including the official Google Play app Market" [87] and therefore was one of the clear choices for this research work. AndroZoo dataset contains over 15 million Android application packages from different App marketplaces, assessed by multiple anti-virus products available through Virus Total API.

Four thousand seven hundred applications (2350 malicious and 2350 benign) were selected from the AndroZoo dataset using clustered random sampling technique with a dex date (creation date on the *classes.dex* file) ranging from 2015 to the current date. These Android application files were downloaded using the script created by Artem Kushnerov [89]. This dataset would be referred to as Dataset 1 in the later parts of this work. For dataset 2, another 2,350 malicious Android applications were randomly selected from the DREBIN dataset to evaluate our methods against existing works. Only the malicious APK files were used from the DREBIN dataset. The 2,350 benign APK files for Dataset 2 were thus, randomly taken from the AndroZoo repository. The number 4700 was coincidental and was selected based on the downloaded files and available time and resources during the candidature. The APK files were randomly split into two sets of 4000 and 700 for both datasets. The set containing 4000 will be referred to as **training** set, and the remaining 700 APK files will be referred to as **test** set in this research work. The **test** set applications were deliberately left out of the training process. The 4000:700 split was again based on the earlier point, that the resources and time were limited, and we started to work with available applications at the time. The 700 file split was done to keep some applications aside to allow us to test the generated embeddings against unseen malware. The number of files chosen for training and evaluation of our automatic feature generation technique was low based on data availability, storage and computational limitations. Although, the size of both datasets 1 and 2 were kept the same for easy comparison of our feature's performance against our dataset and the DREBIN dataset. The datasets that are being used in this research are different from the papers that were found throughout the literature review as their datasets are not easily available. In the field of artificial intelligence-based Android malware detection, there does not seem to be a single standard dataset or evaluation technique in use.

3.3.2 Solution (or Artefact) Design

Before this stage, the main objective will be to set up pre-processing to process the APK and remove the unnecessary files from the APKs that are not needed to generate document embeddings using the *doc2vec* algorithms. This includes unpacking the Android application

packages to access the dex files and decompiling them. The artefact design includes the creation of document embeddings with different files that were extracted. The various file-based embeddings will be used to generate the artefacts of this research work, and in each iteration, two document embedding artefacts will be created. Each file will be used to generate two artefacts based on the two algorithms, PV-DM and PV-DBoW, which were discussed earlier. These artefacts would then be used to train a set of binary classifiers. The performance metrics for each of the classifiers on the artefacts will be recorded at the end of the iteration. Further details about how the artefact performance will be presented for the different detection models that would be trained and evaluated on the research artefacts have been provided in the section **Solution Validation**.

For this research, two files from Android APKs i.e., *AndroidManifest.xml* and *classes.dex* will be used to generate document vectors as artefacts in each iteration of the design cycle. We also use a concatenated form of embeddings where *classes.dex* and *AndroidManifest.xml* file embeddings are joined together. The possible file embedding artefacts that would be created, using both PV-DBoW and PV-DM algorithms, have been listed in Table 3.1. The concatenated embeddings will be generated by combining the embeddings of manifest and other dexdump output types. We would not be using the embeddings of the *classes.dex* files in combination with each other as even though the files are in different formats, they all contain roughly the same information. This would bring the number of possible document embeddings artefacts to seven times two, one each for PV-DM and PV-DBoW.

Table 3.1: List of artefacts created with PV-DBoW and PV-DM algorithms.

#1:	Manifest File Embeddings
#2:	Dex (*text* format) File Embeddings
#3:	Dex (*XML* format) File Embeddings
#4:	Dex (*Hexadecimal* format) File Embeddings
#5:	Manifest + Dex (*text* format) File Embeddings
#6:	Manifest + Dex (*XML* format) File Embeddings
#7:	Manifest + Dex (*Hexadecimal* format) File Embeddings

3.3.3 Solution Validation

After the document embeddings are ready, the quality of these automatically generated feature vectors will be measured by creating different statistical learning models to validate their Android malware detection capabilities. The classification performance of each of the embedding types listed in Table 3.1 will individually be evaluated on different machine learning models. The classification models are trained on these document vectors to classify an input document into their respective classes, i.e., malicious and benign. Then we evaluate the

classification model performance by performing a classification task on a previously unseen malware and benignware document's embeddings.

The malware detection task is essentially a binary classification task where statistical models are explicitly trained to differentiate between two different classes of documents. For this research work, these models will be trained and tested with the document embeddings artefacts, both PV-DBoW and PV-DM. As this study aims to create a system that can perform feature engineering automatically, the binary classification models were chosen from the most commonly used models. This decision was made based on the works we saw during the literature review and was done to compare the results achieved from this research quickly. In an iteration of the design cycle, the artefacts created from the files of interest will be tested on all the binary classification models that have been listed below:

1. Convolutional Neural Nets
2. Support Vector Machines
3. Logistic regression
4. K-Nearest Neighbours
5. Decision Trees

Evaluation and Findings

Initially, the goal was to identify and understand the document embeddings' effectiveness in extracting features from Android malware. Once the embeddings are generated, the goal is to find the best detection models and evaluate their performance against datasets benchmarked by researchers previously. Each of the above listed binary classification models will have two separate tables where the experiment results of both the PV-DBoW and PV-DM artefacts will be recorded. In each iteration of the design cycle, an artefact will be used to train and test all the binary classification models for their malware detection capabilities. Based on the classification model performances on document embedding artefacts, the knowledge regarding the document embeddings performance on the Android malware detection task can be understood.

During the design cycle iterations, the experiments are done to evaluate artefact performance. The performance result for the artefact is recorded in the assigned row for an artefact in the results table of an individual classifier. A sample table structure has been presented in Table 3.2. The left-most column of Table 3.2 shows the specific artefacts and the remaining columns consist of the metrics we recorded. The results of the experiments have

been presented such that the performance of all the artefacts against a malware detection or classification model can be observed together.

Table 3.2: Example results table for individual models.

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest					
Dex	**Plain**				
	XML				
	Hex				
Combinations	**Manifest + Plain**				
	Manifest + XML				
	Manifest + Hex				

Evaluation of document embedding performance at the task of automatic feature creation would require the evaluation of binary classification models that have been trained on these feature vectors. We have **training** and **test** sets for both DREBIN and AndroZoo datasets. The classification models will be trained on the file embeddings generated from the **training** set. This trained model will then be evaluated against the file embeddings generated from the APK files belonging to the **test** set, which consists of 700 unknown Android applications that the binary classification model has not previously seen. This model's performance will be evaluated by measuring the model characteristics of Precision, Recall, Accuracy and F1-score during both the model training and evaluation phases. These metrics were again chosen from the most commonly seen metrics during the literature review for an easy comparison of results. The goal here is to identify the document embeddings that perform the task of binary classification effectively. This would be achieved based on the performance metrics which are shown below:

- **Precision:** This value is a ratio of valid positive results, i.e., True Positives, against the total predicted positive results, i.e., the total number of True Positives and False Positives.

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} \tag{3.1}$$

- **Recall:** This measure represents the ratio of valid positive predictions, i.e., True positive, against the actual valid results, i.e., True Positives and False Negatives.

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} \tag{3.2}$$

- **Accuracy:** This measures the ratio of valid predictions, i.e., True negative and True Positives, against all the predictions made by the model.

$$Accuracy = \frac{TruePositive + TrueNegative}{TruePositive + TrueNegative + FalsePositive + FalseNegative} \tag{3.3}$$

- **F1-Score:** This is also known as F-score (or F-measure) and it evaluates the model accuracy by taking in consideration both precision and recall values. The F1-score is evaluated using the following equation.

$$F1Score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3.4}$$

Presentation of findings

The results of the experiments performed with the different sets of document embedding artefacts will be presented in a tabular format by recording the metrics achieved in each of the design cycle iterations performed multiple times over the binary classification models. The structure of an example findings table can be seen in Table 3.2. These tables would consist of five different columns, namely, 'File Embedding Type', 'Precision', 'Recall', 'Accuracy' and 'F1-Score'. The values that would be stored in these columns are as follows:

- File Embedding Type: As the name suggests, this column would contain the file type of the document embeddings the model is being trained upon.
- Precision: As discussed in the evaluation section, but for each individual experiment iteration on the file embedding type.
- Recall: As discussed in the evaluation section, but for each individual experiment iteration on the file embedding type.
- Accuracy: As discussed in the evaluation section, but for each individual experiment iteration on the file embedding type.
- F1-Score: As discussed in the evaluation section, but for each individual experiment iteration on the file embedding type.

3.4 Summary

This chapter discussed the methodology that is being applied for this research work. We discussed design science research methodology and how it will be applied to solve the research

problem at hand. We discussed how research problems could be divided into design problems and knowledge questions in design science research. This was followed by an explanation of the design problem and the knowledge questions at hand. We also discussed the prior knowledge of *doc2vec* that is pre-existing and would be used to generate an artefact that helps reduce the manual effort for feature engineering from Android applications, benign or malicious.

After the research and knowledge goals were defined, a brief discussion was presented on the datasets used for research work. The files of interest, from which the document embeddings learn to represent an Android application's behaviour, are selected based on the knowledge gained during the literature review. In each iteration of the design cycle, an artefact will be selected, and binary classification models will be trained and tested using the document embedding features of the artefact. The models for classification that would be trained on these embeddings were chosen randomly based on their ability to perform a binary classification task. The artefacts created by the document embedding algorithms will be assessed experimentally using the metrics described earlier in this chapter. We also verify the trained Android malware detection models by using them to predict the classes of document embeddings that the binary classification models have not previously seen. The classification models trained and evaluated on the embeddings artefacts will allow us to quantify the document embedding's ability at the task of Android malware detection.

It is important to note that document embedding algorithms do not solve the problem of Android malware detection by themselves, and they can be used in various other settings. Document embeddings, instead, conserve the document level similarity, which then interacts with binary classifiers. Based on this similarity, the classifiers are trained to differentiate between malicious and benign applications. The experimentation done to understand this is an iterative process where classifier models are trained against different file embeddings in each experimental iteration. The experiments done to understand this interaction better have been presented in Chapter 5.

Chapter 4

Feature Extraction & Background Information

4.1 Introduction

This chapter presents the relevant background information about an Android application and its building blocks, followed by the pre-processing steps required to extract *AndroidManifest.xml* and *classes.dex* files before the features can be learnt automatically. Firstly, we describe the information that is contained in the files of interest. We then present the algorithms used to extract the decompiled files from Android APKs. We also discuss the approach taken for file extraction and decompilation. We then explain how the document embeddings were generated from the files. After generating the document embeddings from the Android APK files, we present 3-dimensional visualisations of the embeddings. This is done by reducing the dimensionality of the high-dimensional embeddings to 3-dimensions using a technique known as isometric mapping. We finish this chapter by briefly describing the malware detection models that will be developed and evaluated during our experiments.

4.2 Building Blocks of an Android application

Android operating system has a Linux kernel at the lowest layer of the operating system [90]. Some specific system functions are added to the vanilla Linux kernel to customise it for a mobile device platform. An Android application package (APK) consists of certain files and resources. The files of interest for our research work are *AndroidManifest.xml* and *classes.dex* files. The manifest file contains the definition of an application's components and the permissions required

by an application to run on an Android device. On the other hand, the Dalvik executable file contains compiled source code and resources required for an application to be installed and executed on the Android operating system. We now describe the files and resources present inside an Android application.

Files and Resources

1. Manifest File - It is probably the most important file inside an Android application package. For an application to be installed on a device, Android OS looks for this file. It lists all the components present inside an application [91]. The manifest file also lists the permissions required by an application, the minimum Android version API required, hardware features that would be used and the other application APIs access it may require. The manifest file is an XML formatted file (AndroidManifest.xml) and lists the components with the below XML tags:
    - `<activity>`
    - `<service>`
    - `<receiver>`
    - `<provider>`
2. Dalvik Executable - The manifest file, java source code of the application and the resources are compiled together into a Dalvik executable file (or classes.dex). Dalvik executable file contains compiled Dex bytecode, which is executed by the Android Runtime [92]. In simple terms, the Dalvik executable file contains all the logical components of an Android application in a compiled form.
3. Application Resources - Android applications consist of various resources like strings, images, audio files, and animations. These resources are handled separately from the source code and are assigned an integer ID by the Android development kit. These integer IDs are used to refer to the resources from within the source code [93].

Figure 4.1: Sample Manifest File - Benign

Figure 4.2: Sample Manifest File - Malicious

```
Class #0            -
  Class descriptor  : 'Lair/protech/AppEntry$1;'
  Access flags      : 0x0000 ()
  Superclass        : 'Ljava/lang/Object;'
  Interfaces        -
    #0              : 'Landroid/content/DialogInterface$OnClickListener;'
  Static fields     -
  Instance fields   -
    #0              : (in Lair/protech/AppEntry$1;)
      name          : 'this$0'
      type          : 'Lair/protech/AppEntry;'
      access        : 0x1010 (FINAL SYNTHETIC)
  Direct methods    -
    #0              : (in Lair/protech/AppEntry$1;)
      name          : '<init>'
      type          : '(Lair/protech/AppEntry;)V'
      access        : 0x10000 (CONSTRUCTOR)
      code          -
      registers     : 2
      ins           : 2
      outs          : 1
      insns size    : 6 16-bit code units
      catches       : (none)
      positions     :
        0x0000 line=160
      locals        :
        0x0000 - 0x0006 reg=0 this Lair/protech/AppEntry$1;
        0x0000 - 0x0006 reg=1 (null) Lair/protech/AppEntry;
  Virtual methods   -
    #0              : (in Lair/protech/AppEntry$1;)
      name          : 'onClick'
      type          : '(Landroid/content/DialogInterface;I)V'
      access        : 0x0001 (PUBLIC)
      code          -
      registers     : 4
```

Figure 4.3: Sample Dexdump - Benign

```
Class #0            -
  Class descriptor  : 'Lcom/qihoo/util/Configuration;'
  Access flags      : 0x0001 (PUBLIC)
  Superclass        : 'Ljava/lang/Object;'
  Interfaces        -
  Static fields     -
    #0              : (in Lcom/qihoo/util/Configuration;)
      name          : 'ENABLE_CRASH_REPORT'
      type          : 'Z'
      access        : 0x0009 (PUBLIC STATIC)
    #1              : (in Lcom/qihoo/util/Configuration;)
      name          : 'ENABLE_PT'
      type          : 'Z'
      access        : 0x0009 (PUBLIC STATIC)
  Instance fields   -
  Direct methods    -
    #0              : (in Lcom/qihoo/util/Configuration;)
      name          : '<clinit>'
      type          : '()V'
      access        : 0x10008 (STATIC CONSTRUCTOR)
      code          -
      registers     : 1
      ins           : 0
      outs          : 0
      insns size    : 7 16-bit code units
      catches       : (none)
      positions     :
        0x0001 line=7
        0x0004 line=8
      locals        :
    #1              : (in Lcom/qihoo/util/Configuration;)
      name          : '<init>'
      type          : '()V'
      access        : 0x10001 (PUBLIC CONSTRUCTOR)
      code          -
```

Figure 4.4: Sample Dexdump - Malicious

4.3 Document embeddings creation

As discussed earlier, we aim to utilise the Android manifest and Dalvik executable files for generating our feature vectors. We discussed the information contained within these files previously in this chapter. We now describe the process used to extract these files and turn them into document embeddings artefacts using the two algorithms of *doc2vec* technique, i.e., PV-DBoW and PV-DM, on which we can conduct our experiments.

File Extraction from APK

Once the applications were collected from the AndroZoo and DREBIN datasets, the following steps were followed for both Datasets 1 and 2, which were described in the Methodology's Section 3.3.1. The first step was to decompile the APKs. After decompilation, the *AndroidManifest.xml* file was extracted as it contains information related to the application components and necessary permissions. This file is used for feature extraction during artefact creation. The manifest file was extracted by firstly decompiling the APK using APKTool [10] and then extracting the manifest file.

Algorithm 1 Data Pre-processing - Manifest File

Require: F_{apk}
Ensure: $Manifest_{xml}$ ▷ AndroidManifest.xml file.

$DevSet_N \leftarrow 4000APKs$
$TestSet_N \leftarrow 700APKs$

while $F_{apk} \in DevSet_N \& TestSet_N$ **do**

if F_{apk} is from folder Benign **then**
$Prefix \leftarrow benign\_$
else if F_{apk} is from folder Malicious **then**
$Prefix \leftarrow malicious\_$
end if

`input-file` = F_{apk}
execute
`apktool d <input-file> -o <decompiled-filepath>`
execute
`find <decompiled-filepath> -maxdepth 1 -type f -name "AndroidManifest.xml"`
execute
`cp <manifest-filepath> <Prefix + manifest-file-output-path>`
execute
`rm -rf <decompiled-filepath>`

end while

After extracting the manifest files, the second step was to extract and process the Dalvik executables. The Android application package (APK) is like a usual archive file but contains all the necessary files for an application to run on Android's Dalvik virtual machine. This archive file has to be unpacked to extract the Dalvik executable file in the package. This is followed by decompilation of this Dalvik executable file using the dexdump tool [11]. This tool provides capabilities to disassemble the Dalvik executable to Dalvik virtual machine byte code and save it to disk in different formats, i.e., plain text file and XML. We also used another tool

Algorithm 2 Data Pre-processing - Dalvik Executable File

Require: F_{apk}
Ensure: $Dex_{plain}, Dex_{xml}, Dex_{hex}$
▷ Dex file in plain, xml & hex formats.

$DevSet_N \leftarrow 4000APKs$
$TestSet_N \leftarrow 700APKs$

while $F_{apk} \in DevSet_N \& TestSet_N$ **do**

if F_{apk} is from folder Benign **then**
$Prefix \leftarrow benign\_$
else if F_{apk} is from folder Malicious **then**
$Prefix \leftarrow malicious\_$
end if

Unzip the F_{apk} to `unzipped-filepath`
execute to find Dex file
`find <unzipped-filepath> -maxdepth 1 -type f -name "classes.dex"`
execute to create F_{plain}
`dexdump -l plain <input-dex-file> -o <plain-output-path>`
execute to create F_{dexml}
`dexdump -l xml <input-dex-file> -o <xml-output-path>`
execute to create F_{hex}
`hexdump <input-dex-file> >> <hex-output-path>`
execute to remove the unzipped files
`rm -rf <unzipped-filepath>`

end while

called hexdump [12] that comes as a command-line utility with Ubuntu. This tool was used to create a hexadecimal dump from the *classes.dex* file.

The process of extracting the manifest and Dalvik executable files can be done in any order. The steps applied for this task were presented in the Algorithm 1 and 2. The decompilation process and extraction of the necessary files for our work were automated by creating a program using Python programming language, which can be found on GitHub at the following URL (https://github.com/TheCyberian/manifestAndDexExtraction) [94]. The extracted files were used as input to *doc2vec* algorithms of PV-DBoW and PV-DM to generate our research artefacts. During the phase of the artefact design, the document embedding features would be learnt from three different types of decompiled Dalvik executables and the manifest files. This was done using Python's gensim library [95].

Learning Document Vectors

Once the dex and manifest files are extracted successfully, the task of learning document vectors can begin. Document vectors can be created using two different algorithms, both of which use an unsupervised shallow neural network to learn fixed-length numeric features from different-sized documents by performing a prediction-based learning task. The document vector algorithms capture the semantics of documents as a side-effect of these prediction tasks and, thus, give better performance than the usual bag-of-words(BoW) based models, as highlighted in the research work by Tomas Mikolov and Quoc Le [13].

The two algorithms, PV-DM and PV-DBoW, which were discussed earlier in the Methodology chapter, learn the document embeddings and the word embeddings of the words present in the training documents. These embedding matrices are initially declared with random values and then updated at each learning epoch by back-propagating the loss until the loss is minimised. These fixed-length embeddings are learnt by maximising the probability of a given context word present in the manifest and Dalvik executable files based on the current target word and document using either the Hierarchical Softmax function (or Negative sampling). In addition to the paragraph (or document) embedding matrix, a word embedding matrix is maintained and updated based on the loss evaluated during the prediction task. We only utilise the paragraph embedding matrix for our task.

We trained 200-dimensional embeddings for the malware detection task. We also developed 30-dimensional vectors for a visualisation task. We used isometric mapping to reduce the 30-dimensional embeddings to 3-dimensions. The choice of using 30-dimensional embeddings was made to lessen complexity during dimensionality reduction. The learnt document embeddings are used to train different binary classification models that would allow

Algorithm 3 Document Embeddings Creation

Require: C_{input} ▷ Corpus of N manifests or dex file dumps
Ensure: $E_{N,200}$ ▷ Document Embedding Vector for classification
Ensure: $E_{N,30}$ ▷ Document Embedding Vector for visualisation

$C_{input} \rightarrow TrainSet_{4000}, TestSet_{700}$

while $F_{apk} \in C_{input}$ **do** ▷ F_{apk} is manifest or dex file dump

if F_{apk} is from TestSet **then**
$Tag \leftarrow test$
else if F_{apk} is from TrainSet **then**
$Tag \leftarrow train$
end if

perform
`Build Vocabulary of words present in the file`
perform
`Train doc2vec models - PV-DM and PV-DBoW`
perform
`Save the document embeddings with its class`

end while

the detection of malware. The classification models used for the detection task have been briefly explained in the upcoming section.

4.4 Visualisations

We performed seven iterations of the design cycle on the different Android files, where we performed experiments on the generated document embeddings-based 'artefacts' for both Datasets #1 and #2. The files used for creating these artefacts are the *AndroidManifest.xml* and *classes.dex*. As mentioned in the methodology chapter, the information in the Dalvik file has been dumped in the form of hexadecimal using hexdump tool [12] and plain text & XML using dexdump tool [11]. Therefore, a discussion will be presented on the four major files that were used to generate our research artefacts, which are:

1. Manifest
2. Dex text format
3. Dex XML format

4. Dex Hex dump

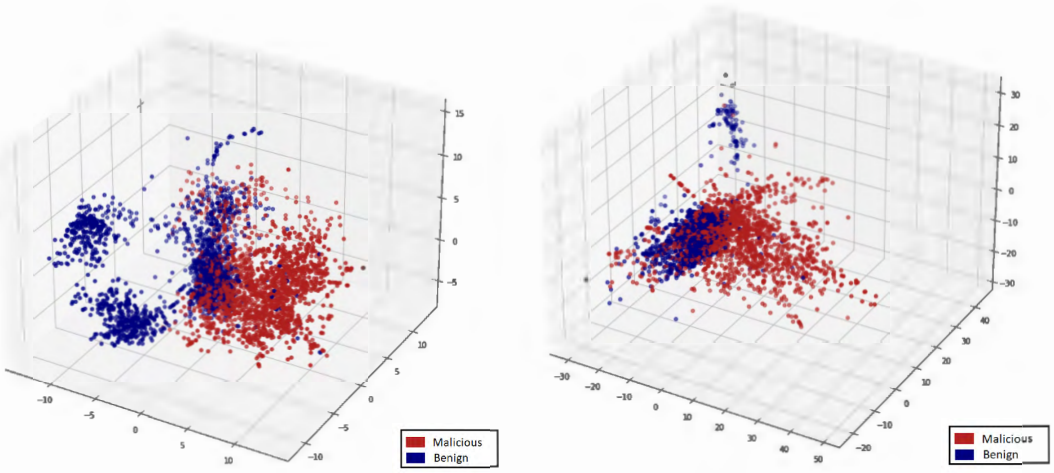
For the other three remaining design cycle iterations, the artefacts were created by concatenating different Dex file (formats) embeddings with Manifest file embeddings. All of the artefacts used during our design cycle iterations were listed in Table 3.1, and we visualise Artefact #1 to #4 in the following parts of this chapter. Each of the seven artefact types is created using two separate ways, i.e., using PV-DBoW and PV-DM algorithms. The previous chapter evaluated these artefact types for the Datasets #1 and #2. Please note that even though we created artefacts using a combination of manifest and Dalvik executable files for the classification task, we do not visualise them in this chapter.

The rest of this chapter discusses the artefacts one by one. We begin by describing the file and the information present in the file that was used to create that specific artefact. We then present the Isometric mapping visualisation of both PV-DBoW and PV-DM algorithm-based artefacts. The visualisation is presented for AndroZoo (Dataset #1) first, followed by the visualisation for the DREBIN (Dataset #2). This chapter also attempts to get a better insight into our experiment results.

4.4.1 Artefact #1: Manifest File Embeddings

This section showcases the visualisations of document embeddings-based artefact generated from manifest files. An *AndroidManifest.xml* file contains permissions required by the application, the activities that the application contains, intents and other important information required to run an application on an Android device. From an overview of a manifest file of a benign vs malicious application, one can notice that malicious applications tend to have a higher number of required permissions when compared to benign applications as could be seen from the sample of benign and malicious application's manifest files that were presented in Figures 4.1 and 4.2.

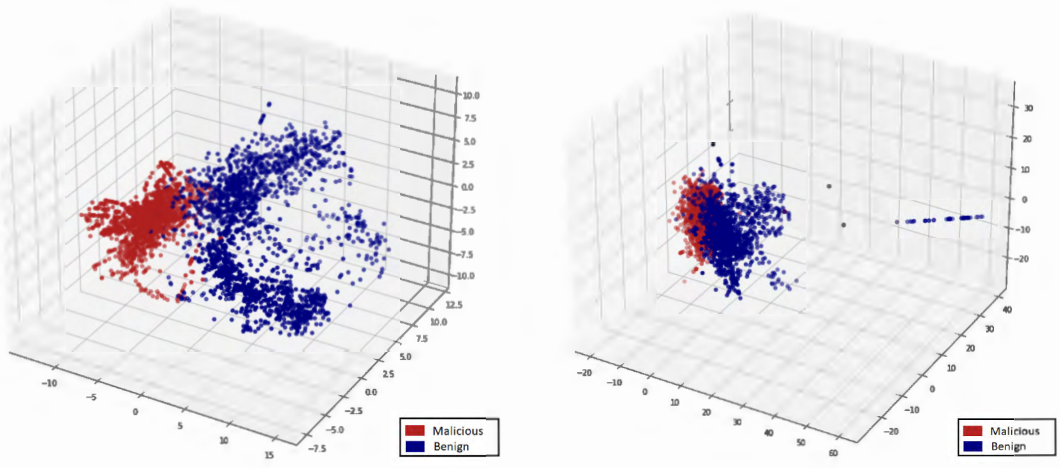
As mentioned earlier, we have created 30-dimensional document embeddings for the visualisation task. These 30-dimensional embeddings were then scaled down to 3-dimensions using Isometric mapping. The Isomap visualisations for APKs belonging to the AndroZoo dataset are shown in Figure 4.5. At a glance, the clustering of malicious and benign classes can be seen in both PV-DBoW and PV-DM algorithm-based document embeddings in the visualisations. The PV-DM algorithm-based embeddings show much overlap in the clusters, and the PV-DBoW embeddings have a higher degree of separation between the malicious and benign classes.



(a) PV-DBoW Embeddings (b) PV-DM Embeddings

Figure 4.5: Artefact #1 - Isometric mapping - AndroZoo

The visualisation of embeddings of manifest files from DREBIN dataset is presented in Figure 4.6. The clustering of embeddings here is much more concentrated when compared to what we saw in Figure 4.5 with a very minimal overlap between the low-dimensional embeddings. This explains why the artefact (or automatically extracted features) performance at the task of Android malware detection was slightly better with DREBIN dataset than against AndroZoo dataset for the same artefact.



(a) PV-DBoW Embeddings (b) PV-DM Embeddings

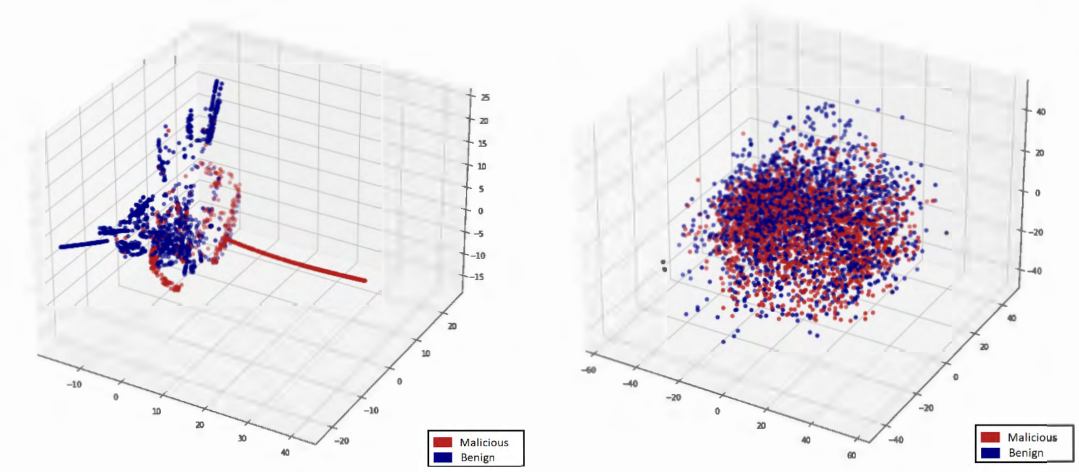
Figure 4.6: Artefact #1 - Isometric mapping - Drebin

4.4.2 Artefact #2: Dex (*text* format) File Embeddings

The *classes.dex* file contains compiled form of resources, application code, manifest and other relevant information packaged into a single file. For this artefact, we utilised a tool called

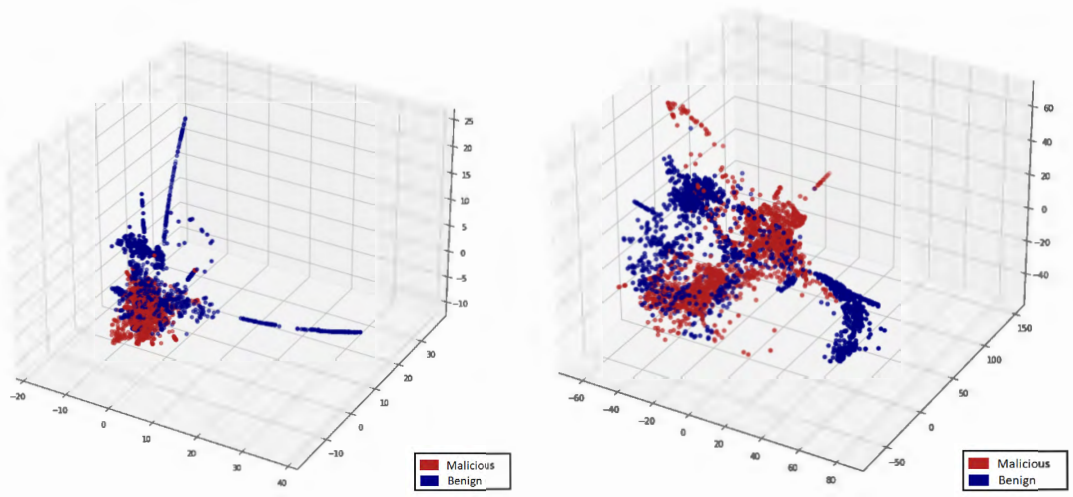
dexdump [11], which dumps the disassembled Dalvik bytecode in a text format (and also XML format, which is used to create the next artefact) for security researchers to analyse the Dalvik bytecode manually. However, finding traces of malicious behaviour in the dexdump of Android applications can be really difficult for an untrained eye. A snapshot of how a benign and malicious application's dex file dump looks like was presented in Figures 4.3 and 4.4.

From the low-dimensional visualisations of the dex file dumps, we can see some clusters forming for both malicious and benign classes, specifically in PV-DBoW-based embeddings of both AndroZoo and DREBIN datasets. Although clustering of malicious and benign classes can be seen in both PV-DBoW and PV-DM-based document embeddings for the DREBIN dataset in Figure 4.8, the PV-DM algorithm-based embeddings for benign and malicious classes in AndroZoo dataset appeared to be grouped. We can see this from the PV-DM embeddings part of Figure 4.7.



(a) PV-DBoW Embeddings (b) PV-DM Embeddings

Figure 4.7: Artefact #2 - Isometric mapping - AndroZoo

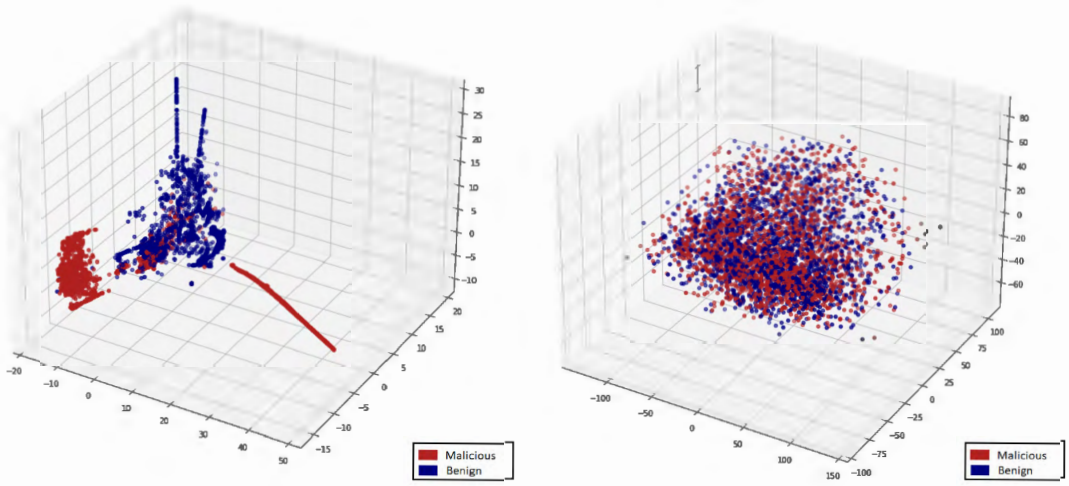


(a) PV-DBoW Embeddings (b) PV-DM Embeddings

Figure 4.8: Artefact #2 - Isometric mapping - Drebin

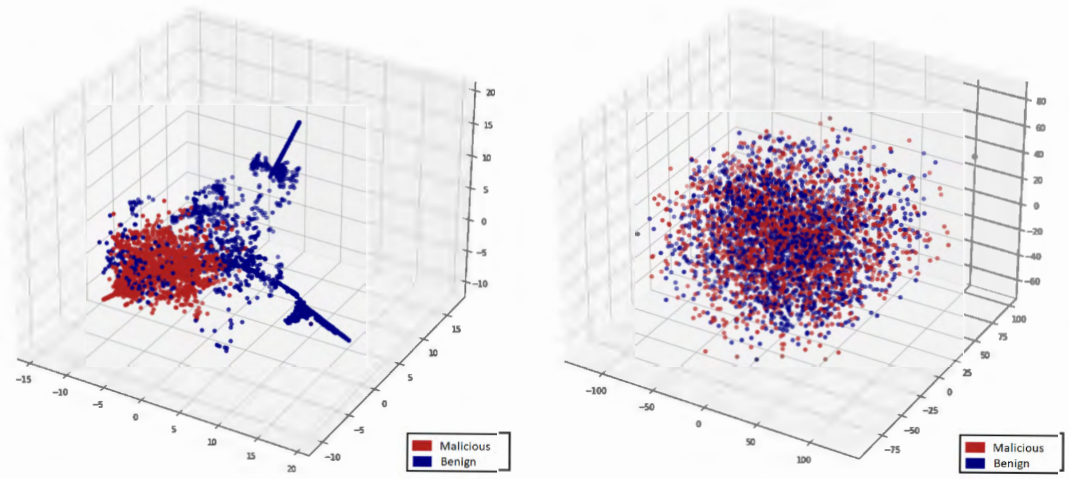
4.4.3 Artefact #3: Dex (*XML* format) File Embeddings

As mentioned previously, *classes.dex* file contains compiled resources required by an Android application to run smoothly. This section presents visualisations of document embeddings generated from *classes.dex* files dumped in an XML format using the dexdump tool. Similar to other artefacts, we created a 30-dimensional document embedding for visualisation tasks which were scaled down to 3-dimensions using Isometric mapping. Although, for the previous artefact (embeddings of Dex file in plain format), PV-DM based embeddings showed clustered malicious and benign classes for DREBIN dataset, the clustering of malicious and benign classes can only be seen in PV-DBoW based document embeddings of this artefact (Dex (XML) file embeddings) for both the Datasets #1 and #2 (refer Figures 4.9 and 4.10).



(a) PV-DBoW Embeddings (b) PV-DM Embeddings

Figure 4.9: Artefact #3 - Isometric mapping - AndroZoo

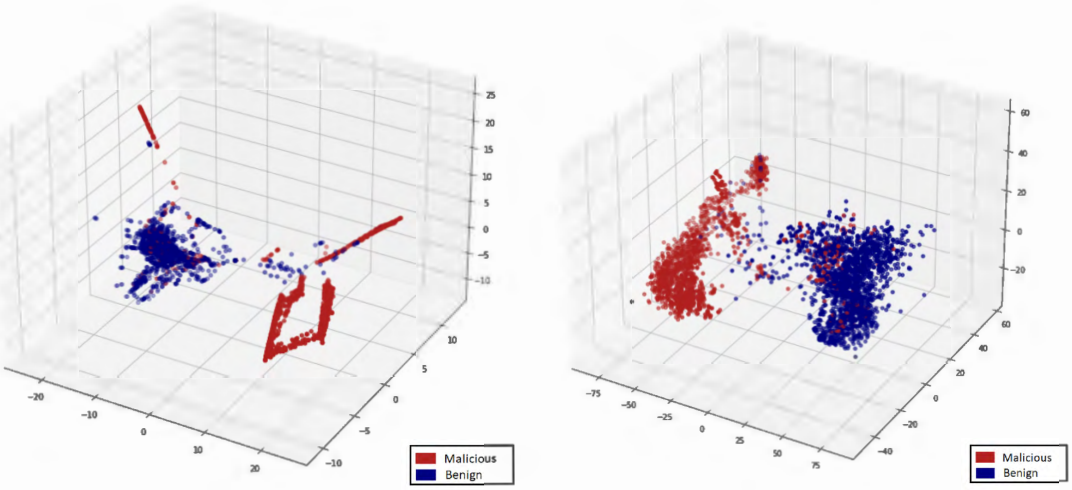


(a) PV-DBoW Embeddings (b) PV-DM Embeddings

Figure 4.10: Artefact #3 - Isometric mapping - Drebin

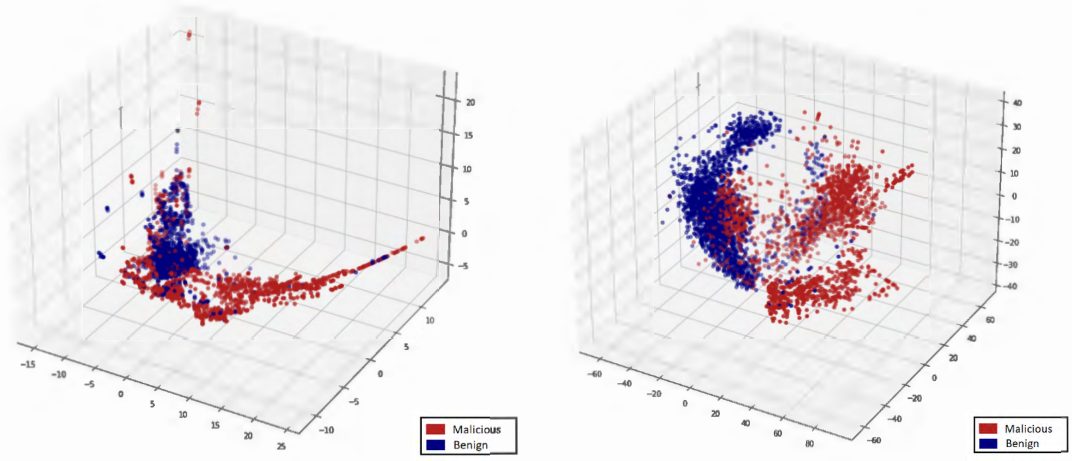
4.4.4 Artefact #4: Dex (*Hexadecimal* format) File Embeddings

For creating this artefact, the file content bytes of *classes.dex* are first dumped in hexadecimal form using hexdump [12] tool. The PV-DBoW and PV-DM algorithms are used on these hex dumps to generate the respective document embeddings. Different clusters of malicious and benign class document embeddings can be seen in the visualisations of both the PV-DBoW and PV-DM algorithm-based Dex (Hexadecimal format) file artefacts. This behaviour was seen for both the DREBIN and AndroZoo datasets (refer to Figures 4.11 and 4.12).



(a) PV-DBoW Embeddings (b) PV-DM Embeddings

Figure 4.11: Artefact #4 - Isometric mapping - AndroZoo



(a) PV-DBoW Embeddings (b) PV-DM Embeddings

Figure 4.12: Artefact #4 - Isometric mapping - Drebin

4.5 Detection Model Development

The malware detection task is essentially a binary classification task where statistical models are explicitly trained to differentiate between two different classes of documents. These models will be trained on the document embedding artefacts for this research work. As this study aims to create an automated feature engineering method, the binary classification models were selected intentionally to evaluate standard methods that appeared in the literature. The classical binary classifier models used for this work have been listed and briefly described here.

1. **Support Vector Machines**

SVM is generally used as a two-class classifier, and it can be enhanced to work as a multi-class classifier. It works based on a linear discriminant function and calculates a decision boundary (or hyperplane) that best separates data points into different classes with the largest margin in a multi-dimensional vector space. The model aims to learn an optimal hyperplane that can classify unknown data into their respective classes. SVM works well when the feature dimensionality is high, and the feature points are not linearly separable.

2. **Decision Trees**
   Decision Trees fall in the family of supervised learning algorithms. It is basically a binary Tree that recursively splits the dataset based on differing rules until we are left with pure leaf nodes, i.e., data with only one class type. It works similar to nested if-then-else conditions, but the model learns these rules instead of pre-programmed rules during the training process.

3. **K-Nearest Neighbours**
   The k-NN model works on the idea that data points in the vector space closer to each other have some similarities and are, therefore, part of the same class. It is most commonly used for the task of pattern recognition. For a classification goal, the model compares the unseen instances with K similar training instances it can find, and the most similar instance's class is returned as the model prediction.

4. **Logistic regression**
   A logistic regression model is an example of a probabilistic discriminative model. The model is used for classification tasks and not regression tasks as the name suggests, and it is an advanced version of a linear regression model. Logistic regression is an efficient algorithm which does not require extensive computations, and its simplicity and low resource requirements make it a popular algorithm used for classification tasks.

5. **Convolutional Neural Nets**
   A class of neural networks that applies the mathematical operation of convolutions to large feature sets are commonly referred to as Convolutional Neural Networks (or CNNs). CNNs were generally applied to computer vision-related problems and practically revolutionised the field. CNNs deal with huge feature dimension sizes and are also apt for the task of dimensionality reduction as well as prediction and classification tasks. The only drawback of using a CNN is that it requires large datasets for sufficiently training the neural network for classification or prediction tasks.

Experiments will be performed using each one of these five models. Each one of the models would be trained and validated on both the artefacts generated from the seven different file embedding artefacts. Development of the detection model involved parameter tuning to achieve optimal model performance metrics. This was achieved using grid search. The trained

classification models were used to predict the classes for the embedding artefacts belonging to the **test** set. We present the results of experiments done on both the **training** and **test** sets in the Chapter 5.

4.6 Summary

In this chapter, we shed light on some background knowledge that would allow a reader to understand better our research artefacts and how they interact with the Android malware detection problem. We described the building blocks of an Android application by describing the application components and information present in the files of interest. This chapter also described the process used to extract and decompile the *AndroidManifest.xml* and *classes.dex* files. We also briefly describe how the document embeddings are learnt from the extracted files. We end this chapter by presenting a short description of the classification models used for the experiments conducted to understand our research artefacts' capabilities on the Android malware detection task. We now conclude this chapter and present the results of our experiments conducted on artefacts created from Datasets #1 and #2 in the next chapter.

Chapter 5

Malware Detection Experiments: Results & Discussion

5.1 Introduction

In this chapter, overall binary classification performances achieved at the task of Android malware detection from experiments performed on our research artefacts are presented. This was achieved by using the document embedding artefacts described in the previous chapter to train multiple binary classifiers. Metrics of Accuracy, Precision, Recall and F1-score are presented for the malware detection models trained and evaluated at the binary classification task. Firstly, the document embedding artefacts generated for the AndroZoo dataset are evaluated. The results of the artefacts against the binary classifiers are presented in a tabular format for analysis, and bar charts of artefact performance during the training and evaluation phases are presented for comparison. It will be followed by evaluating and presenting the results of classification models against the embeddings of applications from the DREBIN dataset in a similar format.

Multiple malware detection models are trained and validated against the artefacts generated from **training** set by splitting the file embeddings into an 80:20 ratio. The model parameters which provide the best performance are found by running a grid search on various model parameters by using **training** set. The model parameters that lead to a binary classification model that achieves the most promising results are used to train a classification model with the best parameters. This classification model is evaluated by using it to detect malware by predicting the classes of the unseen embeddings present in the artefacts generated from the **test** set.

As part of this chapter, we present the results of training and evaluation of 5 binary classification models on artefacts made up of both Dataset 1 and 2 separately. Each classification model is trained and evaluated on all of the seven artefacts listed in Table 3.1 presented in Chapter 3. Each of artefact (file) embeddings were generated using both PV-DBoW and PV-DM algorithms. Therefore, the training and evaluation metrics are presented for malware detection models for both PV-DBoW and PV-DM embedding artefacts in different tables.

5.2 Dataset #1 : AndroZoo

In this section, we evaluate all the artefacts that were created using the files extracted from the Android applications belonging to the AndroZoo dataset. The evaluation of document embeddings-based features at the task of Android malware detection is done by using the research artefacts to train and evaluate all of the classifiers mentioned in Section 3.3.3 of the Methodology chapter. The experiments were done by acquiring the dexdump (in text, XML and hexadecimal formats) and android manifests of 4,700 Android APKs from Dataset 1. These files from the APKs were then split into two sets of 4000, and 700 files, each in **training** and **test** sets, respectively, as described in a previous chapter.

These files were then used to generate 200-dimensional document embeddings from each file type using PV-DBoW and PV-DM algorithms over each design cycle iteration. The results of the malware detection models have been presented by showcasing the performance results of artefacts during the development phase, followed by the performance results during the evaluation phase, where the malware detection models were given artefacts made up of unseen embeddings (i.e., **test** set) whose class the model had to predict. We also present bar charts of performance metrics of binary classification models on individual artefacts during both training and evaluation phases for easy comparison.

5.2.1 Convolutional Neural Networks

We begin by presenting the results of CNN model development and testing on embeddings generated using the PV-DBoW algorithm. This is followed by the presentation of the development and testing process results on embeddings that were created using the PV-DM algorithm. The results are presented in tabular format, and clustered bar charts of the performance metrics were added to visually compare model performances during both the development and testing phases.

PV-DBoW

During the development phase, although the Manifest file document embeddings had a Precision, Recall, Accuracy and F1-score higher than 96.5%, they still had comparatively lower performance metrics when compared to Dalvik executable file document embeddings. It can also be seen that the document embeddings of the hexadecimal dump of the Dalvik executables performed the best when used alone and when these embeddings were concatenated with the embeddings of the Manifest file. The CNN model also worked very well with other file-type embeddings during the development phase, as seen from Table 5.1.

Table 5.1: CNN Model metrics on PV-DBoW - Development

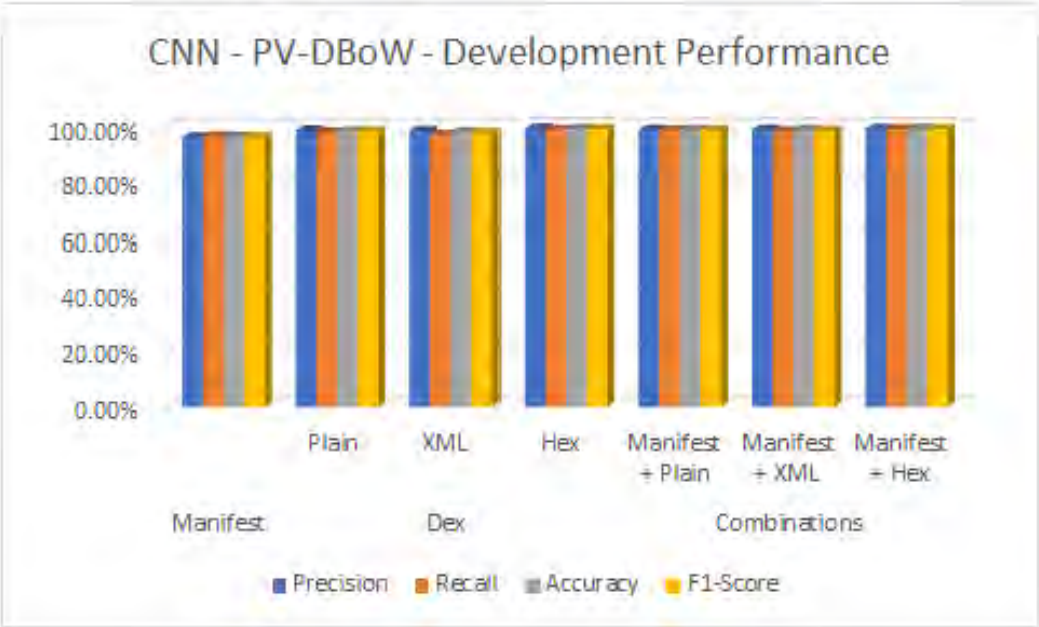
File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		96.5%	96.9%	96.8%	96.5%
Dex	**Plain**	99.0%	98.2%	98.6%	98.6%
	XML	98.8%	97.3%	98.3%	98.1%
	Hex	99.6%	99.2%	99.5%	99.4%
Combinations	**Manifest + Plain**	99.2%	99.1%	99.2%	99.1%
	Manifest + XML	99.2%	98.6%	99.2%	98.9%
	Manifest + Hex	99.6%	99.4%	99.5%	99.5%

During the testing phase, trained models were used to predict classes of various unseen file embeddings based on PV-DBoW and PV-DM algorithms. The manifest file-based document embeddings also had the lowest metrics during the testing phase. We could also see that the hexadecimal dump-based document embeddings could have worked better on unseen file embeddings, as they did during the CNN model development. Instead, the regular text formatted Dalvik file dexdump-based document embeddings outperformed hexadecimal ones. All of the file embedding types, except Manifest based, had all their metrics above 95% even on unseen/test document embeddings, as can be seen from the Table 5.2

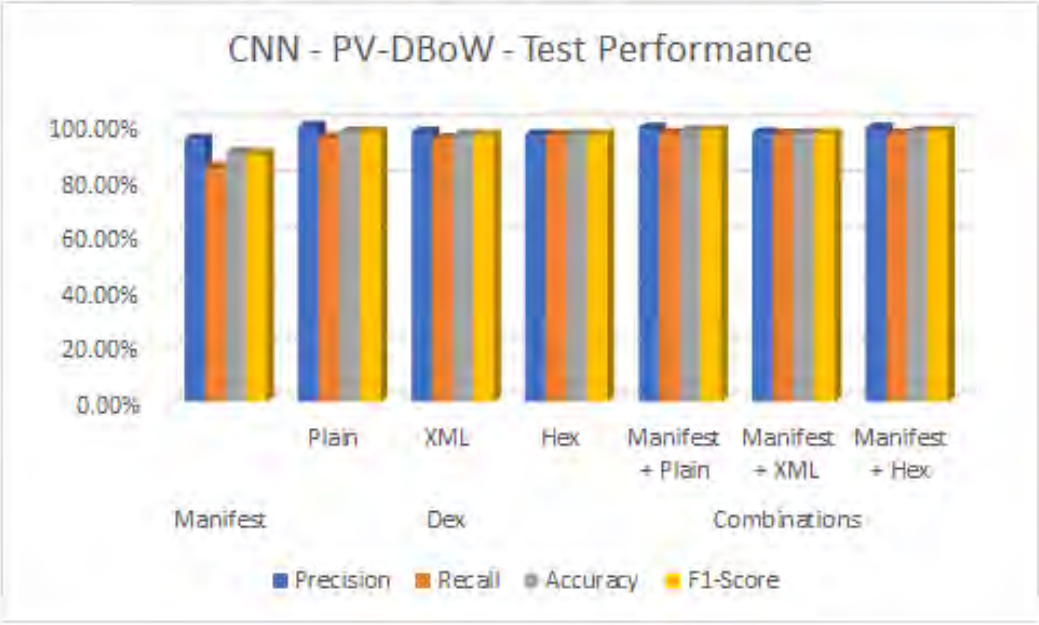
Table 5.2: CNN Model metrics on PV-DBoW - Test

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		94.8%	84.5%	90.0%	89.4%
Dex	**Plain**	99.4%	95.4%	97.4%	97.3%
	XML	97.3%	95.1%	96.2%	96.2%
	Hex	96.5%	96.2%	96.4%	96.4%
Combinations	**Manifest + Plain**	98.8%	96.8%	97.8%	97.8%
	Manifest + XML	97.1%	96.5%	96.8%	96.8%
	Manifest + Hex	98.8%	96.5%	97.7%	97.6%

The comparative results of the development and testing phases of a Convolutional Neural Network based classifier on PV-DBoW algorithm-based artefacts can be seen in Figure 5.1. It can be noticed that even though during the development, plain (text) formatted dex file dump embeddings performed slightly poorer than the hexadecimal dump embeddings of dex file, it outperformed the hexadecimal dump embeddings when it was used to classify unseen/test android files. Based on the metrics for the CNN model performance on PV-DBoW-based embeddings, we can safely infer that the PV-DBoW algorithm-based document embeddings would allow us to train CNN-based binary classifiers that can differentiate between malicious and benign android application document embeddings.



(a) PV-DBoW - Development



(b) PV-DBoW - Testing

Figure 5.1: PV-DBoW artefacts performance on CNN - AndroZoo

PV-DM

The CNN model performance metrics with PV-DM algorithm-based document embeddings are very low in general when compared to the same model performance against PV-DBoW algorithm-based embeddings. In the development phase, the Manifest file and hexadecimal dump-based document embeddings are the only exceptions to this trend, as seen from Figure 5.2. During the testing phase, the manifest file-based embeddings performed poorly, and only the hexadecimal dump-based embeddings achieved Precision, Recall, Accuracy and F1-score consistently higher than 93%.

Compared to clusters that appeared in PV-DBoW-based visualisations, PV-DM-based embeddings were generally noisy. They appeared intertwined when seen in the PV-DM embeddings-related visualisations presented in Chapter 4. Only hexadecimal file PV-DM appeared to be clustered in the visualisations, which is also reiterated by the exceptional classification performance of the model on the hexadecimal file embeddings, which can be seen in Table 5.3 and 5.4

Table 5.3: CNN Model metrics on PV-DM - Development

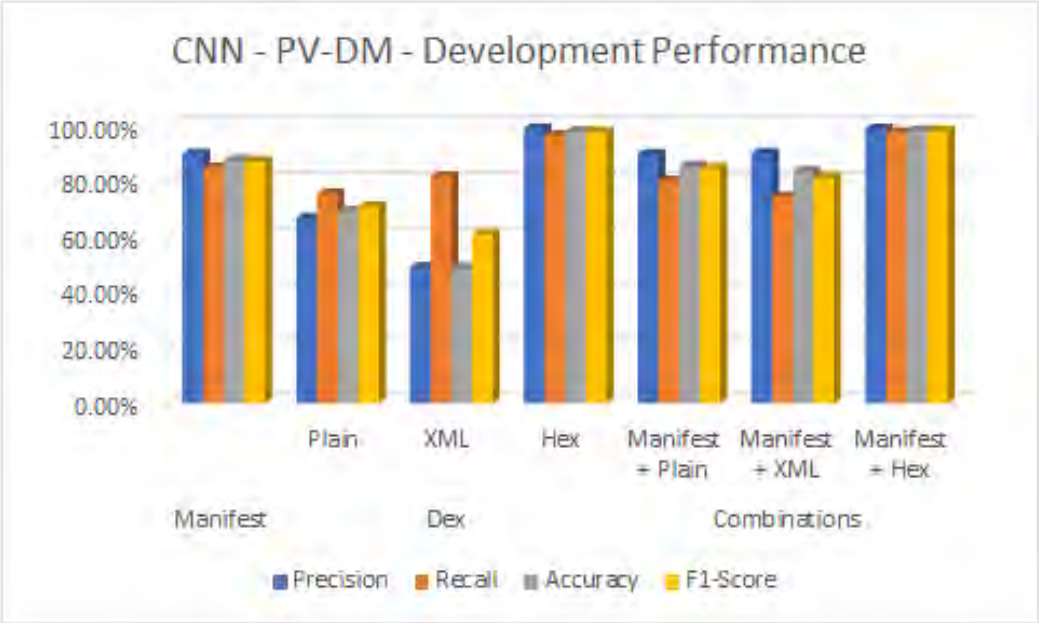
File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		90.0%	85.0%	87.8%	87.4%
Dex	**Plain**	66.9%	75.6%	69.3%	71.0%
	XML	49.1%	81.9%	48.9%	61.0%
	Hex	99.1%	96.8%	98.0%	97.9%
Combinations	**Manifest + Plain**	89.8%	80.3%	85.6%	84.8%
	Manifest + XML	90.2%	74.6%	83.6%	81.6%
	Manifest + Hex	99.0%	97.7%	98.4%	98.3%

Table 5.4: CNN Model metrics on PV-DM - Test

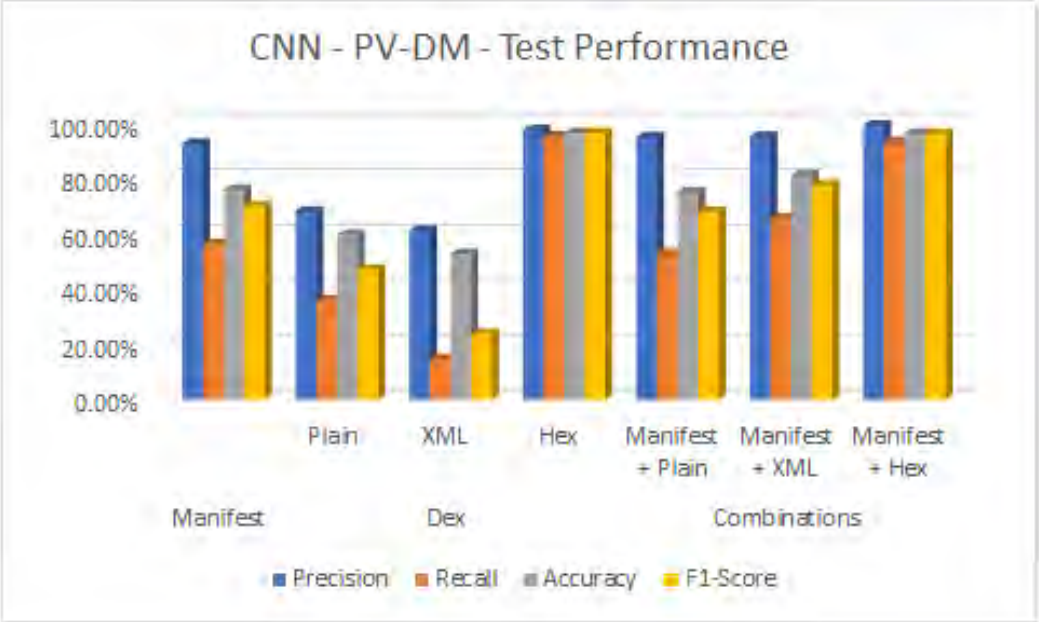
File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		92.4%	56.2%	75.8%	69.9%
Dex	**Plain**	68.1%	36.0%	59.5%	47.1%
	XML	61.1%	14.8%	52.7%	23.9%
	Hex	97.3%	95.1%	96.2%	96.2%
Combinations	**Manifest + Plain**	94.8%	52.8%	75.0%	67.8%
	Manifest + XML	95.0%	65.4%	81.0%	77.5%
	Manifest + Hex	99.0%	93.1%	96.1%	96.0%

The results of the development and testing phases of a Convolutional Neural Network based classifier on PV-DM algorithm-based artefacts can be seen side by side for comparison in

Figure 5.2. It can be clearly seen that even though manifest file and hexadecimal dump of dex file performed better during the development than other file type embeddings. However, when the unseen PV-DM-based file embeddings were to be classified into their respective classes during tests, only hexadecimal dump-based document embeddings provided satisfactory results.



(a) PV-DM - Development



(b) PV-DM - Testing

Figure 5.2: PV-DM artefacts performance on CNN - AndroZoo

5.2.2 Support Vector Machines

This subsection showcases the research artefact's performance on an SVM classifier. We present the tabulated results and comparative bar charts of the artefact performance metrics during SVM model development and testing. This was done for embeddings generated using both PV-DBoW and PV-DM algorithms.

PV-DBoW

In the SVM model development phase, we observed that the manifest file embeddings performance is slightly poorer than the performance of other files PV-DBoW based embeddings. This can be inferred from Table 5.5 which contains the SVM model development results on the seven artefacts. Although the performance metrics of all other models declined slightly during the testing phase, the manifest file embeddings performed the worst. The SVM model was tested on unseen document embeddings, similar to other classification models in the study. The results of testing of the SVM model on PV-DBoW-based artefact embeddings can be seen in Table 5.6.

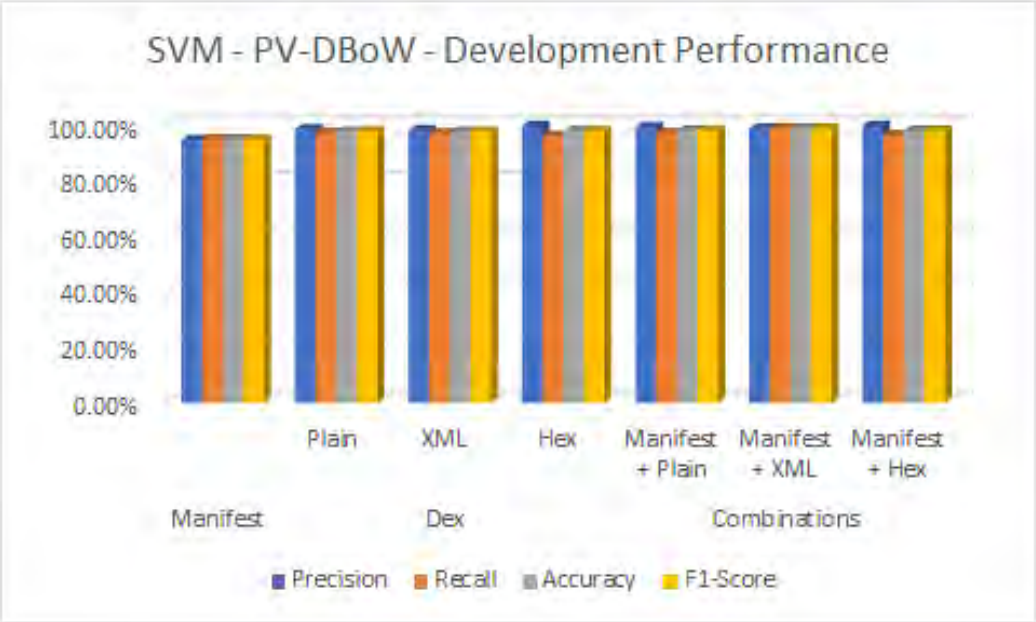
Table 5.5: SVM Model metrics on PV-DBoW - Development

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		94.9%	95.1%	95.1%	95.0%
Dex	**Plain**	98.7%	97.3%	98.0%	98.0%
	XML	98.4%	97.2%	97.8%	97.8%
	Hex	100.0%	96.3%	98.1%	98.1%
Combinations	**Manifest + Plain**	99.5%	97.3%	98.3%	98.4%
	Manifest + XML	99.0%	99.0%	99.0%	99.0%
	Manifest + Hex	100.0%	96.6%	98.2%	98.2%

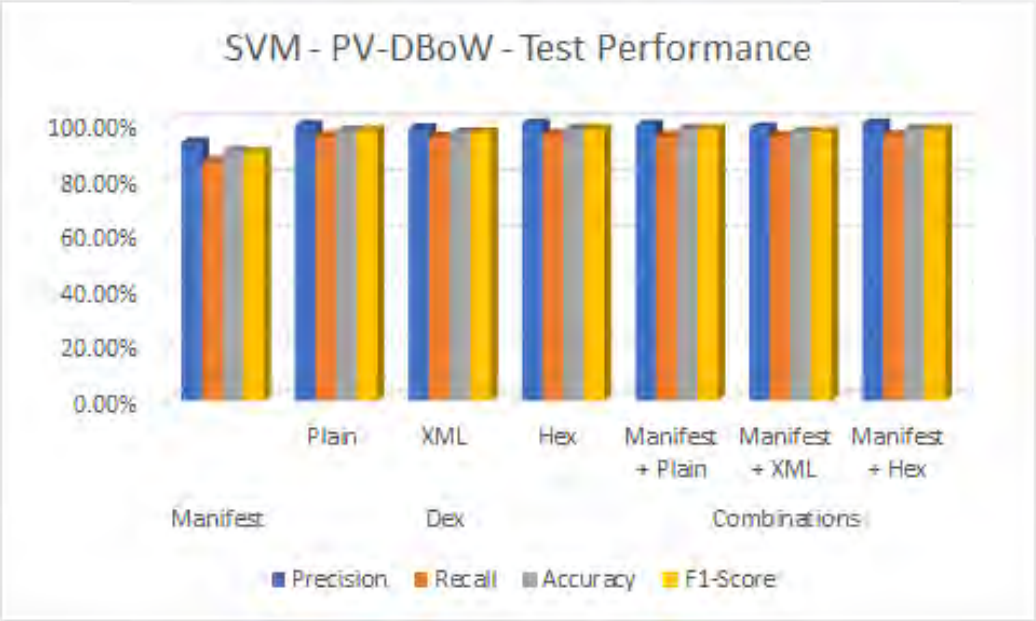
Table 5.6: SVM Model metrics on PV-DBoW - Test

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		93.2%	86.2%	90.0%	89.6%
Dex	**Plain**	99.4%	95.4%	97.4%	97.3%
	XML	98.2%	95.1%	96.7%	96.6%
	Hex	100.0%	96.0%	98.0%	97.9%
Combinations	**Manifest + Plain**	99.4%	95.5%	98.0%	97.9%
	Manifest + XML	98.5%	95.4%	97.0%	96.9%
	Manifest + Hex	100.0%	95.7%	97.8%	97.8%

During the model development phase, the embeddings of the Dex file in plain and XML format had a Precision of 98.7% and 98.4%, respectively. In contrast, the hexadecimal file embeddings had a 100% Precision in both the development and testing phases. It was also noted that the model development done with concatenated Embeddings of Manifest and Dex file in XML format had an overall 99% Precision, Recall, Accuracy and F1-score.



(a) PV-DBoW - Development



(b) PV-DBoW - Testing

Figure 5.3: PV-DBoW artefacts performance on SVM - AndroZoo

Comparison of the PV-DBoW artefacts performances can be seen in the bar chart presented in Figure 5.3. During the testing phase, concatenated embeddings of Manifest and

Dex file in plain format had a slightly better performance compared to the concatenated embeddings of Manifest and XML formatted dex file, which performed comparatively better during development. The standalone hexadecimal format Dex file embeddings had the best performance metrics during the testing phase.

PV-DM

The binary classification performance of the SVM model on PV-DM embeddings was generally poor during both the development and testing phases. Five of the seven artefacts achieved a maximum of 88% F1-score during the development, as seen from Table 5.7. During the testing phase, the lowest F1-score fell to 81%. It must be noted that the embeddings created from the hexadecimal dump of *classes.dex* file performed exceptionally well. We can see from Table 5.8 even during the testing phase, the Precision and F1-score both stayed above 97%.

The Hexadecimal file-based embeddings generated using the PV-DM algorithm performed well with the CNN model too, as can be seen from Tables 5.3 and 5.4 presented earlier. The performance of artefacts built using PV-DM algorithms appears to be substandard, except for the artefact built using a hexadecimal dump of dex files. From the PV-DM embedding visualisations presented in the previous chapters, Section 4.4.4 about Artefact#4 this exception in the performances can be attributed to the class-wise clustering that was present in the Hexadecimal file embeddings. This clustering with low-dimensional embeddings consequently means a better binary classification capability with high-dimensional embeddings.

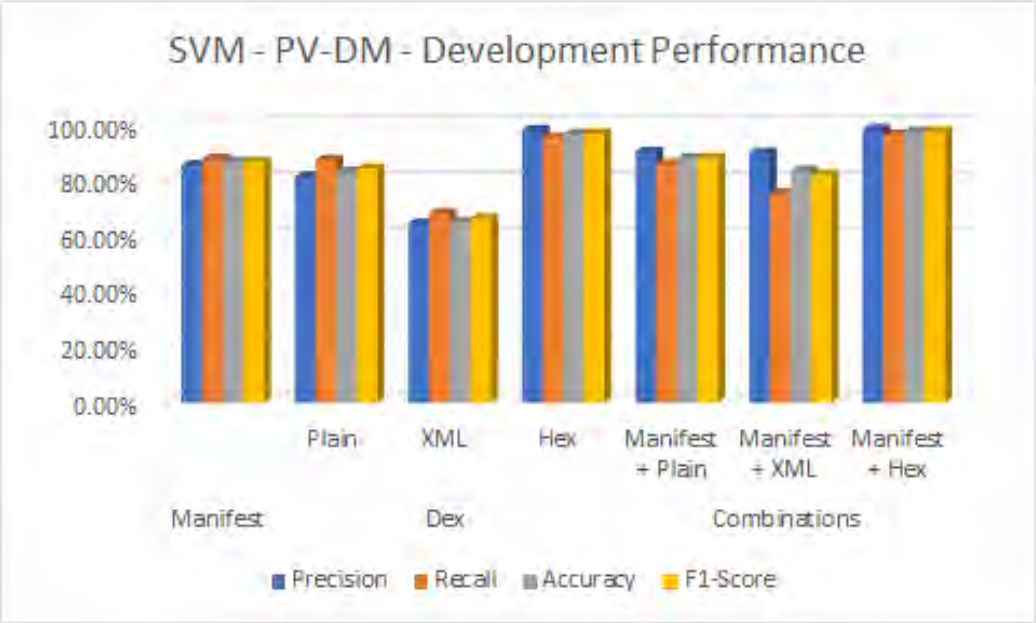
Table 5.7: SVM Model metrics on PV-DM - Development

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		85.6%	88.0%	86.8%	86.8%
Dex	**Plain**	81.5%	87.6%	83.3%	84.5%
	XML	64.7%	68.4%	65.3%	66.5%
	Hex	98.5%	95.6%	97.0%	97.0%
Combinations	**Manifest + Plain**	90.6%	86.2%	88.2%	88.3%
	Manifest + XML	90.2%	75.6%	83.6%	82.2%
	Manifest + Hex	98.7%	96.8%	97.7%	97.8%

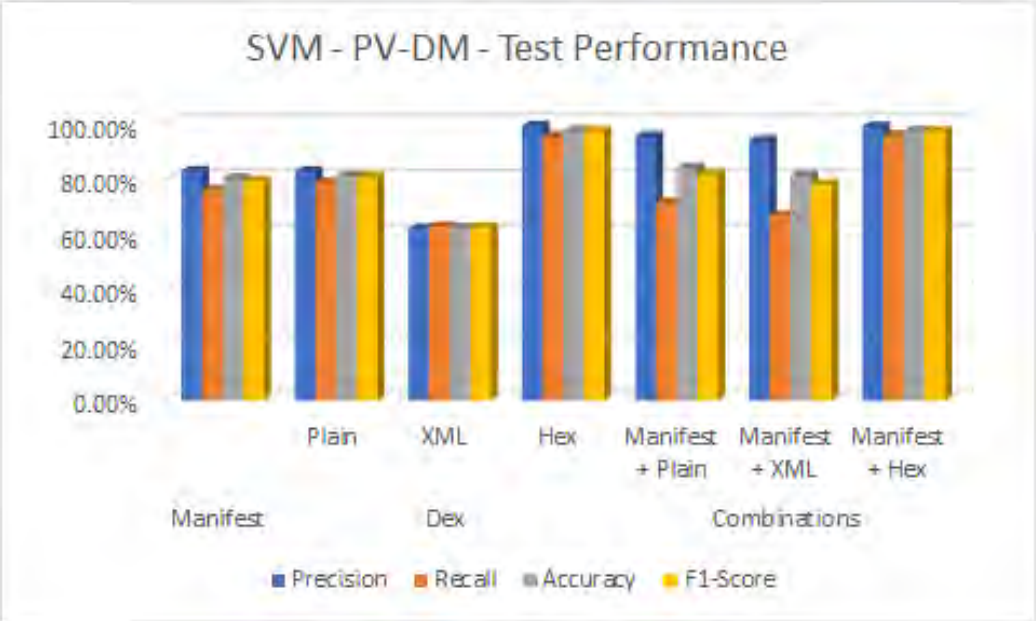
Table 5.8: SVM Model metrics on PV-DM - Test

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		82.8%	76.0%	80.1%	79.2%
Dex	**Plain**	82.8%	78.5%	81.1%	80.6%
	XML	62.1%	62.8%	62.2%	62.5%
	Hex	99.1%	95.4%	97.2%	97.2%
Combinations	**Manifest + Plain**	95.4%	71.4%	84.0%	81.6%
	Manifest + XML	93.9%	66.8%	81.2%	78.1%
	Manifest + Hex	98.8%	95.7%	97.2%	97.2%

From Figure 5.4, we can observe at a glance that the hexadecimal file embeddings (standalone and concatenated) performed the best with all the metrics staying above 95% during both training and testing phases. The embeddings generated using manifest files and plain formatted dex files had similar performance metrics during both phases. The embeddings generated using XML formatted Dex file performed the worst and had an F1-score of mere 62.5% during the testing phase.



(a) PV-DM - Development



(b) PV-DM - Testing

Figure 5.4: PV-DM artefacts performance on SVM - AndroZoo

5.2.3 Logistic Regression

The results of developing and testing a Logistic Regression classification model on the research artefacts related to AndroZoo are presented in this subsection. The embeddings for the artefacts were generated using both PV-DBoW and PV-DM algorithms.

PV-DBoW

Referring Tables 5.9 and 5.10, we can observe that during the development phase, the artefact generated with plain formatted dex file had a 98.9% Precision whereas the concatenated version had a 99.2% Precision predicting the classes of the document embeddings during the development phase. Nevertheless, during the testing phase plain formatted dex file artefact and concatenated version had a 98.5% and 97.9% Precision values, respectively. The F1-score for the LR model trained on plain formatted dex file artefact fell from 98.3% in development to 97.6% in the testing phase. Its also noteworthy that during the development phase, XML formatted dex file and hexdump-based artefacts had similar performance with an F1-score of 96.3% and 96.4%, respectively. Concatenated form of XML formatted dex file and hexdump had 97.5% and 97.9% F1-score respectively, during the development, which came down to 94.7% and 96.1% respectively, during the testing phase.

Similarities to the previous two model metrics on PV-DBoW embeddings can also be observed in the Logistic Regression model results. The artefact created using manifest files performed poorly in PV-DBoW during both the development and testing phases. With Logistic regression, the plain formatted dex file artefact performed consistently better than the embeddings artefact made up of a hexadecimal dump of a dex file. The artefact created with hexadecimal dump worked a little better with the two models previously presented. Concatenated embeddings of manifest file and dex file in plain format had slightly better performance metrics than the standalone plain formatted dex file artefact during the development phase. However, during the test phase, the performance on unseen embeddings was the best with the plain formatted standalone dex file artefact, amongst all the other artefacts.

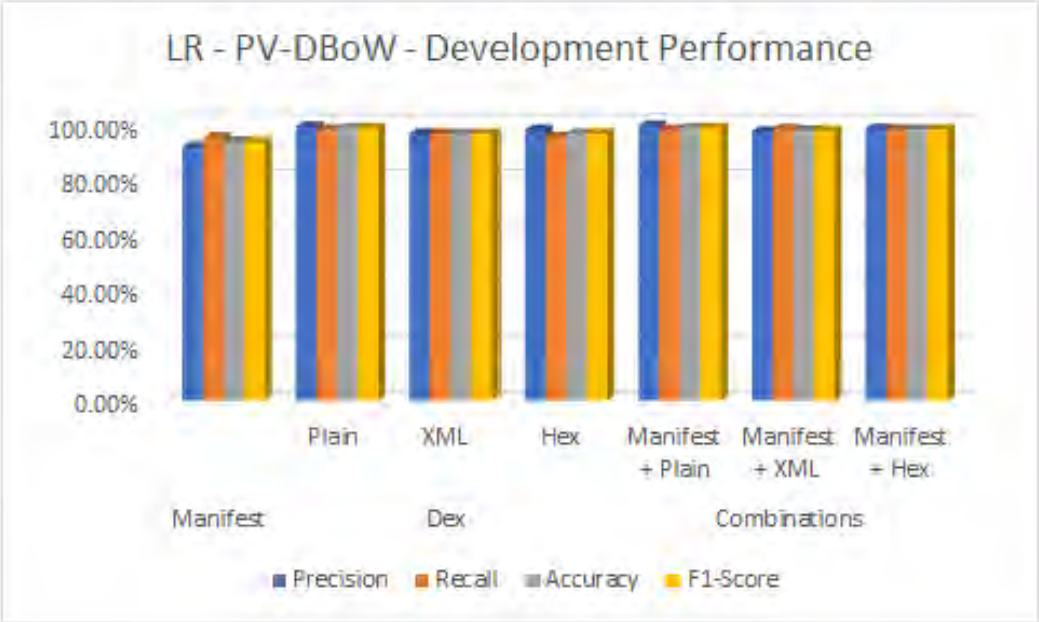
Table 5.9: Logistic Regression Model metrics on PV-DBoW - Development

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		91.6%	95.0%	93.5%	93.2%
Dex	**Plain**	98.9%	97.7%	98.3%	98.3%
	XML	96.3%	96.3%	96.2%	96.3%
	Hex	97.7%	95.1%	96.3%	96.4%
Combinations	**Manifest + Plain**	99.2%	97.7%	98.5%	98.4%
	Manifest + XML	97.1%	98.0%	97.5%	97.5%
	Manifest + Hex	98.0%	97.8%	97.8%	97.9%

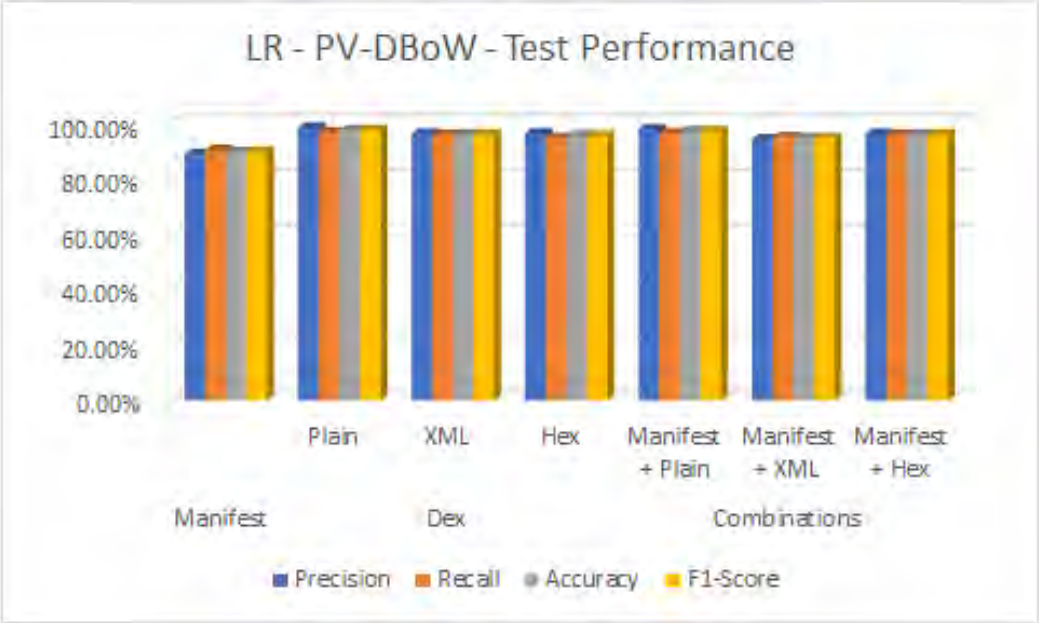
Table 5.10: Logistic Regression Model metrics on PV-DBoW - Test

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		89.0%	90.5%	89.7%	89.8%
Dex	**Plain**	98.5%	96.8%	97.7%	97.6%
	XML	96.2%	96.0%	96.1%	96.1%
	Hex	96.5%	94.8%	95.7%	95.6%
Combinations	**Manifest + Plain**	97.9%	96.5%	97.2%	97.2%
	Manifest + XML	94.3%	95.1%	94.7%	94.7%
	Manifest + Hex	96.2%	96.0%	96.1%	96.1%

In Figure 5.5 showing the comparative result of development and testing phases, we can clearly see that the performance of artefact created with plain formatted dex file was better than other artefacts. This was observed in both the standalone as well as concatenated forms. Throughout the development phase, we saw that the concatenated embeddings performed slightly better than standalone ones in general. However, this trend was not present when these artefacts were assessed in the testing phase. It is also noteworthy that the metrics of artefacts generated using manifest files stayed under 90% during the test phase and had the lowest performance metrics during the development phase amongst all the other artefacts.



(a) PV-DBoW - Development



(b) PV-DBoW - Testing

Figure 5.5: PV-DBoW artefacts performance on Logistic Regression - AndroZoo

PV-DM

Corresponding to the earlier trend seen in PV-DM algorithm embeddings generated with the AndroZoo dataset, the artefact generated using Hexadecimal file dumps performed the best with metrics of 96.7%, 95.6%, 96.1% and 96.1% Precision, Recall, Accuracy and F1-score respectively. None of the other artefacts' performance was comparable to that of the Hexadecimal dump artefact. As can be seen from Table 5.11 containing development

performance metrics, the only artefact that could compare and was actually a better performer than the standalone hexadecimal dump was the artefact created using hexadecimal dump concatenated with manifest file embeddings, with a very small margin. Their exact metrics were; 97.5%, 95.8%, 96.6% and 96.6% in Precision, Recall, Accuracy and F1-score, respectively.

Table 5.11: Logistic Regression Model metrics on PV-DM - Development

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		88.4%	76.1%	83.8%	81.8%
Dex	**Plain**	75.3%	74.3%	74.3%	74.8%
	XML	68.5%	65.8%	67.0%	67.1%
	Hex	96.7%	95.6%	96.1%	96.1%
Combinations	**Manifest + Plain**	90.1%	84.8%	87.8%	87.4%
	Manifest + XML	90.5%	78.8%	84.8%	84.3%
	Manifest + Hex	97.5%	95.8%	96.6%	96.6%

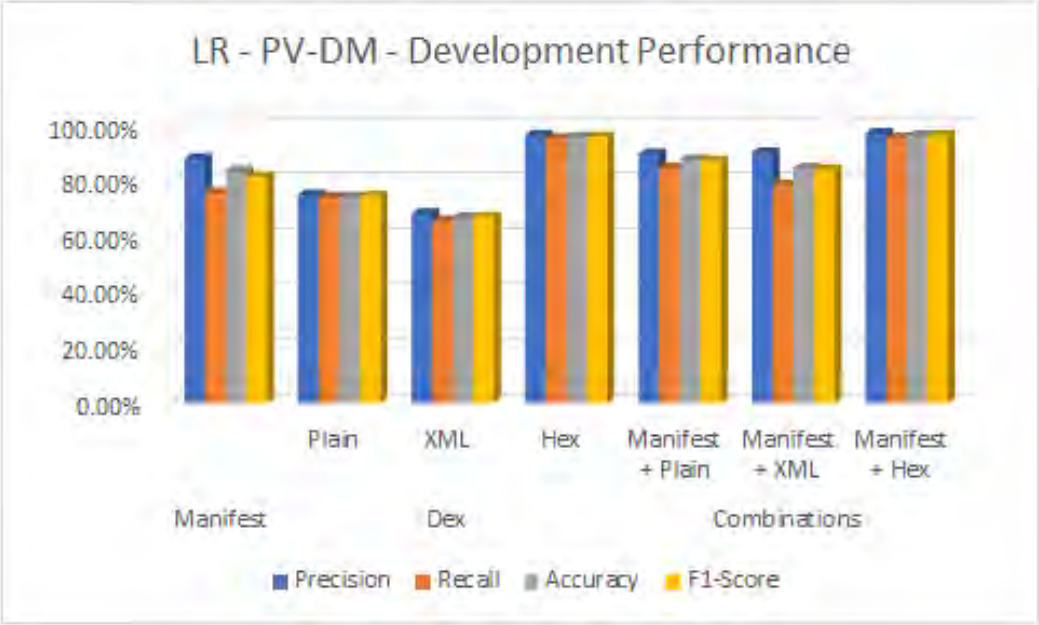
The standalone dex file embeddings of plain and XML format had the lowest performance metrics, even lower than the manifest file-based artefact. The XML formatted dex file artefact was the worst performing with an F1-score of 67.1% during development. During the testing, the plain formatted dex file artefact had a slightly lower F1-score and Recall than the XML formatted artefact, as can be seen from Table 5.12. It should be noted that the F1-score, Precision and Accuracy improved when the dex file embeddings were concatenated with Manifest file embeddings; there was over a 10% improvement in Accuracy and F1-score. Whereas Precision saw an improvement of above 20% for both the artefacts.

Table 5.12: Logistic Regression Model metrics on PV-DM - Test

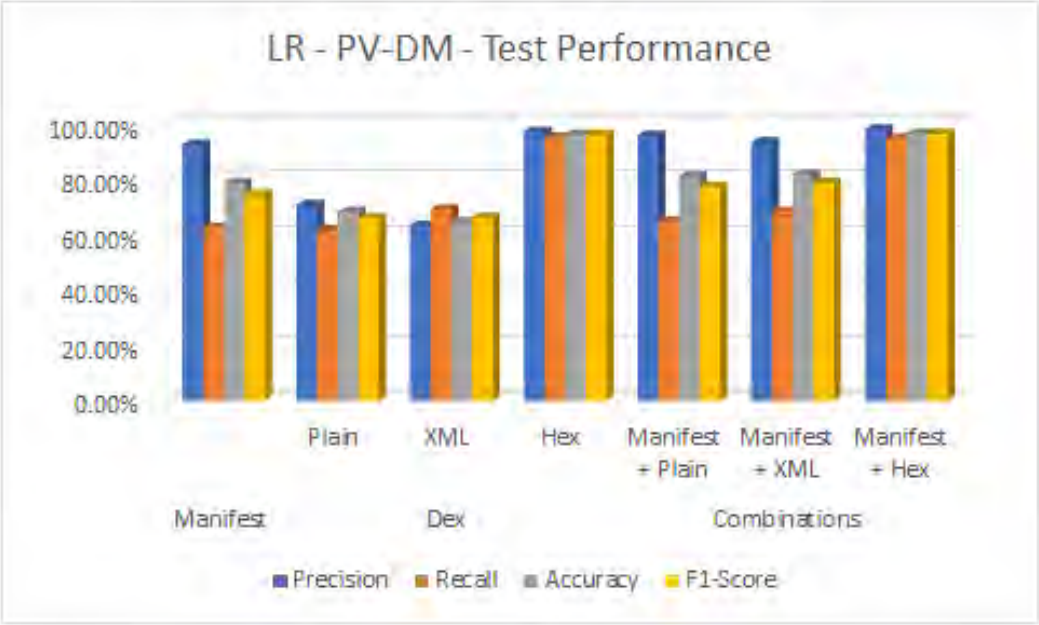
File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		92.4%	62.5%	78.7%	74.6%
Dex	**Plain**	70.8%	61.7%	68.1%	65.9%
	XML	63.3%	69.1%	64.5%	66.1%
	Hex	97.0%	95.1%	96.1%	96.1%
Combinations	**Manifest + Plain**	95.7%	64.8%	81.0%	77.3%
	Manifest + XML	93.3%	68.2%	81.7%	78.8%
	Manifest + Hex	97.9%	94.8%	96.4%	96.3%

From Figure 5.6 that contains the results of the development and testing of the Logistic Regression model against all the PV-DM artefacts, one can see that the artefact of hexadecimal outperformed every other artefact during both the development and testing phases. This behaviour can be seen with both standalone and concatenated forms of the

Hexadecimal filedump of the dex file. Manifest file performed the second best but the performance was not comparable to the performance of hexadecimal-based artefacts. Manifest file embeddings concatenated with various dex file output types provided better results when compared to standalone manifest or dex files, only exception being the hexadecimal filedump artefact.



(a) PV-DM - Development



(b) PV-DM - Testing

Figure 5.6: PV-DM artefacts performance on Logistic Regression - AndroZoo

5.2.4 k-Nearest Neighbors

The subsection discusses the performance metrics of the k-nearest neighbors classifier on our research artefacts. As done for the other models, we evaluate the research artefacts created using both PV-DM and PV-DBoW algorithms against the kNN models.

PV-DBoW

In comparison with the previously described three models, the performance of PV-DBoW embeddings with a kNN model was lower in general as can be seen from the Tables 5.13 and 5.14 below. Amongst the artefact generated, one generated using the manifest file has relatively low performance in both phases. Furthermore, during both the development and test phases, concatenated embeddings of manifest and dex files performed slightly better than the standalone dex file embeddings.

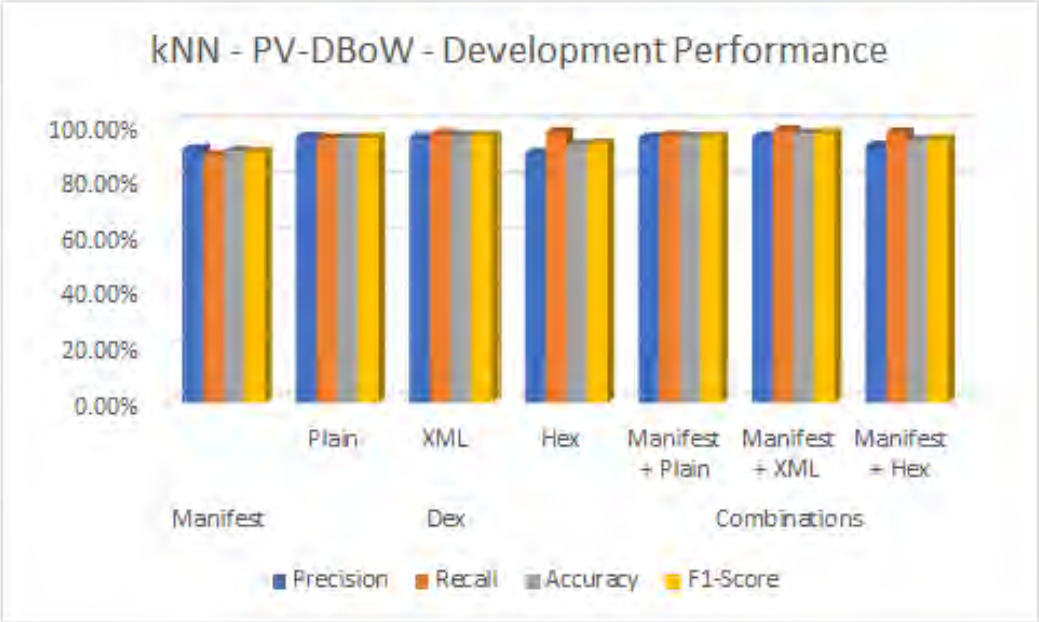
Going through the performance metrics data, the F1-score of the manifest file artefact has dropped by nearly 14% during the test phase. Recall and Accuracy also fell to 70.0% and 79.0%, respectively. On the other hand, the artefact created using a dex file in plain format had a similar F1-score during the development as well as testing phase.

Table 5.13: kNN Model metrics on PV-DBoW - Development

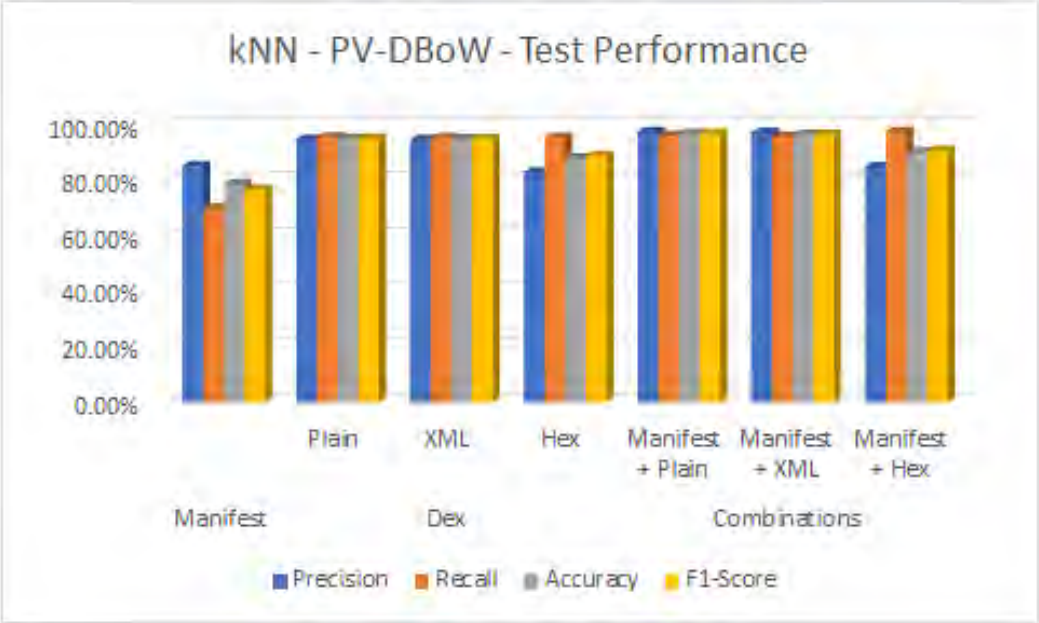
File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		91.4%	89.3%	90.8%	90.4%
Dex	**Plain**	95.8%	95.3%	95.5%	95.6%
	XML	95.5%	97.0%	96.2%	96.2%
	Hex	89.9%	97.5%	93.1%	93.6%
Combinations	**Manifest + Plain**	95.6%	96.3%	95.8%	96.0%
	Manifest + XML	96.0%	98.2%	97.1%	97.1%
	Manifest + Hex	92.6%	97.5%	94.7%	95.0%

Table 5.14: kNN Model metrics on PV-DBoW - Test

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		85.3%	70.0%	79.0%	76.9%
Dex	**Plain**	94.9%	95.7%	95.2%	95.3%
	XML	94.8%	95.4%	95.1%	95.1%
	Hex	83.1%	95.7%	88.1%	88.9%
Combinations	**Manifest + Plain**	97.6%	96.2%	97.0%	96.9%
	Manifest + XML	97.3%	96.0%	96.7%	96.6%
	Manifest + Hex	85.0%	97.7%	90.2%	90.9%



(a) PV-DBoW - Development



(b) PV-DBoW - Testing

Figure 5.7: PV-DBoW artefacts performance on k-Nearest Neighbors - AndroZoo

Analysing the comparative bar diagram of development and testing metrics represented in Figure 5.7, it is evident that the artefacts created with plain and XML formatted dex files had the most consistent performance in both the development and test phases. Similar results were seen for the concatenated form of these two artefacts as well. Only the artefacts created with the manifest and hexadecimal dump of the dex file were not following this trend.

PV-DM

Table 5.15: kNN Model metrics on PV-DM - Development

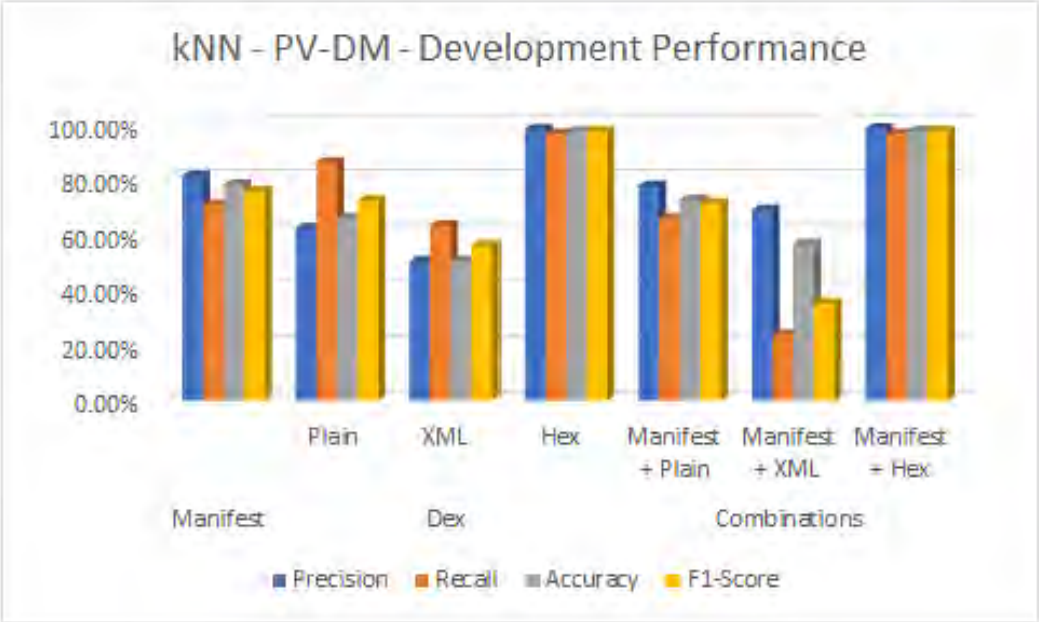
File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		81.1%	70.6%	78.0%	75.5%
Dex	**Plain**	62.2%	85.9%	65.8%	72.2%
	XML	50.2%	63.2%	50.3%	56.0%
	Hex	98.0%	96.3%	97.1%	97.1%
Combinations	**Manifest + Plain**	77.4%	65.8%	72.5%	71.2%
	Manifest + XML	68.8%	23.7%	56.5%	35.3%
	Manifest + Hex	98.2%	96.3%	97.2%	97.3%

The performance of the kNN model on embeddings generated using the PV-DM algorithm was extremely poor during both the development and testing phases. Only two out of seven artefacts had all their metrics above 95.0%; this includes F1-score, Accuracy, Recall and Precision score. This was the observation during the development phase, as seen from Table 5.16. The concatenated and standalone embeddings of XML formatted dex files fall behind in terms of their performance during both the development and testing phases.

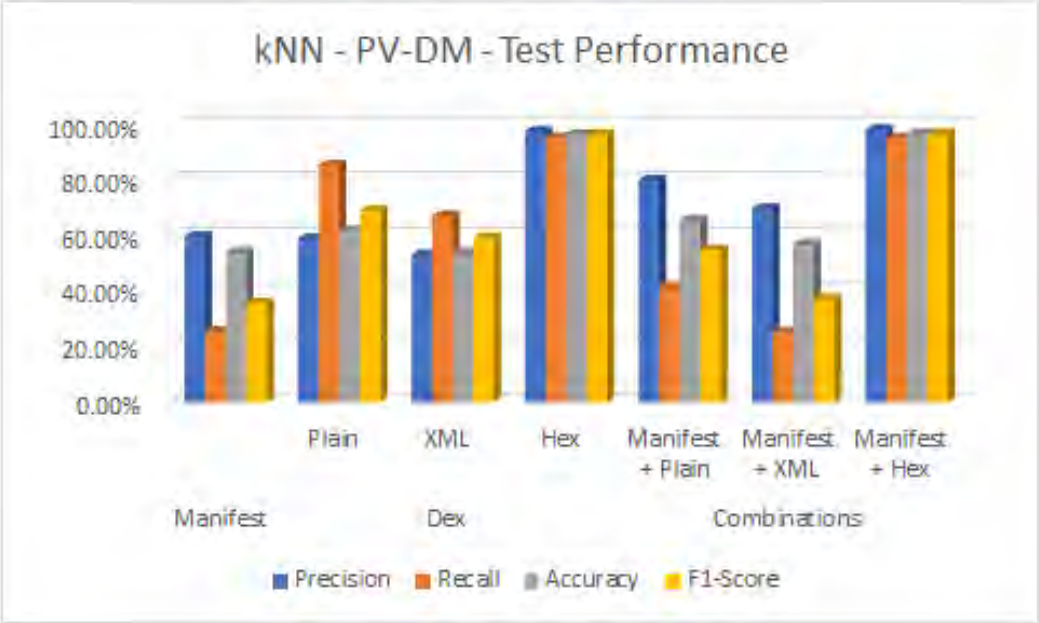
On the other hand, the performance of manifest file-based artefact plummeted during the testing phase, even though the Precision was 60.0% , the F1-score, Accuracy and Recall reduced to a score of mere 36.0%, 54.2% and 25.7% respectively. It should be noted that the embeddings created from the hexadecimal dump of *classes.dex* file and its concatenated form performed really well even though the other artefacts had a miserable performance, as we can see from Table 5.16.

Table 5.16: kNN Model metrics on PV-DM - Test

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		60.0%	25.7%	54.2%	36.0%
Dex	**Plain**	59.2%	85.7%	62.1%	69.3%
	XML	53.2%	67.4%	54.1%	59.5%
	Hex	97.9%	95.7%	96.8%	96.8%
Combinations	**Manifest + Plain**	80.3%	42.0%	65.8%	55.1%
	Manifest + XML	70.0%	25.4%	57.2%	37.3%
	Manifest + Hex	98.5%	95.4%	97.0%	96.9%



(a) PV-DM - Development



(b) PV-DM - Testing

Figure 5.8: PV-DM artefacts performance on k-Nearest Neighbors - AndroZoo

Observing Figure 5.8, one can safely say that only the embeddings created from the hexadecimal dump of the dex file, in both the standalone and concatenated form, performed well during the development phase as well as during testing with PV-DM embeddings. PV-DM artefact of XML formatted dex file has performed the poorest during the development, but the manifest artefacts performance fell further during the testing phase.

5.2.5 Decision Tree

This is the last binary classifier to be trained on the AndroZoo dataset. In this subsection, we discuss the performance metrics of a Decision tree classifier being trained and tested on our research artefacts that were listed in Table 3.1. After this, we train the classifiers mentioned in this section again, but this time against Android applications from the DREBIN dataset.

PV-DBoW

Observing the results present in Tables 5.17 and 5.18, we see that the performance metrics are a bit lower in general, when compared with the four previously discussed binary classifiers. During the development phase, the best performing artefact was the one created by concatenating the manifest file and XML formatted dexdump. Although, this was by a small margin compared to the artefact of concatenated manifest and plain formatted dexdump. In this phase, the worst metrics were observed for the artefacts generated using manifest files with a 83.5% Precision and 82.7% F1-Score As can be seen from the table 5.17, artefacts of plain and XML formatted dexdump stay similar in terms of performance with F1-Scores of 92.1% and 92.0%, respectively. Going through Table 5.18, we can see the performance metrics for all the artefacts that fall during the test phase. Accuracy of the decision tree model trained on plain formatted dex file artefact, standalone and concatenated, fell by over 6% and 8%, respectively, when compared to the metrics from the development phase. The Recall score for both of these artefacts also fell by 13%.

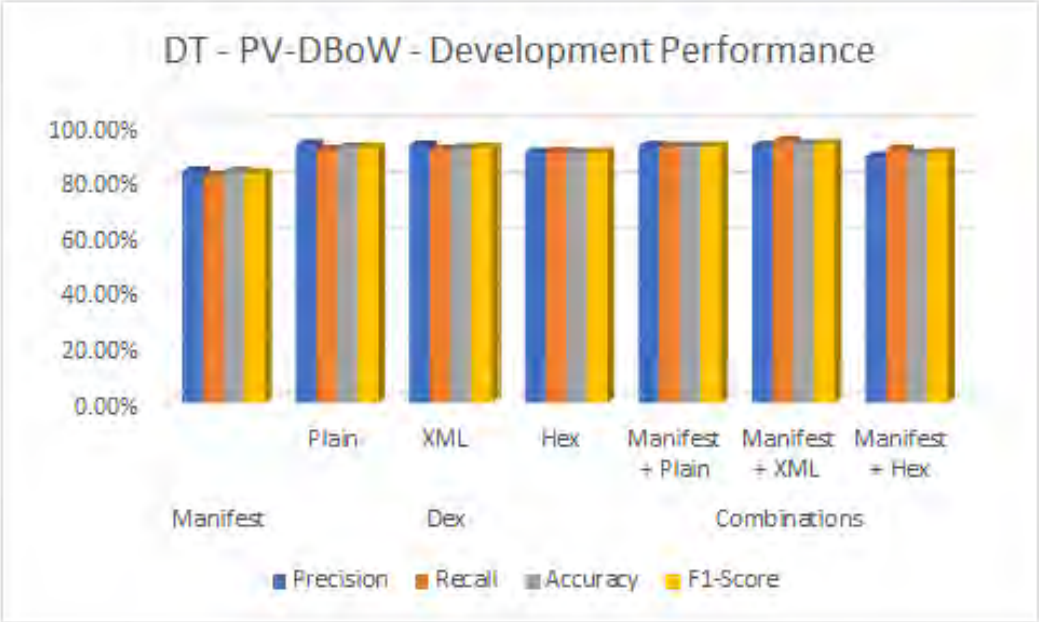
Table 5.17: Decision Trees Model metrics on PV-DBoW - Development

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		83.5%	82.0%	83.3%	82.7%
Dex	**Plain**	93.1%	91.1%	92.1%	92.1%
	XML	92.8%	91.2%	91.8%	92.0%
	Hex	90.1%	90.3%	90.1%	90.2%
Combinations	**Manifest + Plain**	92.5%	92.3%	92.3%	92.4%
	Manifest + XML	92.6%	94.4%	93.2%	93.5%
	Manifest + Hex	88.7%	91.3%	89.7%	90.0%

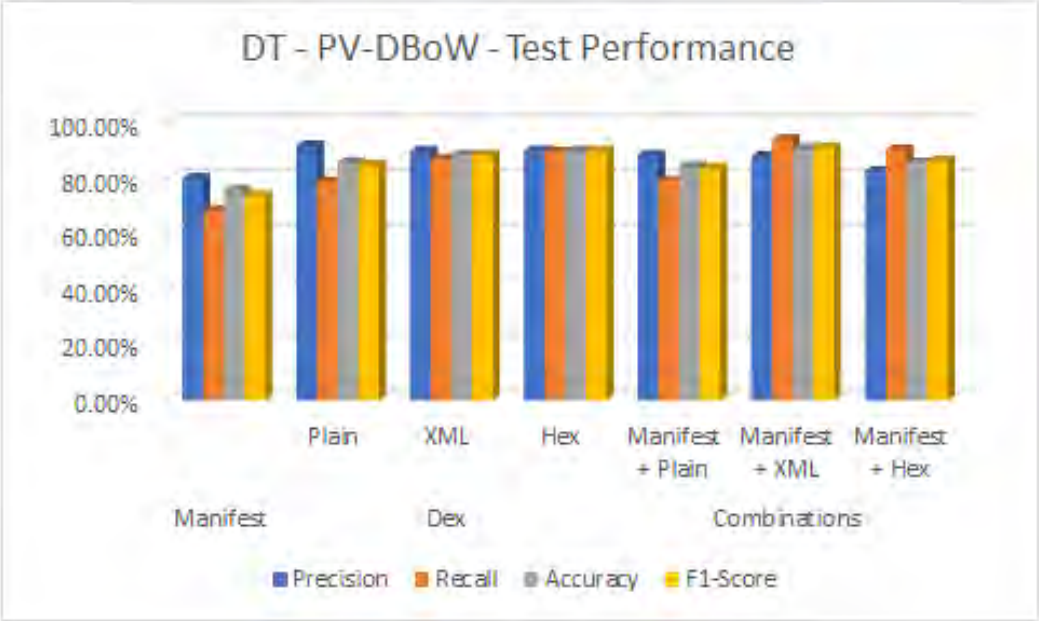
Table 5.18: Decision Trees Model metrics on PV-DBoW - Test

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		80.2%	68.5%	75.8%	73.9%
Dex	**Plain**	92.0%	78.8%	86.0%	84.9%
	XML	89.9%	87.1%	88.7%	88.5%
	Hex	90.2%	89.7%	90.0%	89.9%
Combinations	**Manifest + Plain**	88.5%	79.4%	84.5%	83.7%
	Manifest + XML	88.2%	94.2%	90.8%	91.1%
	Manifest + Hex	82.7%	90.5%	85.8%	86.4%

From Figure 5.9, we can compare the performances of the artefacts. We can see that artefact created using a manifest file had the lowest performance metrics in development, which fell further in the test phase. The performance metrics of artefacts created with XML formatted dex and hexdump of dex file stayed almost the same during the development and testing phases. Whereas Recall of artefacts created using plain, formatted dex file, standalone and concatenated, fell during testing with other metrics staying almost constant in both phases.



(a) PV-DBoW - Development



(b) PV-DBoW - Testing

Figure 5.9: PV-DBoW artefacts performance on Decision Trees - AndroZoo

PV-DM

We noticed in the PV-DBoW results section of the Decision Tree classifier that the binary classification models' performance at the task of malware detection was lower compared to other binary classifiers that were trained as part of this work. Similar observations were made in development and testing phases, using PV-DM algorithm based embeddings as can be seen from the Tables 5.19 and 5.20. The only two Decision Tree models with all their metrics above 90% were trained on hexdump-based artefacts in both the development and test phases.

Similar to what was observed in the kNN model and other previous models, the concatenated and standalone embeddings artefact of XML formatted dex file had the worst performance during both the development and testing phase. It has already been noted that the embeddings created from the hexadecimal dump of *classes.dex* file and its concatenated form with *AndroidManifest.xml* embeddings performed really well even though the other artefacts had a miserable performance.

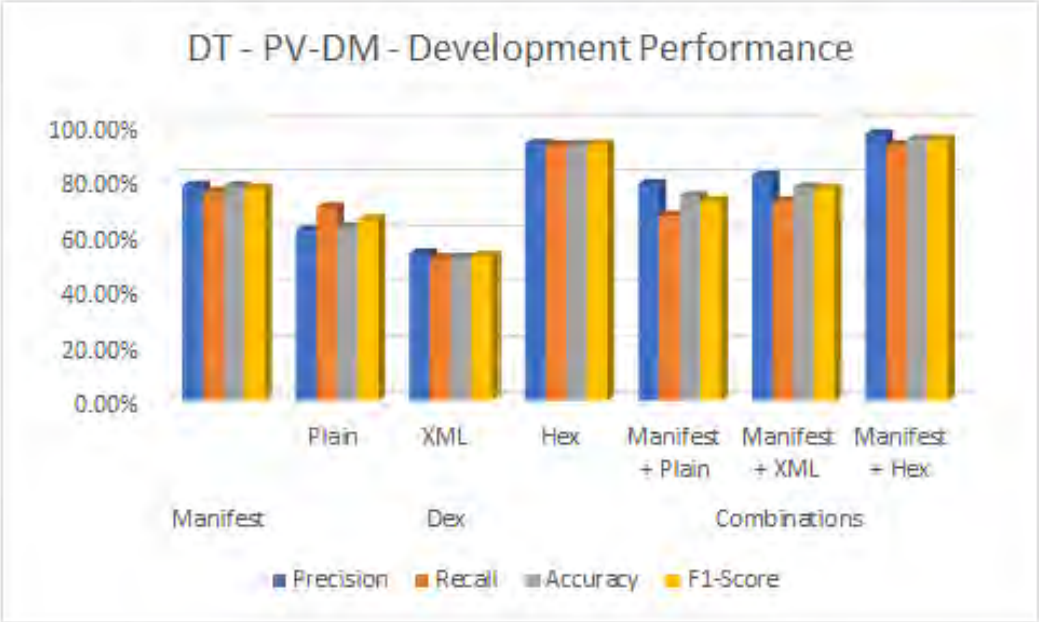
Table 5.19: Decision Trees Model metrics on PV-DM - Development

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		77.5%	75.3%	77.3%	76.4%
Dex	**Plain**	61.5%	69.7%	62.5%	65.3%
	XML	53.1%	51.3%	51.5%	52.2%
	Hex	92.5%	92.1%	92.2%	92.3%
Combinations	**Manifest + Plain**	78.0%	66.7%	73.6%	71.9%
	Manifest + XML	81.1%	71.9%	76.8%	76.2%
	Manifest + Hex	96.1%	92.1%	94.1%	94.0%

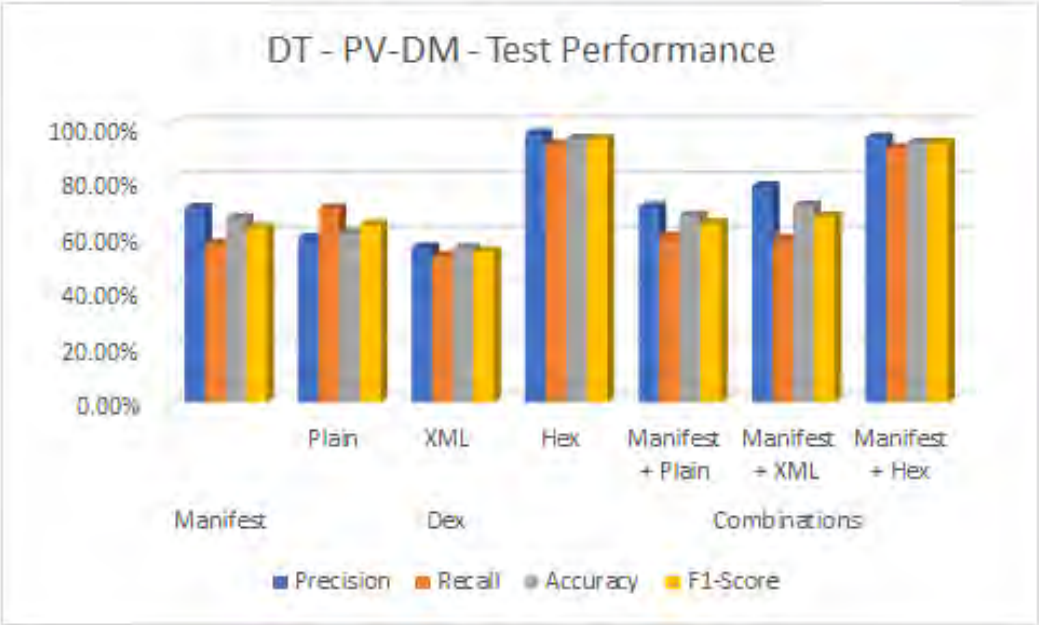
Table 5.20: Decision Trees Model metrics on PV-DM - Test

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		70.1%	57.1%	66.4%	62.9%
Dex	**Plain**	59.4%	70.0%	61.1%	64.3%
	XML	55.8%	52.8%	55.5%	54.3%
	Hex	97.0%	93.1%	95.1%	95.0%
Combinations	**Manifest + Plain**	70.6%	59.7%	67.4%	64.7%
	Manifest + XML	78.0%	58.8%	71.1%	67.1%
	Manifest + Hex	95.5%	91.7%	93.7%	93.5%

When comparing the development and test performance metrics of Decision Tree models trained on PV-DM embeddings, we can see that the artefacts created with manifest file, plain and XML formatted dex files performed equally bad in both the phases. The performance of model with manifest file artefact, fell only during test, the performance of plain and XML formatted dex file artefacts stayed poor consistently. We can also see from the Figure 5.10 that the artefacts with concatenated embeddings had a slightly better performance in development but this fell in testing phase. The performance with Decision Tree was low compared to all other previous models, but the hex performance was still much better than that of the other artefacts.



(a) PV-DM - Development



(b) PV-DM - Testing

Figure 5.10: PV-DM artefacts performance on Decision Trees - AndroZoo

5.3 Dataset #2 : DREBIN

In this section, we analyse each artefact produced using files taken from Android applications that are part of the DREBIN dataset. Similar to how the AndroZoo dataset was evaluated, document embeddings-based features (or artefacts) are evaluated for the goal of detecting Android malware. The experiments were carried out by obtaining the Android manifests and dexdump files for 4,700 Android APKs from Dataset 2 (in text, XML, and hexadecimal

forms). The files from the APKs were then divided into two groups of 4000 and 700 files each in the **training** and **test** sets, similar to how it was done with the applications from Dataset 1.

These files were used for each design cycle iteration to create 200-dimensional document embeddings from each file type using the PV-DBoW and PV-DM algorithms. The performance results of artefacts from the development phase have been shown, followed by performance results from the evaluation phase, where the malware detection models were given artefacts made up of unknown embeddings (i.e., **test** set) whose class the model had to guess. Additionally, for ease of comparison, we provide bar charts of binary classification model performance data on specific artefacts throughout the training and evaluation stages.

5.3.1 Convolutional Neural Networks

We begin the evaluation of our methods against the DREBIN dataset by presenting the outcomes of CNN model testing and development on embeddings produced by the PV-DBoW method. The outcomes of the development and testing procedure on embeddings made using the PV-DM algorithm are then presented. For a visual comparison of model performances during the development and testing phases, clustered bar charts of the performance indicators were added to the results tables.

PV-DBoW

All seven artefacts of the DREBIN dataset performed exceptionally well with Convolutional neural nets during the development phase with a performance score greater than 99%. As can be seen from Table 5.21, the concatenated embeddings of manifest and dex file (in Hex format) showed the best result as they achieved above 99.9% Precision, Recall, Accuracy and F1-score. Comparing the individual artefacts, the dex dump (hexadecimal format) based document embeddings are the only dataset whose performance score was in the range of 98% and hence can be considered as the poorest performer amongst all the other artefacts.

Table 5.21: CNN Model metrics on PV-DBoW - Development

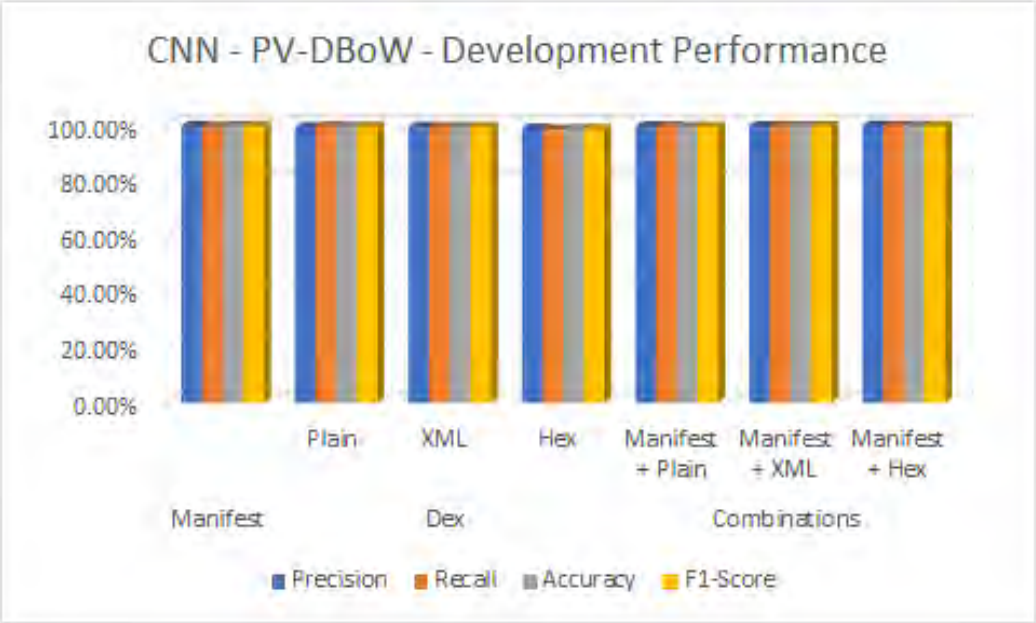
File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		99.8%	99.6%	99.9%	99.8%
Dex	**Plain**	99.4%	99.7%	99.6%	99.5%
	XML	99.6%	99.5%	99.5%	99.5%
	Hex	98.9%	98.6%	98.8%	98.8%
Combinations	**Manifest + Plain**	99.7%	99.7%	99.7%	99.7%
	Manifest + XML	99.8%	99.8%	99.8%	99.8%
	Manifest + Hex	99.9%	99.9%	99.9%	99.9%

On the other hand, we can see the variation in the performance of the artefacts in the testing phase. The artefact generated from the manifest file was one of the best-scoring performers as it received 99.7% Precision, 100% Recall, 99.8% Accuracy and 99.8% F1 score. Similarly, the concatenated embeddings of manifest and dex file (in XML format) also achieved outstanding results with 100% Precision and a 99.8% F1-score and Accuracy. One of the disappointing results was achieved by the embeddings created from the standalone dex file (in hexadecimal format) as its performance was significantly lowered during the testing phase resulting in the overall metric scores in the range of 76%-79% as can be observed from the Table 5.22.

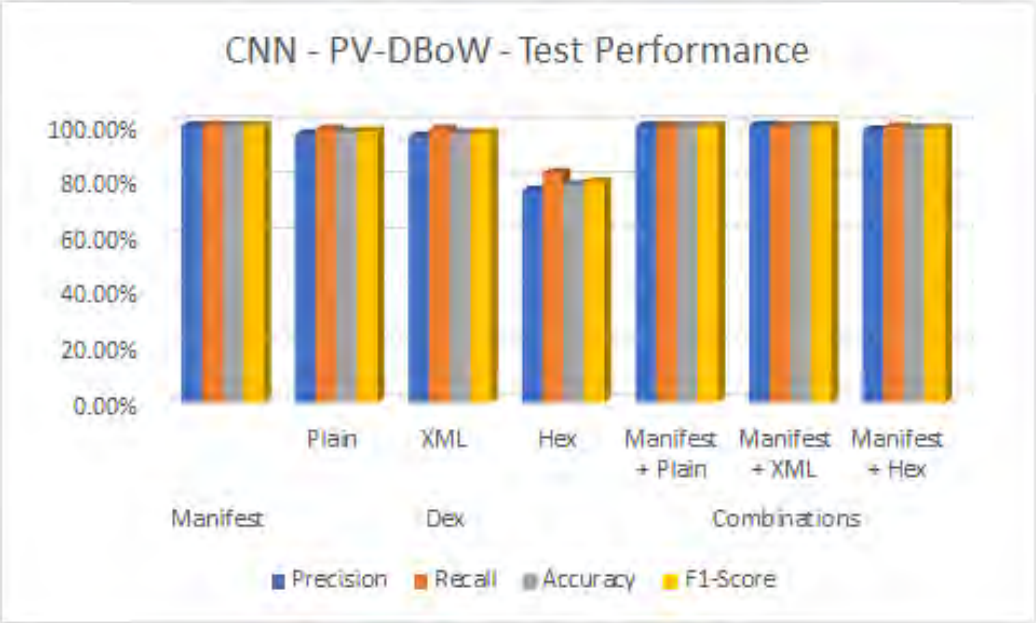
Table 5.22: CNN Model metrics on PV-DBoW - Test

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		99.7%	100.0%	99.8%	99.8%
Dex	**Plain**	96.9%	98.5%	97.7%	97.7%
	XML	96.1%	98.5%	97.2%	97.3%
	Hex	76.5%	82.8%	78.7%	79.5%
Combinations	**Manifest + Plain**	99.7%	99.7%	99.7%	99.7%
	Manifest + XML	100.0%	99.7%	99.8%	99.8%
	Manifest + Hex	98.3%	99.4%	98.8%	98.8%

Comparing the results of both the development and testing phase in Figure 5.11, it is pretty clear that the embeddings generated from the Dalvik executable file in hexadecimal format could not perform well in comparison with the other six artefacts. Manifest embeddings and concatenated embeddings performed consistently better.



(a) PV-DBoW - Development



(b) PV-DBoW - Testing

Figure 5.11: PV-DBoW artefacts performance on CNN - DREBIN

PV-DM

With the artefacts generated using the PV-DM algorithm, the performances were quite similar between the development and testing phases except for the XML formatted dex file, as can be noticed from Figure 5.12. During the development phase, manifest file embeddings achieved quite successful results with an F1-score of 96%. Similar results were achieved when the manifest file embeddings were concatenated with dex file embeddings. The combinations of

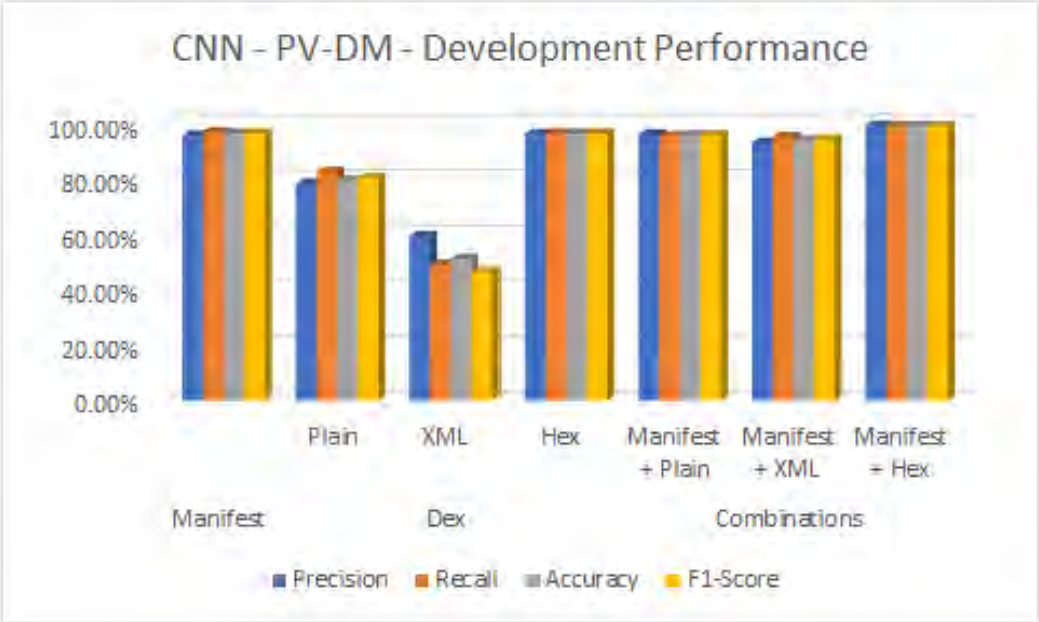
manifest and dex file (in hexadecimal format) embeddings got the significantly best result with 99.2% Precision, 98.9% Recall, 99.1% Accuracy and 99.0% F1-score. However, the standalone embeddings of the dex file in XML format performed the worst, achieving an F1-score of just 46.5%. All seven artefacts during the testing phase achieved similar trends of results. However, the XML embeddings achieved the lowest performance score of 9.7% as can be seen from the Tables 5.23 and 5.24.

Table 5.23: CNN Model metrics on PV-DM - Development

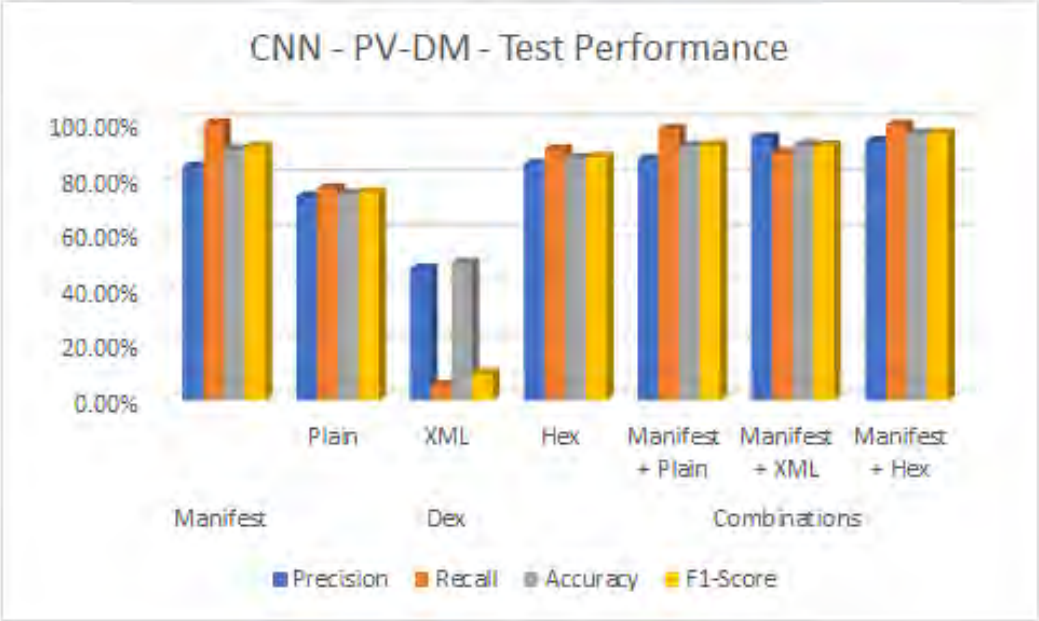
File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		95.7%	96.9%	96.2%	96.3%
Dex	**Plain**	78.3%	82.4%	79.4%	80.3%
	XML	59.4%	49.1%	51.1%	46.5%
	Hex	96.1%	96.4%	96.2%	96.3%
Combinations	**Manifest + Plain**	96.1%	95.4%	95.7%	95.7%
	Manifest + XML	93.0%	95.2%	93.8%	94.1%
	Manifest + Hex	99.2%	98.9%	99.1%	99.0%

Table 5.24: CNN Model metrics on PV-DM - Test

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		84.3%	100.0%	90.7%	91.5%
Dex	**Plain**	73.6%	76.5%	74.5%	75.0%
	XML	47.5%	5.4%	49.7%	9.7%
	Hex	85.2%	90.5%	87.4%	87.8%
Combinations	**Manifest + Plain**	87.0%	98.0%	91.7%	92.2%
	Manifest + XML	94.8%	89.4%	92.2%	92.0%
	Manifest + Hex	93.5%	99.4%	96.2%	96.4%



(a) PV-DM - Development



(b) PV-DM - Testing

Figure 5.12: PV-DM artefacts performance on CNN - DREBIN

5.3.2 Support Vector Machines

The performance of the research artefacts on a Support Vector Machines classifier is demonstrated in this subsection. During the creation and testing of the SVM model, we tallied the findings and created comparative bar charts of the key performance indicators. This was carried out for embeddings produced by the PV-DM and PV-DBoW algorithms.

PV-DBoW

Using the SVM model to analyse the findings during the development phase, it is evident that all of the embeddings performed well, except for the hexadecimal dump-based document embeddings, which received 91.5% Precision and an 88.4% F1 score. Although the Manifest and XML file document embeddings had 100% Recall, their performance was still in the 99% range. On the other hand, concatenated embeddings did quite well in the SVM model, especially the combination of Manifest and Plain embeddings and Manifest and XML embeddings, which received 100% Precision and 100% Recall 100% Accuracy and 100% F1 score as can be seen from the Table 5.25.

Table 5.25: SVM Model metrics PV-DBoW - Development

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		99.7%	100.0%	99.8%	99.8%
Dex	**Plain**	98.0%	99.4%	98.7%	98.7%
	XML	98.0%	100.0%	99.0%	99.0%
	Hex	91.5%	85.5%	89.0%	88.4%
Combinations	**Manifest + Plain**	100.0%	100.0%	100.0%	100.0%
	Manifest + XML	100.0%	100.0%	100.0%	100.0%
	Manifest + Hex	98.9%	99.7%	99.3%	99.3%

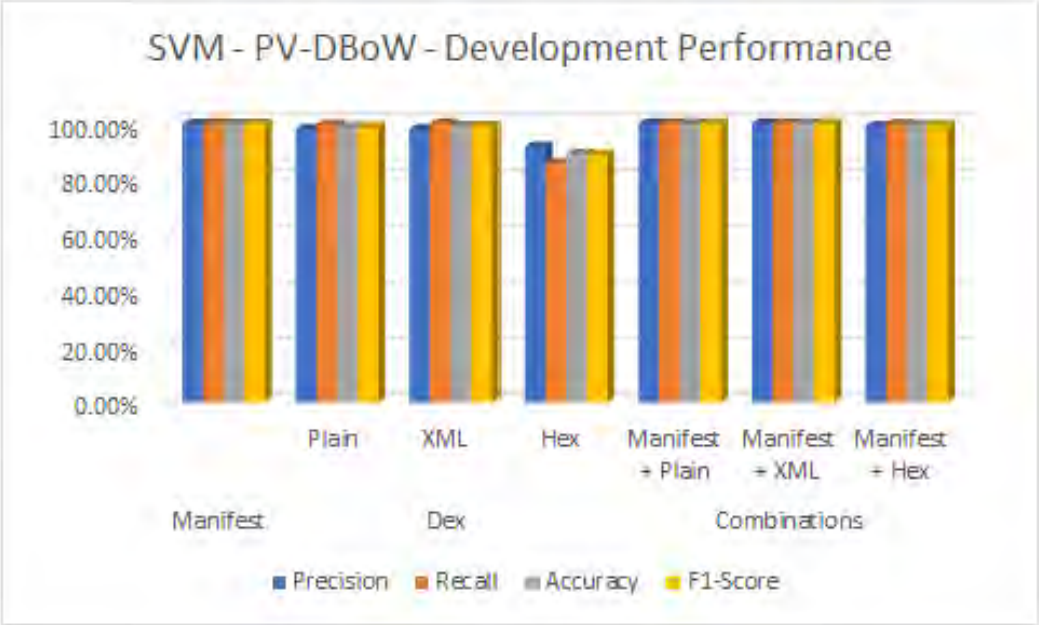
During the testing phase, a similar pattern of outcomes is noticed from the metrics presented in Table 5.26. Except for the concatenated embeddings in Plain format, the performance metrics of the other six artefacts decreased slightly. With only 81.6% of the F1 score, standalone hexadecimal files performed the worst. The best outcome in the test phase was also the concatenated file embeddings in Plain format, which received a perfect score of 100%.

Table 5.26: SVM Model metrics PV-DBoW - Test

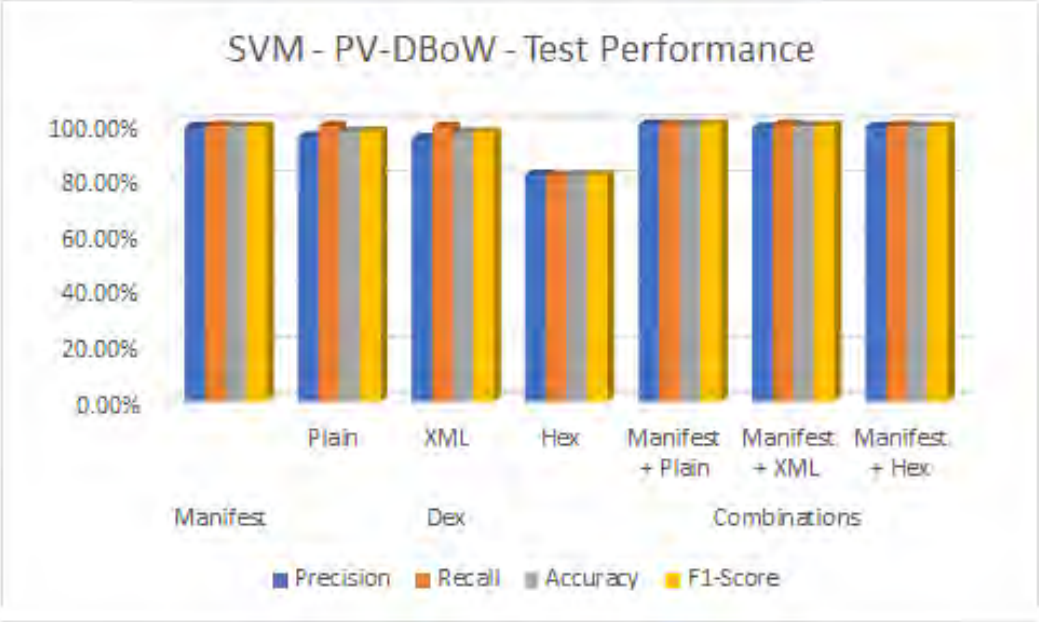
File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		99.1%	99.4%	99.2%	99.2%
Dex	**Plain**	95.6%	99.4%	97.4%	97.4%
	XML	95.0%	99.1%	97.0%	97.0%
	Hex	81.8%	81.4%	81.7%	81.6%
Combinations	**Manifest + Plain**	100.0%	100.0%	100.0%	100.0%
	Manifest + XML	98.8%	100.0%	99.4%	99.4%
	Manifest + Hex	99.1%	99.4%	99.2%	99.2%

From Figure 5.13, we can observe that compared to the development phase, the concatenated embeddings of Manifest and Dex files in XML format performed slightly lower

during the test phase. Overall, the SVM model performed well with Manifest file embeddings and concatenated file embeddings in Plain and Hex formats, achieving above 99% in all four performance metrics.



(a) PV-DBoW - Development



(b) PV-DBoW - Testing

Figure 5.13: PV-DBoW artefacts performance on SVM - DREBIN

PV-DM

The SVM model's performance on PV-DM embeddings was a little disappointing. During the development phase, the Manifest file embeddings worked admirably, with a Precision of

97.4%, a Recall of 97.6%, an Accuracy of 97.6%, and an F1 score of 97.5% as can be seen from Table 5.27. On the other hand, standalone file embeddings in Plain and XML format fared substantially worse. The standalone Dex file embeddings in XML format had the worst results, with an F1 score of only 62.9%.

Table 5.27: SVM Model metrics PV-DM - Development

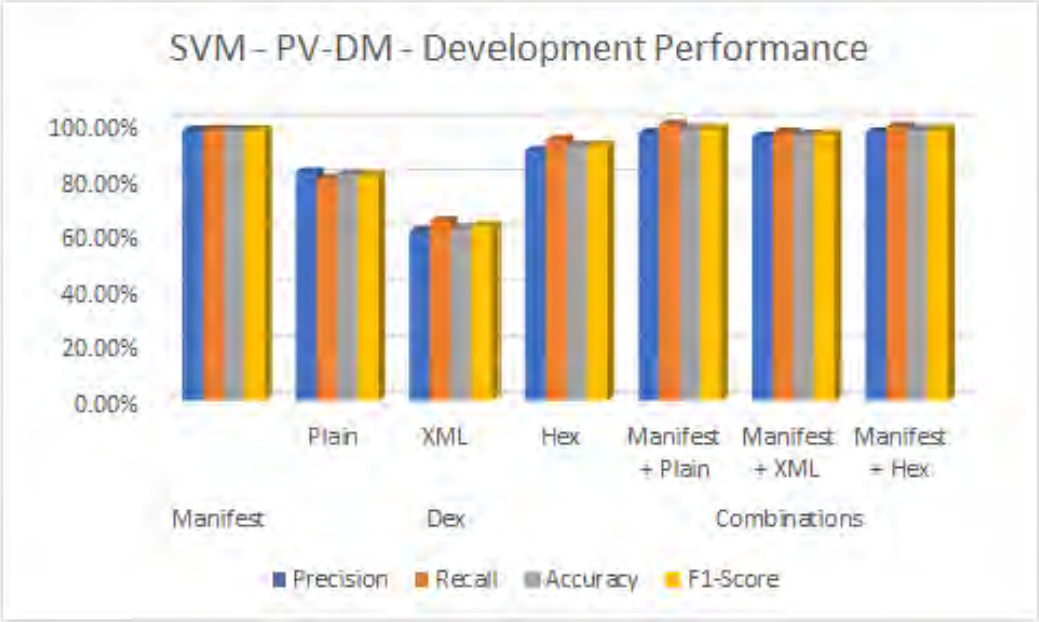
File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		97.4%	97.6%	97.6%	97.5%
Dex	**Plain**	82.2%	80.0%	81.6%	81.1%
	XML	61.1%	64.7%	62.1%	62.9%
	Hex	90.0%	93.9%	91.8%	91.9%
Combinations	**Manifest + Plain**	96.5%	99.4%	98.0%	98.0%
	Manifest + XML	95.2%	96.7%	96.0%	96.0%
	Manifest + Hex	96.7%	98.7%	97.7%	97.7%

Table 5.28 shows that the standalone file embeddings in Plain and XML format continually performed poorly even during the test phase. The performance score of the Dex file in Hex format improved by 2% in the testing phase compared to the development phase, despite the fact that it was the least performing of the seven embeddings with a score of 64.3%. The concatenated file embeddings of Hex format achieved one of the best outcomes, with all four components receiving a score of more than 95%, resulting in an overall F1 score of 97.4%.

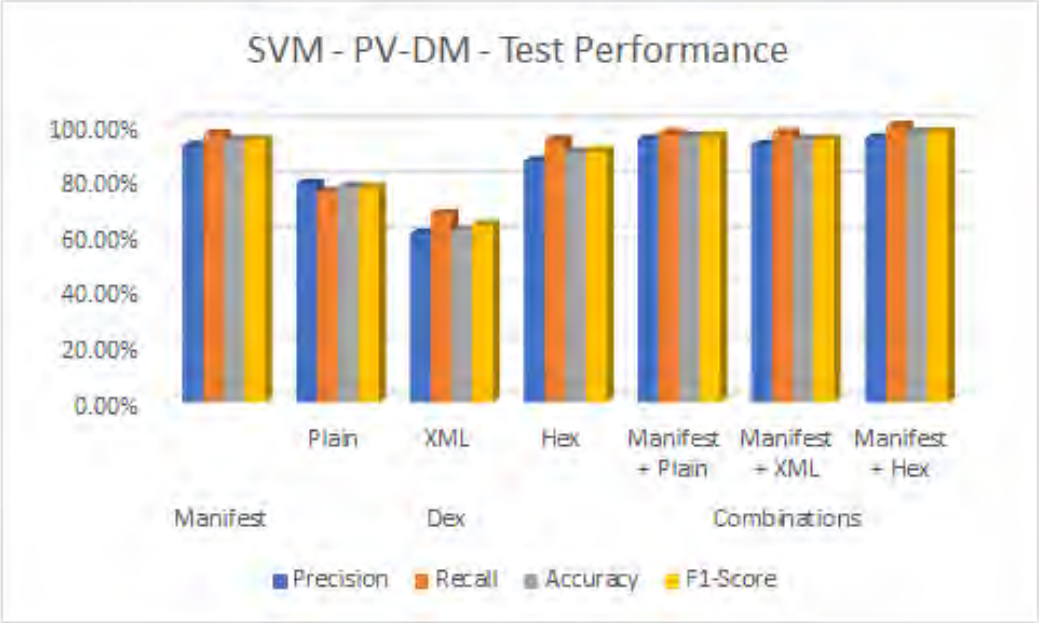
Table 5.28: SVM Model metrics PV-DM - Test

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		92.6%	96.8%	94.5%	94.6%
Dex	**Plain**	78.9%	76.0%	77.8%	77.4%
	XML	61.0%	68.0%	62.2%	64.3%
	Hex	87.1%	94.5%	90.2%	90.6%
Combinations	**Manifest + Plain**	94.9%	97.1%	96.0%	96.0%
	Manifest + XML	92.8%	97.1%	94.8%	94.9%
	Manifest + Hex	95.3%	99.7%	97.4%	97.4%

In both the development and testing phases, all of the concatenated file embeddings worked remarkably well, as seen in Figure 5.14's comparison bar diagram. Similarly, the standalone Manifest file embeddings scored pretty well in terms of performance.



(a) PV-DM - Development



(b) PV-DM - Testing

Figure 5.14: PV-DM artefacts performance on SVM - DREBIN

5.3.3 Logistic Regression

This subsection presents the findings from developing and testing a Logistic Regression classification model on the research artefacts associated with DREBIN. PV-DBoW and PV-DM algorithms were used to create the artefact embeddings.

PV-DBoW

The performance of the artefacts in the Logistic Regression model was very close to the Support Vector Machine model. According to Table 5.29, concatenated embeddings in Plain and XML format performed extraordinarily well, receiving a top score. Similarly, the Manifest file embeddings and the Dex file embeddings in Plain and XML format worked well, scoring above 97% in all four components. On the other hand, hexadecimal embeddings had a variable score, with 93.7% Precision and only 70.2% Recall, resulting in an overall performance score of only 80.3%, making it the least performing file.

Table 5.29: Logistic Regression Model metrics on PV-DBoW - Development

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		99.7%	100.0%	99.8%	99.8%
Dex	**Plain**	98.3%	98.5%	98.3%	98.4%
	XML	97.8%	98.7%	98.2%	98.3%
	Hex	93.7%	70.2%	81.7%	80.3%
Combinations	**Manifest + Plain**	100.0%	100.0%	100.0%	100.0%
	Manifest + XML	100.0%	100.0%	100.0%	100.0%
	Manifest + Hex	99.7%	99.7%	99.7%	99.7%

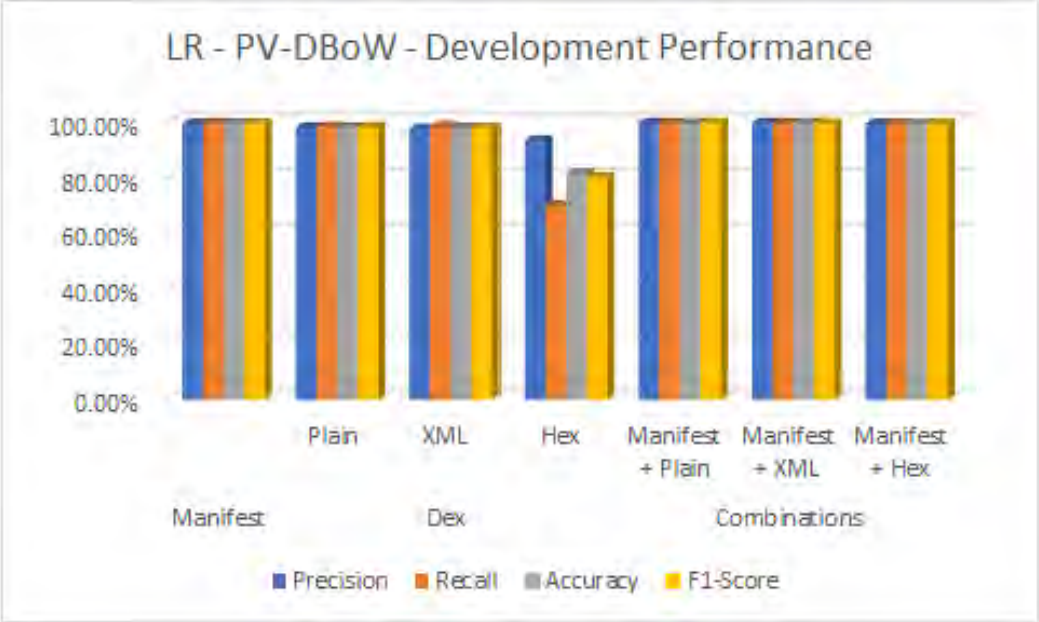
The concatenated file embeddings in all three formats outperformed the Manifest embeddings and the independent Dex file embeddings during the testing phase. A consistent result of 99.7% was obtained for the combination of Manifest and Plain file embeddings. Regarding the standalone file embedding artefacts, the Hexadecimal files did not perform as well as expected, with just 89.8% Precision, 70.8% Recall, 81.4% Accuracy, and a 79.2% F1 score.

Table 5.30: Logistic Regression Model metrics on PV-DBoW - Test

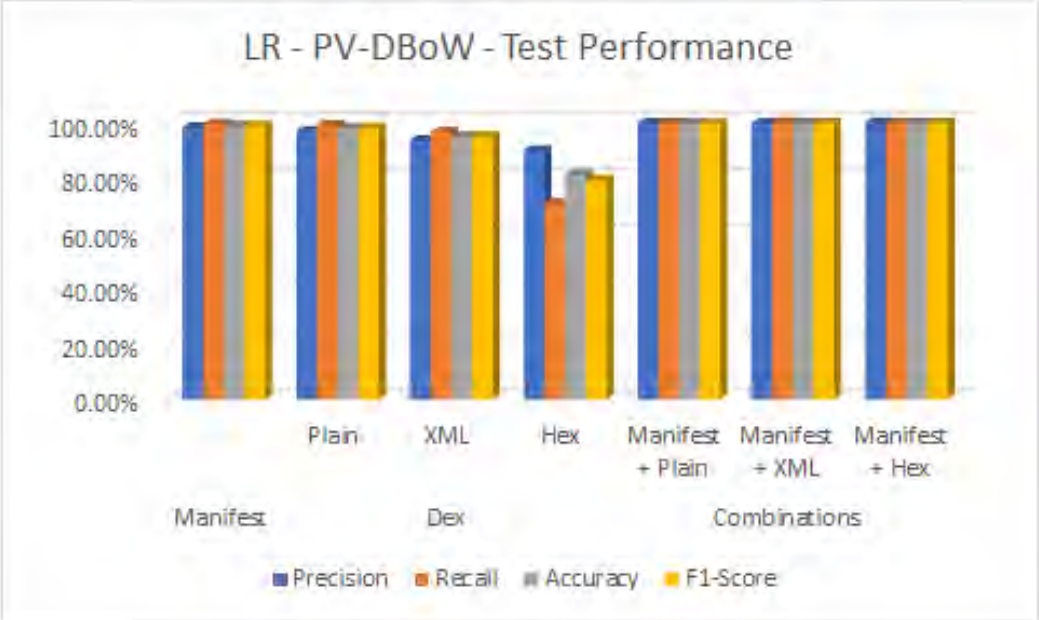
File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		98.3%	99.4%	98.8%	98.8%
Dex	**Plain**	96.9%	99.1%	98.0%	98.0%
	XML	93.6%	96.5%	95.0%	95.0%
	Hex	89.8%	70.8%	81.4%	79.2%
Combinations	**Manifest + Plain**	99.7%	99.7%	99.7%	99.7%
	Manifest + XML	99.7%	100.0%	99.8%	99.8%
	Manifest + Hex	100.0%	99.7%	99.8%	99.8%

In both the development and test phases, the Logistic Regression model did not operate well for standalone Hexadecimal file embeddings, as shown in Figure 5.15. The other

six artefacts performed exceptionally well for the PV-DBoW algorithms with a slight change in performance score suring testing.



(a) PV-DBoW - Development



(b) PV-DBoW - Testing

Figure 5.15: PV-DBoW artefacts performance on Logistic Regression - DREBIN

PV-DM

During the development phase, the Dex file embeddings in XML format performed the worst, with a performance score of only 63.2%. Compared to other artefacts, even plain file embeddings had a significantly reduced range of scores. The combination of Manifest and

Plain file embeddings obtained the highest score, with Precision, Recall, Accuracy, and F1 scores consistently above 97%.

Table 5.31: Logistic Regression Model metrics on PV-DM - Development

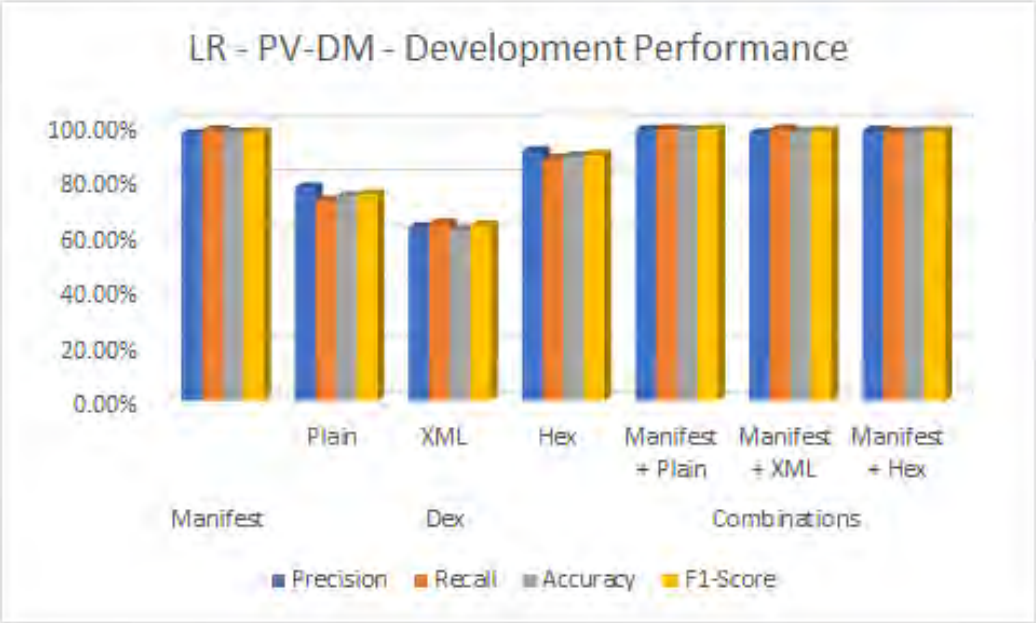
File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		96.0%	97.5%	96.7%	96.8%
Dex	**Plain**	76.8%	72.1%	73.7%	74.4%
	XML	62.5%	63.9%	61.8%	63.2%
	Hex	90.0%	87.2%	88.1%	88.6%
Combinations	**Manifest + Plain**	97.6%	97.8%	97.6%	97.7%
	Manifest + XML	96.3%	97.8%	97.0%	97.0%
	Manifest + Hex	97.3%	96.9%	97.0%	97.1%

Table 5.32 shows that the standalone file embeddings in Plain and XML format continuously performed poorly, particularly in the XML format, where the overall score dropped to 59.2%. The concatenated file embeddings in Hex format performed exceptionally well, with a Recall of 98.8%, and their performance score was better in the testing phase compared to the development phase. Hexadecimal file embedding performance was neither poor nor excellent, as they got 88.6% and 87.1% during the development and testing phases, respectively.

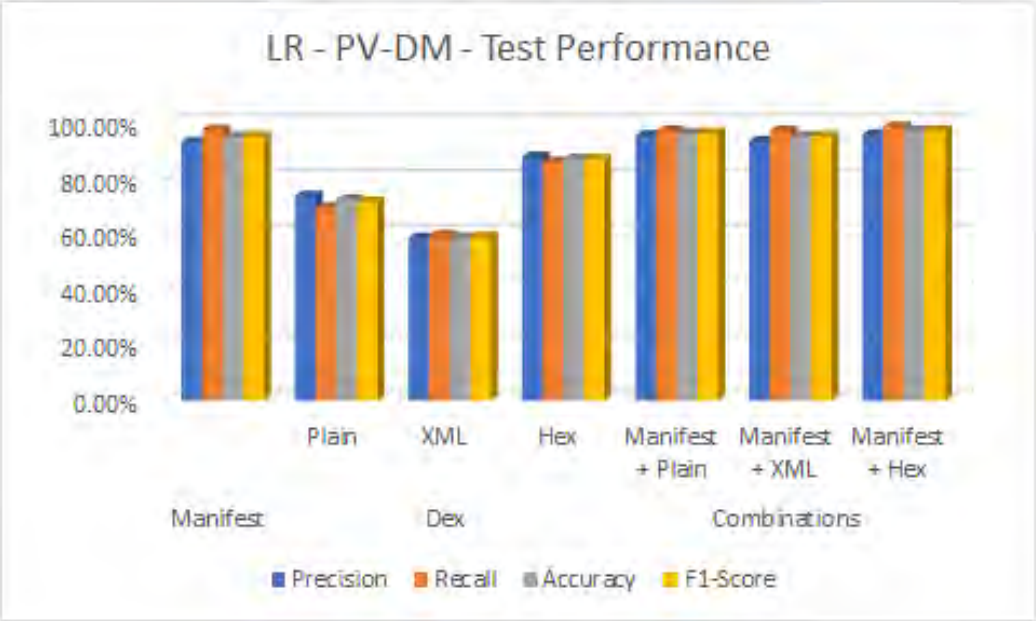
Table 5.32: Logistic Regression Model metrics on PV-DM - Test

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		93.4%	97.7%	95.4%	95.5%
Dex	**Plain**	73.9%	69.7%	72.5%	71.7%
	XML	58.7%	59.7%	58.8%	59.2%
	Hex	88.0%	86.2%	87.2%	87.1%
Combinations	**Manifest + Plain**	95.7%	97.4%	96.5%	96.6%
	Manifest + XML	93.6%	97.4%	95.4%	95.5%
	Manifest + Hex	96.1%	98.8%	97.4%	97.4%

We found that the artefacts generated utilising concatenated embeddings performed better in both the development and testing stages when employing the PV-DM algorithms. Aside from that, the performance of manifest file embeddings was also notably well.



(a) PV-DM - Development



(b) PV-DM - Testing

Figure 5.16: PV-DM artefacts performance on Logistic Regression - DREBIN

5.3.4 k-Nearest Neighbors

The performance metrics of the k-nearest neighbour's classifier on our research artefacts are covered in the following sections. We compare the research artefacts produced by the PV-DM and PV-DBoW algorithms to the kNN models, just as we did for the other models.

PV-DBoW

If we compare the performance of binary classifiers developed and tested so far on the DREBIN dataset, we can see that the performance of PV-DBoW embeddings with the k-Nearest Neighbors model is slightly lower than that of earlier classifiers. From Table 5.33 that contains the performance metrics from the development phase, we can see that the best performance metrics for a standalone artefact was in the case of Manifest file embeddings with a 98.0%, 99.7%, 98.8% and 98.8% Precision, Recall, Accuracy and F1-score respectively. For concatenated embeddings artefacts, the best performance metrics were 99.2%, 99.7%, 99.5%, and 99.4% Precision, Recall, Accuracy and F1-score, respectively, with the artefact made up of Manifest and XML formatted dex file.

Table 5.33: k-Nearest Neighbors Model metrics on PV-DBoW - Development

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		98.0%	99.7%	98.8%	98.8%
Dex	**Plain**	97.1%	98.1%	97.5%	97.6%
	XML	97.9%	97.9%	98.0%	97.9%
	Hex	95.4%	74.4%	84.6%	83.6%
Combinations	**Manifest + Plain**	98.8%	99.0%	98.8%	98.9%
	Manifest + XML	99.2%	99.7%	99.5%	99.4%
	Manifest + Hex	96.9%	98.3%	97.5%	97.6%

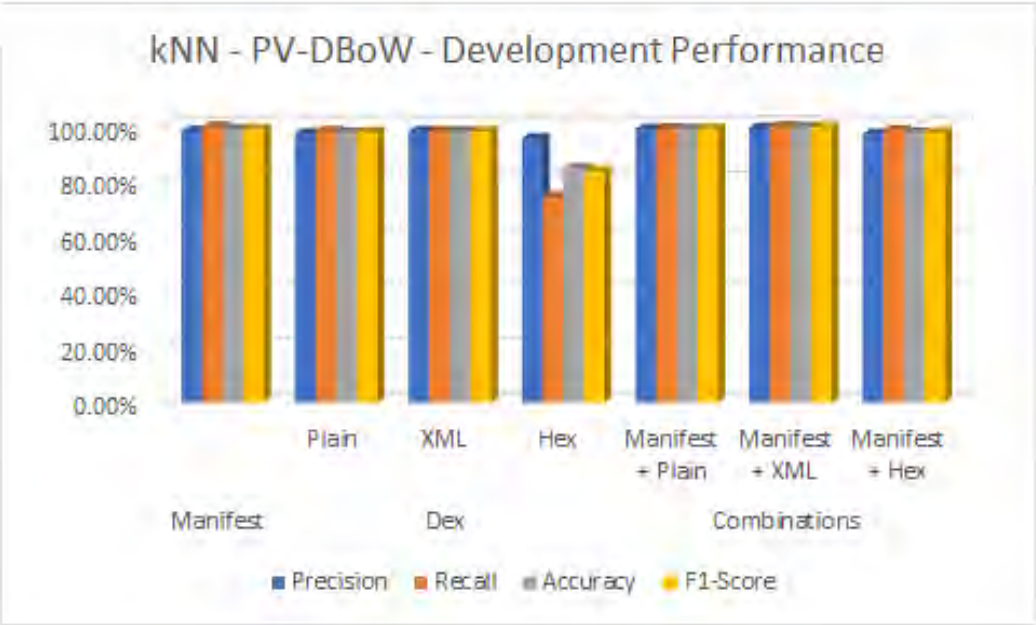
During the testing phase, the performance metric of the F1-score was lowered, for all the artefacts, by a very thin margin, as can be seen from Table 5.34 containing the test results. Overall the model performance did not degrade much during the testing. We notice that the two artefacts of the standalone Manifest file concatenated with XML formatted dex file, which also performed well during development, had a perfect Recall score of 100%.

Table 5.34: k-Nearest Neighbors Model metrics on PV-DBoW - Test

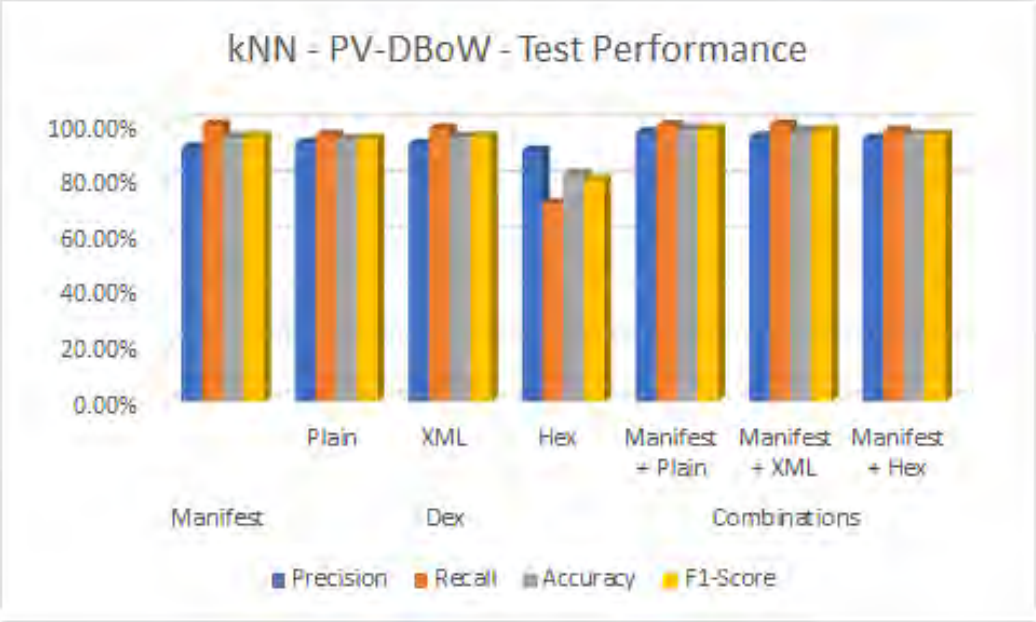
File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		91.8%	100.0%	95.5%	95.7%
Dex	**Plain**	93.3%	96.2%	94.7%	94.7%
	XML	92.9%	98.5%	95.5%	95.7%
	Hex	90.6%	71.7%	82.1%	80.0%
Combinations	**Manifest + Plain**	97.2%	99.7%	98.4%	98.4%
	Manifest + XML	95.6%	100.0%	97.7%	97.7%
	Manifest + Hex	95.0%	97.7%	96.2%	96.3%

From the below Figure 5.17, we can observe that the results during the development and testing phases appear to stay consistent. The performance metrics against the concatenated

embedding artefacts were slightly better than the ones against standalone embedding artefacts. The only poor performance was seen with the artefact created using the hexadecimal formatted dex file dump.



(a) PV-DBoW - Development



(b) PV-DBoW - Testing

Figure 5.17: PV-DBoW artefacts performance on k-Nearest Neighbors - DREBIN

PV-DM

By observing the results from the Tables 5.35 and 5.36 we can safely say that the performance metrics of the kNN model on PV-DM algorithm-based embeddings were inferior to those seen

with PV-DBoW algorithm based embeddings. The results table shows that the Manifest file and its concatenated form with Hexadecimal dump were the only two artefacts with all their metrics above 90%. The performance metrics of the artefact made up of standalone embeddings of Hexadecimal dump of dex file were not all above 90%. However, they still performed better than the remaining four artefacts.

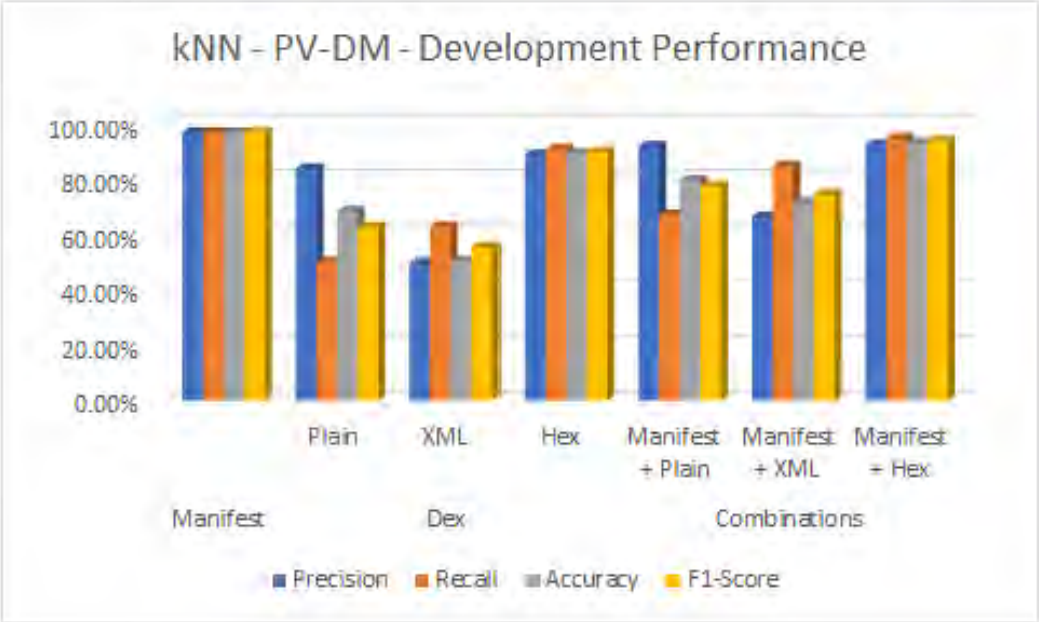
Table 5.35: k-Nearest Neighbors Model metrics on PV-DM - Development

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		96.7%	96.7%	96.7%	96.7%
Dex	**Plain**	83.7%	50.0%	68.5%	62.6%
	XML	49.6%	62.8%	50.5%	55.5%
	Hex	88.8%	90.7%	89.1%	89.8%
Combinations	**Manifest + Plain**	91.8%	67.0%	79.5%	77.5%
	Manifest + XML	66.4%	84.7%	71.5%	74.4%
	Manifest + Hex	92.3%	94.7%	93.1%	93.5%

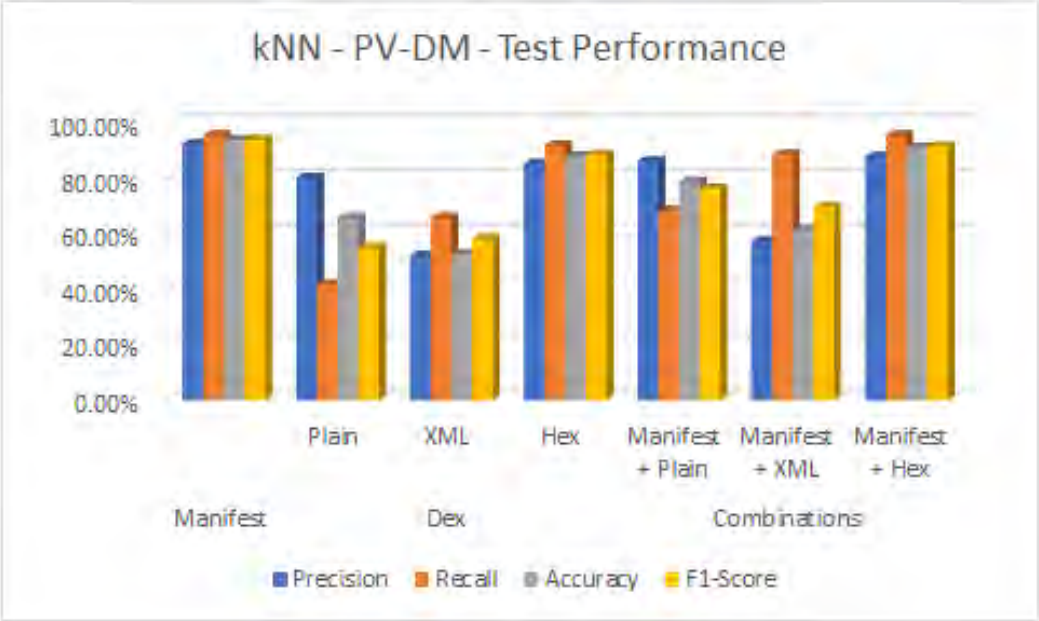
Table 5.36: k-Nearest Neighbors Model metrics on PV-DM - Test

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		92.5%	95.7%	94.0%	94.1%
Dex	**Plain**	80.3%	42.0%	65.8%	55.1%
	XML	52.2%	66.0%	52.8%	58.3%
	Hex	85.4%	92.0%	88.1%	88.5%
Combinations	**Manifest + Plain**	86.3%	68.5%	78.8%	76.4%
	Manifest + XML	57.4%	88.8%	61.5%	69.8%
	Manifest + Hex	87.9%	95.7%	91.2%	91.6%

During the testing phase, the performances stay consistent with what was seen during the development phase, as seen from Figure 5.18. The standalone manifest file and its concatenated form with a Hexadecimal dump of dex files had the best overall performance metrics. The performance of standalone and concatenated embeddings of Plain and XML formatted dex files during both the development and testing phase fell behind.



(a) PV-DM - Development



(b) PV-DM - Testing

Figure 5.18: PV-DM artefacts performance on k-Nearest Neighbors - DREBIN

5.3.5 Decision Tree

This is the final binary classifier that was trained on research artefacts for the task of Android malware detection as part of this study. This section presents the classification model performances on the artefacts generated from the files extracted from the DREBIN dataset.

PV-DBoW

During the development phase, the concatenated form of Manifest and Dalvik executable (in plain text format) file embeddings performed the best with 95.2% F1-score with all other metrics staying above 95% as is evident from the results from Table 5.37. The artefact made up of a standalone hexadecimal dump of Dalvik executable performed the worst with only 74.2% F1-Score and 70.0% Recall. In general, the artefacts made up of concatenated forms of embeddings had better F1-score, Accuracy, Precision and Recall scores when compared to the standalone embeddings. The only standalone embeddings with all metrics above 90% was the artefact made up of Manifest file embeddings.

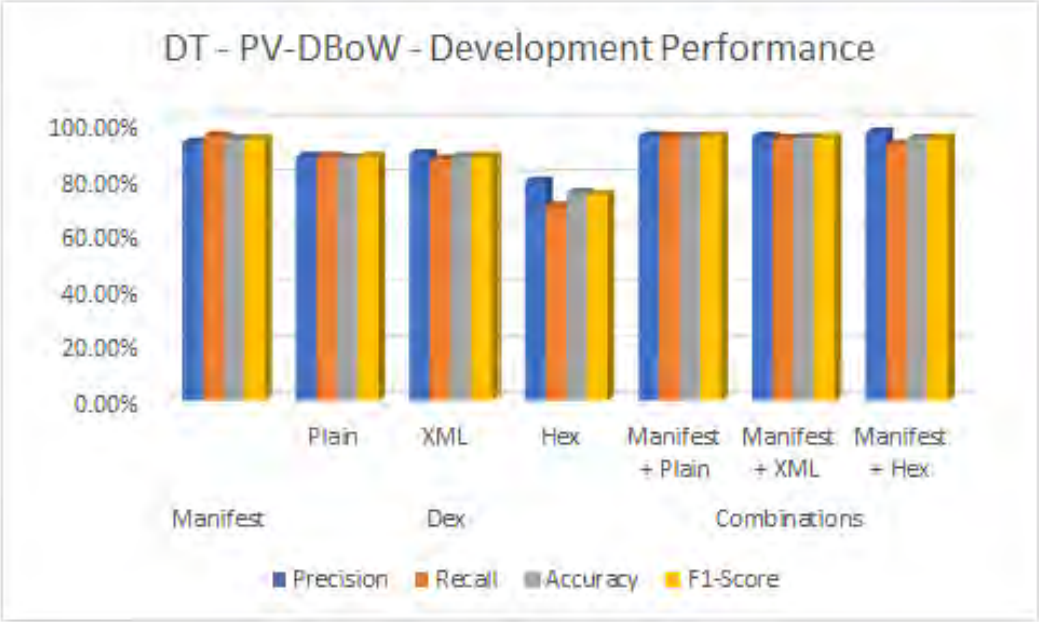
Table 5.37: Decision Tree Model metrics on PV-DBoW - Development

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		93.0%	95.3%	94.2%	94.1%
Dex	**Plain**	87.9%	87.9%	87.5%	87.9%
	XML	88.9%	86.9%	87.8%	87.9%
	Hex	79.0%	70.0%	74.8%	74.2%
Combinations	**Manifest + Plain**	95.3%	95.1%	95.1%	95.2%
	Manifest + XML	95.2%	94.3%	94.7%	94.8%
	Manifest + Hex	96.7%	92.2%	94.3%	94.4%

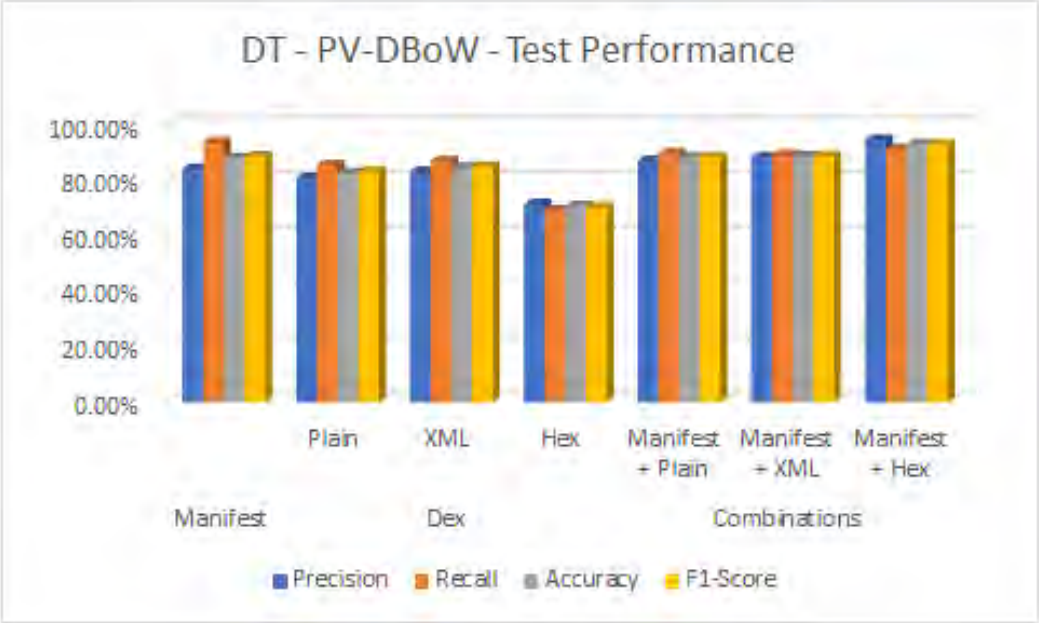
During the testing phase, however, the concatenated form of Manifest and Dalvik executable (in hexadecimal format) file embeddings had the best performance metrics with a Precision value of 94.9%, a Recall of 91.4%, Accuracy of 93.2% and F1-score of 93.1%. The other remaining artefacts saw a performance drop and had all of their metrics going under 90%. The only exceptions to this were the artefacts made up of the standalone manifest file and manifest concatenated with the Dalvik executable (in plain text format) file. These artefacts have a Recall of 94.0% and 90.0%, respectively, with the rest of the performance metric values ranging between 81-89% as can be seen from Table 5.38.

Table 5.38: Decision Tree Model metrics on PV-DBoW - Test

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		84.3%	94.0%	88.2%	88.9%
Dex	**Plain**	81.0%	85.7%	82.8%	83.3%
	XML	83.3%	87.1%	84.8%	85.1%
	Hex	71.6%	69.4%	71.0%	70.5%
Combinations	**Manifest + Plain**	87.2%	90.0%	88.4%	88.6%
	Manifest + XML	88.6%	89.4%	89.0%	89.0%
	Manifest + Hex	94.9%	91.4%	93.2%	93.1%



(a) PV-DBoW - Development



(b) PV-DBoW - Testing

Figure 5.19: PV-DBoW artefacts performance on Decision Trees - DREBIN

We can see from Figure 5.19 that concatenated embeddings performed better than the standalone in general during both the development and test phases. The dex file (in hexadecimal format) standalone embeddings performance dipped further in testing, which was already low in development. In contrast, concatenated forms of manifest and dex files performed well during development and stayed consistent during the testing phase.

PV-DM

If we observe the previous results on the DREBIN dataset, a trend can be seen in PV-DM so far where manifest file artefacts along with dex file (in hexadecimal format) artefacts were outperforming the other two standalone artefacts. Although based on results in Tables 5.39 and 5.40, it can be safely said that the Decision Tree model performance on artefacts was lower, in general, compared to other binary classifier trained thus far for DREBIN dataset. Only the concatenated form of manifest and dex (in hexadecimal format) had a Precision value of above 90%. All the metrics for the other artefacts were in the range of 40%-80% during development. Similar performance metrics were seen during the testing phase with manifest and dex (in hexadecimal format) artefacts outperforming other standalone artefacts, as can be observed from Table 5.40.

Table 5.39: Decision Tree Model metrics on PV-DM - Development

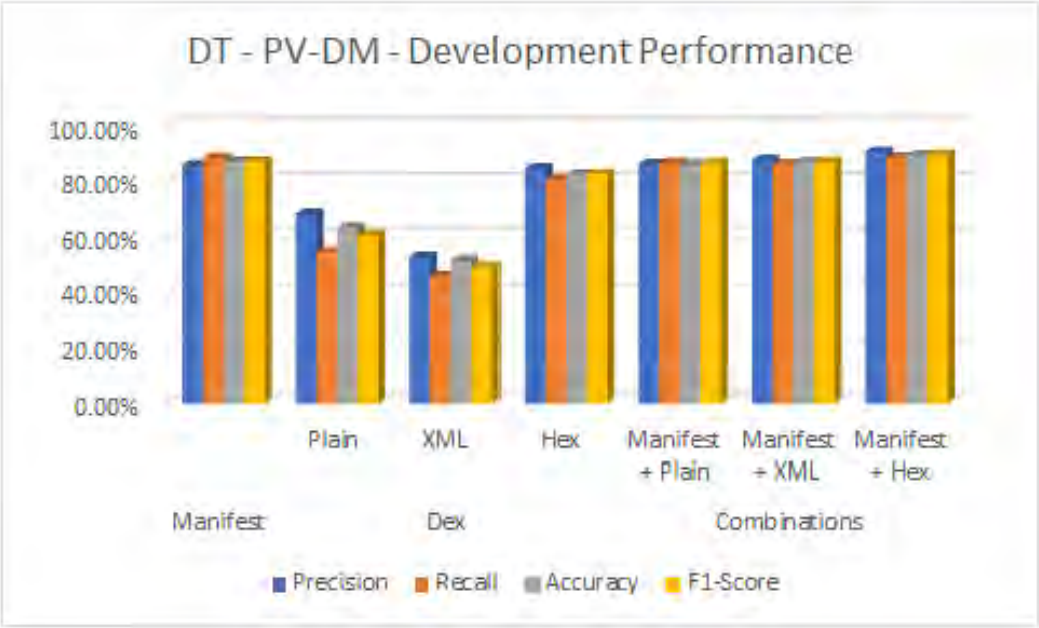
File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		86.1%	88.7%	87.5%	87.4%
Dex	**Plain**	68.5%	54.8%	63.6%	60.9%
	XML	53.1%	46.1%	51.8%	49.4%
	Hex	85.0%	80.9%	82.7%	82.9%
Combinations	**Manifest + Plain**	86.5%	86.9%	86.2%	86.7%
	Manifest + XML	88.0%	86.4%	87.1%	87.2%
	Manifest + Hex	90.8%	88.8%	89.6%	89.8%

The three artefacts made up of concatenated embeddings performed slightly better than the best performers of standalone embedding artefacts. The F1-score was almost similar for both dex files in a plain format as well as in hexadecimal format. The concatenated form of manifest with dex in hexadecimal format had the best F1-score with 89.8% and 89.9% during development and testing, respectively. The second best performer was the artefact made up of a concatenated form of the manifest with dex in plain text format with an F1-score of 86.7% and 87.9% during development and testing, respectively.

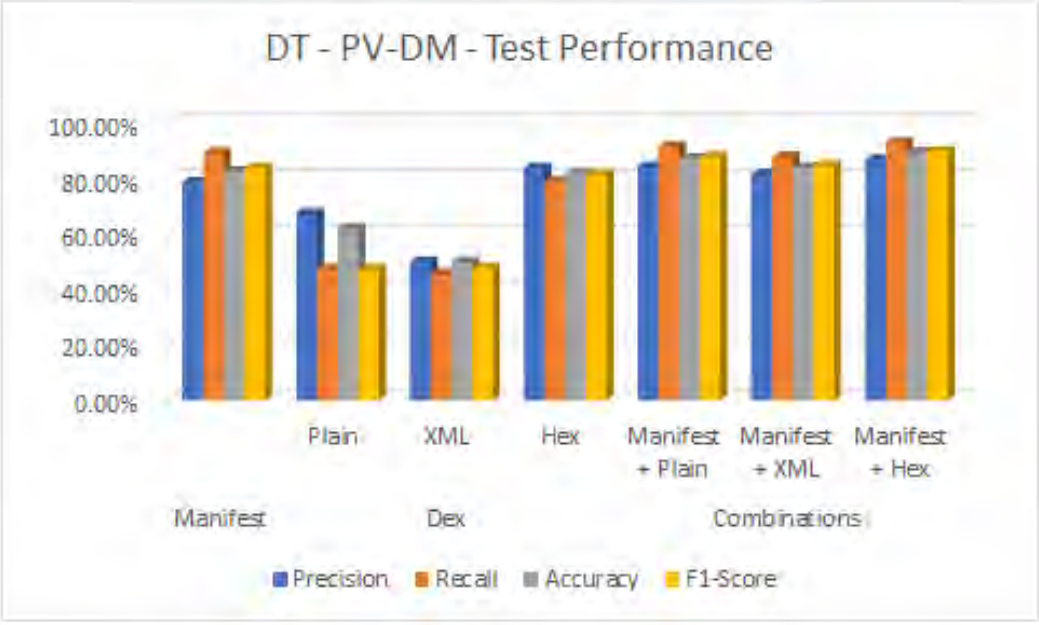
Table 5.40: Decision Tree Model metrics on PV-DM - Test

File Embedding		Precision	Recall	Accuracy	F1-Score
Manifest		78.8%	89.4%	82.7%	83.8%
Dex	**Plain**	67.4%	47.4%	62.2%	47.4%
	XML	50.1%	45.7%	50.1%	47.8%
	Hex	83.9%	79.1%	82.0%	81.4%
Combinations	**Manifest + Plain**	84.4%	91.7%	87.4%	87.9%
	Manifest + XML	81.6%	88.0%	84.1%	84.7%
	Manifest + Hex	86.9%	93.1%	89.5%	89.9%

From Figure 5.20, one can observe that manifest and dex files in hexadecimal format perform better than the other two artefacts of plain and dex files in XML format. The artefacts made up of concatenated forms had an overall better performance in both the development and test phases, with manifest and dex files (in hexadecimal format) having the best metrics during both the development and testing phases.



(a) PV-DM - Development



(b) PV-DM - Testing

Figure 5.20: PV-DM artefacts performance on Decision Trees - DREBIN

5.4 Discussion

This section of the chapter aims to discuss the results of our experiments in respect to our research objectives and questions. The objectives of our research were to explore the effectiveness of automatically generated features in the task of Android malware detection. We also wanted to assess the performances of automatically generated features which were created from different types of documents that are used by malware researchers for static analysis. In our experiments, we developed and validated our detection models against the **training** set and later evaluated the detection models trained on automatically generated features by predicting classes of unseen Android malware embeddings from the **test** set.

With these research objectives in mind, we wanted to answer some or all of the research questions. One of which was whether it was possible to treat static analysis documents (*AndroidManifest.xml* and dexdumps of *classes.dex*) as text (or natural language) documents. We also wanted to investigate whether it was possible to automatically produce feature vectors that accurately depict Android malware and benignware without explicitly using specific features from Android applications. Along with these questions, we wanted to compare the performance of our features against the existing approaches with our experiments. Simply put, we wanted to perform Android malware detection by using our novel feature generation technique that would eliminate the manual feature selection that we encountered in multiple research works during our literature survey. Our research artefacts were designed keeping the above problem in mind. We saw that some of our artefacts (PV-DBoW embedding based) performed better in general compared to another type of artefacts (PV-DM embedding based). Additionally, we saw that the manifest file artefact had a slightly lower performance with the AndroZoo dataset when compared to the manifest file artefact performance with the DREBIN dataset. However, the combined Artefacts of Dalvik executables with manifest files showed a decent performance showing that combining the feature vectors can help improve the detection performances. Against the DREBIN dataset, all the artefacts performed well during development, with hexadecimal-based artefacts being the only exception during evaluation. With the PV-DM-based artefacts, we also saw that hexadecimal dump embeddings performed well with both the AndroZoo and DREBIN datasets.

Table 5.41: Comparison of performance amongst the works that used the DREBIN dataset

Reference	Name	Precision	Recall	Accuracy	F1-Score
[29]	**ANASTASIA**	97.0%	97.0%	96.0%	97.0%
[19]	**DeepClassifyDroid**	96.6%	98.3%	97.4%	97.4%
[60]	**N/A**	99.9%	99.6%	99.7%	99.7%
[56]	**TC-Droid**	94.6%	98.4%	96.6%	96.6%
This Work	**CNN-PVDBoW**	**99.7%**	**100.0%**	**99.8%**	**99.8%**

The results of our experiments suggest that we can use the *doc2vec* technique to automatically generate embedding features that can effectively represent an Android application's behaviour, malicious or benign. The results of our experiments also support the existing knowledge about the Android application behaviour-related information that is present inside *AndroidManifest.xml* and *classes.dex* files. The embeddings generated from these files could effectively represent the behaviours of Android applications to train well-performing binary classification models consistently on them. In Table 5.41, we have presented the performance of one of our artefacts on the DREBIN dataset compared to other research works done on the DREBIN dataset, which were encountered during the literature survey. It should be noted that various cited results have applied different 'protocols' i.e., different dataset sample sizes and methods for feature generation. Our results are equivalent, if not better, than the results of existing works which utilised more complex feature generation approaches. The research work presented in Table 5.41 called ANASTASIA [29] focused on the Dalvik byte code and extracted 560 informative features from the application dex files to train nine different machine learning classifiers. In DeepClassifyDroid [19], five different feature sets were extracted, including Permissions, Intents, API calls and Strings present in the application. In the 2021 work [60] by Peng Xu *et al.*, researchers used a skip-gram word embedding model on the opcode sequences. In TC-Droid [56], word embeddings were generated from static and dynamic analysis reports using AndroPyTool. The last row contains results from our work where the CNN model was trained and evaluated on manifest file embeddings artefact, which was generated using the PV-DBoW algorithm. The results of this CNN model on other PV-DBoW artefacts can be found in Table 5.22.

With our analysis of the experiment results, it was identified that the embeddings of *AndroidManifest.xml* not only worked very well by themselves but also enhanced the detection capabilities when the file embeddings of *AndroidManifest.xml* and *classes.dex* were concatenated together. The results of our experiments from both Datasets showed that detection models trained on PV-DBoW artefacts have a better performance when compared to PV-DM artefacts. The standalone PV-DBoW embeddings artefact created from *AndroidManifest.xml* for the DREBIN dataset, all the performance metrics of Precision, Recall, Accuracy and F1-score of the detection models stayed between 99% and 100% during the development of CNN, SVM and LR models. All the performance metrics stayed between 98%-99% for the kNN model developed on this artefact. For AndroZoo, however, the PV-DBoW embeddings artefact created from hexdump of *classes.dex* had the best performance metrics with 99.6% Precision, 99.2% Recall, 99.5% Accuracy and 99.1% F1-Score with the CNN detection model. On the same artefact, the SVM model achieved 100.0% Precision, 96.3% Recall, 98.1% Accuracy and 98.1% F1-Score. The detection models trained on PV-DM artefacts had poor performance consistently across the experiments. Although the *AndroidManifest.xml* file embeddings gave good malware detection performance,

it was also notable that the Android malware detection results were slightly better when the embeddings were concatenated together. Also, the results of experiments where unseen malware document embeddings were given to the trained Android malware detection models support the idea that the document embeddings that were generated automatically (or using unsupervised learning) can be used to detect previously unseen malware as well. It should be noted that no statistical significance test was conducted, although attempts were made to ensure that the Android applications picked were recent and resembled commonly used application while choosing the applications for the dataset.

The feature embeddings/vectors which were generated automatically using the *doc2vec* algorithms were able to accurately depicted the Android application's behaviour. This can be inferred based on the binary classifiers that were trained and tested on the automatically generated features. Our features outperformed (or performed equally well compared to) the existing methods that also require additional manual efforts to extract the features from the Android Applications. The performance metrics of our CNN model trained on PV-DBoW embeddings created from Manifest files in comparison with other works done on the DREBIN dataset can be seen in Table 5.41. As discussed earlier, document embeddings can conserve document similarities. Moreover, it is well known that many new malware tend to be mutated versions of existing malware, which means that the source code is similar and subtly changed to avoid detection but retains most of the existing functionalities of the parent malware. Our study provided an automatic feature generation technique that can be directly applied to raw Android application files. The generated features do not require any further work and can directly be used to train artificial intelligence models. This would allow the security industry and researchers to keep up with the ever increasing numbers of Android malware that appear in the market every day.

5.5 Summary

In this chapter, we presented the results of experiments performed on our research artefacts using five popular binary classification models for the task of Android malware detection. The classification performance metrics are consistently good across all the artefacts based on PV-DBoW, whereas, with PV-DM algorithm-based artefacts, the performances of the concatenated embeddings were generally better than standalone embeddings. Also, with PV-DM algorithm-based artefacts, only the standalone manifest and dex (in hexadecimal format) file embedding artefacts had better results when compared to the remaining two artefacts. The "good" classification performance here refers to metrics that are **equivalent to** or **better than** the performance metrics of the research works we saw during literature review.

For PV-DBoW embeddings, the detection models had all of their performance metrics higher than 90% for the developed artefacts except when the detection model was trained on artefact created from Hexadecimal dump of *classes.dex* files from applications of the DREBIN dataset. Some detection models achieved perfect Precision and Recall scores during development and testing with DREBIN dataset applications. Higher detection Accuracy and Precision were also achieved with the AndroZoo dataset, with CNN models achieving 96% to 99% of performance scores on all of the artefacts that were developed. PV-DM-based artefacts, however, saw poor performance when compared to PV-DBoW. With F1-Score falling below 50% repeatedly for artefact created with *classes.dex* in XML format.

The results of the experiments show that our proposed methods for automatically creating feature vectors work very well at the task of Android malware detection. It is also noteworthy that we can generate such features without any manual intervention. The classification models trained on our automatically generated features performed consistently better than the manual approaches encountered during the literature review. The trained models had good classification (or Android malware detection) performances even when trained models were evaluated against unseen document embeddings.

We also presented a discussion of our results with respect to our research objectives, and we answered our research questions based on the results of the experiments. In an earlier chapter, we had also visualised the file embeddings where we noticed that the embeddings of the malicious and benign applications tend to cluster towards the embeddings of their own class, even when low-dimensional embeddings were used (whose dimensionality was further reduced for their portrayal in 3-dimensions). The results of our experiments reiterated that the automatically generated feature embeddings could conserve an Android application's behaviour to a reasonable degree that they can be used to train Android malware detection models.

Chapter 6

Conclusion & Future Work

6.1 Introduction

The research study will come to a close in this chapter, which will summarise the significant findings regarding the objectives and research questions and their importance and contribution. Additionally, it will discuss the study's limitations and suggest areas for future investigations.

6.2 Key Findings

This research work aimed to investigate the ability of *doc2vec* embeddings to solve the problem of Android malware detection. This research demonstrated that we could effectively use the document embeddings generated from the Android APK files to train statistical models that can differentiate between malicious and benign Android applications. Another one of the aims of this study was to analyse the performance of *doc2vec* algorithm of PV-DBoW and PV-DM algorithm with different types of documents generated using static analysis. The results indicate that the embeddings generated with the PV-DBoW algorithm were better segregated when plotted in a 3-dimensional vector space. We noted this for most of the documents generated with static analysis compared to the embeddings generated using the PV-DM algorithm. The final aim of this study was to evaluate the binary classification models trained on *doc2vec* embeddings by predicting classes of unseen Android application embeddings. As discussed earlier, the experiments were performed on Android applications from two datasets - AndroZoo and DREBIN. The results show that the NLP technique of document embeddings work effectively when used for predicting the classes of previously unseen Android application embeddings. These findings suggest that we could automatically

generate reliable features using the Android application files with *doc2vec* algorithms. The performance of these features was equivalent, if not better, at the Android malware detection task compared to other research methods we encountered during the literature review, as seen from the results in Table 5.41. This could be attributed to the fact that malware writers do not generally create malware from scracth due to a lack of skill or motivation. They rather use pre-existing malware and modify the source code to achieve their malicios goals or to avoid detection. The source code is thus, shared between multiple malwares i.e., they use similar code (files), and *doc2vec* is well-known for its application on document similarity problems.

6.3 Contributions

This section briefly explains the contributions made by our research work in the field of Android Malware detection. The primary research problem was that the current ML and DL-based Android malware detection methods require a certain degree of knowledge about the Android malware and the operating system to mine the applications for creating reliable feature vectors. Simply put, the feature creation task, a pre-requisite to training supervised classification models, requires manual effort and is a lengthy process of hits and trials. The manual selection of features is also not able to adapt the ever mutating and evolving Android malware. By applying *doc2vec* algorithms to files selected from an Android application, we generated the feature vectors automatically and successfully removed the need for manually identifying the features that can represent an application's behaviour. Additionally, these document embedding-based features can be directly used to train the binary classifiers for the Android malware detection task.

During the literature survey, we identified that artificial intelligence-related works on malware detection generally required knowledge about malwares and the system they are designed to exploit. This knowledge is required so that one may select reliable features that can define application characteristics. These features are then utilised for training malware detection models. We attempted to provide a document embedding-based malware detection approach that generates these features using an unsupervised learning approach using specific files from an Android application. Our proposed method requires a minimum pre-requisite knowledge about an Android application structure and files present in it. These document embedding-based features are used to train the supervised machine learning models that can reliably predict if an application is benign or malicious.

The malicious and benign applications tend to have distinct 'requested permissions'. Malicious applications need access to multiple aspects of an Android device and thus tend to have a higher number of required permissions. In contrast, benign applications have fewer

'requested permissions'. *doc2vec* algorithms conserve document-level information, and the visualisations presented in Chapter 4 show that the embeddings of malicious applications tend to cluster towards each other. This clustering shows that even though *doc2vec* was designed to deal with NLP problems, it can still conserve semantic information for Android application files. Additionally, we noticed during the literature review that the researchers had used Dalvik executable and manifest files from Android APKs to mine for features repeatedly. We also targeted those two files as part of our research work, and the embeddings of these files were used to train the classification models. The results show that the document embeddings created from these files retained the information and were capable of training classification models about the behaviour of an application.

The research outputs during the course of this research work have been listed below:

- Published an article in the proceedings of International Conference on Data Mining Workshops [96].
- Another paper has been sent in for Pattern Recognition Letters Journal and is awaiting review.
- The code base of this research work has been published on GitHub. The relevant URLs for the repositories can be found in the Appendix section of this thesis.

6.4 Limitations

The scope of our research work was set to investigate the application of document embeddings to the problem of Android malware detection. With this research work, we tried to understand the ability of document embeddings to preserve information from Android applications to train effective binary classifiers. We could not verify the capabilities of the document embeddings-based feature engineering approach against polymorphic or metamorphic malwares due to a lack of pre-existing benchmarked datasets consisting of such malware. Designing such a dataset from scratch was not possible due to the limited time available. We limited our analysis to the *AndroidManifest.xml* and *classes.dex* files. We ignored the other files contained in an Android application package, another significant limitation of this study.

6.5 Future works

For future research works, it would be interesting to look into the below-listed areas:

1. Investigating the other files present in the Android APK for the task of Android malware detection: Android APK consists of other resource files including the file containing constant strings and folder containing media. Malicious actors sometimes use them to hide the malware payload. The components of a disassembled application can also be used for creating document embeddings.

2. Android malware family classification using document embeddings: As malware belonging to a family share certain source code and behavioural aspects, document embeddings can be applied to Android malware family classification. Similar source code leads to the creation of similar document vectors, which help find similar documents without complexity.

3. Utilising trained *doc2vec* models to infer vectors of new Android malware: A Doc2Vec model created from a vast corpus of Android malware would be required to infer vectors for new Android malware correctly. We require an extensive dataset that can be created by building on top AndroZoo API.

4. A Web application that allows downloading Android applications by using APK SHA256 hash (or a list of SHA256 hashes) as input. AndroZoo project provides us with a file that contains SHA256 of APKs available with them as well as some other details. This web application will help researchers quickly create/download datasets on demand with an input list of APK hashes, allowing easy replicability of research.

5. Applying document embedding technique to detect polymorphic and metamorphic malware: The first step to verify this would be to create a dataset containing such malware. Polymorphic and metamorphic malware can rewrite their source code; thus, applying the document embedding technique might not provide successful results but it is still interesting to see if we will be able to conserve the presence of code-rewriting ability within the embeddings.

6. Investigating the applicability of more recent state-of-the-art NLP deep learning methods such as Transformers, to Android malware detection and classification.

7. A invetigation of the performance differences that were observed between PV-DBoW and PV-DM embeddings.

8. Conduct an in-depth investigation of overfitting issues that happen with PV-DM embeddings.

9. Study the comparison between the Android malware representations generated by *doc2vec* against the representations generated as part of other works surrounding Android malware detection and classification.

10. Study the comparison with other similarity-based detection methods which utilise other non-textual (non-NLP) techniques.

6.6 Summary

This study proposed a document embeddings-based approach to automatically generate features that can be used to design supervised learning systems, traditional and deep, to detect Android malware. A reliable malware detection system should have high accuracy and Precision when classifying previously unseen malware. In order to detect Android malware, this work provided a novel feature engineering strategy that did not require identification or selection of features. The features were used to train multiple classifiers against two independent datasets during our experiments. The experiments conducted using a variety of binary classifiers produced results on several of the generated artefacts that demonstrated great accuracy and Precision. Our features, which were automatically generated, outperformed DREBIN [28] which had a 94% detection rate with the numerous features used to train the detection models. Detection models created as part of this research work performed equivalent to state of art models which utilise complex feature generation processes such as that of [60] which achieved 99.9% Precision, 99.6% Recall, 99.7% Accuracy and 99.7% F1-score by utilising complex features or that of [6] which utilised n-grams of opcodes and achieved over 96% detection accuracy. One of our models, which was presented in Table 5.41, achieved 99.8% F1-score and Accuracy. The remaining models also see consistent detection performance during the training and against previously unseen file embeddings. This behaviour was seen across PV-DBoW embeddings-based artefact results presented in Chapter 5. As the dataset we used differed slightly from the researchers', a direct comparison can not be made. However, we can safely say that *doc2vec* malware embeddings can be utilised for Android malware detection. During the research, we saw that no standard (updated) dataset was available to benchmark methods and suggested future work to solve that issue. An automatic feature generation approach helps quickly identify Android malware, especially previously unknown ones. This feature engineering approach also creates a better defence as the detection models can quickly be retrained by adding new samples to the dataset.

Appendix A

GitHub Repositories: Source Code and Experiment Results

1. Android Malware Detection repository
   `https://github.com/TheCyberian/doc2vecMalwareDetection`
2. Result Notebooks
   `https://github.com/TheCyberian/doc2vecMalwareDetection/tree/master/classificationModelNotebooksAndResults`
3. Embedding Visualisations
   `https://github.com/TheCyberian/documentVectorVisualisation_MalwareEmbeddings`

Appendix B

IEEE ICDMW 2021 Publication

The PDF file of the publication in IEEE **International Conference on Data Mining Workshops (ICDMW) 2021** has been added from the next page as part of this Appendix. The paper can also be found online on the following URL:
`https://ieeexplore.ieee.org/abstract/document/9680047`.

Static Analysis for Android Malware detection with Document Vectors

Utkarsh Raghav
School of IT and Systems
University of Canberra
ACT, Australia
Utkarsh.Raghav@canberra.edu.au

Elisa Martinez-Marroquin
School of IT and Systems
University of Canberra
ACT, Australia
Elisa.Martinez-Marroquin@canberra.edu.au

Wanli Ma
School of IT and Systems
University of Canberra
ACT, Australia
Wanli.Ma@canberra.edu.au

2021 International Conference on Data Mining Workshops (ICDMW) | 978-1-6654-2427-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICDMW53433.2021.00104

***Abstract*—With the increase of smart mobile devices in use, the number of malware targeting the mobile platforms has been increasing. As the major market player in the industry, Android OS has been the favourite target of perpetrators targeting mobile platforms. The current machine learning and deep learning approaches for android malware detection utilize various feature creation methods. The majority of these feature creation methods use frequency-based vectors created from different files present in the android application package (APK). These frequency-based feature creation methods fail to preserve the semantic information that is present in those files. In this paper we propose a method that utilises the static analysis and natural language processing (NLP) technique of document embeddings to generate feature vectors that can represent the information contained in android manifests and dalvik executables files present inside an APK. These embeddings are then used to train binary classifiers which can effectively differentiate between a benign or malicious android application. Our proposed method in the experiments has outperformed the other related works on the test datasets.**

***Index Terms*—android malware detection, document embeddings, android, malwares, cybersecurity**

I. Introduction

As the digital age progresses, cyberspace is expanding, and related attack surfaces are on a rise. The smart mobile devices we all depend upon for our day to day activities are one such attack surface. With a control on over 70% of mobile market share [1], the Android operating system (Android OS) is the major player and therefore, a major target for malicious actors. The number of android malware have seen a significant rise in recent times, and traditional methods of signature-based detection fall short. With about 400,000 android malwares being sampled every month as of March 2020 [2], some of these malwares are custom made but the majority consist of mutated or morphed variants of similar malwares helping them evade detection systems.

Different machine learning (ML) and deep learning (DL) models have been utilised by researchers to overcome the shortfalls of hash-signature based malware detection. The techniques of static and dynamic analysis have been used to extract features for training these ML and DL models. Dynamic analysis is resource intensive as it requires the application to be executed in a sand-boxed environment to assess the application behaviour on the device and the network traffic generated by it, whereas during static analysis the application source code is explored with the goal of searching for code blocks that may represent malicious behaviours.

For the android platform, researchers have looked at the various files present inside an android application package (APK) and experimented with them to identify the best features. The best classification results so far, have been achieved with the dalvik executables and the android manifest files by creating frequency-based vectors from these files. Dalvik executable (or *classes.dex*) is the file that is executed by the android runtime when a user opens an application on their mobile device [3] and therefore, contains information required for execution. The android manifest (or *AndroidManifest.xml*) on the other hand contains information required by the Android OS to execute the application, such as permissions and hardware access needed by the application to work as intended [4]. The previous works, explained further in Section II, have utilised feature creation methods which do not preserve semantic information contained in these files but rather focus more on the presence of code level features that can define an applications behaviour.

The research work *Semantics-aware malware detection* by Christodorescu *et al.* [5] on Windows malwares used the observation that "certain malicious behaviours appear in all variants of a certain malware". Our work extends this intuition to android application files (*AndroidManifest.xml* and *classes.dex*) which contain the permissions and class definitions that depict the behaviours of an android application. We propose utilizing the NLP text classification technique of document embeddings [6] to create feature vectors from these two files that represent the semantic information contained in these documents, numerically. This is done by using certain prediction tasks, which have been further explained in the Section III.

In our work, *AndroidManifest.xml* and *classes.dex* files have been extracted from 2,234 android APKs. Document embeddings of these 2,234, *AndroidManifest.xml* and *classes.dex* files each, have been created. Experiments have been carried out to train classifiers using document embeddings of *AndroidManifest.xml* as features, document embeddings of *classes.dex* as features and a combination of document embedding of both of these files as features. Three different binary classifier models were trained on half of these files i.e., 1,117. The other half

978-1-6654-2427-1/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDMW53433.2021.00104

were used for measuring the classification model performance against unseen document embeddings (as test set).

The models used for the binary classification included a Convolutional Neural Network (CNN), a Support Vector Machine (SVM) and a Logistic Regression (LR) classifier. The target variables are categorical and fall into distinct categories of either *malicious* or *benign*, therefore, Logistic Regression and Support Vector Machine models are suited for the evaluation of the created document embeddings. The Convolutional Neural Network was also trained with a *sigmoid* activation on the output layer which lets the model predict binary outputs from a given set of input features.

The rest of the paper is organised as follows: Section II contains a literature review of related works; Section III discusses the existing techniques which were used in this work; Section IV describes the proposed methods and the steps taken to conduct this work; Section V describes the performed experiments and the achieved results and finally Section VII concludes our research work.

II. Related Works

In this section, we present a literature review of android malware detection methods currently in use.

In the work by Yerima *et al.* [7], three Bayesian classifiers were trained which utilised *classes.dex* file and *AndroidManifest.xml* for features. One of the classifiers utilised permission-based features extracted from *AndroidManifest.xml* and gave 89% classification accuracy, second classifier utilised code-based features extracted from *classes.dex* files which gave a 92% classification accuracy. The final classifier used a mix of features from both the files and gave the highest classification accuracy amongst the three, i.e., 93%. Their work was based on a dataset containing 2000 android applications (1000 benign and 1000 malicious samples).

Urcuqui-López *et al.* proposed a similar framework for android malware analysis [8]. They developed a permission analyser which assessed 558 *AndroidManifest.xml* files against 330 permissions. The analyser was programmed to assign 1 against the permission if it existed in the manifest file and a 0 otherwise. They trained and tested 6 different ML models with K-Nearest Neighbours, SVM and Decision tree giving the highest classification accuracy of 94% on this feature set and a lowest classification accuracy of 90% with a Naïve Bayes classifier.

Both works [7] and [8] displayed that the *AndroidManifest.xml* files contain information which can be utilised for malware detection and classification. They both utilised a permission analyser to create feature vectors, where Urcuqui-López *et al.* [8] used 330 permissions whereas Yerima *et al.* only took 131 'standard permissions' [7] in consideration when creating their feature vectors. Additionally, work done by Yerima *et al.* [7] also shows that *classes.dex* files contain classification features that are worth looking into when it comes to malware detection. In their work, they disassembled the application and extracted android and java API calls present in the code. They also looked for Linux system commands and other android commands while choosing the code based features.

In the work of Canfora *et al.* [9], the researchers utilised a dataset of 5560 benign and 5560 malicious android applications. They disassembled the dalvik executable file into *.smali* files to get access to opcode sequences. These sequences were then split into n-grams with various 'n' values. The best classification accuracy of 96% was achieved with the Random Forest classifier using n value of 2 (or bigrams).

A similar approach of training on opcode sequences was taken by McLaughlin *et al.* [10] which generated one hot encoded opcode sequence vectors. This approach was tested on three separate datasets with the best results shown for testing carried out on dataset with 2123 applications, 863 benign and 1260 malicious applications. In McLaughlin's work, 98% classification accuracy was achieved with 99% and 95% measured for precision and recall scores respectively, but these metrics dropped when this was tested against an unknown dataset. The model performance against unknown dataset was about 69% accuracy, with 67% precision and 74% recall scores.

All the methods presented in the above papers gathered features from Android APKs with static analysis but the features do not represent the document as a whole, they rather represent the frequency with which these features appear in malicious and benign APK files. In the current study, document embeddings (or document vectors) would be utilised as features for the classification model training. Document embeddings are trained as a by-product of a prediction task which leads to the creation of vector representation of the documents which are closely mapped in the vector space, if they are created from documents that contain semantically similar information.

In our work, these embeddings were trained using the distributed bag of words (PV-DBoW) algorithm which was presented by Le *et al.* in their paper 'Distributed Representations of Sentences and Documents' [6]. This algorithm is briefly explained in the next section. The application of PV-DBoW to APK files results in fixed-length document embeddings which conserve the semantic information present at the document level.

III. Feature extraction and classifiers

In this section, we provide some background information on the existing techniques which have been utilised for our work. These are Document Vectors, CNN, LR and SVM, briefly explained as follows.

A. Document Vectors

Paragraph Vectors, also known as document vectors (*doc2vec*), are vector representations of the information contained within a single document. These vectors are created using two separate unsupervised learning algorithms presented in the research work of Le *et al.* [6], called distributed memory (PV-DM) and distributed bag of words (PV-DBoW).

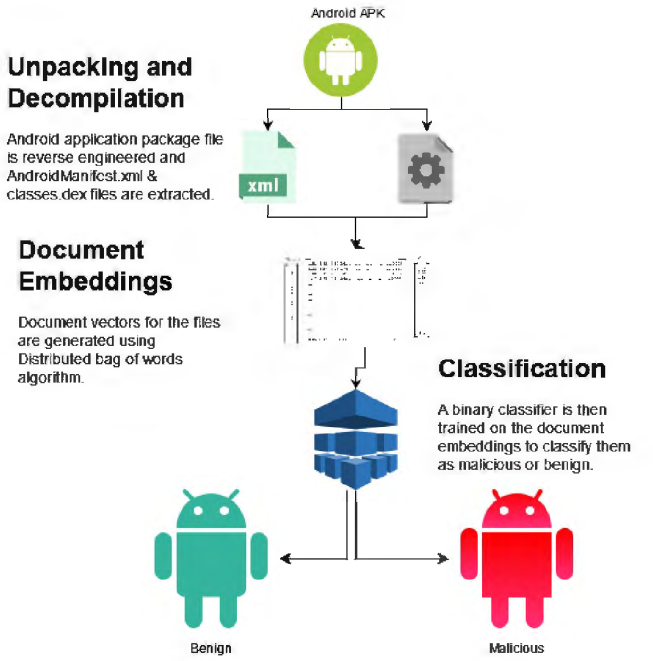


Fig. 1. Runtime processes overview

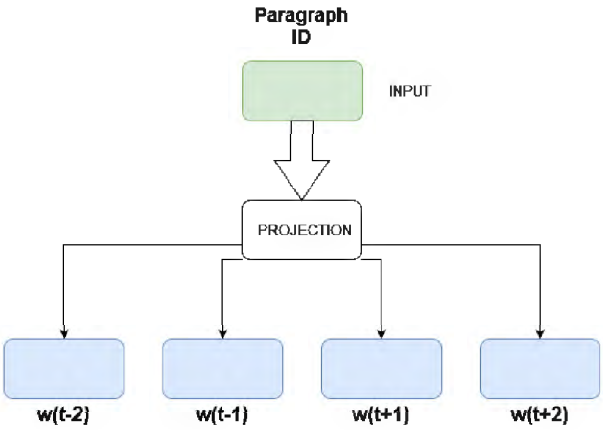


Fig. 2. PV-DBoW Model

The algorithm of PV-DBoW model is presented in Fig 2. This algorithm works by taking paragraph ID vector as an input and then using it to predict the context word vectors of the words (*w(t-2), w(t-1), w(t+1)* and *w(t+2)*) present in the document. The document vectors are randomly initialized and are then updated consistently as the algorithm learns, making them effectively represent the information contained inside each document that is present in the corpus. The second algorithm PV-DM is presented in Fig 3 which takes paragraph ID, and other context word vectors *w(t-2), w(t-1), w(t+1)* and *w(t+2)* as input, meanwhile trying to predict the target word *w(t)*.

By performing these prediction tasks, *doc2vec* helps generate fixed-length vector representations of different sized documents, paragraphs and even sentences which represent the document numerically. Please note that the prediction task is not what we are interested in, instead our focus is on the documents vectors that are created as the by-product of these prediction tasks. In our experiments, gensim package's Doc2Vec model's python implementation has been used [11].

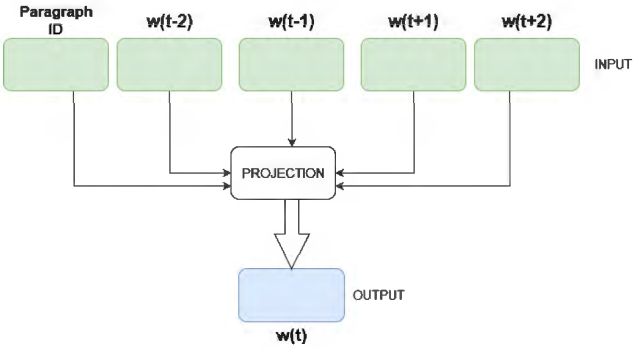


Fig. 3. PV-DM Model

B. Convolutional Neural Networks

CNNs are a specific architecture type of neural networks which are typically made up of one or more convolution and pooling layers put together. When talking about CNN, it is generally being referred to a 2 dimensional convolutions neural networks but, for the purposes of our experiments, single dimensional (1-D) convolution neural networks were utilised on document embeddings generated from malware and benign files.

At the input layer, a set of 1-dimensional feature vector is provided. This input is passed through a 1-D convolutional layer followed by 1-D max pooling. The output of the pooling layer is again passed through a 1-D convolutional layer followed by another 1-D max pooling. The output of the second pooling layer is then flattened and passed through a fully connected layer which in turn is passed through a *sigmoid* function which squeezes them and outputs a value between the range of 0 and 1.

In our experiments, the tensorflow package's Keras module's python implementation of 1-dimensional convolutional layer (Conv1D) and 1-dimensional Max pooling (MaxPooling1D) has been used [12].

C. Logistic Regression

Logistic regression is basically a modification of linear regression, modelled to handle classification problems by giving discrete predictions as outputs. These predicted values are mapped to probabilities by using a sigmoid function which squeezes the output value to be between 0 and 1.

In our experiments, Scikit-learn package's implementation of Logistic Regression classifier has been used from the 'sklearn.linear_model' module [13].

D. Support Vector Machines

Support Vector Machines or SVM belong to a class of supervised learning algorithm that is generally utilised for classification tasks. SVM operates with the goal of learning an optimal hyper-plane that can separate two classes while maximising the margin.

In our experiments, scikit-learn package's python implementation of SVC or support vector classifier has been used from the 'sklearn.svm' module [13].

IV. Proposed Method

The proposed method for this work firstly, requires us to acquire information from *classes.dex* and *AndroidManifest.xml* files by analysing benign and malicious android applications. After this, the document vectors of the information contained in these two files are generated using PV-DBoW (Paragraph Vector – Distributed Bag of Words) algorithm, and then a label defining the document vectors behaviour (malicious or benign) is tied along with the document vector representation. This is followed by document vectors being fed to Logistic Regression, SVM and CNN as input for the learning process. Previously unseen document vectors embeddings are then classified into malicious or benign, with the learnt models. Document vector is vector representation of the information contained inside a document or a file, in this case *AndroidManifest.xml* and *classes.dex* files. The representations are mapped in the vector space closer to each other if the documents contain similar content.

For instance, the *AndroidManifest.xml* file contains the permissions requested by the android application to run on a device and give a picture of application behaviour. A vector representation of documents containing similar permissions would be closely mapped in a vector space and would therefore, allow for easy classification by a simple binary classifier.

A. Steps involved

The Fig 1 represents the major tasks undertaken to create the model. Fig 4 represents the ratio in which document embeddings are split for model training and evaluation. The embeddings of *AndroidManifest.xml* and *classes.dex* files were split equally into train and test sets. The training dataset and testing dataset were kept equal to have a high number of unseen document embeddings to gauge the performance of classification models against them. The classification models were trained and validated against the embeddings present in the train set and evaluated against previously unseen embeddings i.e., the embeddings from test set.

The below steps were performed to create and train the binary classification models:

1) Unpacking and decompiling the APKs
2) Document Embeddings generation from dex and android manifest files
3) Feature labelling
4) Classifier Training
5) Evaluation

B. Unpacking and decompiling the APKs

The android APKs are first unpacked using the 'apktool' [14] which reveals the *AndroidManifest.xml* and *classes.dex* file. However, before these files can be read, they are decompiled using *dexdump* utility [15]. These files are then extracted from each of the android application and place in an individual folder for manifest files and dex files. These folders have subfolders 'test' and 'train' within them assigning the files to test or train dataset beforehand. The extracted files are equally distributed in the respective folders, ensuring that if an application's manifest is part of the test set, then the application's dex file should also be part of the test set and vice versa.

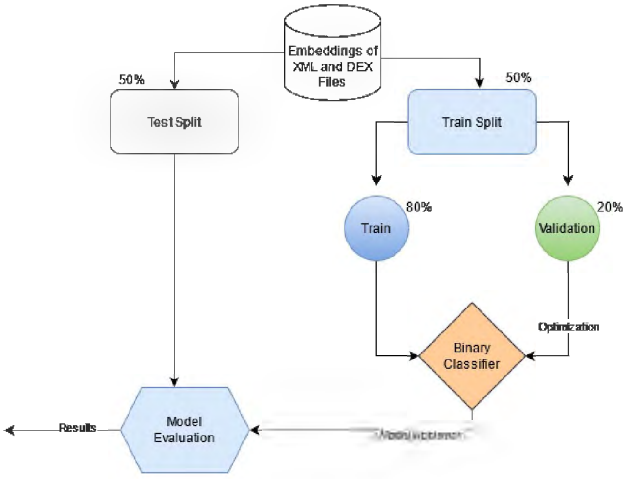


Fig. 4. System Architecture

C. Document Embeddings generation from dex and android manifest files

Features are extracted using static analysis of files extracted during the unpacking and decompilation step. Document embeddings of *AndroidManifest.xml* and *classes.dex* are generated as features and are grouped together for classification model training and testing. 200 dimensional vectors have been used for this research work.

a) Document Embeddings of classes.dex: Document vectors of information present in *classes.dex* are created using Gensim package of Python. The package, Gensim, contains methods to implement both the doc2vec algorithms that were discussed by the researcher team that created doc2vec, PV-DM (Paragraph Vectors – Distributed Memory) and PV-DBoW (Paragraph Vectors – Distributed Bag of Words). For this work, only PV-DBoW is used, which learns document vector by predicting *classes.dex* file content at the output layer, given the document ID at the input layer.

b) Document Embeddings of AndroidManifest.xml: One document vector of *AndroidManifest.xml* holds information present in it about a single android application. Document embedding of *AndroidManifest.xml* are generated by analysing

the manifest file statically. The content of manifest is read after decompiling the file. To learn document vectors of information contained inside *AndroidManifest.xml*, a neural network tries predicting manifest files content given the document ID at the input layer and updates the vector till prediction accuracy is enhanced.

c) Combined Embeddings: Document vectors of *AndroidManifest.xml* and *classes.dex* file are combined together to form a single document vector of 400 dimensions.

D. Feature labelling

The document vectors created in the last step contain the information, which was present in the document. The labels for these vectors must be maintained separately when training the document embeddings as the Doc2Vec model does not implicitly conserves it. We used *namedtuple()* function from Python's *collection* module to create a custom datatype which stored the document's text, its tag (which is needed by doc2vec's gensim implementation) and its label (malicious or benign). This datatype was used when training the document embeddings and after the vectors are learnt, these are labelled with malicious (1) or benign (0) labels by referring the label parameter of the datatype.

E. Classifier Training

Once the document vectors are arranged as features and labels, classification model training can begin. For this work, Support Vector Machine, Logistic Regression and Convolutional Neural Network have been used for classification modelling purposes. Labelled document embeddings are fed into a classifier as an input for learning. A document embedding and its respective label is then learnt by the classification model.

F. Evaluation

The trained model performances are evaluated against the test split as shown in Fig. 4. The trained binary classifiers in previous step, are then tasked with predicting labels for these previously unseen file embeddings and their accuracy, precision and recall values are recorded.

V. Experiments

In this section, we evaluate the model metrics of accuracy, precision and recall. This section also discusses the data set used, the experimental procedures involved and their results.

A. Dataset

The dataset for these experiments consists of 2,234 android applications taken from AndroZoo repository [16]. The applications were selected using the dex date (creation date on dex file) ranging between 2015 to 2021 and the APK size was limited to 5 MB. These were downloaded using *az* script created by Artem Kushnerov [17].

had been decompiled and unpacked to access the *classes.dex* files and the *AndroidManifest.xml* files. Document embeddings for both the files were generated using the doc2vec algorithm which was described in the *Feature Extraction And Classifiers* section.

The dataset was divided into two equal sized test and train sets i.e., 1117 files in each split. The classification models were trained with an 80-20 train-validation split followed by an evaluation of the trained models against the document embeddings of the remaining 1117 files of the test split. This was done to gauge the model performance against previously unseen document embeddings generated from the files extracted from android applications.

B. Experimental Procedure

In this subsection, the overall classification model performance is studied using the generated document embeddings by performing a classification task. Metrics of accuracy, precision and recall are evaluated. The first task is to generate document embeddings from *AndroidManifest.xml* and *classes.dex* file, this is done by acquiring dexdump and Android manifests by static analysis of 2,234 Android APKs of which 1,185 were malicious and remaining were benign. Initial 32000 bytes are statically read from each of the (*AndroidManifest.xml* and *classes.dex*) files, acquired from the same APK. These 32000 bytes of data are used to generate 200 dimensional document vectors for each file using PV-DBoW algorithm. These document vectors are assigned a class label representing whether it comes from a Malicious or a Benign android application.

These document embeddings are then split into two sets of 1117 file embeddings each, train and test set. The models are trained and validated against the train set being split into an 80-20 ratio. The trained models with best results are then tested against the test set. The evaluation metrics are calculated for document vectors of *AndroidManifest.xml* files being inputted to SVM, LR and CNN classifiers alone, then for document vectors of *classes.dex* files being inputted into the classifiers and lastly, for a combination of document vectors generated from both the files.

C. Experiment Results

Experiment results for malware detection are presented in this section. The training and evaluation results have been grouped together in Table I, with three entries each for Accuracy, Precision and Recall scores for Logistic Regression, SVM and CNN models. The results of model testing on unseen data set have been separately placed in Table II.

Table I shows that a better overall training accuracy was achieved when a 400 dimensional combined(dex + manifest) vector was used for binary classification between a malware and benignware. The least accurate predictions were under Logistic Regression model which was trained using the document embeddings of *classes.dex* file alone. However, when SVM and CNN are utilised, classification accuracy remains stable across all three feature types and dimension sizes.

Similar trends can be seen in Table I Precision and Recall rows which summarize the Precision and Recall scores. However, the Recall scores for SVM, Logistic Regression and CNN models trained on manifest file features alone are slightly better than the ones trained on dex file features alone.

TABLE I
MODEL PERFORMANCE METRICS ON TEST TRAIN SPLIT DATA SET.

Model	Metrics	File Embeddings		
		dex	manifest	dex+manifest
LR	Accuracy	0.8303	0.9598	0.9330
	Precision	0.8250	0.9652	0.9316
	Recall	0.8534	0.9568	0.9396
SVM	Accuracy	0.9598	0.9598	0.9642
	Precision	0.9572	0.9572	0.9426
	Recall	0.9655	0.9655	0.9913
CNN	Accuracy	0.9877	0.9899	0.9944
	Precision	0.9837	0.9891	0.9973
	Recall	0.9864	0.9864	0.9891

Please note that the results for Logistic Regression, SVM and CNN contained in Table I represent the performance of trained models against test train split data.

As stated earlier, the prediction accuracy of CNN during model training remains steady for all the feature types as can be seen in Table I. Although, during the training the best results were gained with manifest file features alone. This is due to the fact that the *AndroidManifest.xml* file contains required permissions, hardware access and other privileges needed by an application to execute its full functionality on a device. In figures 5 and 6 showing a snapshot of manifest files of benign and malicious applications, one can clearly see that the number of permissions (*uses-permission android:name*) requested by a malicious application is comparatively higher than a benign application.

The model, after training completion, was given a set of unknown document embeddings for classification. These classification metrics have been summarised in Table II. It must be noted that, when unknown document embeddings extracted from dex files were used for classification, the model performance degrades. The best model performance against unknown malwares was achieved by a combination features extracted from both dex and manifest files.

VI. DISCUSSION

Our results have reiterated the effectiveness of manifest and dalvik executable file features. They have also proven that the document embeddings can be effectively used to conserve information present in these files which can be in turn used as classification features.

The classifier trained on the features created by our proposed method, outperformed the classifiers which were trained by using same files in the work of Yerima *et al.* [7]. Similar to our work, their 3 Bayesian models were trained separately on features from *AndroidManifest.xml* files, from *classes.dex* files and a mixture of features from both files. Their method achieved 89% classification accuracy with *AndroidManifest.xml*, 92% classification accuracy with *classes.dex* and 93% classification accuracy with combination of these two. The embeddings created as part of our work, other than LR model trained on *classes.dex* embeddings alone, had significantly higher accuracy than the regular features

Fig. 5. Benign application - AndroidManifest.xml

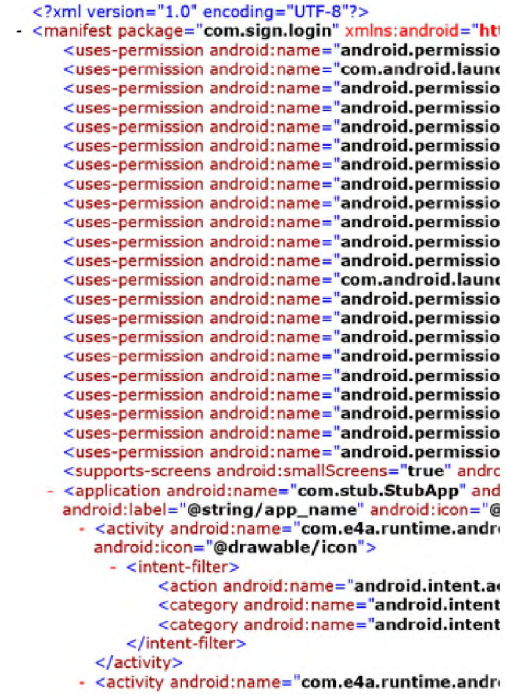


Fig. 6. Malicious application - AndroidManifest.xml

TABLE II
MODEL PERFORMANCE METRICS ON UNKNOWN DATA SET

Model	Metrics	File Embeddings		
		dex	manifest	dex+manifest
LR	Accuracy	0.6598	0.8084	0.8236
	Precision	0.6871	0.7849	0.8009
	Recall	0.6593	0.8802	0.8887
SVM	Accuracy	0.6821	0.7824	0.8343
	Precision	0.6976	0.7492	0.8063
	Recall	0.7082	0.8870	0.9055
CNN	Accuracy	0.6150	0.7932	0.8120
	Precision	0.6687	0.8068	0.7969
	Recall	0.5447	0.8027	0.8668

created from these files.

The classification models trained on android manifest files alone, also outperformed the models which were trained on permission-based features extracted from the *AndroidManifest.xml* in the work of Urcuqui-López *et al.* [8]. In their work, the highest classification accuracy of 94% was achieved from K-Nearest Neighbours, SVM and Decision tree models. Our work, also conserved the information present in the *AndroidManifest.xml* file as a whole and not just that of permissions and therefore, yielded better accuracy, precision and recall scores.

All the experiments in [7], [8] and [10] were done on three different android malware datasets with different number of applications and are not available to be used by other people. We have conducted our tests on applications available from AndroZoo [16] dataset, which is publicly available for the researchers to "engage in reproducible experiments".

The creation (or training) of document embeddings from over 2000 dex and manifest files took about 18 minutes in total (dex $\approx$ 15 minutes and manifest $\approx$ 3 minutes) on an Intel(R) Core i7-8665U CPU with 16 GB RAM. Contrary to this, the methods presented in related works require manual creation of feature vectors which is in itself time consuming. Although python's *gensim* [11] module implementation of doc2vec provides a function *infer_vector()* which can be used to infer the embeddings of new documents from a pre-trained doc2vec model in relatively short time duration, this work trained the doc2vec model for all the files together and later split them into test and train sets for ease of implementation.

In the work of Christodorescu *et al.* [5], the researchers initially developed "malicious behaviour templates" which allowed matching these templates with the code sequences present in files being tested for malicious behaviours. Our approach doesn't require the creation of these sequence templates manually but instead learns the semantic information representing malicious and benign behaviours present in the files by using the doc2vec algorithm to generate the document embeddings.

VII. CONCLUSION

In this paper, we report our work in investigating three different binary classifiers trained on three different combinations of document embeddings for android malware detection. The classifier models gave a highest accuracy score of 0.99 on validation data when document embeddings of dex and manifest files were combined together. On the other hand, when the same model was tested against previously unseen document embeddings an accuracy score of 0.83 was achieved. In this work, we have demonstrated that the information present in malware files can be extracted as text and the semantic information present in these files can be learnt efficiently with doc2vec model as document embeddings. These embeddings can be used as feature vectors which allow binary classifiers to effectively distinguish between malicious and benign files.

The malware detection method proposed in our work has outperformed the other approaches. The classification accuracy achieved was higher with document embeddings of *AndroidManifest.xml* when compared to *classes.dex* file alone. A combined document vector extracted from these files outperforms the models trained using single files. The SVM model achieved better classification against previously unseen malware file than the other models when it was trained on combined embeddings of dalvik executable and manifest file. The CNN model achieved better overall classification metrics on combined file embeddings generated from android manifest and dex files.

In future, work could be done on android malware family classification using document embeddings. The document embeddings primarily conserve document similarity information which could be helpful in classifying android malwares into their respective families using a much larger data set than the current one.

REFERENCES

[1] Mobile Operating System Market Share Worldwide — Statcounter Global Stats. https://gs.statcounter.com/os-market-share/mobile/worldwide.

[2] AV-Test. Security report 2019/2020. 2019.

[3] Android Runtime (ART) and Dalvik — Android Open Source Project. https://source.android.com/devices/tech/dalvik.

[4] App Manifest Overview — Android Developers. https://developer.android.com/guide/topics/manifest/manifest-intro.

[5] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. *Proceedings - IEEE Symposium on Security and Privacy*, pages 32–46, 2005.

[6] Quoc V. Le and Tomas Mikolov. Distributed Representations of Sentences and Documents. *31st International Conference on Machine Learning, ICML 2014*, 4:2931–2939, may 2014.

[7] Suleiman Y Yerima, Sakir Sezer, and Gavin Mcwilliams. Analysis of Bayesian classification-based approaches for Android malware detection. *IET Information Security (IET INFORM SECUR)*, 2013.

[8] Christian Urcuqui-López and Andrés Navarro Cadavid. Framework for malware analysis in Android. *Sistemas y Telemática*, 14(37):45–56, aug 2016.

[9] Gerardo Canfora, Andrea De Lorenzo, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Effectiveness of opcode ngrams for detection of multi family android malware. In *Proceedings - 10th International Conference on Availability, Reliability and Security, ARES 2015*, pages 333–340. Institute of Electrical and Electronics Engineers Inc., oct 2015.

[10] Niall McLaughlin, Jesus Martinez Del Rincon, Boo Joong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupe, and Gail Joon Ahn. Deep android malware detection. In *CODASPY 2017 - Proceedings of the 7th ACM Conference on Data and Application Security and Privacy*, pages 301–308. Association for Computing Machinery, Inc, mar 2017.

[11] Doc2Vec Model — gensim. https://radimrehurek.com/gensim/auto_examples/tutorials/run_doc2vec_lee.html.

[12] Module: tf.keras — TensorFlow Core v2.6.0. https://www.tensorflow.org/api_docs/python/tf/keras.

[13] sklearn.svm.SVC — scikit-learn 0.24.2 documentation. https://scikit-learn.org/.

[14] Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps. https://ibotpeaches.github.io/Apktool/.

[15] Ubuntu Manpage: dexdump - Display information about Android .dex files. http://manpages.ubuntu.com/manpages/bionic/man1/dexdump.1.html.

[16] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[17] Artem Kushnerov. Artemkushnerov · github. https://github.com/ArtemKushnerov/az, 2018.

Bibliography

[1] AV-Test. “Security Report 2019/2020”. In: (2019). URL: www.av-test.org.

[2] *Mobile malware jumped 75 percent in 2014: Report*. URL: https://www.cnbc.com/2015/01/14/mobile-malware-jumped-75-percent-in-2014-report.html (visited on 03/21/2022).

[3] Igor Santos et al. *N-GRAMS-BASED FILE SIGNATURES FOR MALWARE DETECTION*. Tech. rep. 2009.

[4] Peter Teufl, Udo Payer, and Guenter Lackner. “From NLP (Natural Language Processing) to MLP (Machine Language Processing)”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6258 LNCS (2010), pp. 256–269. ISSN: 03029743. DOI: 10.1007/978-3-642-14706-7_20.

[5] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. Tech. rep. 2014. arXiv: 1301.3781v3. URL: https://arxiv.org/abs/1301.3781.

[6] Gerardo Canfora et al. “Effectiveness of opcode ngrams for detection of multi family android malware”. In: *Proceedings - 10th International Conference on Availability, Reliability and Security, ARES 2015*. Institute of Electrical and Electronics Engineers Inc., 2015, pp. 333–340. ISBN: 9781467365901. DOI: 10.1109/ARES.2015.57.

[7] Niall McLaughlin et al. “Deep android malware detection”. In: *CODASPY 2017 - Proceedings of the 7th ACM Conference on Data and Application Security and Privacy*. Association for Computing Machinery, Inc, 2017, pp. 301–308. ISBN: 9781450345231. DOI: 10.1145/3029806.3029823.

[8] A. Martín et al. “ADROIT: Android malware detection usingmeta-information”. In: (2016). URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=%7B%5C&%7Darnumber=7849904 (visited on 04/02/2021).

[9] Shahid Alam. “Applying Natural Language Processing for detecting malicious patterns in Android applications”. In: *Forensic Science International: Digital Investigation* 39 (2021). ISSN: 2666-2817. DOI: 10.1016/J.FSIDI.2021.301270.

[10] *Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps.* https://ibotpeaches.github.io/Apktool/. URL: https://ibotpeaches.github.io/Apktool/ (visited on 07/26/2021).

[11] *Ubuntu Manpage: dexdump - Display information about Android .dex files*. http://manpages.ubuntu.com/manpages/bionic/man1/dexdump.1.html. (Visited on 07/21/2021).

[12] *Ubuntu Manpage: hexdump, hd — ASCII, decimal, hexadecimal, octal dump*. URL: https://manpages.ubuntu.com/manpages/trusty/man1/hexdump.1.html (visited on 04/30/2022).

[13] Quoc V. Le and Tomas Mikolov. "Distributed Representations of Sentences and Documents". In: *31st International Conference on Machine Learning, ICML 2014* 4 (2014), pp. 2931–2939. arXiv: 1405.4053. URL: http://arxiv.org/abs/1405.4053.

[14] Anne-Wil Harzing. *Publish or Perish*. URL: https://harzing.com/resources/publish-or-perish.

[15] Michael Sikorski and Andrew Honig. *Practical Malware Analysis*. No Starch Press, 2012, pp. 10–11. ISBN: 9781593272906.

[16] Ashish Aggarwal and Pankaj Jalote. "Integrating static and dynamic analysis for detecting vulnerabilities". In: *Proceedings - International Computer Software and Applications Conference* 1 (2006), pp. 343–350. ISSN: 07303157. DOI: 10.1109/COMPSAC.2006.55.

[17] Shweta Bhandari et al. "DRACO:DRoid Analyst COmbo An Android Malware Analysis Framework". In: 15 (). DOI: 10.1145/2799979.2800003. URL: http://dx.doi.org/10.1145/2799979.2800003..

[18] Dong Jie Wu et al. "DroidMat: Android malware detection through manifest and API calls tracing". In: *Proceedings of the 2012 7th Asia Joint Conference on Information Security, AsiaJCIS 2012*. 2012, pp. 62–69. ISBN: 9780769547763. DOI: 10.1109/AsiaJCIS.2012.18.

[19] Yi Zhang, Yuexiang Yang, and Xiaolei Wang. "A novel android malware detection approach based on convolutional neural network". In: *ACM International Conference Proceeding Series* (2018), pp. 144–149. DOI: 10.1145/3199478.3199492.

[20] James Scott. "Signature Based Malware Detection is Dead". In: (2017). URL: www.ICITForum.org.

[21] *What is Hashing and How Does it Work? — SentinelOne*. URL: https://www.sentinelone.com/cybersecurity-101/hashing/.

[22] Dragoş Gavriluţ et al. “Malware detection using machine learning”. In: *Proceedings of the International Multiconference on Computer Science and Information Technology, IMCSIT ’09*. Vol. 4. 2009, pp. 735–741. ISBN: 9781424453146. DOI: 10.1109/IMCSIT.2009.5352759.

[23] El Mouatez Billah Karbab, Mourad Debbabi, and Djedjiga Mouheb. “Fingerprinting android packaging: Generating DNAs for malware detection”. In: *DFRWS 2016 USA - Proceedings of the 16th Annual USA Digital Forensics Research Conference*. Digital Forensic Research Workshop, 2016, S33–S45. DOI: 10.1016/j.diin.2016.04.013. URL: http://dx.doi.org/10.1016/j.diin.2016.04.013.

[24] Christian Urcuqui-López and Andrés Navarro Cadavid. “Framework for malware analysis in Android”. In: *Sistemas y Telemática* 14.37 (2016), pp. 45–56. ISSN: 1692-5238. DOI: 10.18046/syt.v14i37.2241.

[25] Mohammad Al-Fawa’reh et al. “Malware Detection by Eating a Whole APK”. In: *2020 15th International Conference for Internet Technology and Secured Transactions, ICITST 2020* (Dec. 2020). DOI: 10.23919/ICITST51030.2020.9351333.

[26] Suleiman Y Yerima, Sakir Sezer, and Gavin Mcwilliams. “Analysis of Bayesian classification-based approaches for Android malware detection”. In: *IET Information Security (IET INFORM SECUR)* (2013). ISSN: 1751-8709. DOI: 10.1049/iet-ifs.2013.0095. URL: www.ietdl.org.

[27] Ali Ghasempour, Ovye John Abari, and Nor Fazlida Mohd Sani. “Permission Extraction Framework for Android Malware Detection”. In: *Article in International Journal of Advanced Computer Science and Applications* 11.11 (2020). DOI: 10.14569/IJACSA.2020.0111159. URL: www.ijacsa.thesai.org.

[28] Daniel Arp et al. *DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket*. Tech. rep. 2014. URL: http://dx.doi.org/doi-info-to-be-provided-later.

[29] Hossein Fereidooni et al. “ANASTASIA: ANdroid mAlware detection using STatic analySIs of applications”. In: *2016 8th IFIP International Conference on New Technologies, Mobility and Security, NTMS 2016* (2016). DOI: 10.1109/NTMS.2016.7792435.

[30] Vasileios Syrris and Dimitris Geneiatakis. “On machine learning effectiveness for malware detection in Android OS using static analysis data”. In: *Journal of Information Security and Applications* 59 (2021), p. 102794. ISSN: 2214-2126. DOI: 10.1016/J.JISA.2021.102794.

[31] R. Vinayakumar et al. “Detecting Android malware using Long Short-term Memory (LSTM)”. In: *Journal of Intelligent and Fuzzy Systems*. Vol. 34. 3. IOS Press, 2018, pp. 1277–1288. DOI: 10.3233/JIFS-169424.

[32] Hongliang Liang, Yan Song, and Da Xiao. “An end-To-end model for Android malware detection”. In: *2017 IEEE International Conference on Intelligence and Security Informatics: Security and Big Data, ISI 2017* (2017), pp. 140–142. DOI: 10.1109/ISI.2017.8004891.

[33] Yoshua Bengio et al. *A Neural Probabilistic Language Model*. Tech. rep. 2003, pp. 1137–1155.

[34] Dali Zhu et al. “A transparent and multimodal malware detection method for android apps”. In: *MSWiM 2019 - Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems* (Nov. 2019), pp. 51–60. DOI: 10.1145/3345768.3355915. URL: https://doi.org/10.1145/3345768.3355915.

[35] Tianliang Lu et al. “Android malware detection based on a hybrid deep learning model”. In: *Security and Communication Networks* 2020 (2020). ISSN: 19390122. DOI: 10.1155/2020/8863617.

[36] Zhenlong Yuan, Yongqiang Lu, and Yibo Xue. “Droiddetector: Android malware characterization and detection using deep learning”. In: *Tsinghua Science and Technology* 21.1 (2016), pp. 114–123. ISSN: 18787606. DOI: 10.1109/TST.2016.7399288.

[37] *Droidbox – Android Application Sandbox – The Honeynet Project*. URL: https://www.honeynet.org/projects/active/droidbox/ (visited on 04/03/2022).

[38] Hyunjae Kang et al. “Detecting and classifying android malware using static analysis along with creator information”. In: *International Journal of Distributed Sensor Networks* 2015 (2015), p. 479174. ISSN: 1550-1329. DOI: 10.1155/2015/479174. URL: https://koreauniv.pure.elsevier.com/en/publications/detecting-and-classifying-android-malware-using-static-analysis-a.

[39] Abdelmonim Naway and Yuancheng Li. “Android Malware Detection Using Autoencoder”. In: *arXiv* (Jan. 2019), arXiv:1901.07315. arXiv: 1901.07315. URL: https://ui.adsabs.harvard.edu/abs/2019arXiv190107315N/abstract.

[40] Yangxu Jin et al. “Android Malware Detector Exploiting Convolutional Neural Network and Adaptive Classifier Selection”. In: *Proceedings - International Computer Software and Applications Conference* 1 (June 2018), pp. 833–834. ISSN: 07303157. DOI: 10.1109/COMPSAC.2018.00143.

[41] Lichao Zhao et al. “Deep neural network based on android mobile malware detection system using opcode sequences”. In: *International Conference on Communication Technology Proceedings, ICCT* 2019-October (Jan. 2019), pp. 1141–1147. DOI: 10.1109/ICCT.2018.8600052.

[42] Wei Wang et al. “DroidEnsemble: Detecting Android Malicious Applications with Ensemble of String and Structural Static Features”. In: *IEEE Access* 6 (May 2018), pp. 31798–31807. ISSN: 21693536. DOI: 10.1109/ACCESS.2018.2835654.

[43] Roni Mateless et al. “Decompiled out based malicious code classification”. In: *Future Generation Computer Systems* 110 (Sept. 2020), pp. 135–147. ISSN: 0167-739X. DOI: 10.1016/J.FUTURE.2020.03.052.

[44] Subash Poudyal et al. “A Multi-Level Ransomware Detection Framework using Natural Language Processing and Machine Learning Evolutionary Data Mining View project Computer vision and image processing View project A Multi-Level Ransomware Detection Framework using Natural Language Processing and Machine Learning”. In: (). URL: https://www.researchgate.net/publication/336251881.

[45] Shahid Alam, R Nigel Horspool, and Issa Traore. “MAIL: Malware Analysis Intermediate Language-A Step Towards Automating and Optimizing Malware Detection”. In: (2013). DOI: 10.1145/2523514.2527006. URL: http://dx.doi.org/10.1145/2523514.2527006.

[46] El Mouatez Billah Karbab and Mourad Debbabi. “MalDy: Portable, data-driven malware detection using natural language processing and machine learning techniques on behavioral analysis reports”. In: *Digital Investigation* 28 (2019), S77–S87. ISSN: 17422876. DOI: 10.1016/j.diin.2019.01.017. arXiv: 1812.10327.

[47] Kartik Khariwal, Jatin Singh, and Anshul Arora. “IPDroid: Android malware detection using intents and permissions”. In: *Proceedings of the World Conference on Smart Trends in Systems, Security and Sustainability, WS4 2020* (July 2020), pp. 197–202. DOI: 10.1109/WORLDS450073.2020.9210414.

[48] Weina Niu et al. “OpCode-Level Function Call Graph Based Android Malware Classification Using Deep Learning”. In: *Privacy and Security Challenges for Internet of Things* Sensor Data Processing (2020). DOI: 10.3390/s20133645. URL: www.mdpi.com/journal/sensors.

[49] R. Vinayakumar, K. P. Soman, and Prabaharan Poornachandran. “Deep android malware detection and classification”. In: *2017 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2017* 2017-January (Nov. 2017), pp. 1677–1683. DOI: 10.1109/ICACCI.2017.8126084.

[50] Ali Aghamohammadi and Fathiyeh Faghih. “Lightweight versus obfuscation-resilient malware detection in android applications”. In: *Journal of Computer Virology and Hacking Techniques* 16.2 (2020), pp. 125–139. ISSN: 22638733. DOI: 10.1007/s11416-019-00341-y. URL: https://doi.org/10.1007/s11416-019-00341-y.

[51] El Mouatez Billah Karbab and Mourad Debbabi. “PetaDroid: Adaptive Android Malware Detection Using Deep Learning”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 12756 LNCS (2021), pp. 319–340. ISSN: 16113349. DOI: 10.1007/978-3-030-80825-9_16. URL: https://link.springer.com/chapter/10.1007/978-3-030-80825-9%7B%5C_%7D16.

[52] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. “DroidChameleon: Evaluating Android Anti-malware against Transformation Attacks”. In: (2013).

[53] Elmouatez Billah Karbab and Mourad Debbabi. *Resilient and Adaptive Framework for Large Scale Android Malware Fingerprinting using Deep Learning and NLP Techniques*. Tech. rep. 2021. arXiv: 2105.13491v1. URL: https://tinyurl.com/y4qdtuy9.

[54] El Mouatez Billah Karbab et al. “MalDozer: Automatic framework for android malware detection using deep learning”. In: *DFRWS 2018 EU - Proceedings of the 5th Annual DFRWS Europe* (2018), S48–S59. ISSN: 17422876. DOI: 10.1016/J.DIIN.2018.01.007.

[55] ElMouatez Billah Karbab et al. *Android Malware Detection using Machine Learning*. Vol. 86. Advances in Information Security. Cham: Springer International Publishing, 2021. ISBN: 978-3-030-74663-6. DOI: 10.1007/978-3-030-74664-3. URL: https://link.springer.com/10.1007/978-3-030-74664-3.

[56] Nan Zhang et al. “Hybrid sequence-based Android malware detection using natural language processing”. In: *International Journal of Intelligent Systems* 36.10 (Oct. 2021), pp. 5770–5784. ISSN: 1098111X. DOI: 10.1002/INT.22529.

[57] *strace*. URL: https://strace.io/.

[58] Tianshi Mu et al. “An Android Malware Detection Method Using Deep Learning Based on API Calls”. In: *Proceedings of 2019 IEEE 3rd Advanced Information Management, Communicates, Electronic and Automation Control Conference, IMCEC 2019* (Oct. 2019), pp. 2001–2004. DOI: 10.1109/IMCEC46724.2019.8983860.

[59] Dali Zhu et al. “FSNet: Android Malware Detection with only One Feature”. In: *Proceedings - IEEE Symposium on Computers and Communications* 2019-June (June 2019). ISSN: 15301346. DOI: 10.1109/ISCC47284.2019.8969711.

[60] Peng Xu, Claudia Eckert, and Apostolis Zarras. “Detecting and categorizing Android malware with graph neural networks”. In: *Proceedings of the ACM Symposium on Applied Computing* (Mar. 2021), pp. 409–412. DOI: 10.1145/3412841.3442080.

[61] Peng Xu. *Android-COCO: Android Malware Detection with Graph Neural Network for Byte-and Native-Code*. Tech. rep. 2021. arXiv: 2112.10038v2.

[62] Peng Xu et al. “HawkEye: Cross-Platform Malware Detection with Representation Learning on Graphs”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 12893 LNCS (2021), pp. 127–138. ISSN: 16113349. DOI: 10.1007/978-3-030-86365-4_11.

[63] Mohammad Reza Norouzian et al. “Hybroid: Toward Android Malware Detection and Categorization with Program Code and Network Traffic”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 13118 LNCS (2021), pp. 259–278. ISSN: 16113349. DOI: 10.1007/978-3-030-91356-4_14/COVER. URL: https://link.springer.com/chapter/10.1007/978-3-030-91356-4%7B%5C_%7D14.

[64] Peng Xu, Claudia Eckert, and Apostolis Zarras. “hybrid-Falcon: Hybrid Pattern Malware Detection and Categorization with Network Traffic and Program Code”. In: (Dec. 2021). DOI: 10.48550/arxiv.2112.10035. arXiv: 2112.10035. URL: https://arxiv.org/abs/2112.10035v2.

[65] Aditya Grover and Jure Leskovec. “node2vec: Scalable Feature Learning for Networks”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (). DOI: 10.1145/2939672. URL: http://dx.doi.org/10.1145/2939672.2939754.

[66] Tatiana Frenklach et al. “Android malware detection via an app similarity graph”. In: *Computers & Security* 109 (Oct. 2021), p. 102386. ISSN: 0167-4048. DOI: 10.1016/J.COSE.2021.102386.

[67] Abdurrahman Pektaş and Tankut Acarman. “Deep learning for effective Android malware detection using API call graph embeddings”. In: *Soft Computing* 24.2 (Jan. 2020), pp. 1027–1043. ISSN: 14337479. DOI: 10.1007/S00500-019-03940-5/TABLES/6. URL: https://link.springer.com/article/10.1007/s00500-019-03940-5.

[68] Mahmood Yousefi-Azar et al. “Byte2vec: Malware Representation and Feature Selection for Android”. In: *The Computer Journal* 63.8 (Aug. 2020), pp. 1125–1138. ISSN: 0010-4620. DOI: 10.1093/COMJNL/BXZ121. URL: https://academic.oup.com/comjnl/article/63/8/1125/5618685.

[69] Kewen Zou et al. “ByteDroid: Android Malware Detection Using Deep Learning on Bytecode Sequences”. In: *Communications in Computer and Information Science* 1149 CCIS (2020), pp. 159–176. ISSN: 18650937. DOI: 10.1007/978-981-15-3418-8_12/COVER. URL: https://link.springer.com/chapter/10.1007/978-981-15-3418-8%7B%5C_%7D12.

[70] Nan Zhang et al. “Deep learning feature exploration for Android malware detection”. In: *Applied Soft Computing* 102 (Apr. 2021), p. 107069. ISSN: 1568-4946. DOI: 10.1016/J.ASOC.2020.107069.

[71] Stuart Millar et al. “Multi-view deep learning for zero-day Android malware detection”. In: *Journal of Information Security and Applications* 58 (May 2021). ISSN: 22142126. DOI: 10.1016/J.JISA.2020.102718.

[72] Oluwafemi Olukoya, Lewis Mackenzie, and Inah Omoronyia. “Security-oriented view of app behaviour using textual descriptions and user-granted permission requests”. In: *Computers & Security* 89 (Feb. 2020), p. 101685. ISSN: 0167-4048. DOI: 10.1016/J.COSE.2019.101685.

[73] Alper Egitmen et al. “Combat Mobile Evasive Malware via Skip-Gram-Based Malware Detection”. In: *Security and Communication Networks* 2020 (2020). ISSN: 19390122. DOI: 10.1155/2020/6726147.

[74] Zhuo Ma et al. “Droidetec: Android Malware Detection and Malicious Code Localization through Deep Learning”. In: (Feb. 2020). DOI: 10.48550/arxiv.2002.03594. arXiv: 2002.03594. URL: https://arxiv.org/abs/2002.03594v1.

[75] Annamalai Narayanan et al. “Apk2vec: Semi-Supervised Multi-view Representation Learning for Profiling Android Applications”. In: *Proceedings - IEEE International Conference on Data Mining, ICDM* 2018-November (Dec. 2018), pp. 357–366. ISSN: 15504786. DOI: 10.1109/ICDM.2018.00051. arXiv: 1809.05693.

[76] Xinjun Pei et al. “Combining multi-features with a neural joint model for Android malware detection”. In: *Journal of Intelligent & Fuzzy Systems* 38.2 (Jan. 2020), pp. 2151–2163. ISSN: 1064-1246. DOI: 10.3233/JIFS-190888.

[77] Tieming Chen et al. “DroidVecDeep: Android Malware Detection Based on Word2Vec and Deep Belief Network”. In: *KSII Transactions on Internet and Information Systems (TIIS)* 13.4 (Apr. 2019), pp. 2180–2197. ISSN: 1976-7277. DOI: 10.3837/TIIS.2019.04.025. URL: http://doi.org/10.3837/tiis.2019.04.025.

[78] Trung Kien Tran and Hiroshi Sato. “NLP-based Approaches for Malware Classification from API Sequences”. In: *Asia Pacific Symposium on Intelligent and Evolutionary Systems* (2017). ISSN: 1098-6596. eprint: arXiv:1011.1669v3.

[79] Yuta Nagano and Ryuya Uda. “Static Analysis with Paragraph Vector for Malware Detection”. In: (2017). DOI: 10.1145/3022227.3022306. URL: http://dx.doi.org/10.1145/3022227.3022306.

[80] Ryo Ito and Mamoru Mimura. “Detecting unknown malware from ASCII strings with natural language processing techniques”. In: *Proceedings - 2019 14th Asia Joint Conference on Information Security, AsiaJCIS 2019* (2019), pp. 1–8. DOI: 10.1109/AsiaJCIS.2019.00-12. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=%7B%5C&%7Darnumber=8827005.

[81] Abhilash Hota and Paul Irolla. “Deep neural networks for android malware detection”. In: *ICISSP 2019 - Proceedings of the 5th International Conference on Information Systems Security and Privacy* (2019), pp. 657–663. DOI: 10.5220/0007617606570663.

[82] Ming Fan et al. “CTDroid: Leveraging a Corpus of Technical Blogs for Android Malware Analysis; CTDroid: Leveraging a Corpus of Technical Blogs for Android Malware Analysis”. In: *IEEE TRANSACTIONS ON RELIABILITY* 69.1 (2020). DOI: 10.1109/TR.2019.2926129. URL: http://ieeexplore.ieee.org..

[83] Yue Liu et al. “Deep Learning for Android Malware Defenses: a Systematic Literature Review”. In: *J. ACM* 37 (2021), p. 51. DOI: 10.1145/1122445.1122456. arXiv: 2103.05292v1. URL: https://doi.org/10.1145/1122445.1122456.

[84] *Methodology - definition of methodology by The Free Dictionary*. URL: https://www.thefreedictionary.com/methodology (visited on 03/15/2022).

[85] Herbert A. (Herbert Alexander) Simon. *The sciences of the artificial*. eng. 3rd ed. Cambridge, Mass: MIT Press, 1996. ISBN: 9780262691918.

[86] Roel Wieringa. *Design science methodology for information systems and software engineering*. eng. Heidelberg: Springer, 2014. ISBN: 9783662438381.

[87] University of Luxembourg. *Androzoo home*. 2016. URL: https://androzoo.uni.lu/ (visited on 07/21/2021).

[88] Kevin Allix et al. “AndroZoo: Collecting millions of Android apps for the research community”. In: *Proceedings - 13th Working Conference on Mining Software Repositories, MSR 2016* (2016), pp. 468–471. DOI: 10.1145/2901739.2903508.

[89] Artem Kushnerov. *ArtemKushnerov · GitHub*. 2018. URL: https://github.com/ArtemKushnerov (visited on 07/21/2021).

[90] *Android Architecture — Android Open Source Project*. URL: https://source.android.com/devices/architecture/ (visited on 03/02/2022).

[91] *App Manifest Overview — Android Developers*. URL: https://developer.android.com/guide/topics/manifest/manifest-intro.

[92] *Android Runtime (ART) and Dalvik — Android Open Source Project*. https://source.android.com/devices/tech/dalvik. URL: https://source.android.com/devices/tech/dalvik (visited on 07/29/2021).

[93] *Application Fundamentals — Android Developers*. URL: https://developer.android.com/guide/components/fundamentals (visited on 03/02/2022).

[94] *TheCyberian/manifestAndDexExtraction*. URL: https://github.com/TheCyberian/manifestAndDexExtraction.

[95] *Doc2Vec Model — gensim*. https://radimrehurek.com/gensim/auto_examples/tutorials/run_doc2vec_lee.html. URL: https://radimrehurek.com/gensim/auto%5C_examples/tutorials/run%5C_doc2vec%5C_lee.html (visited on 08/31/2021).

[96] Utkarsh Raghav, Elisa Martinez-Marroquin, and Wanli Ma. "Static Analysis for Android Malware detection with Document Vectors". In: *IEEE International Conference on Data Mining Workshops, ICDMW* 2021-December (2021), pp. 805–812. ISSN: 23759259. DOI: 10.1109/ICDMW53433.2021.00104.