A Performance-Sensitive Malware Detection System Using Deep Learning on Mobile Devices

PROJECT REPORT

Submitted by

ATHIRA JOSHI

REG NO : TKM20CSCE03

In partial fulfillment for the award of the degree of

MASTER OF TECHNOLOGY
IN
COMPUTER SCIENCE AND ENGINEERING

Under the guidance of
Dr. ANSAMMA JOHN
Dr. MANU J PILLAI

Thangal Kunju Musaliar College of Engineering
Kerala

AUGUST 2022

Thangal Kunju Musaliar College of Engineering
Dept. of Computer Science & Engineering

C E R T I F I C A T E

This is to certify that,this report titled ***A Performance-Sensitive Malware Detection System Using Deep Learning on Mobile Devices*** is a bonafide record of the **Project** presented by **ATHIRA JOSHI(TKM20CSCE03)**, under our guidance and supervision, in partial fulfillment of the requirements for the award of the degree, **M.Tech Computer Science & Engineering** in **APJ Abdul Kalam Technological University** .

Coordinators

Dr. Ansamma John
Professor
Computer Science & Engineering

Dr. Manu J Pillai
Assistant Professor
Computer Science & Engineering

Head of the Department

Dr. Dimple A Shajahan
Associate Professor
Computer Science & Engineering

ACKNOWLEDGEMENT

A successful project is a fruitful culmination of efforts by many people, some directly involved and some others indirectly, by providing support and encouragement. Firstly I would like to thank the almighty for giving me the wisdom and grace for making my project a memorable one. I thank him for steering me to the shore of fulfillment under his protective wings.

I express my sincere gratitude to **Dr. T A Shahul Hammed** , Principal of T.K.M College of Engineering for giving me an opportunity to present my project. I would like to thank **Dr. Dimple A Shajahan**, Associate Professor and Head of the Department, CSE, TKMCE, for her constant support and encouragement throughout the project work.

With a profound sense of gratitude, I would like to express my heartfelt thanks to my guides **Dr. Ansamma John**, Professor, CSE, TKMCE and **Dr. Manu J Pillai**, Assistant Professor, CSE, TKMCE for their expert guidance, cooperation and immense encouragement. I also extend my thanks to the entire faculty and staff of the Department of Computer Engineering, TKMCE, who has encouraged me throughout this work.

I also express my thanks to my loving parents, brother and friends, for their support and encouragement in the successful completion of this project work.

Athira Joshi

Abstract

Concerns about the threat of piracy and the spread of viruses have been raised as a result of the rise in popularity of Android applications. Through applications, malware is frequently propagated in the mobile environment. People frequently grant applications rights without fully understanding their purpose. Android devices are used by almost 2.8 billion people worldwide. Therefore, the requirement for a reliable Android malware detection system is unavoidable. In order to provide a real-time and responsive detection environment for Android mobile devices, this paper proposes MobiTive, an efficient Android malware detection system. A more practical and secure application solution is MobiTive, which checks and monitors APK files before installation. MobiTive is concentrating on mobile and server-side platforms. APK files are mined for API calls and Manifest files of both malware and benign apps, and this raw data is then fed into convolutional neural networks. After being trained independently, networks including GRU, BI-GRU, STACKED GRU, and LSTM were shown to be of good performance. Its shown that BI-GRU have the highest performance. As quantization is done for mobile device adaptation, network model size varies. Deep Learning model must be converted to Tensorflow-lite model due to limited computing power and resource availability in order to function well on the mobile platform. MobiTive has been evaluated for effectiveness on a variety of mobile devices, and it achieves detection accuracy of 90.26% percent.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As there are more Android devices and programmes (apps) available nowadays, security and privacy concerns are growing as well. Users keep their personal information on mobile devices using a variety of well-known social, shopping, and banking apps. Malware is one of the biggest risks on the Internet today (such as viruses, worms, and Trojan horses). The ease with which new viruses can be produced utilising generation tools causes the amount of malware to quickly increase. As a result, during the past 10 years, hackers have focused more on mobile apps, which is now one of the biggest security dangers in the industry. These assaults may result in account theft, personal data loss, financial loss, denial-of-service attacks, and many other negative effects.
Detecting Android malware is a major problem. Malware (Malicious software) is mostly designed to infect individual computers or an entire organization's network. Users tend to expect a safe environment that is provided by the app markets. Users consider their app sources are all trustable and secure. Android malware detection approaches are mainly such as signature based approaches, behavior-based approaches, data-flow analysis-based approaches and currently Machine learning-based approach is one of the most promising techniques in detecting Android malware. Deep learning has achieved tremendous success in Android malware detection due to the availability of data and hardware. When compared to traditional methods of machine learning techniques, deep learning techniques are trained through feature learning rather than task-specific algorithms.
The "WannaCry" ransomware spread to over 100 countries and cost 8 billion US dollars in damages. Additionally, these new variants of malicious code behave similarly to benign code, making them more difficult to detect, posing a significant problem for anti-virus companies. Despite the fact that several analysis methods have been researched to deal with malware varieties, they are insufficient to handle the rising avoidance methods used in malware.Given how many real-world Android apps there are - for example, there are more than 3 million Android apps available on the Google Play Store and complete detection of all of them takes a lot of time. Moreover,the apps from unofficial markets and third-party resources are more vulnerable. The security of these kinds of apps is indeed unpredictable and uncontrollable.Additionally, third-party and unofficial app stores offerings are more open to attack. These types of apps security is in fact erratic and unmanageable.
The two most common methods for detecting malware are (i) signature-based methods and (ii) anomaly-based methods. The detection of known malware is highly quick and effective with signature-based techniques. A set of signatures is used by the majority of commercial anti-malware approaches used today to identify dangerous applications. A file's or its parts' cryptographic hash or a sequence of bytes in the file could both be represented by a signature.If the signature is absent from the dataset, it indicates that the file is not harmful at all (malware). The majority of malware, however, has the ability to create new variations each time it is executed and a new signature is created. As a result, signature-based techniques are unable to identify malware that is not already in the database. The need for human expertise to update the signature database with new signatures is another another drawback of signature-based detection techniques.
When identifying these apps, the traditional server-side based malware detection method

has unavoidable downsides : (1) it takes a long time to upload the apps to the server before installation, especially for large apps; (2)Internet uploading is not a secure method. For instance, hackers could change the malware during the uploading process so that a false "benign" result is given. The users will therefore install the malware. The necessity for a last line of protection on mobile devices is therefore inevitable. Here, this aims to perform Android malware detection on mobile devices rather than on the server side to address the serious issue.

Although machine learning-based methods have outperformed previous methods in detecting Android malware, the goal of this work is to transfer trained deep learning (DL) models from the server to mobile devices. While a computationally intensive deep learning software could be executed effectively on a server with GPU support, such deep learning models typically cannot be deployed and executed directly on other platforms supported by small mobile devices due to a variety of computation resource limitations, such as the computation power, memory size, and energy.

Developers compile their source code and other components into an Android application package, which is then executed by Android devices (APK). APKs are compressed application files for the Android platform that include resources, assets, a manifest file (androidManifest.xml), Dex files, and more. The meta-data for Android apps is contained in the manifest file, which specifies the package name and application ID, app components (such as Intent filters, activities, and services), permissions, and device compatibility (e.g., uses-feature and uses-sdk).

Dex files have the extension for Dalvik executable files, which can be transformed from Java bytecode using an alternative instruction set and run on the Dalvik virtual machine in Android OS. There are 21 different types of items in a Dalvik executable file, which can be further broken down into programme information and metadata. The header, checksum, signature, and other metadata are used to convey information about the Dalvik executable file. The size and offset values of programme information (such as class definition identifiers, method identifiers, type identifiers, and string identifiers), as well as the map offset, which offers explicit mappings between static information, come next (e.g., strings and method names).

MobiTive is a device-end solution that effectively defends mobile devices against malware threats in real-time by utilising binary characteristics and specialised deep neural networks. Instead of identifying malware on public servers or keeping track of it after installation, this research intends to detect it directly on mobile devices as a pre-installed and run-time solution. In this work, TensorFlow Lite is used to migrate the deep learning models.

The application market, the Android OS platform, and actual device-end factors can be used to categorise the existing security procedures.When an APK is uploaded, the official market for Android, or Google Play Store, offers a security verification. Google, for example, offers security supported by its machine learning system. Some reputable third-party markets offer security checks for the uploaded programmes as well. For instance, ApkMirror executes a protection service offered by GuardSquare in addition to performing the signature verification. However, the majority of the security check services currently offered by third-party markets are quite basic and constrained. Only a signature verification is present in some of them, which is readily evaded.

Users must therefore install and utilise apps they download from third-party markets at their own security risk. The most well-known security programmes, such as Avast and AVG,

primarily offer their antivirus services by keeping an eye on the privacy-sensitive elements (such as run-time permission requests) and checking the app's signature against their local or cloud virus database. In addition to external protection, Android OS offers several robust built-in security features, such as application sandboxing, etc. Every application has its own unique execution environment thanks to the application sandbox mechanism. As a result, an application's assault can only target the components that it has specifically requested. The attack will never be able to use the Bluetooth functionalities if the programme does not require Bluetooth permissions and actions like activities.

A DL model frequently goes through either quantization, platform migration, or both steps before being distributed to end-user applications after it has completed the training phase and is prepared to be deployed to a target device. This is because to the training phase's extensive computational and energy requirements. More data is required to train the network until it achieves optimality as the model size and task complexity increase, which could take days or even weeks of training on high performance GPU clusters. On the other hand, the environment in which DNN models are deployed is typically resource-constrained and has low levels of computing, power, etc.

A DL model frequently goes through a modification phase to respond to specific software and hardware constraints of a target platform due to environment variations between a target platform (e.g., mobile phones, green energy embedded systems) and training platform (e.g., generally equipped with GPUs). When moving a big DL model trained on a cloud system to a mobile or IoT device with low processing power, quantization, which lowers the precision of a DL model to increase computation efficiency, memory consumption, and storage space, has become a frequent approach.

Chapter 2

Related Works

2.1 Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach[1]

Security on smartphones is seriously threatened by the development of mobile malware. The most recent machine learning-based malware detection technologies (such DREBIN, DROIDAPIMINER, and MAMADROID) are no longer useful due to sophisticated attackers' ability to adapt by severely undermining classifiers by contaminating training data. In this study, Chen. [1] investigate the viability of building crafted malware samples, look at three different threat models to see how machine-learning classifiers can be tricked, and come to the conclusion that adding carefully crafted data to training data can significantly lower detection accuracy. They suggested KUAFUDET, a two-phase learning augmenting technique that learns mobile malware through adversarial detection, to address the issue. Both an online detection phase, which makes use of the training set's features, and an offline training phase are included in KUAFUDET.

2.2 Droiddetector: Android malware characterization and detection using deep learning[2]

A deep learning-based online Android malware detection engine (DroidDetector) developed by Yuan et al.[2] can automatically determine if an app is malicious or not. It rigorously test DroidDetector against thousands of Android apps and conduct an in-depth investigation of the factors that deep learning fundamentally leverages to identify malware. The findings demonstrate that deep learning is useful for defining Android malware and is more successful when more training data is available. DroidDetector exceeds conventional machine learning methods with a detection accuracy of 96.76%. An analysis of ten well-known antivirus programmes shows how urgent it is to improve our capacity for Android malware detection.

2.3 Adaptive and scalable Android malware detection through online learning[3]

A novel online machine learning-based system called DroidOL is proposed by Narayanan et al.[3] to solve the issue of malware population drift and successfully detect malware. A cutting-edge graph kernel is used to capture security-sensitive app behaviours as inter-procedural control-flow sub-graph features in order to achieve precise detection. An online passive-aggressive classifier is utilised to carry out scalable detection and to adapt to the drift and evolution in malware population. DroidOL outperforms two cutting-edge malware approaches by more than 20% in their typical batch learning scenario and by more than 3% when they are continually re-trained in a large-scale comparison investigation with more than 87,000 programmes.

2.4 MalDozer: Automatic framework for Android malware detection using deep learning[4]

MalDozer is a deep learning-based system for automatically identifying malware families and detecting it on Android devices. MalDozer intelligently extracts and learns the malicious and benign patterns from the actual samples to identify Android malware, starting from the raw sequence of the app's API method calls. MalDozer can be used as a general-purpose malware detection solution that is installed on mobile and even Internet of Things (IoT) devices in addition to servers. MalDozer is evaluated here using several Android malware datasets, including 38 K benign apps and 1 K to 33 K malware apps. With an F1-Score of 96 to 99 percent and a false positive rate of 0.06 to 2 percent, the results demonstrate that MalDozer can accurately detect malware and attribute them to their actual families.

2.5 Apk2vec: Semisupervised multi-view representation learning for profiling Android applications[5]

A semi-supervised Representation Learning (RL) framework called apk2vec was created by Yiu et al. [5] to automatically create a compact representation (also known as a profile or embedding) for a certain app. The three distinct qualities listed below, in particular, make apk2vec a superior option for extensive app profiling: As a semi-supervised embedding technique, it may use labels associated with apps (such as malware family or app category labels) to build high quality app profiles. Additionally, it combines RL with feature hashing, which makes it possible to effectively create profiles of apps that stream over time (i.e., online learning). After that, a range of applications might be made use of the semi-supervised multi-view hash embeddings of programmes.

2.6 Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic Android malware analysis[6]

DroidScope, an Android analysis platform that carries on the heritage of virtualization-based malware analysis, was introduced by Yan et al. [6]. DroidScope reconstructs both the OS-level and Java-level semantics simultaneously and fluidly, in contrast to current desktop malware analysis solutions. DroidScope exports three tiered APIs that correspond to the three tiers of an Android device: hardware, OS, and Dalvik Virtual Machine in order to simplify custom analysis. In addition to building on DroidScope, it also created a number of analysis tools to gather comprehensive native and Dalvik instruction traces, profile API-level activities, and monitor data leaks through both Java and native components using taint analysis. These tools have shown to be efficient in analysing malware samples from the real world and have manageably low performance overheads.

2.7 FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps[7]

FLOWDROID, a brand-new and incredibly accurate static taint analysis for Android applications, was introduced by Steven et al. [7]. Context, flow, field, and object sensitivity

enable the analysis to limit the frequency of false alarms while allowing the analysis to handle callbacks fired by the Android framework with suitable handling. FLOWDROID is able to simultaneously retain great efficiency and precision because to novel on-demand algorithms. Additionally, they suggested DROIDBENCH, an open test framework for assessing the efficacy and precision of taint-analysis methods designed exclusively for Android apps. FLOWDROID is a package of tests that uses SecuriBench Micro, DROIDBENCH, and a number of well-known Android test apps to discover a significant portion of data leaks while minimising the incidence of false positives. FLOWDROID obtains a 93 percent recall rate on DROIDBENCH.

2.8 StormDroid: A Streaminglized Machine Learning-Based System for Detecting Android Malware[8]

Nowadays, malware assaults are more likely to target mobile devices because of the rising trend in app downloads. Effective malware detection (MD) continues to be a serious difficulty despite the enormous security and privacy concerns it received. This study addresses this issue by developing StormDroid, a streaminglized machine learning-based MD system. StormDroid's foundation is machine learning, which has been strengthened with a novel combination of contributed features that have been observed over a sizable body of data. Additionally, the entire MD process has been stream-lined to support large-scale analysis, creating an effective and scalable MD technique that monitors app behaviours both statically and dynamically. In comparison to cutting-edge antiviral systems, the combination of donated features enhances MD accuracy by about 10% when tested on almost 8,000 apps.

Chapter 3

Methodology

3.1 Overview of MobiTive

MobiTive is suggested as a means of accomplishing the objective, and its functionality can be separated into two primary categories (i.e., parts of server side and mobile side). The system's feature preparation, DL model training, model migration, and quantization are all found in the first section. The second phase involves deploying the migrated/quantized models on mobile devices. Multiple features (such as manifest attributes, API calls, and opcode sequences) were extracted from decompiled apps in the prior study [12]. The paper suggests a new feature extraction method that is immediately extract and vectorize the Android Permissions and API calls from APK files rather than decompiling APK into source code like smali code. In Fig. 3.1, an overview of MobiTive is presented.

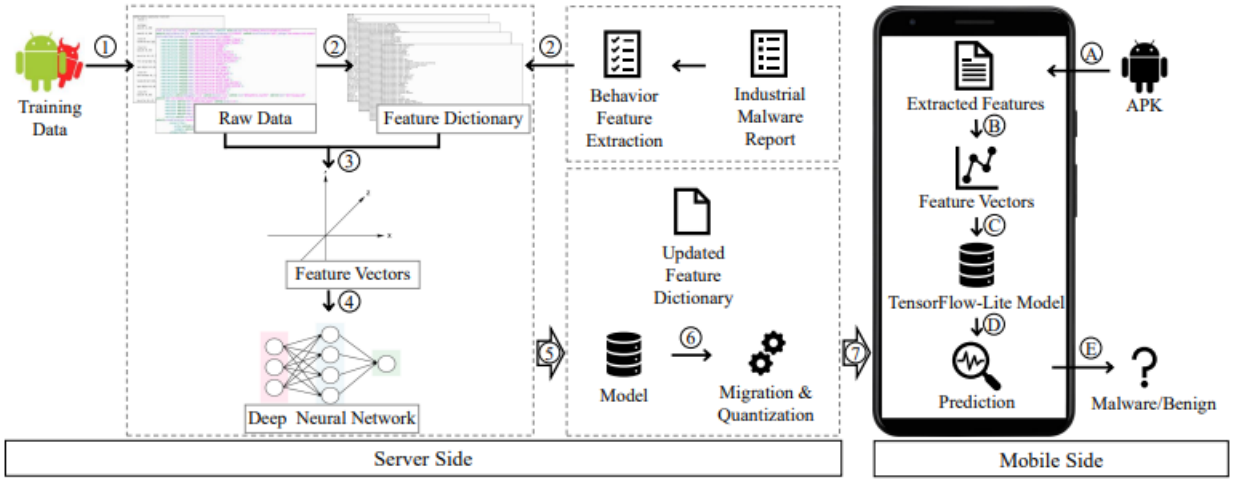

Figure 3.1: Overview of MobiTive

Here, create the feature dictionary by combining a performance-based feature selection mechanism and a behavior-based feature updating method (step 2). The first portion enables to offer a trained DL model and a feature dictionary (step 5) for the second part using the tailored deep neural networks and extracted feature vectors (steps 3 and 4). Then convert the pre-built DL model to a TensorFlow Lite model to make the model mobile device-adaptive. Additionally, a quantization phase - a general method for reducing model size that also offers decreased latency with no loss in accuracy is offered as a performance improvement for mobile devices (step 6). The second section downloads the quantized DL model and feature dictionary into portable devices, as shown in Fig.3.1. Following that, MobiTive can extract feature vectors from an application that has been downloaded from the market or a third-

party market and give the finished result to MobiTive (steps A-C). After making predictions using the loaded DL model, determine if the downloaded Android app is malware or not using the predictive output (steps D-E).

3.2 Feature Preparation

In the device-end scenario, manifest properties are chosen based on the performance-based feature selection technique (Feature Selection). The permissions that an app needs to have in order to access data from other apps are declared in the Android manifest file. A behavior-based approach based on industrial malware reports is also proposed here in order to update the feature dictionary and enhance the representatives (Feature Updating). Instead of decompiling the package to acquire the APK's functionality, unzip it to save time. Extract the features from the unzipped binary files' raw data (Feature Extraction).
1) **Feature Selection**: Android malware may often be found using manifest information like used permissions, intents, and hardware features. The feature extraction process benefits from the ease with which existing tools can decode the AndroidManifest.xml file from the APK file. It is a lightweight feature type that a system like MobiTive, which is performance-sensitive, might use. Since practically all malicious behaviours will be represented through API calls, they are more representative and significant feature categories in terms of usefulness. In addition to the specific API call, an API call sequence may also include additional semantics, such as an opcode code sequence. However, because it involves studying source code or smali code, the extraction process for these two feature categories takes a long time. Selecting unzipping APK to extract raw data first and choosing API calls and manifest properties as feature types due to the better extraction performance compared to others is done by comparing the performance (time cost) of all these potential feature types based on the two different extraction steps, namely raw data and feature vector extraction (steps 1 and 3). Create a feature dictionary (step 2) using the two feature types chosen based on performance comparison in order to obtain the feature vectors (step 3). Build the manifest property dictionary specifically by according to the guidelines in the official Android handbook. Remove the obfuscated API calls first. Second, remove any API calls, like the View loading API, that aren't connected to potentially dangerous actions.
2)**Feature Updating**: A more thorough feature coverage of dangerous activities produces greater benefits because the efficacy of machine learning-based detection systems heavily rely on the chosen features. MobiTive. Collects hundreds of reports on commercial malware from Symantec Threats to enhance the feature coverage of dangerous behaviour. Performs a manual analysis of the gathered text-based data and, as an addition to the features chosen, summarises 23 categories of fundamentally possible dangerous activities. It should be noted that the malware reports describe the malicious actions and the essential code aspects, such as manifest properties and API calls. Additionally, three co-authors execute a behavior-based feature understanding and verification to validate the manual results. Consequently, in addition to the features in the original feature dictionary, there are a total of 46 additional API calls and 12 new manifest attributes that are changed for the new feature dictionary. Include the package name in the new API requests as well. Extract all the API calls contained in a package named "android/net/Uri," for instance, if a new API call has this name.

Input		
Reshape	input	(None, 2915,)
	output	(None, 1, 2915)
GRU/LSTM	input	(None, 1, 2915)
	output	(None, 128)
Dropout		
Softmax Classification		

Table 3.1: Deep Neural Network Architecture: GRU and LSTM

3)**Feature Vector Extraction**: As mentioned in feature selection, the traditional feature vector extraction methods cause a lot of time due to the cost of decompiling and extracting from source code such as Java code and smali code. To improve the extraction performance, we propose a novel feature vector extraction method from binary code instead of source code. Specifically, by analyzing the inner architecture of Dalvik binary file (classes.dex), its found that there exists an API table, which is used to match the executable symbols and API strings. Also extract API calls by parsing the API table in classes.dex file based on the address and offset defined in the metadata. Meanwhile, to get access to the information in binary format AndroidManifest.xml, firstly generate a standard output with a XML decoder, Axmldec. By analyzing the decoded manifest file, the manifest properties can be extracted.

3.3 DL Model Construction

1) **DL Model Training**: Customize the RNN models to adopt the device-end scenario and enhance performance, as shown in Table 3.1 Deep Neural Network Architecture: GRU and LSTM and Deep Neural Network Architecture: Bidirectional GRU and LSTM. For simple RNNs in Table DeepNeural Network Architecture: GRU and LSTM the first computational layer is an LSTM/GRU layer with 128 neural units.Comparing GRU and LSTM, its found that Bidirectional GRU gives highest accuracy among other networks.The sequence processing paradigm known as a bidirectional GRU, or BiGRU, consists of two GRUs. One processing the

information forward and the other processing it backward. Only the input and forget gates are present in this bidirectional recurrent neural network. The long-term dependency problem is addressed by both LSTM and GRU; the differences are in the quantity of operations and processing time. Newer, faster, and less expensive computationally is GRU.A BRNN is made up of two RNNs, one of which starts moving forward from the beginning of the data sequence and the other which starts moving backward from the beginning of the data sequence. A BRNN's network blocks may be LSTMs, GRUs, or straightforward RNNs. The backward training procedure is supported by a second hidden layer in BRNNs. Then, there will be a dropout layer, the dropout rate is 0.5. At last, the result is passed to a softmax classifier function to get the final training result.

2) **DL Model Migration and Quantization**: Convert and deploy pre-trained DL model to mobile devices conversion of the pre-trained model to a TensorFlow Lite model, which is

supported by Android operating system (step 6) Transfer the TensorFlow model to a mobile readable TensorFlow Lite model with a TensorFlow Lite converter. Apart from the model migration, also quantize pre-trained model to improve the performance on mobile devices, which does not influence the accuracy of detection significantly.Reducing the precision of the weights, biases, and activations to make them less memory-intensive is known as quantization. The neural network is frozen when training is complete, making it impossible to adjust its parameters. Subsequently, parameters are quantized. The post-training parameters are not changed; instead, the quantized model is deployed and used to do inference. In the experiments, measure the performance of accuracy and time cost affected by the model migration and quantization.

3.4 Real-Time Detection System

Before conducting a real-time detection, the quantized TensorFlow Release the lightweight model and feature dictionary. to the detection system in advance (step 7). There are three important steps before completing the forecast. The initial step is feature preparation for MobiTive. the moment an APK file step A, MobiTive first unzips the file into the original Classes.dex, AndroidManifest.xml, and other assembly files other sources. API call characteristics and manifest features will be taken out appropriately. An API parser is implemented to based on the classes.dex, directly extract the API calls. knowledge of Dalvik's binary code. given that the raw binary Since AndroidManifest.xml cannot be directly analysed, we AXML [34], a third-party decoder library, to obtain the decoded manifest document.All of the two different feature types are combined into a vector and sent into the TensorFlow Lite model (step C). MobiTive can make the prediction in step D using the quantized model, and in step E it can provide feedback based on the final forecast outcome. The system can raise a warning based on the prediction result to assist users in preventing the installation of the identified malware and further save its information (such as name, version, and checksum) in a local database. In addition to the activities on local devices, there are still two other options: reporting the information about malicious programmes to the relevant market and syncing the malware information with the update server. To deploy an update for the MobiTive in practice, the service provider firstly need to collect the new detected malware and update the training dataset. After updating, it will be able to obtain a new pre-trained model on server. Then, the new model can be packed as a system patch and deployed to devices within an update directly. As a result, the updated system surely will improve the effectiveness and robustness of the protection on device based on the new delivered model. The MobiTive serves the API parser as an external lib file. Technically, there are three key components that make up MobiTive's classification functionality on Android smartphones. In contrast to the well-known high level API offered by Keras, bytebuffer is the fundamental data structure used in computation using TensorFlow Lite on Android devices. As a result, the input vector and model will first be converted into bytebuffer format. Second, by loading the model into a TensorFlow Lite interpreter, the result matrix may be obtained by feeding the interpreter's input bytebuffer. The final prediction result can then be acquired by applying an argmax function to the result matrix.

Chapter 4

Experimental results and discussion

4.1 Experiment Environment

On a server running Ubuntu 16.04, which has two Intel Xeon E5-2699 V3 CPUs, 192GB of RAM, and an NVIDIA GeForce 2080Ti GPU, the tests are conducted. To evaluate the efficacy and accuracy of the method on real mobile devices, we've selected six different Android mobile devices. There are four of them, including the Nexus 6P, Huawei Mate 10, HTC U11, and LG G6, whereas the Huawei P30 is a flagship model and the LG G6 is a low-profile model (Samsung Galaxy J7 Pro). The server's system is implemented in Python 3. Additionally, to access the unprocessed data and functionality, seven different kinds of currently accessible tools—axmldec, AXML, ApkTool, AndroGuard, Dex2jar, Soot, and FlowDroid—were used. The C++ project axmldec can decode a binary manifest file into a readable XML format file. The processing of binary Android XML files is the sole purpose of the AXML library. It was produced in Java and can be used as an external library in an Android application. The apk file can be decompiled using the reverse engineering programme ApkTool, which also produces resources like manifest and smali files. AndroGuard, a Python utility, has the ability to decode resources and translate bytecode into Java code. With the help of AndroGuard, we can also easily construct the call graphs (CG) and data-flow graph for an Android app.Tools for working with Java.class and Android.dex files are available through the Dex2jar project. The Java optimization framework Soot can be used to extract the call graph (CG). FlowDroid is a static taint analysis tool for Android apps. It is utilised to produce an interprocedural control-flow graph (ICFG). In addition to the aforementioned modern tools for feature extraction, JitPack is a cutting-edge package repository for JVM and Android projects that can compile the project into a ready-to-use artefacts (i.e., jar and aar). The deep neural networks and training projects are implemented using the Keras, Numpy, Scikit-learn, and TensorFlow libraries.

4.2 Effectiveness of Feature Extraction, Feature Updating, Feature Category Selection, and Neural Network Selection

1) Performance evaluation of feature extraction: In order to demonstrate the performance benefits of chosen features, the feature extraction period in this experiment is divided into two portions, APKraw datafeature and feature preparation phase technological methods.
a) Dataset: Datasets ae collected from Drebin and VirusShare that are labelled as Malware. Drebin has an Original size of 5,560 and de-duplicated size 3,561 and also VirusShare has a size of 20,000 and when de-duplicated it has a size of 12,660. Benign datasets are collected from Benign apps as random
b) Setup: This experiment first evaluates the extraction time (APK→raw data) of 2 different raw data types. Here, assess the ability to extract two distinct feature types (i.e., manifest characteristics and API requests) from two different categories of raw data (i.e., decompiling and unzipping). Here, utilise an XML tag parser to retrieve manifest properties from the manifest file that ApkTool decompiled for the decompiled manifest and smali files. To extract

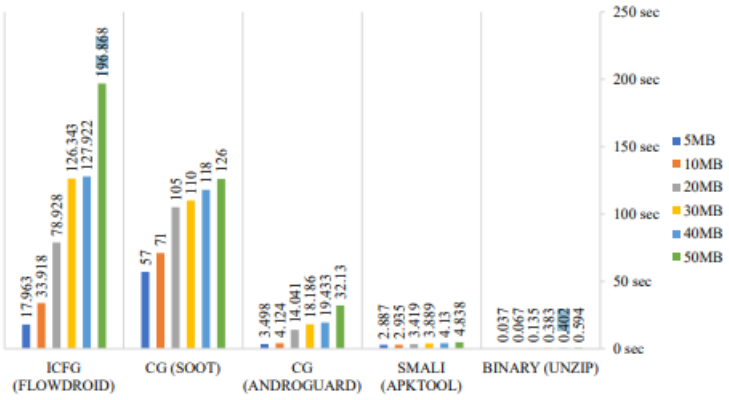

Figure 4.1: Extraction time of different raw data types

API calls, directly match the Smali files and API call dictionary to get the desired result. Therefore, by comparing each smali file to the opcode list, extract the opcode sequences for each one. Applying Axmldec will allow you to extract manifest properties from the binary manifest file. Extract API calls by directly loading the API table with the offset and size specified in the Dalvik binary's metadata.

4.2.1 Results

The outcomes are illustrated from the two perspectives below (APK-raw data and raw data-feature).
1) Extracting raw data : The ICFG and CG extraction times are too long for performance-sensitive approaches like MobiTive to allow. In particular, extracting ICG with FlowDroid takes an average of 17.963 seconds for 5MB apps and 196.868 seconds for 50MB apps. When using Soot to generate CG, it takes 126 seconds for 50MB apps and 57 seconds for 5MB apps, while when using AndroGuard, it costs 32.13 seconds and 3.498 seconds, respectively. On CG extraction, AndroGuard performs better than Soot.
a) Dataset: As can be seen in Table 4.1 of the used dataset, this collection includes more than 16,000 Android applications for evaluation purposes. These apps contain Android malware, while some others were picked up clean from the Google Play Store. On the legitimate market, they might be malware, nevertheless. Upload the files to VirusTotal, an online antivirus service with over 60 security scanners, to do a verification in order to filter the suspected malware as much as feasible.

In conclusion, the following studies will require a sizable dataset of both benign and malicious samples. There are many duplicated examples because the dataset was compiled from several sources. Therefore, among harmful and benign apps, we run a hash check to remove redundant programmes. Due to the API parser's capabilities, several cases of failed data prepossessing, which includes processes for feature vector generation and decompiling raw data, were encountered. The other errors are just brought on by the faulty APK packages;

Datasets	Labels	Original Size	Deduplicated Size
Drebin	Malware	5,560	3,561
VirusShare	Malware	20,000	12,660
Total	Malware	25,560	16,221
Total	Benign	25,284	16,000

Table 4.1: Used Dataset

Categories	Neural Networks	Accuracy	Precision	Recall
Manifest	CNN	79.89%	79.50%	80.12%
API Calls	CNN	83.17%	82.33%	83.90%
Two Types	CNN	85.11%	85.06%	85.16%
Two Types	LSTM	86.56%	86.72%	86.40%
Two Types	GRU	86.75%	86.78%	86.72%
Two Types	Stacked LSTM	86.64%	86.83%	86.46%
Two Types	Stacked GRU	86.67%	87.00%	86.36%
Two Types	Bidirectional LSTM	86.61%	86.67%	86.56%
Two Types	Bidirectional GRU	86.78%	87.00%	86.57%

Table 4.2: Detection results of feature categories and networks

simply remove them. In order to execute the tests that follow, here have chosen 1000 benign and 1000 malicious samples from the dataset, respectively. Divide these 500 malicious and 500 benign apps into three groups for training purposes: training data (80%) and testing data (20%).
Results: The accuracy of the updated feature version on CNN, LSTM, and GRU is 85.11%, 86.56%, and 86.78%, respectively, as shown in Table 4.2 Detection Results of Feature Updating. After feature update, there is about a 1%–5% improvement over the previous findings. Accept updating features that have been condensed from potentially dangerous actions as a part of the input feature set based on the outcome.
2) Accuracy evaluation of feature category selection and deep neural network selection: In this experiment, two recently updated feature categories' effects on detection accuracy are evaluated individually in order to determine the relationship between chosen features and the efficacy of various deep neural networks. Second, look into how different deep neural networks' computational architecture affects the precision of detection.

The accuracy of the three CNN models is 79.89%, 83.17%, and 85.11%, as given in Table 4.2 Detection Results of Feature Categories and Networkstable. Here, as the input with two feature types has the best outcome, it is decided to employ manifest attributes and API calls combined as an input bundle in approach after comparing the accuracy of feature categories.
3) Network selection: RNN models typically perform more accurately than CNN models. The fact that RNNs have an internal state (memory) that can take into account the correlation between the various feature positions is one potential explanation. This internal state will enable RNN to retain the most corresponding feature positions by enabling long-term retention of highly prospective connected features during the training phase. CNN, however,

takes into account each distinct feature position separately throughout training. GRU and bidirectional GRU perform similarly to other RNN models in terms of accuracy (86.75 vs. 86.78%), which is higher. Additionally, they exhibit a higher recall than precision (87.00% vs. 86.57% for Bidirectional GRU and 86.78% vs. 86.72%). Compare the original pretrained model's size to that of the quantized and non-quantized models as well. By converting the original pre-trained model to the TensorFlow Lite model, its size is reduced by three times for RNNs and five times for CNN. The migrating model size for both the RNN and CNN models will remain the same throughout the experiment, regardless of whether the quantization setup is activated or not. On average, the RNN models have a better result than CNN. GRU models have a better accuracy than the LSTM models on our dataset.

4) Performance evaluation of RNN models on real device: In this experiment, in addition to examining the effectiveness of unzipping and feature extraction, we also assess the effectiveness of prediction using several RNN models on actual devices. It is clear that the pre-trained model using GRU performs best among all of them. Bi-GRU has the best accuracy (90.26%) among the others and hence it was selected as the network for training Mobitive. It was determined to be reasonable to choose the Bi-GRU model to assess the effectiveness and accuracy of the technique on actual mobile devices after taking into account all circumstances. The model has been trained and saved for prediction on the server side. This has be quantized and migrated to a mobile platform using tensorflow lite converter. The tensorflow model is implemented on the mobile side for further predictions when an unknown apk is introduced.

Chapter 5

Output Screenshots

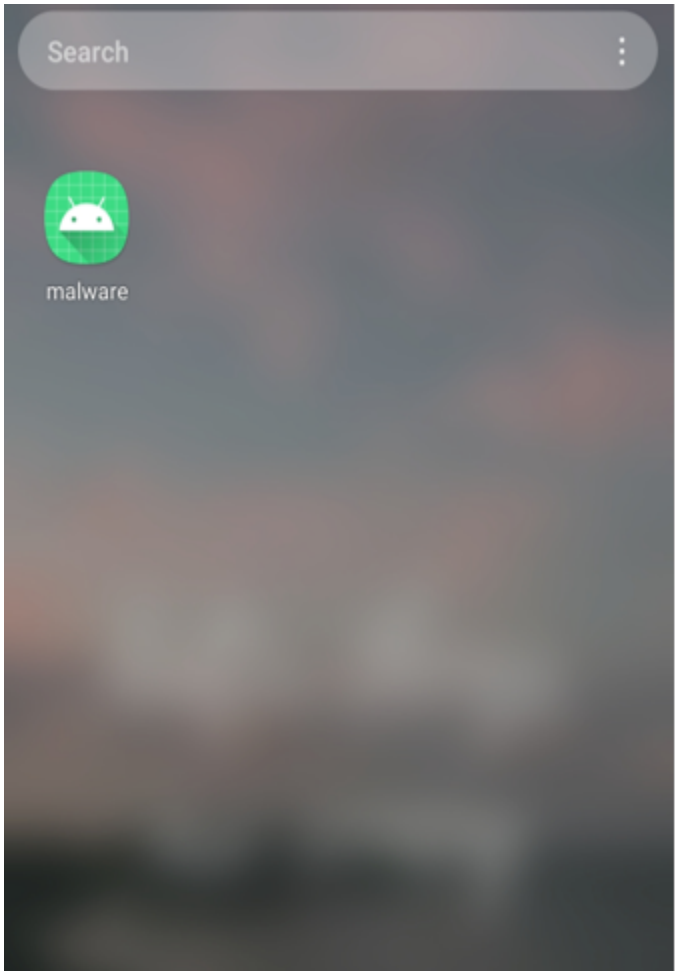

Figure 5.1: The app icon of Mobitive
The Figure 5.1 shows the app icon of the malware detection system in Android mobile phones.

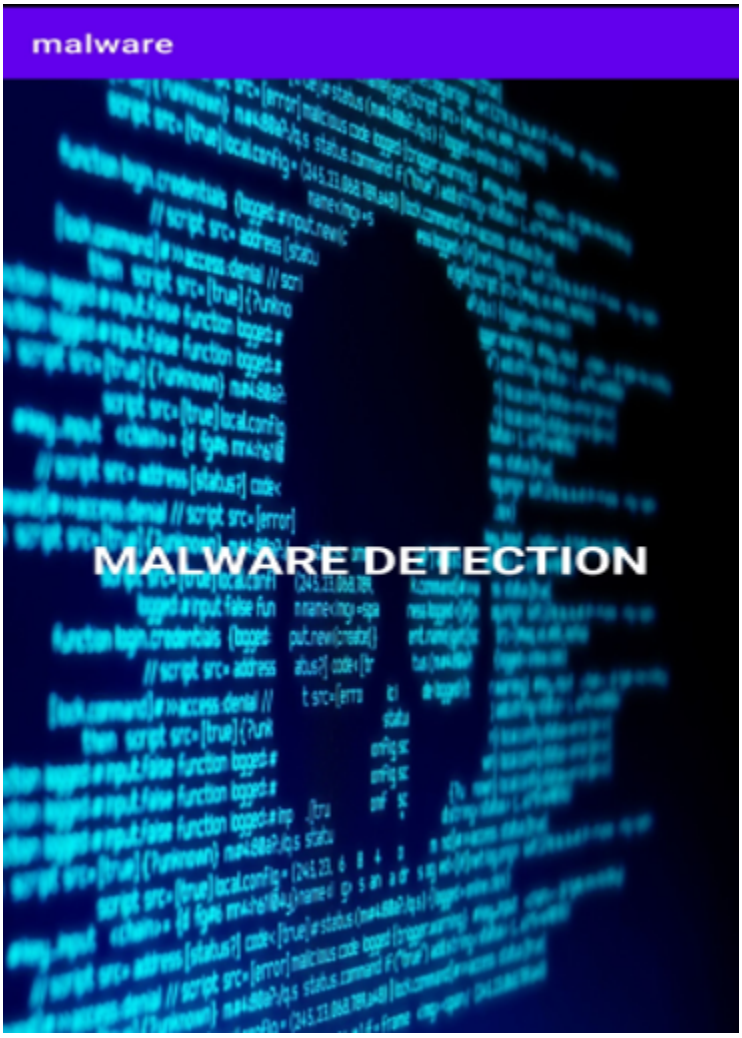

Figure 5.2: The Splashscreen of Mobitive

A 2 seconds splashscreen has been introduced to show the motive of the app as shown in figure 5.2.

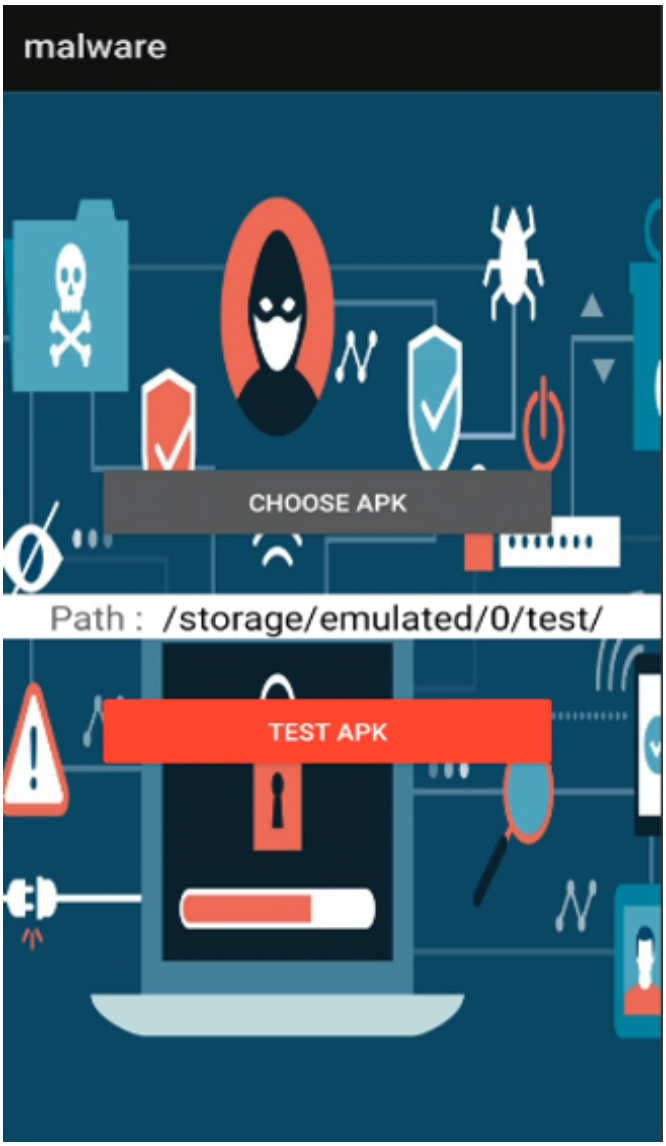

Figure 5.3: The Userhome of Mobitive

Figure 5.3 shows the Userhome where the apk files are tested to know whether its a malware or not.

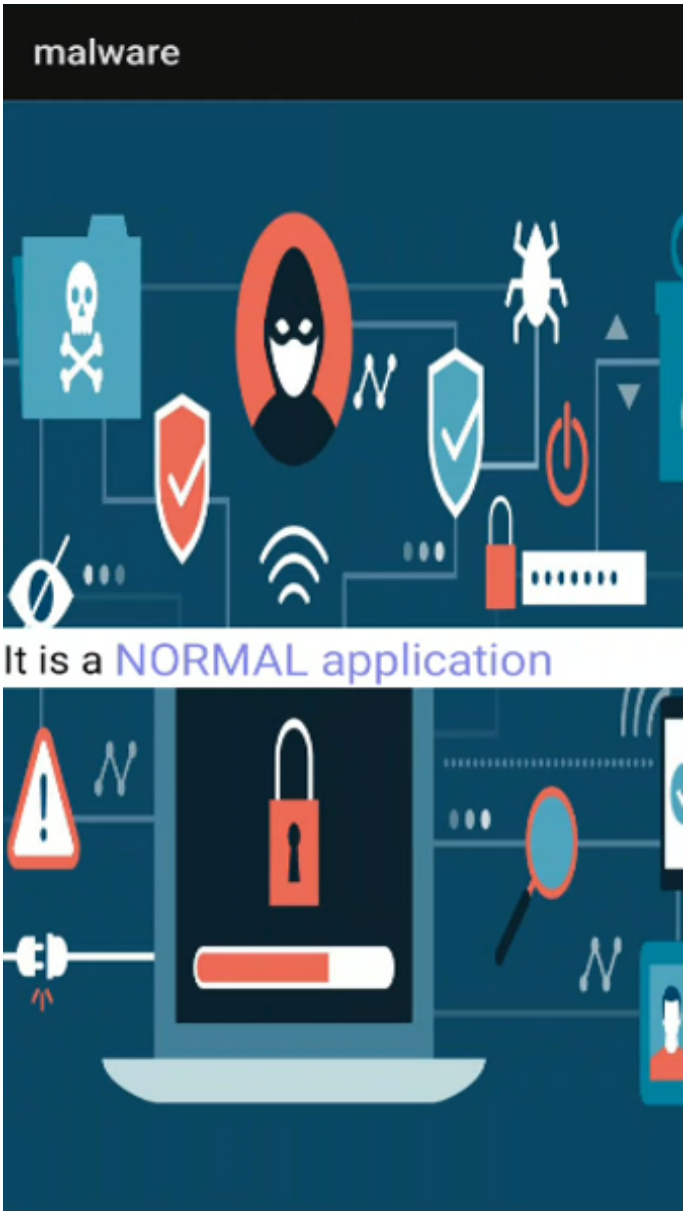

Figure 5.4: The Output of Mobitive
This shows the output of a Normal Apk when tested.

Figure 5.5: The Output of Mobitive
This shows the output of a Malware Apk when tested.

Chapter 6

Performance Evaluation

6.1 Settings

From the dataset, 80 % of the Apks are selected as the training set, and the remaining is the testing set. The evaluation metrics selected are Mean Absolute Error (MAE), Mean Square Error (MSE), Root Mean Square Error (RMSE), Precision, Accuracy and Recall as shown in Equations 6.1, 6.2, 6.3, 6.4 and 6.5.

$$MAE = \frac{\sum_{i=1}^{N} |pr_i - ar_i|}{N} \tag{6.1}$$

$$MSE = \frac{\sum_{i=1}^{N} (pr_i - ar_i)^2}{N} \tag{6.2}$$

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} (pr_i - ar_i)^2}{N}} \tag{6.3}$$

$$Precision = \frac{|True\ positives|}{|Actual\ results\|} \tag{6.4}$$

$$Recall = \frac{|True\ positives|}{|Predicted\ results\|} \tag{6.5}$$

The performance Evaluation results are shown in Table 6.1. This shows that using Bi-directional GRU it was able to obtain an accuracy of 90.26% which is comparatively higher than other predicted models like CNN, LSTM and GRU. When predicted in the mobile platform it was found that it predicts the output in realtively lesser time.

Metric	Proposed Model
MSE	0.1279
MAE	0.12
RMSE	0.3577
Precision	85.1%
Recall	99.1%
Accuracy	90.26%

Table 6.1: Performance evaluation

Chapter 7

Limitations and discussions

7.1 Feature selection

The few carefully chosen feature categories (i.e., API requests and manifest attributes) almost certainly won't result in huge overhead whenever it operates on an Android smartphone, there is overhead. However, t he two restricted feature kinds will only reveal a little amount of data about Android malware. In the future, we hope to include additional useful features types that also have modest performance costs. In the meanwhile, it is crucial to spot emerging virus families in real-world situations. Neither dynamic nor static techniques are true, in fact can fully ensure the effectiveness of protection against new pieces of malware. A potential approach to finding more fresh Malware is when two different techniques are combined. In the future, we'll work to make it easier to spot new malware by creating a fresh adaptive approach that is also an open a query for this neighbourhood.

7.2 New malware family detection

It goes without saying that finding new malware families in actual use is a crucial responsibility for any malware detection technology. However, neither dynamic nor static approaches are completely capable of ensuring the efficacy of defence against fresh malware samples. For instance, MobiTive would share the same drawback as other static analysis-based malware detection systems due to the small training dataset, which is distinct from the dynamic analysis approaches. In particular, while taking into account a new malware family, it is possible that the dangerous traits are completely different from the material already available.Although learning-based algorithms can occasionally be used to detect new malware types, the trained classifier may not be able to make the proper choice due to a lack of expertise. As a result, by integrating different methodologies, this work may attempt to increase the ability to recognise new samples in the future. Additionally, make an effort to increase your capacity to spot new infections by developing new adaptive strategies. An will help the community by revealing potential solutions to this open problem.

7.3 Against adversarial attack

Indeed, adversarial attacks will affect deep learning-based systems (such voice/image recognition), making it difficult to maintain the robustness of these systems. The deep learning-based systems for malware detection and voice/image recognition, however, differ in a number of ways.
(1) First and foremost, as opposed to voice/image Recognizing this, adversarial assaults in malware detection cannot quickly disable all of the applications' functionality. Specifically, in order to counteract the current hostile attacks against Malware detection is always produced by tampering with the Instead of altering actual capabilities, target malware applications with non-triggered code snippets (such as dead code). Although it can provide adversarial samples to avoid detection, a high rate of misclassification by the classifier, it is presently

impractical because such an assault can be quickly identified by utilising other methods like static data flow analysis to remove these features that add dead code introduces from a threat. Additionally, it is demonstrated by a lack of In the ongoing studies, there are actual samples of adversarial malware.
(2) In contrast to malware detection systems on other platforms (such as Windows/Linux), our solution abstracts the full Android app with a condensed feature list rather than embedding the entire package programme, forcing attackers to modify their malware programmes to work around MobiTive. In actuality, attackers find it difficult to gather the precise feature list. As a result, it is nearly impossible to get around MobiTive as easily as deep learning-based voice/image recognition systems under the restriction of maintaining the functionalities in the malware applications, given that the majority of our chosen features (i.e., manifest properties and API calls) are defined by the official/reliable third-party developers. In comparison to image/voice classification, adversarial assaults on deep learning-based malware face domain-specific difficulties.

7.4 Dynamic behavior analysis

Static analysis plays a significant role in previous and present cyber security research, and there are more research articles on Android malware detection than there are on dynamic analysis, to the best of our knowledge and after a thorough examination of the literature (static analysis vs. dynamic analysis). Dynamic behaviour analysis may, in fact, produce a more accurate result than static analysis for a given detection task (e.g., fewer false positives); however, there are a number of drawbacks that must be explored in order to determine whether it may be used to particular scenarios.(1) First and foremost, there are fewer situations for dynamic behaviour analysis-based malware detection systems because of the high cost of computational resources, which prevents these systems from meeting customers' demands for performance and efficiency. For instance, it's common practise to employ a performance counter when performing programme analysis for malware/bug detection tasks. Android applications feature a more complex HCI mechanism than conventional Windows/Linux programmes do. In other words, developing programmes on Windows/Linux is significantly easier than producing high-quality test benchmarks with good coverage to the corner cases. Even if we are able to collect the standards, the time spent developing and implementing them will conflict with the goal of meeting user demand for efficiency. (2) Next, how well the predefined behaviours are covered has a significant impact on the detection efficiency. Of other words, the security of the system will no longer be guaranteed once the dangerous behaviour in the target virus is not clearly identified by the detection system. (3) Third, unlike MobiTive, a system based on dynamic behaviour analysis may experience problems due to the way it operates (i.e., before installation as opposed to during runtime). Since most users lack the expertise of security researchers, social engineering-based malware, for instance, can easily keep the user's privacy information on the device and deceive them into uploading it. In the end, we believe Android malware detection methods based on dynamic behaviour and static analysis have their place given the variety of usage scenarios and targets.

Chapter 8

Conclusion

Due to the huge amount of data that is stored on smartphones in current society—both personal and corporate—malware developers are drawn to create programmes that can sneak onto users' devices and steal data or carry out dangerous actions. This was coupled by the limitations of current protection strategies, such as the weak or nonexistent screening in third-party markets and the ineffective screening in the Google Play store. antivirus programmes that still use signature-based databases are only capable of identifying known threats. Here suggests an Android malware detection system that uses deep learning technology to counter the threats of Android malware in order to thwart malicious applications that are becoming more numerous and sophisticated. In addition to the shortcomings in the current protection method, the extraordinary rise in Android malware over the past few of years necessitates an effective solution to contain Android malware. This work offers MobiTive, a preinstalled malware detection system for Android mobile devices that is performance-sensitive. MobiTive can directly offer a solid detection accuracy and quick responsive (i.e., less than 3 seconds on average) detection service on mobile devices based on the effectiveness of chosen features and the efficiency of feature extraction. MobiTive has been evaluated for effectiveness on a variety of mobile devices, and it achieves detection accuracy of 90.26 percent. Extensive experiments were conducted on a real world dataset and the performance of the proposed system is compared with that of a model trained with other networks. The result obtained on the selected metrics: Precision, Recall, Mean Absolute Error, Mean Squared Error and Root Mean Square Error, shows that the proposed system has better performance. Future works can be done by introducing new feature selections and better model.

References

[1] S. Chen, M. Xue, L. Fan, S. Hao, L. Xu, H. Zhu, and B. Li, "Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach," Computers Security, 2018.

[2] Yuan, Y. Lu, and Y. Xue, "Droiddetector: Android malware characterization and detection using deep learning," Tsinghua Science and Technology, 2016.

[3] A. Narayanan, L. Yang, L. Chen, and L. Jinliang, "Adaptive and scalable Android malware detection through online learning," in IJCNN, 2016.

[4] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "MalDozer: Automatic framework for Android malware detection using deep learning," Digital Investigation, 2018.

[5] Narayanan, C. Soh, L. Chen, Y. Liu, and L. Wang, "Apk2vec: Semisupervised multi-view representation learning for profiling Android applications," in ICDM, 2018.

[6] L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic Android malware analysis," in USENIX Security, 2012.

[7] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in PLDI, 2014.

[8] L. Li, A. Bartel, T. F. Bissyande, J. Klein, Y. Le Traon, S. Arzt, ´ S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in Android apps," in ICSE, 2015.

[9] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of Android malware in your pocket." in NDSS, 2014.

[10] B. Wu, S. Chen, C. Gao, L. Fan, Y. Liu, W. Wen, and L. Michael, "Why an Android app is classified as malware? Towards malware classification interpretation," TOSEM, 2020.

[11] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, and X. Li, "An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms," in ASE, 2020.

[12] R. Feng, S. Chen, X. Xie, L. Ma, G. Meng, Y. Liu, and S.-W. Lin, "Mobidroid: A performance-sensitive malware detection system on mobile platform," in ICECCS, 2019

[13] Chen, M. Xue, L. Fan, S. Hao, L. Xu, H. Zhu, and B. Li, "Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach," Computers Security, 2018

[14] K. Basu, P. Krishnamurthy, F. Khorrami, and R. Karri, "A theoretical study of hardware performance counters-based malware detection," TIFS, 2020.

[15] Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu et al., "Deepgauge: Multi-granularity testing criteria for deep learning systems," in ASE, 2018.

[16] X. Zhang, X. Xie, L. Ma, X. Du, Q. Hu, Y. Liu, J. Zhao, and M. Sun, "Towards characterizing adversarial defects of deep learning software from the lens of uncertainty," in ICSE, 2020.

[17] B. David, X. Xie, L. Ma, L. Zhou, Y. Liu, C. Xu, and J. Zhao, "Cats are not fish: Deep learning testing calls for out-of-distribution awareness," in ASE, 2020.

[18] K. Xu, Y. Li, R. H. Deng, and K. Chen, "Deeprefiner: Multi-layer Android malware detection system applying deep neural networks," in EuroSP, 2018.

[19] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for Android malware detection using various features," TIFS, 2018.

[20] S. Chen, M. Xue, L. Fan, L. Ma, Y. Liu, and L. Xu, "How can we craft large-scale Android malware? an automated poisoning attack," in AI4Mobile, 2019