ROBUST MOBILE MALWARE DETECTION

MAHBUB E KHODA

A dissertation submitted in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

FEDERATION UNIVERSITY AUSTRALIA
School of Engineering, IT and Physical Sciences

SEPTEMBER 2020

Declaration

This thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Mahbub E Khoda

4 September 2020

Acknowledgment

All praise be to Allah, the almighty, the most merciful. I praise and thank Him abundantly for giving me the capability, intellect, and opportunity to undertake this research.

I express my profound indebtedness to my supervisors Joarder Kamruzzaman, Iqbal Gondal, and Ashfaqur Rahman for their constant support, effective guidance, insightful advice, helpful criticisms, valuable suggestions, and endless patience towards the completion process of this thesis. I was given enough freedom to explore the research areas of my choice and guided when I felt lost. Without their insights and encouragements, this research would not have been completed.

My sincere gratitude goes to my mother Zahanara Khatun for her endless love and inspiration throughout my life. I am especially thankful to my brother Neyamul Kabir who has been a guardian and a father figure for me since my childhood. He took care of me in the absence of my father, provided financial support, and was always there for me whenever I needed him. I also thank my beloved wife Sadia Islam for her love, care, patience, and understanding.

An important acknowledgment goes to Capstone Editing who provided copyediting and proofreading services, according to the guidelines laid out in the university-endorsed national 'Guidelines for Editing Research Theses'. I thank Federation University Australia for granting me the Australian Government Research Training Program (RTP) Stipend and RTP Fee-Offset Scholarship, and CSIRO for Data61 top-up scholarship. Finally, I thank all the staffs, graduate students, and friends at the School of Engineering, Information Technology and Physical Sciences, Gippsland, for their support and encouragement during the last few years.

ABSTRACT

The increasing popularity and use of smartphones and hand-held devices have made them the most popular target for malware attackers. Researchers have proposed machine learning-based models to automatically detect malware attacks on these devices. Since these models learn application behaviors solely from the extracted features, choosing an appropriate and meaningful feature set is one of the most crucial steps for designing an effective mobile malware detection system. There are four categories of features for mobile applications. Previous works have taken arbitrary combinations of these categories to design models, resulting in sub-optimal performance. This thesis systematically investigates the individual impact of these feature categories on mobile malware detection systems. Feature categories that complement each other are investigated and categories that add redundancy to the feature space (thereby degrading the performance) are analyzed. In the process, the combination of feature categories that provides the best detection results is identified.

Ensuring reliability and robustness of the above-mentioned malware detection systems is of utmost importance as newer techniques to break down such systems continue to surface. Adversarial attack is one such evasive attack that can bypass a detection system by carefully morphing a malicious sample even though the sample was originally correctly identified by the same system. Self-crafted adversarial samples can be used to retrain a model to defend against such attacks. However, randomly using too many such samples, as is currently done in the literature, can further degrade detection performance. This work proposed two intelligent approaches to retrain a classifier through the intelligent selection of adversarial samples. The first approach adopts a distance-based scheme where the samples are chosen based on their distance from malware and benign cluster centers while the second selects the samples based on a probability measure derived from a kernel-based learning method. The second method achieved a 6% improvement in terms of accuracy.

To ensure practical deployment of malware detection systems, it is necessary to keep the real-world data characteristics in mind. For example, the benign applications deployed in the market greatly outnumber malware applications. However, most studies have assumed a balanced data distribution. Also, techniques to handle imbalanced data in other domains cannot be applied directly to mobile malware detection since

they generate synthetic samples with broken functionality, making them invalid. In this regard, this thesis introduces a novel synthetic over-sampling technique that ensures valid sample generation. This technique is subsequently combined with a dynamic cost function in the learning scheme that automatically adjusts minority class weight during model training which counters the bias towards the majority class and stabilizes the model. This hybrid method provided a 9% improvement in terms of F1-score.

Aiming to design a robust malware detection system, this thesis extensively studies machine learning-based mobile malware detection in terms of best feature category combination, resilience against evasive attacks, and practical deployment of detection models. Given the increasing technological advancements in mobile and hand-held devices, this study will be very useful for designing robust cybersecurity systems to ensure safe usage of these devices.

Acronyms

ADB	Android Debug Bridge
API	Application Programming Interface
APK	Android PacKage
AUC	Area Under the Curve
BERT	Bidirectional Encoder Representation from Transformers
CC	Category Center
CD	Constructive Divergence
CE	Cross Entropy
CERT	Cyber Emergency Response Team
CNN	Convolutional Neural Network
DAE	Deep AutoEncoder
DBN	Deep Belief Network
DNN	Deep Neural Networks
DVM	Dalvik Virtual Machine
FL	Focal Loss
FN	False Negative
FP	False Positive
ICC	Inter Component Communication
KBL	Kernel Based Learning
KNN	K-nearest neighbors
MFE	Mean False Error
MFNE	Mean False Negative Error
MFPE	Mean False Positive Error

MRV	Malware Recomposition Variation
MSE	Mean Squared Error
OS	Operating System
RBF	Radial Basis Function
RBM	Restricted Boltzman Machine
ReLU	Rectified Linear Unit
RF	Random forest
ROC	Receiver Operating Characteristic
SMOTE	Synthetic Minority Over-sampling Technique
SMS	Short Message Service
SSL	Secure Sockets Layer
SVM	Support Vector Machine
TN	True Negative
TP	True Positive
UI	User Interface
XML	Extensible Markup Language

Nomenclature

Ω	Global feature list
E	Energy function of a neural network
v	Visible unit data
h	Hidden unit data
c	Visible unit bias
b	Hidden unit bias
p	Activation probability of a neuron
CD_k	K-step constructive divergence
α	Learning rate
$\|\|w\|\|$	Length of vector w
$sign(.)$	Sign (positive or negative) of a value
γ	Free parameter for RBF function
$loss()$	Loss function of a machine learning algorithm
$\hat{x}$	Adversarial sample crafted from original sample x
$\nabla_x(y)$	Derivative of y with respect to x
Υ	Maliciousness indicator score
W_{ui}	Weight value for UI callback event
W_{be}	Weight value for background event
Γ	Contextual feature value of an API function
$\mathbb{K}$	Maximum separation value between user-aware and unaware API call
k	Steepness parameter for a sigmoid function
$\mathbb{U}$	Set of all contextual values of user aware and unaware calls of an API function

$E_n(X)$	Entropy of a random variable X
$I_g(X\|Y)$	Information gain of a random variable X given Y
ϑ	Relevancy threshold
δ_x	Perturbation added to sample x
y'	Predicted class label by a classifier
y_t	Target class label of an adversarial attack
J_M	Jacobian matrix
S_d	Distance score of an adversarial sample
$Dist_k$	Distance of a sample from cluster center k
$P_{A,B}$	Platt's probability
N_+	Number of positive samples
N_-	Number of negative samples
X_{mal}	Malware feature matrix
X_{ben}	Benign feature matrix
N_{mal}	Number of malware samples
N_{ben}	Number of benign samples
ρ	Imbalance ratio
Idx_m	List of indices of mutable features
v_b	Centroid of class b
$M_{C_i}(\mathbf{x})$	Membership degree of a sample $\mathbf{x}$ to a class C_i
P_w	Dynamic minority class weight

Table of Contents

List of Tables

List of Figures

Publications from this Research

Published works

1. Mahbub E Khoda, Tasadduq Imam, Joarder Kamruzzaman, Iqbal Gondal, Ashfaqur Rahman "Robust Malware Defense in Industrial IoT Applications using Machine Learning with Selective Adversarial Samples", *IEEE Transactions on Industry Applications* (Rank: JCR-Q1, impact factor=3.488)

2. Mahbub E Khoda, Joarder Kamruzzaman, Iqbal Gondal, Tasadduq Imam "Mobile Malware Detection-An Analysis of the Impact of Feature Categories", *International Conference on Neural Information Processing (ICONIP), 2018* (CORE/ former ERA rank: A)

3. Mahbub E Khoda, Joarder Kamruzzaman, Iqbal Gondal, Tasadduq Imam, Ashfaqur Rahman "Mobile Malware Detection: An Analysis of Deep Learning Model", *2019 IEEE International Conference on Industrial Technology (ICIT)* (CORE/former ERA rank: B)

4. Mahbub E Khoda, Joarder Kamruzzaman, Iqbal Gondal, Tasadduq Imam, Ashfaqur Rahman "Selective Adversarial Learning for Mobile Malware", *18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications (TrustCom) 2019* (CORE/ former ERA rank: A)

5. Mahbub E Khoda, Joarder Kamruzzaman, Iqbal Gondal, Tasadduq Imam, Ashfaqur Rahman "Mobile Malware Detection with Imbalanced Data using a Novel Synthetic Oversampling Strategy and Deep Learning", *16th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob) 2020* (CORE/former ERA rank: B)

Awaiting decision

1. Mahbub E Khoda, Joarder Kamruzzaman, Iqbal Gondal, Tasadduq Imam, Ashfaqur Rahman "Malware Detection in Edge Devices with Fuzzy Oversampling and Dynamic Class Weighting", *Journal of Applied Soft Computing* (Rank: JCR-Q1, impact factor=5.472)

Chapter One

Introduction

Smartphones, tablets, and other hand-held mobile devices are being increasingly used in mobile commerce, online transactions, and sharing of personal data due to their ease of use, continuous connectivity, and convenience [1–4]. Statista [5] reports that the number of smartphone users worldwide surpassed 3.5 billion in 2020. This widespread use has made mobile devices an inevitable target of cyber-crime through dissemination of malware [6]. In 2019, Cyber Emergency Response Team (CERT) Australia reported responding to more than 13,672 cyber-attacks that amounted to an estimated annual loss of US$328 million [7]. Such attacks are shifting more towards mobile and handheld devices [8, 9]. Malware developers target these devices to steal information and gain privileged access, which is then used for criminal activities. Therefore, detecting malware in these platforms is becoming increasingly important.

Malware detection in such platforms offers some unique challenges. A significant amount of work has been conducted in the field of malware detection for desktop and wired environments, but, some key differences between the desktop and mobile environments necessitate addressing this issue separately for mobile platforms. For example, the application structure, techniques, and execution methods are very different in mobile devices, and the operating systems for mobile devices are optimized for particular execution environments. As a result, the applications (apps) developed

for these systems are fundamentally different. Thus, specific techniques for malware detection tailored to suit these environments is essential. The structure and organization of such a system (Android) and its applications are described in detail in Sec. 2.1 and 2.2. Device location is another point of difference between wired and mobile platforms. By their nature mobile devices connect to various networks including public, private, personal, and work networks. Naturally, security settings vary in such networks, making mobile devices vulnerable to various novel exploits and attacks that differ from the wired devices where change of location is usually not an issue.

Researchers have proposed several mobile malware detection systems based on machine learning models. Considering the requirements of an effective and robust mobile malware detection system and our study on the current literature, we have identified a number of important research challenges. These are discussed below.

1.1 Research Challenges and Motivation

For designing a machine learning-based malware detection system, analyzing the applications and extracting meaningful and effectual features from them is a crucial step. Features are properties of applications used to train a machine learning model. From a security point of view, an application is analyzed in two different ways namely, static analysis and dynamic analysis. Static analysis decodes the application file and parses them to extract features while dynamic analysis executes the application in a suitable environment and records its run-time behavior. Static features of an application have three sub-categories: permissions, Inter Component Communication (ICC), and Application Programming Interface (API) calls. Additional information can also be associated with some of the feature categories, for example, an API call can be initiated by user interaction or by a system-generated event. These are known as contextual information about features and can provide additional assistance for analyzing application behavior. A detailed description of feature categories and, static

and dynamic analysis is presented in Sec. 2.9.

A number of works in the literature have proposed malware detection systems using these feature categories. For example, some works considered the permission features [10–12], some considered API features [13, 14], and a few considered ICC features [15] that exposed application collusion attacks (detailed in Sec.3.2). However, these works chose the feature categories in a randomized fashion without paying attention to the individual effect of these categories. Since these feature categories represent different characteristics of mobile applications, it is necessary to undertake a systematic investigation of the behaviors of these feature categories regarding malware detection. Previous works have not considered this. Also, an attempt to find the best combination is necessary because a blind combination of all the feature categories may not lead to good performance. For example, dynamic features complement the permission features very well, whereas, our experiments showed that some feature categories introduce redundancy when combined with others.

Despite the promising performance of machine learning-based malware detection systems, ensuring the reliability and robustness of these systems remains an important issue. Specifically, the recent discovery of sophisticated evasive techniques to break detection systems makes this an attractive research topic. For example, a machine learning-based system can be evaded through adversarial samples crafted by carefully modifying a previously developed malware. A successful adversarial sample is able to go undetected by a detection system even though the original version was correctly identified by the same system. Though adversarial retraining is proposed as a defense mechanism against these attacks, using an excessive number of arbitrary samples for retraining can further lead to performance degradation since the model starts to over-fit. Existing works choose the samples in a random manner, which often leads to a sub-optimal choice of the samples for retraining. Selecting an appropriate set of samples that will make a system resilient and significantly enhance the detection performance is a major research challenge here.

Moreover, class-balanced datasets used in most prior studies reflect an unrealistic assumption since in the real world benign applications outnumber malware samples by a significant margin making the actual data distribution highly imbalanced. As a result, imbalanced data handling ability is one of the primary requirements for a robust detection system. Existing techniques in the literature cannot be applied here since they modify features randomly and generate invalid samples with broken functionality [16, 17]. Therefore, a customized data balancing technique for generating synthetic malware samples preserving valid functionality is required. Additionally, combining algorithm-level techniques with the data balancing for improved performance needs to be investigated.

1.2 Research Objectives

The principal aim of this research is to design a robust mobile malware detection system. Considering the various research opportunities outlined in previous sections, the following major research objectives were identified:

1. Systematic investigation of the characteristics of individual feature categories and their impact on mobile malware detection. Additionally, extract the contextual information, embed it in the feature space, and examine its effect on classification performance. This objective leads to the following research questions:

   RQ-1: How does combining different categories of features affect classifier performance, and which combination provides the best performance?

   RQ-2: Does embedding contextual information into the feature space improve classifier performance?

2. Addressing the vulnerability of malware detection systems to adversarial attacks and formulating strategies to make a malware detection system robust against

such attacks. The corresponding research questions addressed this objective:

RQ-3: What are the adverse effects of adversarial samples on a malware detection system?

RQ-4: How to selectively choose samples for adversarial retraining to make classifier robust against adversarial attacks?

3. Handling imbalanced data problem while avoiding generation of invalid synthetic samples. The research questions pertaining to this objective are:

   RQ-5: How to balance the data with valid synthetic malware samples that preserve code functionality?

   RQ-6: Can the performance be improved by a hybrid method combining the novel synthetic sampling in training set preparation and a dynamic cost schema in model training?

1.3 Research Methodology

This section describes the overall research methodology that will be undertaken in this work to address the above-mentioned objectives.

1.3.1 Dataset Collection

To validate our proposed approaches through experiments, publicly available datasets will be used and for completeness, more recent and newer samples will be collected. For this purpose, the most prominent datasets used in mobile malware detection research (i.e., Malgenome and Drebin [14, 18]) will be used and processed to make them usable for machine learning models. However, these datasets contain only a small portion of recent malware. Hence, the Androzoo [19] tool will be used to obtain a more recent set of samples and they will be processed accordingly. Section 2.6 describes the characteristics of the datasets in detail.

1.3.2 Feature Extraction

Since our first research objective addresses the impact of different feature categories on mobile malware detection, an exhaustive list of static and dynamic features from different categories will be extracted in our work. Section 3.1 discusses the feature extraction process and the characteristics of different feature categories in detail.

1.3.3 Innovative Model Design and Evaluation

Our research will adopt innovative approaches to achieve the research objectives. For example, to tackle adversarial attacks, we will explore the possibility of using a selective sampling mechanism that finds out the best possible set of adversarial samples which will be used to retrain a machine learning model to make it robust. Additionally, we will explore designing an innovative over-sampling method and/or modification of machine learning algorithms to better handle imbalanced data problem. The well-known stratified 10-fold cross-validation [20] approach will be used for performance evaluation. A range of widely used performance measures including accuracy, recall, F1-score, and G-mean will be used for quantitative analysis. Section 2.8 defines the performance measures used in this thesis. Additionally, the Wilcoxon Signed-Rank Test [21] will be used to verify the statistical significance of the improvements achieved by our proposed approaches compared to recent competing methods.

1.4 Overview of Contributions

To accomplish the major research objectives mentioned in Sec. 1.2, this work presents a number of original contributions as illustrated in Fig. 1.1. The individual contributions are divided into blocks that work towards a robust mobile malware detection system. Block 1 in Fig. 1.1 represents the works for Objective 1 which investigates the individual feature categories and identifies the combination that provides the best detection performance. Knowledge from this contribution provides the baseline

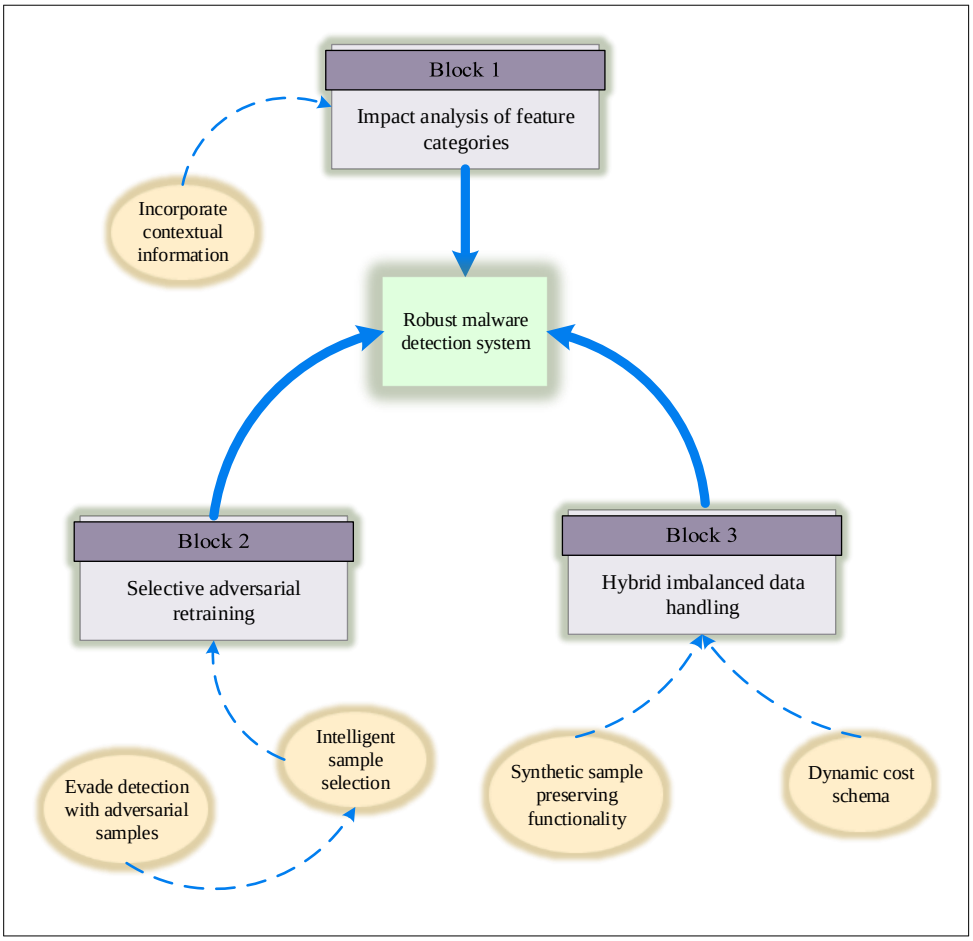


Figure 1.1 An illustration of the overall contribution of this thesis.

for later works. Block 2 represents our contribution to making a mobile malware detection system robust against adversarial attacks. Block 3 outlines the works for Objective 3 which addresses the imbalanced data problem. The overall contribution of this thesis is elaborated on below:

1. A comprehensive investigation of the impact of various feature categories in mobile malware detection is presented. The best combination of feature categories the is, the combination that best enhances the performance of a machine learning-based malware detection system is identified in the process. Additionally, we incorporated contextual information of API call feature into this work. Contextual information can provide additional knowledge regarding the usage of certain features and application behavior. Existing works use contextual infor-

mation in an isolated manner which is not suitable for use in combination with other features. We formulated a mechanism that embeds this contextual information into the feature space. This facilitates the usage of this information in combination with other feature categories, further improving the performance of a malware detection system. We found ICC feature category adds redundancy to the feature space and degrades classification performance when compared with other feature categories. Findings from this work have been published in [22] and [23].

2. An improved selective adversarial retraining strategy is proposed to achieve resilience against adversarial attacks. It is shown that if too many adversarial samples are used for retraining, as has been done in the prior works, the model starts to over-fit and performance degrades. Hence, careful selection of the samples used for retraining is required. To address this issue, two different approaches are proposed. First, a sample selection strategy is proposed based on their distance from malware and benign cluster center. Second, a kernel-based selective sampling technique is proposed to identify the adversarial samples that have a higher probability of being incorrectly classified. Experiments show that using the kernel-based samples for retraining improve detection accuracy by 6% compared to the existing random selection-based techniques and makes the classification system robust against such attacks. This work has been published in [24] and [25].

3. With a view to handling imbalanced data for mobile malware detection, a hybrid of data-level and algorithm-level method is proposed. For data balancing, a novel malware over-sampling technique is formulated that generates synthetic malware samples with valid functionality, thereby mitigating the issues of prior works. This technique improves the fuzzy set theory-based over-sampling technique as it does not generate invalid samples as in [16]. Further, we devised a cost schema that dynamically assigns weight to the minority class samples (i.e.,

malware) which simultaneously counters the bias of the classifier towards the majority class and reduces model instability. This technique provided a 9% improvement in terms of F1-score compared to existing techniques. The findings from this work have been published in [26].

1.5 Organization of the Thesis

The overall organization of this thesis is outlined below.

Chapter 2 presents an overview of the relevant research works in mobile malware detection. The operating system and application structure of the most popular mobile platform (Android) is presented to demonstrate the particular characteristics of this domain. Different machine learning tools used for mobile malware detection models are discussed. A detailed study of the feature categories is presented. Related research works associated with these feature categories are also presented and their limitations identified. Advanced evasive techniques that break down malware detection systems are discussed and the scarcity of relevant works in this domain is highlighted. The inapplicability of existing techniques is evidenced and the necessity of domain-specific research is established.

Chapter 3 shows the characteristics of individual feature categories for mobile malware detection. It investigates classification performance using all possible combinations of the feature categories and identifies the combination that provides the best results. It also details our proposed technique for embedding the contextual information into the feature space. Performance improvement is verified through extensive experimentation. This achieves our first objective and provides the baseline for subsequent works.

Chapter 4 presents the works related to our second research objective. It illustrates a robust defense technique against adversarial attacks through intelligent adversarial retraining. We proposed two novel sample selection strategies in this

chapter to find the best possible subset of adversarial samples for retraining. Details of these strategies are described in this chapter. The increased robustness of the resulting detection model compared to the random selection used in previous work is demonstrated through experimental results.

Chapter 5 presents the works for our third research objective. It investigates the imbalanced data problem for mobile malware detection systems and proposes a hybrid of data-level and algorithm-level technique. The detail of our novel synthetic malware over-sampling strategy is presented here. At the algorithm level, a dynamic cost schema is proposed and how it automatically adjusts the minority class weight based on validation class performance is presented. The results of extensive experiments with different levels of data imbalance are shown to validate the performance improvements.

Chapter 6 concludes the thesis by summarizing the overall contributions and discusses new research directions based on the findings from this work.

Chapter Two

Literature Review

As highlighted in Chapter 1, designing a robust malware detection system for mobile platforms is the primary research goal of this thesis. The mobile platforms include smartphones, tablets, and other hand-held devices which are increasingly being used for work purpose as well as personal use. [27–32]. Among different mobile operating systems including Windows and iOS, Android is currently the most popular operating system for smartphones and hand-held devices [33–37]. Research in [38] and [39] showed that there are two billion Android devices active monthly and this has allowed the Android system to capture more than 80% of the mobile operating system market. Hence, this work focuses on the Android platform. However, the techniques proposed in this work are equally applicable to other platforms. The following section describes Android operating system and its application structure. This is followed by a discussion on mobile malware, their types, and their propagation techniques. Thereafter, a detailed review of the existing works on mobile malware detection is presented and their limitations discussed.

2.1 Android Operating System

Figure 2.1 shows the architecture of the Android operating system. Android OS is written in Java and C/C++. It has five major sections:

Linux Kernel: The bottom-most layer of the Android OS is a modified Linux kernel. It provides a level of abstraction between the device hardware and contains all the necessary device drivers.

Libraries: This section sits on top of the Linux kernel and consists of a set of libraries including WebKit for web browsing, SSL for internet security, and SQLite for database management.

Android Runtime: This section provides a key component called *Dalvik Virtual Machine*, which is a modified java virtual machine specifically designed and optimized for Android applications. This enables every Android application to run in its own process sandbox. This section also provides some core Android libraries that are used by developers to write Android applications.

Application Framework: The application framework section provides many high-level services to Android applications in the form of java classes. Application developers can use these services in their applications.

Applications: All the applications that are used by the user sit in this layer. When a new application is installed it gets added to this layer.

An Android application is generally written in Java code which is then compiled to Java bytecode. This bytecode is further converted to Dalvik executable that is optimized for Android platforms and can be executed in the Dalvik Virtual Machine.

In the following section, we present the structure of an Android application (app) which will be useful to understand the feature extraction process for malware detection.

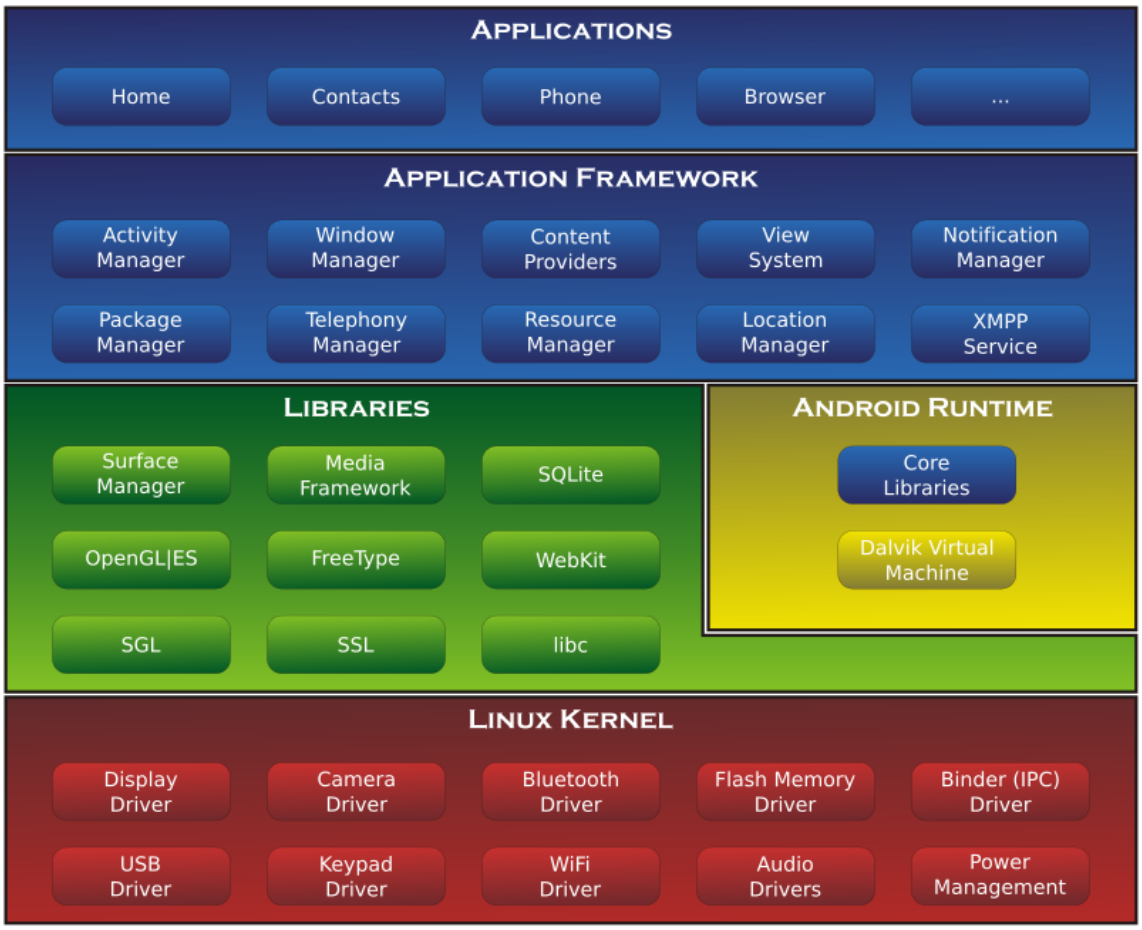


Figure 2.1 Android OS architecture [40]

2.2 Android Application Structure

Figure 2.2 shows the structure of an Android application. These applications are stored in a compressed format known as Android PacKage (APK) file. This is a zip file consisting of Dalvik bytecodes (classes.dex files), compiled and plain resources, assets (e.g., music, texts), and XML manifest file. Dalvik bytecodes is the source code of the app compiled and optimized for Android systems. The resources folder contains graphics, strings, and screen layouts. The complied resources folder contains the compiled binary version of the files in the resources folder. The native libraries folder contains the outer libraries required for the app and the assets folder contains other files required in their raw format. Signatures and certificates are also included

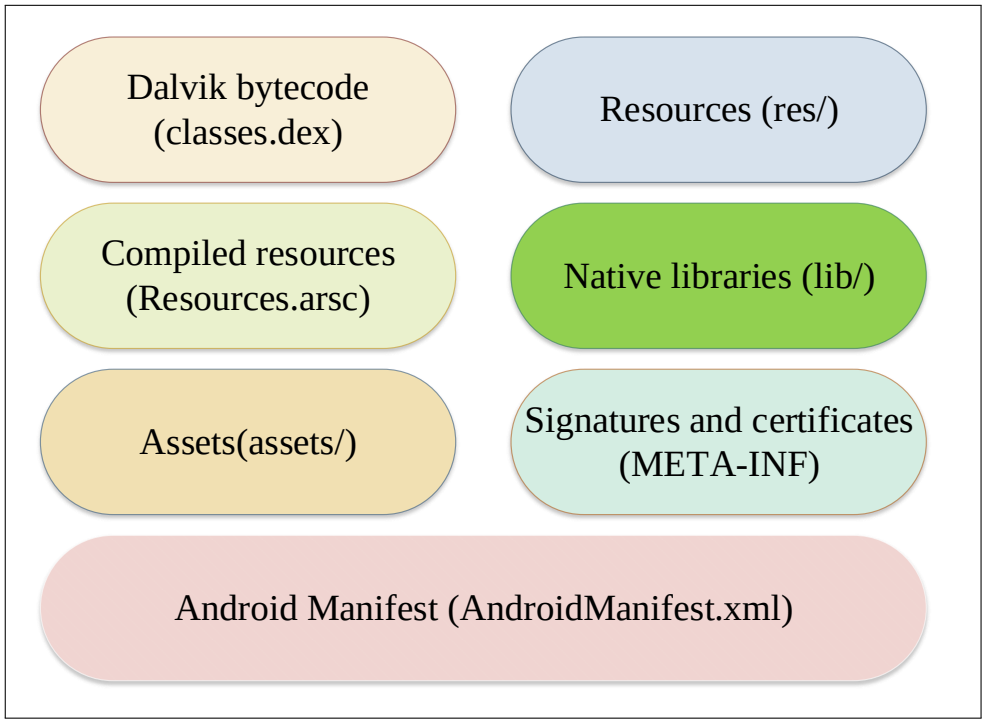


Figure 2.2 Android Application Structure

in the application package to verify the application and the developer. Finally, the androidmanifest.xml, also known as the manifest file, specifies the meta information such as, requested permission and application components.

Android applications are made of four kinds of components that serve different purposes. These components must be declared in the manifest file before they can be used in the code. Following is a brief description of these components.

Activity: Activity is the component that interacts with the application users. It presents the user interface and takes different commands through the interface.

Service: Services are used to perform tasks in the background. Generally, computation-intensive tasks are performed through the services.

Content Providers: Content providers are used for sharing data among the applications. This data can be application-specific or system generated.

Broadcast Receivers: Broadcast receiver of an application is the component used to

receive broadcast message objects from other components. An application can register broadcast receivers to listen to various events including the battery level change, the screen shut off, and change in the network level. These events are broadcast through objects known as intents.

2.3 Mobile Malware Types

The first malware attack for mobile systems was reported in 2004 for Symbian OS [41]. However, due to the ever-increasing popularity of Android system since its launch in 2007, it has become the most popular target operating system among malware attackers, with new attacks being launched every few weeks [41–47]. Below we briefly describe the types of malware targeting Android systems.

Trojans and viruses. Trojans are malicious applications that appear like legitimate apps. When executed, they can cause severe damage to the system [48]. Trojans are one of the major threats to the Android system since they take control of the system and steal critical information such as banking information, credit card numbers, and contacts. Some versions of Trojans can take root control (elevated access) of the system and gain unauthorized access to sensitive files and memory.

Spyware and adware. Spyware applications secretly steal user information and upload them to the attacker that may later be exploited for financial gain or future attacks [49]. This user information is also used by adware: a form of malware that shows intrusive advertisements and notifications on the screen [50]. These malware applications cause privacy leaks and often get installed by disguising themselves as legitimate apps.

Phishing apps. Phishing apps use the same principles as web phishing and uses them on mobile websites and apps [51]. These apps covertly steal user information such as account passwords and credit card details. Some Trojanized apps also use phishing schemes disguising as system updates or gaming applications.

Botnets. Botnets are networks of devices infected with malicious apps known as bots [52]. Through bots, attackers can take control of many infected mobile devices over a network. This network of devices can be used to launch powerful and sophisticated attacks to commit digital crimes.

2.4 Propagation of Malware

Android malware uses various sophisticated methods to propagate through mobile devices [53]. Below we list the most commonly used Android malware propagation techniques.

Infected apps and websites. The most common way to spread mobile malware is through infected apps and malicious websites. Malware developers repackage popular apps with hidden malicious activities. Additionally, malicious websites exploit system vulnerabilities for malware propagation.

Third party app store. Third-party app stores have less security as compared to the official ones. These third-party app stores are one of the main platforms for malware developers to distribute malicious apps. Most of the repackaged apps are found in these app stores with appealing offers (e.g., paid functionalities unlocked) to lure users into installing them.

Spams. Spams are another widely used technique for malware propagation. An attacker sends emails to users with attractive offers that contain malicious links. A device gets infected once an unsuspecting user clicks such links.

Adware. Adware applications display appealing on-screen ads to mobile users. Sometimes these ads are targeted based on user's information collected by other malicious software. When clicked, these ads can take a user to a malicious site or can exploit other vulnerabilities.

Worms. Worms have self-propagating functionality. Once a device is infected, a

worm can propagate to other devices through different vulnerability-exploits.

Dynamic loading and fake update. Some malware apps keep their malicious functionality hidden before installation. When installed, it loads the malicious codes through dynamic loading. Fake update is another way to load malicious codes. The application appears to perform a necessary update while the device gets infected in the background.

2.5 Machine Learning Techniques in Mobile Malware Detection

As mentioned previously machine learning models have been widely deployed for automated malware detection systems due to the infeasibility of manual inspection for various reasons. Also, we have seen from the current literature that researchers have adopted a variety of machine learning models for malware detection systems. Most notable among them are Support Vector Machine (SVM), K-nearest neighbors (KNN), Decision Tree, and Random Forest. Additionally, Deep Neural Networks (DNN) based detection systems have gained recent popularity among the malware researchers due to their promising performance in different fields such as voice recognition, image classification, object detection, and pattern matching [25, 54–79]. We have also used the deep neural network-based classifier in our experiments along with other classifiers. Below we describe the various machine learning models most commonly used for malware detection systems.

Deep Neural Network (DNN)

A neural network is a machine learning architecture that is structured with layers of neurons which are the primary computing units of the network. Multiple hidden layers between the output and input layers make a neural network deep allowing more complex relationships among the features to be modeled. The input layer takes the

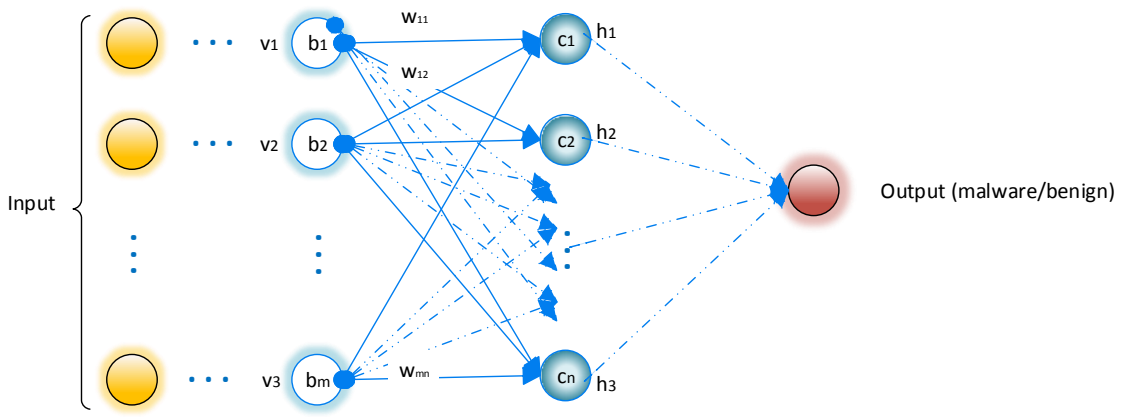


Figure 2.3 RBM with n hidden and m visible units in two layers (b = biases for visible units, c = biases for hidden units, w = connection weights)

feature vectors as input. Each subsequent layer of the network takes input from the previous layer and transforms it into some abstract representation which is fed as the input for the next layer. The network hierarchically extracts complex relationships among the features through the learning process. Deep neural networks were popularized by Hinton et al. who showed the fundamentals of the training process of a Deep Belief Network (DBN) in [80]. DBN architecture is constructed from multiple layers of Restricted Boltzman Machines (RBMs) [81]. An RBM consists of a visible layer and a hidden layer where each node in the visible layer is connected to every node in the hidden layer. The energy function of the network is given by

$$E = -\mathbf{h}^T W \mathbf{v} - \mathbf{c}^T \mathbf{v} - \mathbf{b}^T \mathbf{h} \tag{2.1}$$

where W is the matrix representing the weights of connections between visible and hidden layer, $\mathbf{v}$ and $\mathbf{h}$ are vectors representing data on the visible unit and the hidden unit respectively, $\mathbf{b}$ and $\mathbf{c}$ are the bias vectors for two the layers. The activation probability of visible and hidden units are given by

$$p(h_j = 1|\mathbf{v}) = sig(b_j + W_j\mathbf{v}) \tag{2.2}$$

$$p(v_i = 1|\mathbf{h}) = sig(c_i + \mathbf{h}^T W_i) \tag{2.3}$$

where i is the index of a visible unit and j is the index of a hidden unit. Here, sigmoid function ($sig()$) is shown as example of the activation function for the nodes. Several other activation functions are also used, REctified Linear Unit [82] (RELU) is another popular activation function in this regard.

Figure 2.3 shows an RBM structure with n hidden and m visible units. Multiple RBMs are stacked on top of one another to construct a DBN where the hidden layer of the first RBM serves as the visible layer of the second RBM and so on. After training each of the RBMs individually DBN fine-tunes the parameters through k-step Constructive Divergence (CD-k) method [83] where the gradient θ of the log-likelihood of one training pattern $\mathbf{v}^0$ is approximated by the following equation

$$CD_k(\theta, \mathbf{v}^0) = \sum_{\mathbf{h}} p(\mathbf{h}|\mathbf{v}^k)\frac{\delta E(\mathbf{v}^k, \mathbf{h})}{\delta\theta} - p(\mathbf{h}|\mathbf{v}^0)\frac{\delta E(\mathbf{v}^0, \mathbf{h})}{\delta\theta} \tag{2.4}$$

To update the connection weights, first, the positive gradient denoted by $\mathbf{v}\mathbf{h}^T$ is computed. Afterward, using the previous activation $\mathbf{h}$, subsequent visible and hidden layer activation is computed denoted by $\mathbf{v}'$ and $\mathbf{h}'$ respectively. Thereafter, the negative gradient $\mathbf{v}'\mathbf{h}'^T$ is computed and the weights of the network are updated using the following equation

$$\Delta W_{ij} = \alpha(\mathbf{v}\mathbf{h}^T - \mathbf{v}'\mathbf{h}'^T) \tag{2.5}$$

here α denotes the learning rate.

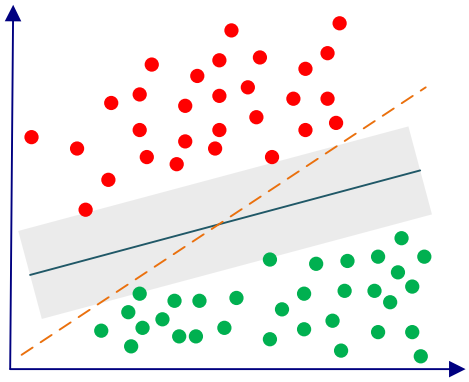


Figure 2.4 Illustration of an SVM classifier. Dashed line represents a candidate hyper-plane while the solid line represents the optimal hyper-plane.

Support Vector Machine (SVM)

Support vector machine is based on the statistical learning theory developed by Vapnik [84]. It is a supervised learning algorithm that categorizes samples by calculating a separating hyper-plane.

A hyper plane in an n-D feature space can be represented by the following equation:

$$f(\mathbf{x}) = \mathbf{x}^T\mathbf{w} + b = \sum_{i=1}^{n} x_i w_i + b = 0 \tag{2.6}$$

where $\mathbf{x}$ and $\mathbf{w}$ are n-dimensional vectors and b represents intercepts of the vertical axes. Dividing by $||\mathbf{w}||$, we get

$$\frac{\mathbf{x}^T\mathbf{w}}{||\mathbf{w}||} = -\frac{b}{||\mathbf{w}||} \tag{2.7}$$

indicating that the projection of any point $\mathbf{x}$ on the plane onto the vector $\mathbf{w}$ is always $-b/||\mathbf{w}||$, i.e., $\mathbf{w}$ is the normal direction of the plane, and $|b|/||\mathbf{w}||$ is the distance from the origin to the plane. Note that the equation of the hyper plane is not unique.

The n-D space is partitioned into two regions by the plane. Specifically, we define a mapping function $y = sign(f(\mathbf{x})) \in \{1, -1\}$,

$$f(\mathbf{x}) = \mathbf{x}^T\mathbf{w} + b = \begin{cases} > 0, & y = sign(f(\mathbf{x})) = 1,\ \mathbf{x} \in M \\ < 0, & y = sign(f(\mathbf{x})) = -1,\ \mathbf{x} \in B \end{cases} \tag{2.8}$$

Any point $\mathbf{x} \in M$ on the positive side of the plane is mapped to 1 (i.e., malware), while any point $\mathbf{x} \in B$ on the negative side is mapped to -1 (i.e., benign). A point $\mathbf{x}$ of unknown class will be classified to M if $f(\mathbf{x}) > 0$, or B if $f(\mathbf{x}) < 0$.

Non-linear Separability

Figure 2.4 shows a simple classification scenario for SVM where the samples are linearly separable. The dashed line represents a candidate hyper-plane whereas the solid line optimizes the separation margin between the two classes. In case of a more complex scenario where samples are not linearly separable, SVM maps the data into a higher dimensional feature space where constructing a separating hyper-plane is more feasible. Table 2.1 shows some popular kernels used for this purpose.

Table 2.1 Popular support vector machine kernels

Kernel	**Expression**	**Comment**
Polynomial	$K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^T\mathbf{z} + 1)^d$	Maps the original n-dimensional samples to n^d dimension
Gaussian	$K(\mathbf{x}, \mathbf{z}) = \exp(-\frac{\|\mathbf{x}-\mathbf{z}\|^2)}{2\sigma^2}$	σ demotes the standard deviation of the sample points.
RBF	$K(\mathbf{x}, \mathbf{z}) = \exp(\gamma\|\mathbf{x} - \mathbf{z}\|^2)$, $\gamma > 0$	$\|\mathbf{x} - \mathbf{z}\|$ denotes euclidean distance and γ is a free parameter used to control over-fitting

K-Nearest Neighbors (KNN)

KNN [85] is a very popular classification technique for its simplicity and its highly competitive performance. KNN is a non-parametric and lazy algorithm in that, it does not make any assumption about the structure of the data rather the structure is derived from the data itself, and also, it does not do any kind of generalization using the training data. KNN works based on a majority voting scheme for classification. To classify a new sample KNN will inspect the class label of its k nearest neighbors

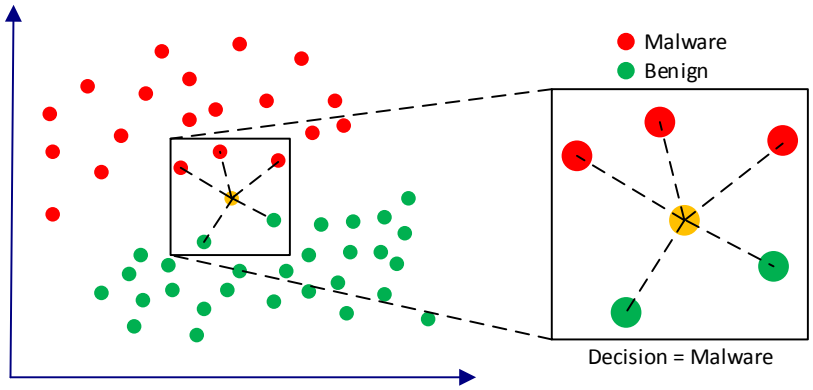


Figure 2.5 K-Nearest Neighbors classifier. Yellow dot represents an unknown sample which is classified as malware based on the majority voting from its nearest k(=5) neighbors.

based on some distance measure (i.e., Manhattan, Euclidean, Minkowski). The new sample will be assigned to the class which is most common among those nearest neighbors. Figure 2.5 illustrates the working principle of a KNN classifier. Here, the algorithm considers 5 nearest neighbors of a sample for the majority voting scheme. In Fig. 2.5, red dots represent malware samples and green dots represent benign samples. The yellow dot is an unknown sample that is classified as malware since 3 of its 5 nearest neighbors are malware.

Decision Tree

A decision tree is a data structure that looks like a tree consisting of nodes and leaves representing attributes and class labels respectively. Each node in the tree selects an attribute to split the data into two subsets that become two branches of the node. To classify an unknown sample the tree is traversed according to the feature value to the samples until a leaf node is reached. The sample is classified as per the label of the leaf node.

Random Forest

Random forest (RF) [86] is an extension of decision tree classifiers that fits many classification trees to a dataset and combines them to predict an outcome in response

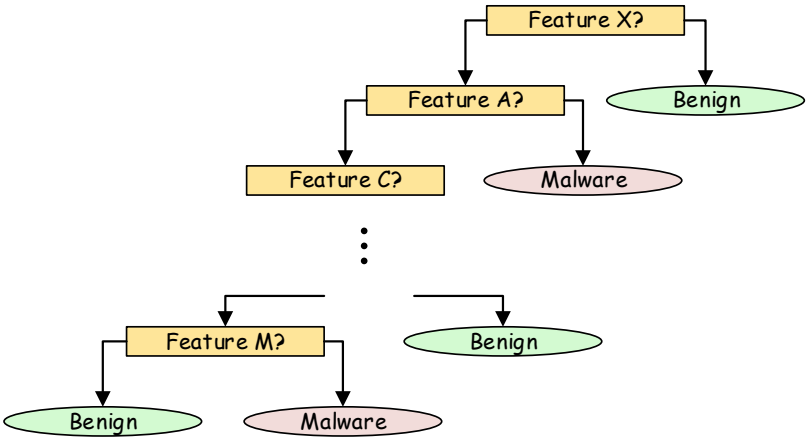


Figure 2.6 Decision tree classifier. Each node subdivides the tree based on a particular feature until a leaf node (decision) is reached.

to an input. The RF algorithm starts off by choosing many bootstrap samples from the data. Approximately 63% of typical bootstrap of the original observations occurs at least once [87]. However, those observations in the original data that do not occur in the bootstrap sample are called out-of-bag observations. A classification tree is mapped to each bootstrap sample. However, each considers only a small number of randomly chosen variables for the binary partitioning. Lastly, the trees will be fully grown and each will be utilized to predict the out-of-bag observations. The calculation of the predicted class of an observation is based on the majority vote of the out-of-bag predictions for that observation, with the ties split randomly.

2.6 Dataset

Malicious application sets for research and experimental purpose is mainly obtained through research communities such as Contagio database and Virustotal [88, 89]. Some researchers also collect and share datasets on request such as the Malgenome dataset [18]. However, the most prominently used dataset in Android malware research is the Drebin dataset [14] which is available to the researchers upon request and used by most of the major works in Android malware detection [54, 56, 90–98]. The key characteristics of this dataset are listed below.

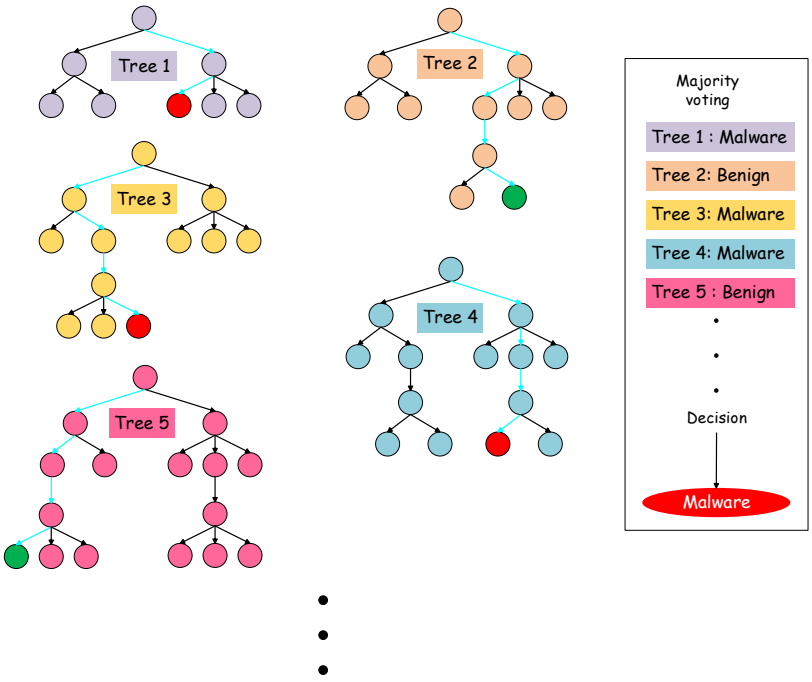


Figure 2.7 Random forest classifier. The final outcome of the classifier is based on majority voting from a collection (forest) of trees.

- The dataset comprises of 5,560 malware and over 120,000 benign applications.
- The samples in the dataset have been collected from August 2010 to October 2012.
- It provides the static features extracted from the samples including permission, API calls, and Android activities.

Due to its prominence in the research community, we also used this dataset for our work. Additionally, samples from Androzoo [19] were included to account for more recent malware techniques that are developed between 2016 and 2019. Further processing for specific experiments was conducted which are described in their due course.

2.6.1 Data Pre-processing and Cleanup

Even though the Drebin dataset provides the extracted features along with the malware application files, we have noticed a lot of noisy data in the feature vectors. For

example, some applications record an API call feature with the name of ';' (a single semicolon). Since these features are extracted by parsing outputs from older versions of third party static analysis tools, it is possible that these noises were generated during the analysis or the parsing phase. To mitigate this, we extracted features for these files with more recent and advanced analysis tools. However, the dataset contains some very old application files which became obsolete since none of the current execution environments (mobile or emulator) will accept them. These applications either raised exception during the analysis phase or resulted in empty feature files and were consequently removed from the dataset. A similar process was conducted for Androzoo dataset for double-checking however, it did not create much of an issue since most of the files are very recent. We extracted both static and dynamic features for all the files and the specifics of feature extraction and vector creation are described in the next Chapter.

2.7 Tools for Extracting Features from Applications

Several tools are available for extracting static and dynamic features from a mobile application. These are commonly known as reverse engineering tools. In the following, we describe some of most popular reverse engineering tools for Android applications.

2.7.1 Static Feature Extraction Tools

Apktool [99] is one of the most popular tools for extracting static features from an Android application. It is a Java programming language-based tool that can decompile an APK file and decode its resources. As the APK files are nothing but a compressed zip file, traditional software can be used to unzip its resource files such as AndroidManifest.xml, classes.dex and META-INF files. However, the files are encoded and cannot directly be used for analysis. Apktool can decode these files and make them readable. The decoded AndroidManifest.xml file can be used to

extract various features such as permission, Android activities, Intents, and Broadcast receivers while the classes.dex files can be used to extract API calls and call graphs. Androguard [100] is another reverse engineering tool for static analysis developed by Google and gaining more popularity for its sophistication and ease of use. While the files decoded by Apktool needs to be manually parsed, Androguard does this automatically for feature extraction.

2.7.2 Dynamic Feature Extraction Tools

For dynamic feature extraction, the application needs to be executed either on an actual device or an emulated environment. Cuckoodroid [101] is one of such tools for automatic dynamic analysis of Android apps. It executes an Android application on a sandboxed environment and monitors its dynamic behavior. However, in our work, we implement our own dynamic analysis system for greater flexibility. We executed the apps on an emulator [102] and logged the system calls while random events were generated using the Monkey Tool [103] during the execution.

2.8 Performance Metrics for Malware Detection Systems

Several performance metrics are used for machine learning-based malware detection systems. Below we describe the most widely used performance measures along with their formulae. In the following equations, TP is the number of true positives, i.e., malware accurately predicted; FP is the number of false positives, i.e., benign apps that were predicted as malware; TN is the number of true negatives, i.e., benign apps accurately predicted and FN is the number of false negatives, i.e., malware that were predicted as benign apps.

Accuracy: Accuracy is the measure of how accurately a classifier can detect malware and benign applications:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \times 100\% \tag{2.9}$$

Precision: Precision indicates the percentage of actual malware among the apps that are classified as malware:

$$Precision = \frac{TP}{TP + FP} \times 100\% \tag{2.10}$$

Recall: It indicates the percentage of malware that were correctly classified. It is also known as *sensitivity* or *true positive rate*:

$$Recall = \frac{TP}{TP + FN} \times 100\% \tag{2.11}$$

Specificity: It is also known as *true negative rate*. It indicates the portion of actual benign apps that are correctly identified.

$$Specificity = \frac{TN}{TN + FP} \times 100\% \tag{2.12}$$

G-mean: The geometric mean (G-mean) is the square root of the product of class-wise sensitivity. This measure tries to maximize the accuracy of each of the classes while maintaining a balance.

$$G\text{-}mean = \sqrt{specificity \times sensitivity} \tag{2.13}$$

F-score: F-score is a measure of accuracy that tries to balance the precision and recall by calculating their harmonic mean.

$$F1\text{-}score = 2.\frac{Precision \times Recall}{Precision + Recall} \tag{2.14}$$

ROC curve and AUC: Receiver Operating Characteristic (ROC) curve and Area Under the Curve (AUC) are two of the most commonly used performance measures for

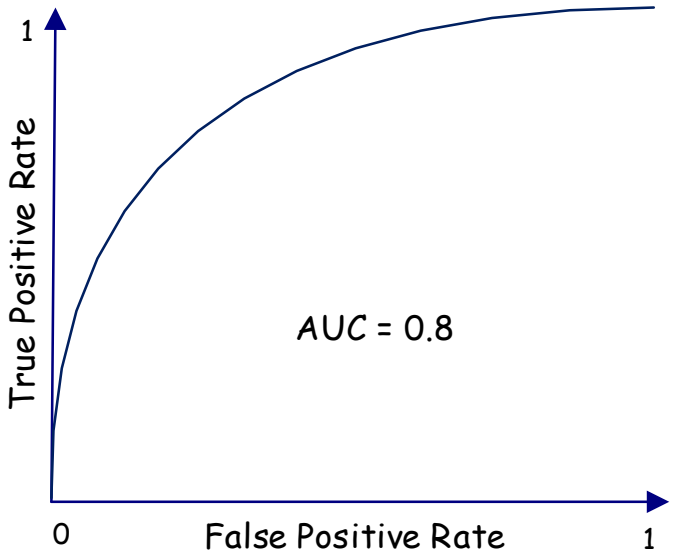


Figure 2.8 Example of an ROC curve and AUC.

machine learning models [104–106]. These two measures give a visual description of the model's performance. Figure 2.8 shows an example of an ROC curve and AUC. ROC is a probability curve that illustrates the classifier's capability to distinguish between the samples of different classes while AUC is a probability measure that represents the area under the ROC curve. For example, an AUC value of 0.8 means that the area under the ROC curve is 0.8 and there is an 80% probability that the model will be able to distinguish between benign and malware samples.

2.9 Mobile Malware Detection Systems Based on Feature Categories

The analysis techniques of machine learning-based malware detection can be classified into two types: static and dynamic, dealing with static and dynamic features respectively [107]. The static features can further be divided into three categories which will be described in Sec. 2.9.1. There is another approach known as hybrid analysis that combines static and dynamic analysis techniques. Table 2.2 summarizes the properties, advantages, and disadvantages of these analysis techniques. Below we

Table 2.2 Summary of Android malware detection techniques based on feature types.

Analysis technique	Properties	Advantage	Disadvantage
Static analysis	Requires to disassemble and decode the files from the application package. The decoded files are then parsed to extract static features.	Does not require the application to be executed. It is computationally less expensive and relatively faster.	Vulnerable to several attacks including code obfuscation and dynamic code loading.
Dynamic analysis	Requires an application to be executed in a real or emulated environment. Different events are generated to emulate run-time behavior which is dynamically logged for extracting the features.	Suitable for detecting attacks such as code obfuscation and dynamic code loading.	Requires the application to be executed. Sometimes can fail to detect dynamic attacks due to the corresponding events not being generated during the analysis phase.
Hybrid analysis	Combines the features extracted from static and dynamic analysis in an intelligent manner.	Can cover a wide range of attacks in both static and dynamic situations.	The process is complex and it is the most computationally expensive analysis method.

present the most notable works in the literature based on these techniques.

2.9.1 Static Analysis

Static analysis methods find malicious activities in an application by analyzing the application binary and code without actually running the application [108–110]. For static analysis, the Android Application Package (APK) file (described in Sec. 2.2) is disassembled and its different components are parsed to extract the features. Static features of an Android application can be divided into three categories: permission, Application Programming Interface (API) calls, and Inter Component Communication (ICC) features. These features are extracted from the androidmanifest.xml and classes.dex files included in the APK package.

Different works have considered different categories of features to develop static mal-

ware detection systems. In [10] Aiman et al. introduced a basic categorization system for Android applications. The work extracted the permission of the applications and studied two categories of application: business and tools. The work classified the applications by applying the k-means clustering algorithm and achieved 71% recall rate.

In [11] Suleiman et al. proposed a Bayesian classifier based mobile malware detection system. The work extracted permission features from a dataset of 2000 samples and achieved an overall accuracy of 93%. Veelasha et al [12] proposed a technique that, in addition to the requested permissions in the manifest file, analyzed the application code to extract the permissions that were actually used in the code. Applying a data mining approach, the work found contrasting properties between the patterns of requested and used permission. The work concludes that these contrasts among benign and malware applications can be useful for malware detection. However, the major drawback of the above approaches is that they only considered the permission feature category. These methods were effective for the early generation of malware apps since they often tried to gain over-privilege by requesting more permissions than necessary. However, later malware applications came with smarter techniques to conduct malicious activities. As a result, later approaches additionally considered different other feature categories.

Li and Li in [13] proposed a static analysis method that focused on the Android code organization. It first parsed the classes.dex file to extract the API calls and afterward built a characteristic tree based on the usage and calls of those APIs. It devised a novel method for analyzing similarities among these trees and proposed that it could be useful for malware classification since malware samples have similar characteristics. The work, however, did not report its performance in an actual malware detection scenario.

Drebin [14] is another static analysis method that included a wide range of static

features for malware detection. The work categorized the static features extracted from an application into eight sets: hardware features, requested permissions, app components, filtered intents, restricted API calls, used permissions, suspicious API calls, and network addresses. Using an SVM classifier, the work achieved an overall accuracy of 94%. Drebin is one of the most prominent works in the literature that extensively considered a broad range of static features including permission, API calls, and ICC. However, since it did not consider the dynamic behavior, it is vulnerable to code obfuscation and dynamic code loading attacks [111].

In addition to extracting the API calls, further information about the behavior of an application can be obtained by analyzing the context of an API call. Android systems rely heavily on call back methods. These call back methods are related to the events that cause an API function to be invoked. For example, an API method (e.g., sendSMS()) could be called when a user clicks on a button. In this case, a call back method such as “onClickListener" will be tied to the button click event and sending SMS will be handled by this callback method. On the other hand, in a potential malicious case, the same API could be called based on some system-generated event such as a change in the signal strength. In this case, the SMS will be sent in a different context and handled by a different callback method (e.g., a broadcast receiver). In the above scenario, the user is aware of the first call but unaware of the second. These are known as contextual information and a detailed discussion on this is presented in Sec. 3.3. A few works in the literature considered this contextual information for malware detection.

AppContext [112] proposed a method that analyzed the call graph of an application to trace back the entry point of the API calls. These entry points are related to the activation events of an API call. It also extracted the associated environmental information such as if a database query is made or if the system time is obtained before an API call. All this information was encoded to construct a feature vector. It then used an SVM based classifier to identify maliciousness of an API call. Though

the work reported a 95% recall rate, the dataset used in the work is fairly small (only 202 malicious apps) and the method is very expensive computationally. Moreover, the work poses a 5MB file size constraint making it very unlikely to be applicable in real-world scenarios.

Narayanan et al. [113] introduced a graph similarity-based approach for malware detection based on API and contextual information. The work first obtained the call graph of an application where each node in the graph represents an API call. It then traced the activation events of these API calls which were used to determine the context, i.e., whether the user was aware or unaware of the call. Each node of the graph is then re-labeled based on this contextual information. A contextual graph kernel is then computed from this labeled graph and unknown samples are classified according to the similarity of the graphs. The work was further improved in [114] that considered multiple views of the graph in addition to the API dependency graph such as, permission dependency graph and information source-sink graph. The later work reported an overall accuracy of 98%. One major difference of their works from others is that they considered the structural information of the application. However, capturing all the structural information is an overwhelmingly difficult task due to the heavy reliance on call-back methods and execution jumps of Android applications. Moreover, the execution structure can easily be changed for applications by simply altering the execution order, or using code obfuscation and dynamic code loading techniques which will render this type of detection method ineffective.

The third category of static features is known as Inter Component Communication (ICC) features. In Sec. 2.2 we described the different components of an Android application. ICC mechanism allows these components to communicate with each other which is known as Inter Component Communication. The communicating components can be from the same or different application(s). Android operating system facilitates this ICC communication to enable the reuse of codes so that the functionality of a component can be used and shared by others. This reduces the burden from

the application developers [115]. However, researchers have found that malware may abuse this ICC mechanism to evade detection and to conduct malicious attacks (e.g., collusion attacks) [116–119]. Different tools like Amandroid [120], Epicc [121], and IC3 [122] have been developed to extract the communicating components (sources and sinks) using ICC mechanism within an Android application. There are also tools like IccTA [123] that identify taints of ICC leaks by constructing precise control flow graphs.

ICCDetector [15] is a technique that proposed detection of malicious applications using the ICC features. It designed a parser on top of Epicc [121] to extract ICC related features such as component, explicit intent, implicit intent, and intent filters. The method used an SVM to classify malicious behaviors and obtained 88.8% accuracy.

Feizollah et. al. in [124] considered “Android intents" (explicit and implicit) as ICC features and combined it with required permissions. They argued that the “Android intents" are semantically significant features that could be used to reveal hidden malicious behaviors more effectively when compared to other features such as permissions. Using the Bayesian Network algorithm the work evaluated different detection approaches that considered the features separately and in combination and achieved 91% accuracy.

The methods considering ICC features are effective in revealing particular malicious intentions such as information leaks and collusion attacks. However, they also add redundancy in the feature space and adversely affect the classifier since all the communicating components are not related to security threats. Hence, we see that the approaches in [15, 124] are outperformed when other features are considered in combination with ICC.

2.9.2 Dynamic Analysis

Dynamic analysis requires the application to be executed in a suitable environment such as an actual device or an emulator [125–131]. During the execution of an application, the dynamic behavior of the application is monitored, logged and analyzed. Dynamic analysis can be used to detect attacks involving code obfuscation and dynamic code loading.

Different features are considered as dynamic within an Android system including Android system calls, network communication properties, and run-time usage of system components. Among these, system calls are the most commonly used features in dynamic analysis [132–134]. Every application requests resources and services from the Android operating system using system calls. For example, opening, reading or writing files, reading connection status, etc. There are more than 250 system calls available in Android operating system from Linux kernel [135]. Below we present the works in the literature that adopted the dynamic analysis approach.

Xiao et al. [132] proposed a technique for malware detection based on the sequences of system calls. The work trains two deep neural networks one for the malicious class and one for the benign application class. When a new application is tested, two similarity score is computed from these two networks and the application is classified based on this score. The work obtained 93.7% accuracy.

Deep et al. [136] proposed an Android malware classification system that extracted system calls as features. It computed the value of a particular feature based on its Term Frequency-Inverse Document Frequency (TF-IDF) value [137]. In this method, the system calls that occur more commonly in a particular class get higher values and the ones that occur rarely get lower values. It also incorporated a Rough Set and Statistical Test-based feature selection and was able to reduce the false positive rate to 1% using Random Forest as the classifier.

Abderrahmane et al. [138] proposed a system call based Android malware detection

system where each application is analyzed in a remote server. The server installs and executes the application in a simulated environment and the system calls of the application are logged during this execution time. The work considers system calls from the Linux kernel and represents each of the applications by a 250×250 matrix. Each entry in the matrix represents the dependency of a pair of system calls based on their distance on the execution sequence. The method reported 93.29% accuracy using a deep neural network-based classifier

Afonso et al. [133] proposed a dynamic malware detection system that logs the frequency of Android system calls to detect malicious behavior. The work evaluated the performance of the system using different classifiers including Random Forest, Naive Bayes, and Decision Tree, and was able to achieve 96.82% accuracy for malware detection. The work however is limited to a certain version of API in Android system.

Crowdroid [135] was proposed by Burguera et al. that classifies malicious applications based on dynamic behavior. It first installs an analyzing application in the device which then attaches a tracing process (strace) to an application to log the system calls. This log is sent to a remote server where K-means clustering algorithm is used for classification. One major drawback of this system is that it requires the user to install an additional application and also it is highly dependent on the number of users using this application.

Marko et al. [134] developed the Maline system that extracted the system call sequence of an Android application. It recorded the frequency of each system call and also formed a dependency graph among them based on their paired distance in a sequence. The work experimented with different classifiers including SVM and Random Forest and achieved an overall accuracy of 96% on Drebin dataset.

Bhatia and Kaushal [139] proposed a system that also used system calls for malware classification. The work applied Decision Tree and Random Forest as classifiers and obtained 88% accuracy. However, the dataset considered in the work is very small

(only 50 malicious applications) and hence, the applicability of the system in real scenarios needs to be evaluated.

Even though dynamic analysis is necessary for uncovering certain attacks such as dynamic code loading and code obfuscation, the methods proposed here has some major drawbacks. To reduce the risk of infecting an actual device, extraction of dynamic features is performed within a sandbox or an emulated environment. There are malware applications that can detect these execution environments and can evade detection by hiding their malicious functionalities during the analysis phase. Also, during the automated execution process, random events are generated to explore different functionalities, execution sequences, and dynamic behavior of an application. It is not always possible to exhaustively generate all the events so that each and every execution sequence of an application is covered.

2.9.3 Hybrid Approach

Static analysis reveals a lot of information about an application including over-privilege, security-sensitive operations, and information leaks. However, it fails to detect code modification and intrusion of virus. Static analysis cannot capture dynamic loading methods because the corresponding code remains encrypted which is decrypted and fetched in the memory during execution. This characteristic can also be used by malware developers to evade static detection. Dynamic analysis, on the other hand, can address these obfuscations, repackaging, and encryption attempts since they are exposed during the execution time. However, dynamic analysis has its drawback which is mentioned in the earlier section. As a result, malware researchers have opted for hybrid methods that combine both static and dynamic analysis techniques for a comprehensive analysis of an application.

Jang et al. [140] devised a hybrid approach considering both malware-centric and creator-centric information. As features, it extracts certificate serial number, permis-

Table 2.3 Summary of key works in mobile malware detection

Work	Feature types	Contribution	Dataset	Performance	Limitations
Yerima et al. [11]	Static (Permission)	Proposed a bayesian classifier-based system that utilized permission features for malware detection.	Malgenome	Accuracy = 93%	Only considers permission features. Vulnerable to new generation of malware.
Samra et al. [10]	Static (Permission)	Proposed a basic categorization system of Android application based on k-means clustering and permission features.	Personal	Recall = 0.71	
Ganesh et al. [90]	Static (Permission)	Designed a system that sends data to a remote server where it is transformed into an image format. Uses a Convolutional Neural Network (CNN) for classification.	Personal	Accuracy = 93%	Classification can be misled during data trasmission.
Karbab et al. [94]	Static (API)	Designed MalDozer system that identifies malware families by analyzing the sequence of API calls.	Drebin	Precision = 0.96 Recall = 0.96 F1-score = 0.96	Vulnerable to code obfuscation and dynamic code loading attacks.
Li et al. [91]	Static (API)	Utilized a weight-adjusted deep learning technique for malware detection.	Drebin	Recall = 0.94	
Hou et al. [142]	Static (API)	Developed AutoDroid that extracts APIs from smali codes and uses a Deep Belief Network for malware detection.	Personal	Accuracy = 96.66%	
Zhang et al. [93]	Static (API)	Presented DeepClassifyDroid that extracted five different feature sets through static analysis and used CNN for malware detection.	Personal	Accuracy = 97.4% Precision = 0.97 Recall = 0.98 F1-score = 0.97	
Nix and Zhang [143]	Static (API)	Extracted API call sequence and used a CNN for learning feature representations which are then used for classification.	Contagio	Accuracy = 99.4% Precision = 1.00 Recall = 0.98	
Narayanan et al.[114]	Static (Context)	Utilizes graph similarity that matches the dependency graph of an application with that of benign and malicious samples.	Personal	Accuracy = 98%	Considered context in isolation. Cannot be combined with other feature categories
Yang et al. [112]	Static (Context)	Devised a malware detection technique that extracted environment information associated with sensitive API calls.	Personal	Accuracy = 94.8%	Considered dataset is very small. 5MB app size constraint limits its applicability.
Xu et al. [15]	Static (ICC)	Proposed a technique to detect collusion attacks among malicious applications by utilizing ICC features.	Drebin	Accuracy = 97.4	Only considers ICC features. Vulnerable to a range of other attacks.

Table 2.3 summary of key works cont.

Work	Feature types	Contribution	Dataset	Performance	Limitations
Su et al. [54]	Static (Permission, API)	Utilizes a deep belief network for malware detection using permission and API call features.	Drebin	Accuracy = 96%	Vulnerable to code obfuscation and dynamic code loading attacks.
Li et al. [95]	Static (Permission, API)	Applies a deep neural network-based system on required permission and API calls for malware detection.	Drebin	Pecision = 0.97 Recall = 0.94 F1-score = 0.96	Considers a very small feature set.
Wang et al. [144]	Static (Permission, API)	Uses a Deep Autoencoder (DAE) to extract features and a convolutional neural network for malware detection.	Personal	Accuracy = 99.8% Recall 0.99 F1-score = 0.99	Depends on the DAE's ablity to extract meaningful features.
Arp et al. [14]	Static (Permission, API, ICC)	Introduced an SVM based malware detection system that considered a wide range of static feature categories including permission, API calls, and ICC features.	Drebin	Accuracy = 94%	Constructs a very sparse feature space than can lead to over-fitting.
Jiang et al. [145]	Static (Permission, API, ICC)	Introduced a fine-grained dangerous permission feature-based detection system that identifies the most dangerous features and uses them for malware detection.	Personal	F1-score = 0.94 Recall = 0.94	Vulnerable to dynamic loading and code obfuscation attacks.
Dimjašević et al.[134]	Dynamic	Proposed a malware detection system based on the frequency of system calls and their dependency.	Drebin	Accuracy = 96%	Exhaustive coverage of execution sequences is not feasible.
Martinelli et al. [97]	Dynamic	Utilizes a Convolutional Neural Network on system call sequences of applications for malware detection.	Drebin	Accuracy = 95%	
Liang et al. [146]	Dynamic	Considers system call sequences as text and detects malware by extracting themes from the text.	Personal	Accuracy = 93.1% Precision = 0.96 F1-score = 0.87	Vulnerable to malware attacks that can suppress activities in emulated environments.
Yuan et al. [147]	Hybrid	Proposed a hybrid feature based malware detection system that utilized permission, sensitive API calls, and dynamic behaviors.	Malgenome	Accuracy = 96.8	Uses coarse grained features.
Yuan et al. [55]	Hybrid	Introduced a deep belief network-based hybrid approach for Android malware detection.	Contagio	Accuracy = 96.5	Considers a very small feature set.
Alshahrani et al. [98]	Hybrid	Developed the DDefender system that utilizes data from a client-side application for malware classification.	Drebin	Accuracy = 95%	Requires additional application to be installed.

Table 2.3 summary of key works cont.

Work	Feature types	Contribution	Dataset	Performance	Limitations
Vinayakumar et al. [148]	Hybrid	Proposed the use of a Long-Short Term Memory-based detection system for Android malware detection.	Personal	Accuracy = 97.5%	Suffers from long training and classification time.

sions, API sequence, and the combination of system commands and forged files. For classification, it computes the similarity score based on these features. It then matches this score against a pre-computed score of benign and malicious behavior. The work reduced the false-negative rate to 1.77%. However, there are a few drawbacks of such approaches. Since it uses the serial number as a feature, the detection system can potentially be broken by obtaining a new serial number. Also, malware creators can alter the sequence of API calls to bypass the detection while being capable of carrying out malicious behavior [141].

Tong and Yan [149] adopted another hybrid approach that dynamically collected the data and statically analyzed them. The work first created a database for malicious patterns and benign patterns by calculating the frequency and weight of sequential system calls at different depths based on a prefixed threshold. Then it inspected the system call sequences of the unknown app and matched them with a known signature database. Decision was made based on whether the sequence matched the malicious pattern or benign pattern. Kapratwar [150] designed a hybrid method taking the permissions as static feature and system calls as dynamic feature. The work studied different machine learning approaches including Decision Tree, Random Forest, and Naive Bayes.

The above approaches evidenced that considering a hybrid of static and dynamic features is more effective in malware detection rather than adopting a static or dynamic approach on its own. However, these methods are evaluated in a small scale environment considering a minimal subset of the features and they are largely dependent on the number of applications that are already analyzed to build their database.

In table 2.3 we provide a summary of the most notable works in the literature detailing their characteristics, key contributions, and performance. This concludes the literature review pertaining to our first research objective. In the following, we review an evasive attack technique, known as adversarial attack, which breaks classification systems and challenges their robustness.

2.10 Adversarial Attack

Machine learning models can detect malicious applications with state-of-the-art performance if an optimal combination of feature categories is provided. However, recent literature shows than such detection systems can be evaded with attacks using adversarial samples. Even though adversarial samples were first introduced to attack machine learning classifiers in the image processing domain, similar principles can be applied to evade malicious application detection systems as well. In this section, we present an overview of the adversarial attack and defense against such attacks proposed in the current literature.

2.10.1 Adversarial Samples

Adversarial samples are artificial samples that are crafted by introducing small perturbations to the original samples. Szegedy et al. [151] first discovered that several machine learning models, including neural networks, are vulnerable to adversarial samples. The authors showed that carefully introducing perturbations into an original sample can mislead a machine learning classifier into misclassifying that sample. This careful introduction of perturbation is different in several aspects from adding random noise into the dataset which is traditionally used to make a machine learning model robust. Firstly, the perturbation is not introduced randomly rather it is carefully calculated. Secondly, the purpose of introducing this perturbation is to specifically mislead the classifier unlike random noises that may or may not affect the

classifier performance. And thirdly, the perturbation is kept at a minimum so that the crafted sample and the original are not easily distinguishable. Following the work in [151], several other methods to craft such adversarial samples have been proposed in the literature. Some of the notable works in this regard are described below.

2.10.2 Adversarial Sample Crafting Techniques and Defense Mechanisms

Most of the approaches to craft adversarial samples suggest to minimize the distance between the adversarial sample and the sample to be modified, i.e., minimize the perturbation required to shift the classification to an incorrect class [152]. Szegedy et al. [151] introduced a method to craft adversarial samples by minimizing the following loss function

$$loss(\hat{f}(x+r), \hat{y}) + c.|r| \tag{2.15}$$

where x is the original sample, $\hat{y}$ is the desired outcome, r is the perturbation added to the sample and c is a parameter used to balance the distance between the original sample and the perturbation. The author proposed to solve this using a box-constrained L-BFGS [153]. It essentially tries to find the minimal amount of perturbation (r) needed to misclassify the sample x to class $\hat{y}$ rather than to its original class.

Goodfellow et. al proposed a fast gradient sign method in [154] that uses gradients of the output with respect to the input features for adversarial sample crafting. The method adds a small value, ϵ, to each of the features of a sample so that the sample is misclassified. The method solves the following equation for adversarial sample crafting.

$$\hat{x} = x + \epsilon \cdot sign(\nabla_x(loss)) \tag{2.16}$$

Equation (2.16) computes the gradient of the loss function with respect to the input. However, only sign (+ve or -ve) of the gradient is used to determine whether to add or subtract the perturbation. When the gradient is positive it adds a small value to the features and when the gradient is negative the value is subtracted. This allows the method to add values in the direction of the gradient, i.e., and increase the loss function value so that a sample is misclassified.

Su et. al [155] proposed a technique that modified only a single feature for adversarial sample crafting. Similar to other adversarial samples crafting methods, it tries to keep the adversarial sample as close to the original sample as possible. However, unlike other methods that try to find minimal perturbation, this method limits the perturbation to a single feature. Nevertheless, different properties of a single feature can be modified. For example, in case of an image classification task with pixels as features, the x- and y-coordinate, red, green, and blue (RGB) channels of a single pixel is modified. It uses a genetic algorithm-based approach that starts with a set of randomly formed candidate samples and progresses in an iterative fashion. In each iteration, it forms a new set of candidate solutions by modifying the current set of samples. The process stops when a candidate sample is successful to be misclassified.

Additionally, some works studied the physical aspect of adversarial samples. For example, Brown et. al [156] developed a technique that creates printable adversarial patches that can be stuck beside an actual image to fool a classifier. Athalye et. al [157] proposed adversarial samples crafting techniques for 2D and 3D classifiers. It showed an example where a turtle was 3D printed and was able to be misclassified as a riffle.

There are very limited works in the literature that have reported adversarial example crafting techniques and their impact on malware detection systems, while even less attention has been focused on adversarial attacks in *mobile* malware domain. Most of the earlier works in this domain proposed techniques only for crafting the adver-

sarial samples to fool a malware detection system. A few of the later works studied the defense strategies against such samples. Most notable works in this regard are presented below.

Anderson et al. [158] adopted a reinforcement learning-based approach for crafting adversarial examples. It modified a malicious sample in an iterative fashion to bypass malware detection systems. In each iteration it modifies the malicious sample by performing a single action from a set of acceptable actions that includes adding a function, manipulating existing section name, and modifying header checksum. This iterative process continues until the sample is misclassified by the target model. The work achieved 16% evasion rate on a model which is trained for 100,000 round on original data. The work did not devise a defense mechanism however, it proposed that a model should be retrained on the adversarial samples to make it robust against adversarial attacks.

Dang et al. [159] proposed an adversarial malware sample crafting technique using three tools namely, a detector, a morpher, and a tester. The detector is the classification system that the attacker wishes to evade, the morpher is a tool that modifies a malicious sample and tester is a tool that checks whether a modified sample still contains the malicious functionality or not. A malicious sample is considered a successful adversarial example if the tester verifies it as a malicious sample but the detector accepts it as a benign sample. The method carries out a series of morphing operations to obtain several modified copies of a malware sample. Afterward, it adopted a hill-climbing approach where the number of morphing steps was leveraged as a scoring mechanism to find the closest sample that is able to evade the detection. The work reported a 100% evasion rate by the adversarial malware samples crafted with this technique. However, it did not propose any defense strategy.

Yang et al. [160] introduced a Malware Recomposition Variation (MRV) approach based on the semantic analysis of various existing malware applications for crafting

adversarial malware samples. Through phylogenetic analysis, the work observed how different features of malicious applications evolve over different versions. To craft adversarial malware samples, it used a semantic-feature mutation technique that automatically mutated the malware bytecode to inject malicious behaviors into an application component. However, the method proposed in this work requires access to the malware binaries and source code. Further, it also requires a specific component of an Android application, i.e., broadcast receiver, which can easily be detected since it is a common characteristic of a malicious application to invoke sensitive APIs from broadcast receivers.

Grosse et al. [161] studied adversarial example crafting specifically for Android malware detection. 63% of the adversarial samples crafted in the work were able to successfully bypass the malware detection system. The work also explored adversarial training as a defense mechanism against adversarial attacks where adversarially crafted malware samples are used to retrain a classifier to make it robust. It showed promising results against adversarial attacks where several architectures of deep neural networks along with varying ratios of malware and benign samples were considered. However, retraining a classifier with too many adversarial samples can lead to overfitting and degrade the performance and the work did not consider how to optimally select these samples for retraining.

Unlike adversarial retraining, Papernot et al. [162] proposed a different defense mechanism based on distillation. In this method, a classification model M is first trained using the original data and the probability of the samples belonging to a certain class is recorded. A second classification model M' is then trained using these probabilities as the label. However, this mechanism is not as effective in malware detection as in other domains such as image processing and computer vision. Other defenses against adversarial attacks include classifying adversarial samples as a separate class and differentiating them based on their statistical property [163, 164].

Table 2.4 Summary of key research and their main contributions on adversarial attacks in mobile malware detection.

Work	Contribution	Comments
Anderson et al. [158]	Utilizes a reinforcement learning-based approach that iteratively modifies a malware sample to craft adversarial examples.	It achieves a 16% evasion rate. Does not suggest any defense machanism.
Dang et al. [159]	Utilizes a detector, a morpher, and a tester to craft adversarial malware samples.	Achieves a 100% evasion rate. Doest not propose any defense technique.
Yang et al. [160]	Adopts Malware Recomposition Variation (MRV) approach based on the semantic analysis of malware samples. Injects malicious byte codes into applications to craft adversarial samples.	Requires access to malware source code.
Grosse et al. [161]	Uses saliency map-based adversarial sample crafting technique and achieves 63% evasion rate. Proposes adversarial retraining as a defense mechanism.	Sub-optimal performance due to random choice of samples.

The effectiveness of adversarial malware attack and its defense in mobile malware detection settings is a relatively unexplored area unlike image processing and computer vision domain. Furthermore, intelligent selection of samples has shown improved performance in different areas of machine learning applications [165, 166]. However, such selective sampling is yet to be applied for adversarial retraining in mobile malware detection domain. In our second research objective, we address this issue and improve the robustness of malware detection systems against adversarial attacks using a novel selective sampling strategy.

The following section presents the literature review for our final research objective that deals with imbalanced data distribution.

2.11 Imbalanced Data Problem for Mobile Malware Detection

The machine learning models descried earlier are designed to expect a balanced data distribution, i.e., it expects that the number of benign and malware samples are nearly equal. If the dataset is imbalanced, the performance of such models can suffer to a great extent. A dataset is considered imbalanced if the number of samples in one class is significantly higher than that of another. This characteristic of data distribution occurs naturally in malware detection systems since in the real world the benign applications outnumber malicious applications to a great extent. In fact, Xu et al. [16] reported a class imbalance in the order of 2 in 1000 in mobile malware detection. Even though various fields of research including telecommunication, text classification, and image processing have tried to deal with data imbalance [167–175], the problem becomes highly critical in the field of malware detection because the cost of incorrectly classifying a malware (i.e. false negative) is significantly more than that of a benign application. Misclassifying a malicious application could result in a harmful application being launched in the market potentially leading to privacy leaks, information stealing, and financial loss. Existing methods in the literature for handling imbalanced data problems can be divided into three categories namely, data-level approach, algorithm-level approach, and hybrid approach [176]. Table 2.5 summarizes the properties, advantage, and disadvantages of these approaches. In the following, we present the most notable works in the literature.

2.11.1 Data Level Approaches

Data-level approaches or data augmentation is one of the most common approaches to handle imbalanced data [16, 169, 177–182]. This approach reduces data imbalance by changing the distribution of the training data. Traditional approaches of data augmentation include over-sampling [17, 183] and under-sampling [174, 184] which

Table 2.5 Imbalanced data handling approaches overview.

Approach	Properties	Advantage	Disadvantage
Data-level methods	Tries to balance the dataset by changing the distribution of the data, e.g., removing majority class samples or creating synthetic minority class samples.	Does not depend on a particular algorithm. Simpler to implement.	Often lead to poor generalization, information loss, and may generate *invalid samples*.
Algorithm-level methods	Modifies an underlying algorithm to handle imbalanced data, e.g., modified loss function.	Relative importance on majority and minority classes can be adjusted.	Solution is algorithm specific.
Hybrid methods	Strategically combines both data-level and algorithm-level methods.	Generates balanced data distribution and has the capability of assigning relative importance to each classes. Can be tailored to be used in various domains.	The process is complex and computationally expensive.

in their simplest form refer to simply copying random minority class samples and discarding random majority class samples, respectively [182]. However, these simple techniques have major drawbacks. In the case of random over-sampling, since mere copies of minority class samples are created, the classifier often gets redundant information. This leads to over-fitting and poor generalization, i.e., the classifier doesn't perform well on unseen test data. Random under-sampling on the other hand, discards samples from the majority class that lead to information loss, i.e., the samples that are critical to learn the characteristics of the data often get removed from the training data. To mitigate these problems researchers have opted for devising intelligent over-sampling and under-sampling methods to minimize redundancy and information loss.

SMOTE (Synthetic Minority Over-sampling Technique) [17] proposed by Chawla et al. is one of the most notable intelligent over-sampling techniques in this regard. Rather than simply creating copies, it generates artificial samples for the minority class. Figure 2.9 illustrates how SMOTE algorithm works. For each minority class

sample, it selects a number of nearest samples from the same class using K-NN rule. Afterward, it randomly interpolates the features between the neighboring samples to create artificial samples. In Fig. 2.9, green and red dots represent majority and minority samples respectively. The yellow dots are the interpolated samples between neighboring minority samples.

Several variants of the SMOTE algorithm were proposed to further improve its performance [185, 186]. Before creating artificial samples these works examined the neighbors from the majority class and prevented over-sampling in overlapping regions to minimize generation of noisy data.

Jo and Japkowicz [187] proposed another intelligent over-sampling technique that considered multiple clusters of the minority class. The work first clustered the samples and then performed over-sampling within each cluster to improve both intra- and inter-class imbalance.

These methods however are only suitable for continuous features. Also, since they do not pay any particular attention to the characteristics of the features, an immutable feature can be modified by these methods resulting in invalid artificial sample creation.

Lee et al. [188] proposed a two-phase learning approach to classify highly-imbalanced datasets. In the first phase, the proposed method down-sampled the majority class until it has at most 5000 samples. This down-sampled dataset is used to pre-train a classifier which is then fine-tuned using the complete dataset in the second phase. The work reported improved minority class performance, however, major drawbacks in this work are determining the creation process of the down-sampled dataset and finding the optimal choice of the minimum number of samples.

Mani and Zhang [189] proposed a k-nearest neighbor (K-NN) based under-sampling approach. It devised multiple methods that removed a majority class sample from the training data based on its distance from the minority samples. For example, their NearMiss-1 method calculated the average distance of a majority sample from

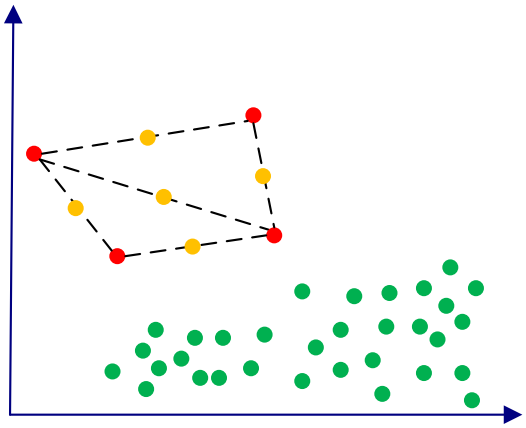


Figure 2.9 SMOTE algorithm for handling imbalanced datasets. Red and green dots represent minority and majority classes respectively and yellow dots are the interpolated samples.

three nearest minority samples and only kept the samples that have the smallest distance. NearMiss-2 on the other hand calculated the average distance of a majority sample from three farthest minority samples. The work presented a comparative study among these methods. However, the choice of different parameters such as, how many minority samples to use for distance calculation is a major drawback in this work.

Additionally, Barandela et al. [190] proposed a technique that removes majority samples from the class boundaries. Using K-NN rule based on Wilson's editing [191] the work removes the samples that are misclassified from the training data. Kubat et al. [184] proposed a technique known as *one-sided-selection* that carefully selects a representative subset of the majority class by removing noise, redundancy and borderline samples that results in a more balanced training set.

2.11.2 Algorithm Level Approaches

In algorithm-level methods, the learning model is modified so that the bias imposed on the classifier due to the prior probability of imbalanced data is mitigated. This is most commonly achieved by designing a loss function that penalizes the misclassification of minority class samples more than that of the majority class.

Wang et al. [192] showed that the traditional Mean Squared Error (MSE) poorly captures the classification performance when high data imbalance exists. The work introduced a new loss function, Mean False Error (MFE), by combining two components of MSE, i.e., Mean False Positive Error (MFPE) and Mean False Negative Error (MFNE). The method showed improved performance as compared to traditional MSE loss. However, the datasets considered were fairly small and the improvement is highly problem-specific [193].

Lin et al. [194] proposed Focal Loss (FL) function that modifies the traditional cross-entropy (CE) loss function. The work multiplied the CE loss function with a modulating factor. This factor downgrades the easily classified samples to reduce their effect on the classifier. Concurrently, it assigns more importance to the minority class samples to improve classification performance. This method is also transferable to hard sample problems, however, in [195], the traditional CE loss performed better in some datasets suggesting that the performance improvement by this method is specific to particular problem domains.

Wang et al. [196] introduced a cost-sensitive deep learning technique to handle imbalanced data problem. It modified the traditional CE loss function and a pre-defined cost matrix was incorporated with it. The cost matrix forces the loss function to pay more attention to correctly classify minority class samples. The work implemented a cost scheme that specified the false negative error to be $2\times$ the cost of false positive error. However, optimally computing the cost matrix is a very complex and time consuming task that can be impractical for large datasets.

Zhang et al. [197] introduced a deep representation learning-based technique for imbalanced datasets. The Category Center (CC) method proposed in the work incorporates transfer learning, deep feature extraction, and nearest neighbor classification. It first trains a deep neural network (DNN) on a balanced dataset. The output from the penultimate layer of this network is used as a deep feature representation of the

samples which are then used for clustering the samples. The same network is used for extracting deep features from the imbalanced dataset. This is used to compute the distance of the samples from the pre-computed cluster centers and the samples are classified accordingly. The limitation of this work is that it greatly depends on the DNN's capability to extract discriminating features and requires a large balanced representative dataset.

Ding et al. [198] studied the possibility of using very deep neural networks for imbalanced data. The work studied the performance of deeper networks with 16 to 50 hidden layers against shallower 6 to 10 hidden layer networks. The work suggested that additional hidden layers can make it easier to locate local minimum and accelerate convergence. However, adding hidden layers exponentially increases computing and time complexity that results in impractical solutions.

2.11.3 Hybrid Approaches

Hybrid-methods strategically combines data-level and algorithm-level approaches that balance the underlying data distribution and counters classification bias towards the majority class.

LMLE [199] was introduces by Huang et al. to learn discriminative deep representations of imbalanced data. The work introduced a combination of quintuplet sampling and triple-header hinge loss function to produce deep representations of the features which preserve same-class locality and increase discrimination between classes. This deep representation provides well-formed clusters that are used by a K-NN based system for classification. The method showed good performance on the highly imbalanced CelebA dataset [200]. However, the technique proposed in the work is both complex and computationally expensive. Moreover, the method requires pre-clustered data for quintuplet sampling which is difficult to obtain in the absence of suitable feature extractor.

Dong et al. [201] proposed an end-to-end deep learning method for addressing high

class-imbalance. The work introduced a combination of hard-sample mining and class rectification loss and used mean sensitivity scores as the primary performance metric. At the data-level, the proposed method selects samples that are correctly classified with low confidence and misclassified with high confidence. At the algorithm-level, it assigns more weights to the highly imbalanced classes and reduces weights to the less imbalanced ones. The work, however, does not generalize well to different domains especially when the problem consists of low level of class imbalance and fails to provide any performance gain.

Data imbalance in mobile malware detection problem has been acknowledged in various works [202, 203]. However, no explicit attempt has been made to deal with the problem until recently. Xu et al. [16] proposed fuzzy-SMOTE for imbalanced mobile malware detection which is based on the fuzzy set theory. It generates synthetic examples for the minority class in the fuzzy region where the minority examples have a low membership degree. Classification models pay more attention to the enhanced minority class when trained with these synthetic samples. Consequently, the decision boundary of the minority class is enlarged and the classification bias towards the majority class is reduced. The work, however, assumes a continuous feature domain, and also, it does not discriminate between mutable and non-mutable features when generating synthetic samples that leads to invalid sample generation. Changing a non-mutable feature will result in the application losing certain malware functionalities.

Oak et al. [204] proposed a technique that utilizes Bidirectional Encoder Representations from Transformers (BERT) [205] for malware classification with imbalanced data. The proposed method utilizes a monitoring service [206] to record the sequence of various operations performed by an Android application. The extracted information is high-level representations, e.g., *a file is opened on the device*, which are then encoded as integer values. This results in a dataset where each application is represented by a sequence of integers which is used for classification. However, a major

drawback of the method is that it relies heavily on the operation sequence generated by the monitoring service. Generating such sequence requires an application to execute in a controlled environment and its not always possible to exhaustively cover every possible execution sequence of an application. Further, some malicious applications are intelligent enough to detect such environment and suppress its malicious functionalities.

Table 2.6 Summary of key research and their contributions on mobile malware detection with imbalanced data.

Work	**Approach type**	**Contribution**	**Limitations**
Xu et al. [16]	Data level	Adopts a fuzzy set theory-based approach that creates synthetic samples from the malware applications that are in the fuzzy region.	Generates invalid samples.
Oak et al. [204]	Data level	Utilizes a monitoring service that records sequences of operations performed by an application and analyzes them for classification	Greatly depends on the efficacy of the monitoring service.
Songqing Yue [207]	Hybrid	Coverts malware binaries to grey scale image. Utilizes a weighted softmax function for classification using CNN.	Limited by the conversion process which may incur loss of information.
Chen et al. [208]	Algorithm level	Utilizes an imbalanced data gravitation classification model that computes gravitation of a sample using weighted Euclidean distance for classification	Cannot handle high level of imbalance.

Songqing Yue [207] proposed a method where malware binaries are converted into digital grey-scale images. It devised a Convolutional Neural Network (CNN) based classification system that incorporated a weighted softmax loss function for imbalanced malware classification. In contrast to the traditional softmax loss where misclassifi-

cation of each sample is weighted equally, the work scaled the misclassification cost of different classes according to their imbalance ratio. This counters the bias towards the majority class and improves the performance of minority class classification. The performance of the model, however, can be limited by the conversion process that can lead to loss of important characteristics of an application. Also, choosing the appropriate scaling of the loss function is a challenging task in this method.

Chen et al. [208] proposed a method that analyzed network traffic with machine learning models to classify malicious behavior with an imbalanced dataset. The work monitored the network packets of mobile applications and utilized the statistical properties of mobile traffic to identify malicious applications. However, the work cannot handle high level of imbalance and the performance declines when the level of imbalance increases [209].

2.12 Open Research Challenges

Mobile malware detection has become an attractive research area due to the undeniable popularity of smartphones and other hand-held devices, and the increased malware attacks in these devices. Because of the large number of smartphone applications (over five million [210]), automatic malware detection systems based on machine learning models have been widely used. However, several key factors affect the performance, resilience, and robustness of these models, and our extensive review of the current literature reveals the following open research challenges.

- The existing machine learning-based techniques consider random feature categories when designing their model. Not considering the effect of feature categories in a systematic manner can render a classification system ineffective since different categories of features represent different behavioral aspects of mobile applications. For example, many systems consider permission and API calls as feature categories while ignoring the dynamic category. These models are

incapable of identifying obfuscation or dynamic loading-based attacks. On the other hand, a blind combination of these features is also ineffective since adding redundancy can negatively impact model performance. Moreover, additional information about some features (i.e., contextual information of API calls) can provide further assistance in distinguishing different functional aspects of a mobile application. Incorporating such information with the associated features and using them in combination with others will significantly improve a model's performance. A few prior works have explore this idea, but they only considered it in isolation. Thus, a model that comprehensively considers a wide range of behavioral aspects while incorporating contextual information can be of great advantage for effective malware detection, which is lacking in the current literature.

- Newer malware applications are surfacing everyday and specific techniques to evade detection systems are also becoming smarter. Preventive measures need to be enforced against adversarial attacks and successful adversarial samples can effectively render a detection system as useless. The literature proposes proactively retraining the models with self-crafted adversarial samples as a preventive measure. However, this does not provide a full-proof solution since retraining with all or an indiscriminate choice of adversarial samples can lead to performance degradation due to over-fitting. Intelligent methods that can selectively choose higher quality samples are necessary to ensure robustness against such attacks.

- The classification models developed in most prior works assume a balanced data distribution in terms of the number of malicious and benign applications. Section 2.11 showed that imbalanced data is an inherent characteristic of this problem domain. Cross-domain approaches cannot be applied here since malware detection in mobile platforms has unique characteristics, and this is a major drawback of prior works in this area which are very limited. For exam-

ple, over-sampling by feature interpolation for data balancing cannot directly be applied here since the resulting samples often represent applications with broken functionality which makes them invalid samples. To ensure training with valid samples, application characteristics have to be preserved, and, hence, it is necessary to address this issue with domain-specific constraints.

- Building in-device detection mechanism and knowledge transfer are some additional research challenges in this domain. For instance, computing resource and limited power constrains mobile devices from hosting a training environment for a malware detection system. This can be overcome by keeping the bare minimum in the device and acquiring knowledge from a server (e.g., transferring the connection weights of a neural network). Upgrading the system by optimizing the knowledge acquisition process as the model updates to account for newly exposed attacks is another related research problem. However, these issues are related to research on resource-constrained environments and are outside the scope of this thesis.

2.13 Conclusion

This chapter discussed the importance of malware detection in mobile platforms while illustrating their characteristics and usage. Due to the inherent dependency of machine learning-based systems on application features, models could result in sub-optimal performance if different categories of features are used in a randomized fashion. The feature categories need to be carefully investigated since they aid in exposing different aspects of application behavior. Further, these models need to employ defense mechanisms against evasive techniques designed to deceive such detection methods. Also, considering real-world characteristics such as data imbalance, incorporating techniques to handle this will ensure models' practical applicability. In this regard, a detailed review of the literature was presented in this chapter. Reported

works' contribution, advantages, and limitations were discussed and open research challenges were identified. With a view to designing a robust mobile malware detection system, the next chapter addresses the fundamental issue of feature categories, their characteristic, and impact on machine learning systems for mobile malware detection.

Chapter Three

Impact of Feature Categories in Mobile Malware Detection

The previous chapter identified that the appropriate choice of feature categories is one of the most crucial steps in designing an effective mobile malware detection system using machine learning. Section 2.9 introduced four different feature categories and described the related works. Based on the studies of current literature we have found that no work has systematically considered all of these feature categories and thoroughly analyzed their individual and collective impact on mobile malware detection. In this chapter, we conduct a systematic study on this issue and investigate the behaviors of these feature categories. We also identify the combination that gives the best classification performance. Additionally, we formulate a technique to embed the contextual information of API calls into the feature space to facilitate its combined use with other feature categories. The following section describes our detection model and its components. Section 3.2 then undertakes a detailed discussion on the feature categories and application behaviors they capture. Section 3.3 details the contextual information, and Sec. 3.4 presents the experiments and results.

3.1 Detection Model

Figure 3.1 shows our mobile malware detection system which comprises two phases, training and testing. The training phase prepares the classification model for testing, which consists of two components, feature extraction and classifier training. The feature extraction component extracts various categories of features from the training set of malicious and benign samples. The extracted features are used to encode each of the applications into (either benign or malicious) feature vectors which are fed to the classifier for training. This trained classifier is used in the testing phase for classification of unknown applications. The components of our detection system are described below.

3.1.1 Feature Extraction Component

The feature extraction component consists of two parts for extracting static and dynamic features respectively. Each application is analyzed individually by this component and the features extracted from an application are saved in an individual feature file. Thus, each application has a feature file associated with it that lists the only features present in that particular application. Additionally, all the features are added to a global feature set, Ω, which lists all the unique features extracted from all the applications and is later used for feature vector formulation. Several feature extraction tools were introduced in Sec. 2.7, however, the specific tools used in our work along with the feature extraction process are described below.

Static Feature Extraction

We used a reverse engineering tool called Androguard [100] for static feature extraction. Androguard provides a Python library that can be used to analyze an application to extract permissions, API calls, and ICC features. After listing all the API calls in an application, we filtered it through the list provided by Pscout [211]

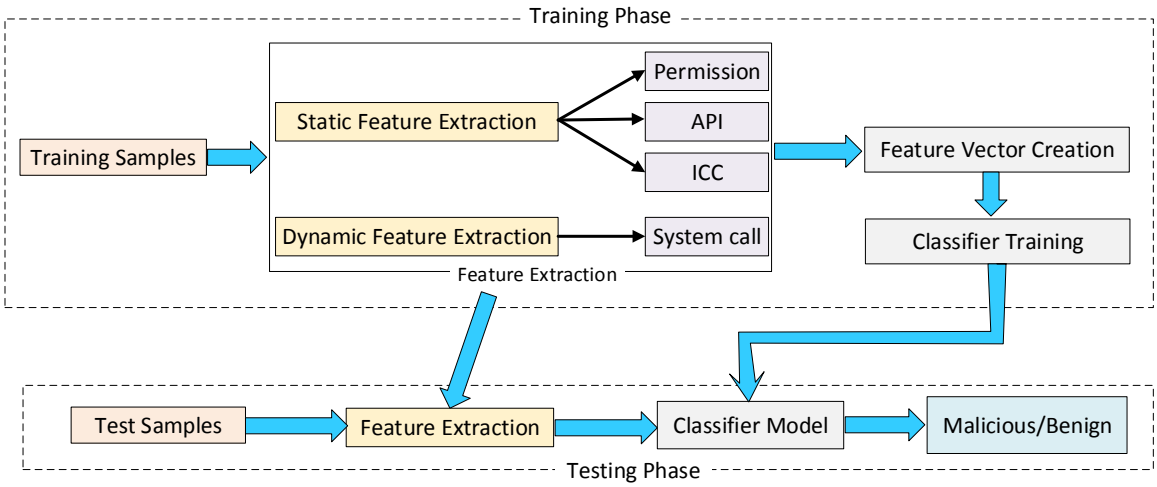


Figure 3.1 Mobile malware detection system

for extracting the security-sensitive API calls. In the process, we extracted 600 permissions, 1000 API calls, and 87000 ICC features. Algorithm 1 describes our static feature extraction process. For each application in the dataset, this algorithm calls Alg. 2 to 4 for extracting permission, ICC, and API call features respectively. Each application is assigned an individual feature file that lists all the extracted features for the application.

Algorithm 1 Static Feature Extraction

Input:
Apps ← application set for feature extraction
Output:
Ω ← global feature set

1: Import Androguard library for static analysis.
2: **for each** Application in *Apps* **do**
3: Create $FeatureFile$ for the application to list the extracted features.
4: Call "AnalyzeAPK()" from Androguard library to obtain the decoded APK object ($DecodedApk$) and the "Analysis" object (Dx).
5: Call Algorithm 2 with $FeatureFile$, $DecodedApk$, and Ω to extract permission features.
6: Call Algorithm 2 with $FeatureFile$, Dx, and Ω to extract API features.
7: Call Algorithm 3 with $FeatureFile$, $DecodedApk$, and Ω to extract ICC features.
8: **end for**

Algorithm 2 Permission Feature Extraction

Input:
$DecodedApk \leftarrow$ the decoded APK file
$FeatureFile \leftarrow$ feature file for the application
Output:
$\Omega \leftarrow$ global feature set

1: Import Androguard library.
2: Call "get_permissions()" function from Androguard to extract the required permissions for $DecodedApk$.
3: List the extracted permission in $FeatureFile$.
4: Add the extracted permission to the global feature set Ω.

Algorithm 3 ICC Feature Extraction

Input:
$DecodedApk \leftarrow$ the decoded APK file
$FeatureFile \leftarrow$ feature file for the application
Output:
$\Omega \leftarrow$ global feature set

1: Call "get_activities()" function on $DecodedApk$ to extract activities.
2: Call "get_services()" function on $DecodedApk$ to extract services.
3: Call "get_receivers()" function on $DecodedApk$ to extract broadcast receivers.
4: Call "get_providers()" function on $DecodedApk$ to extract content providers.
5: Call "get_intent_filter()" function on each activity, service, broadcast receiver, and content provider to extract intent filters.
6: List the extracted features in $FeatureFile$.
7: Add the extracted features to the global feature set Ω.

Algorithm 4 API Feature Extraction

Input:
$Dx \leftarrow$ the "Analysis" object from Androguard
$FeatureFile \leftarrow$ feature file for the application
Output:
$\Omega \leftarrow$ global feature set

1: Call "get_classes()" function on Dx to get all the classes of the application code
2: Call "get_methods()" function get all the method calls for each class
3: Call "is_external()" function for each method to determine if it is an external API call
4: Filter the external API calls with the list of PScout [211] to extract the security sensitive API calls.
5: List the extracted APIs in $FeatureFile$
6: Add the extracted APIs to the global feature set Ω.

Dynamic Feature Extraction

Since extracting dynamic feature requires executing the application, we run each application in an emulator to avoid the risk of an actual device being infected. Genymotion [102] emulator was used for this purpose which provides a free license for personal use. A Python script was written to automate the process of installing and running the application utilizing the Android Debug Bridge (ADB) [212]. Monkey tool [103] was used for generating random events. We injected 1000 events during the execution of each application. The strace [213] utility from Linux was used to attach a tracing process during the application run-time and system calls made by the application were listed in a log file. Afterward, the log files were parsed to extract dynamic features. We extracted 122 system calls as dynamic features from our dataset. Algorithm 5 lists our process of dynamic feature extraction.

Algorithm 5 Dynamic Feature Extraction

Input:
$Apps \leftarrow$ application set for feature extraction
Output:
$\Omega \leftarrow$ global feature set

Run Genymotion emulator
for each *Application* in *Apps* **do**
    Open $FeatureFile$ for the application to list the extracted features.
    Install and run the *Application* on Genymotion using ADB commands.
    Identify the process id (PID) of the running application.
    Create $logFile$ to log the dynamic behaviors.
    Attach strace to the process for logging system call.
    Inject random events using Monkey tool.
    Terminate application execution.
    Write output from strace to $logFile$.
    Pull the $logFile$ out from the emulator memory.
    Parse $logFile$ to extract system call features.
    List the system calls in $FeatureFile$.
    Add the the system calls to the global feature set Ω.
end for

3.1.2 Feature Matrix Formation

After the features were extracted, each of the applications were then encoded as a feature vector. A feature vector is a binary vector whose length is the number of total unique features extracted from all the samples (i.e., the length of the global feature vector Ω). Each component (i.e., index) of this vector corresponds to a feature. For each application, a feature vector is first initialized with 0 in all of its indices. Afterward, the feature file associated with the application was parsed and the value of the indices corresponding to the features listed in the feature file was set to 1. Hence, an entry of 1 in the feature vector means the corresponding feature is present in the application and 0 means it is absent. These feature vectors were compiled as a matrix representing the dataset where each row of the matrix represented a feature vector for an app. Algorithm 6 shows the process of feature matrix creation.

Algorithm 6 Feature Matrix Creation

Input:
$Apps \leftarrow$ application set for feature matrix creation
$\Omega \leftarrow$ global feature set
Output:
$FeatureMat$ ▷ The feature matrix

```
1:  FeatureMat ← Φ                                   ▷ Initialize feature matrix
2:  for each Application in Apps do
3:      Open FeatureFile of the Application
4:      Vec ← allocate memory size = length(Ω) + 1   ▷ Allocate one extra index
                                                       for class label
5:      Vec ← initialize all indices to 0
6:      if Application is malware then               ▷ Set last index to 1 if it's a malware
7:          Vec[size] = 1
8:      end if
9:      for each Feature in FeatureFile do
10:         idx ← index of Feature in Ω
11:         Vec[idx] = 1
12:     end for
13:     Append Vec to FeatureMatrix
14: end for
15: return FeatureMatrix
```

3.2 Feature Category Characteristics

In Sec. 2.9, we briefly introduced different feature categories of an Android application. This section discusses the characteristics of these feature categories in detail. We also discuss the behavioral aspects captured by these feature categories and how they influence a malware classification model.

Permission

Every Android application must acquire the necessary permissions before executing the corresponding functionality. Each Android application needs to declare the required permissions in its manifest file. The user is prompted with these permissions before the installation process begins. Figure 3.2 shows the installation screen of an application that displays the required permissions. There are two types of permission for an Android application: the official Android permissions and developer-defined permissions that are used to protect the components of an application.

Permissions are one of the most commonly used feature categories in mobile malware detection since acquiring necessary permissions is the first barrier a malicious application has to overcome. Figure 3.3 shows the most commonly occurring permissions in malware and benign applications in our database as described in Chapter 2. The figure shows that both malware and benign applications most commonly request permission for accessing the internet, which is understandable since most applications (malware or benign) require internet access to carry out basic functionalities. However, a lot of the malicious applications also ask for permission to send and receive SMSs which is abused for stealing money. Permission to check whether the phone has completed booting is another commonly requested permission by malware applications which aids them to carry out activities secretly in the background.

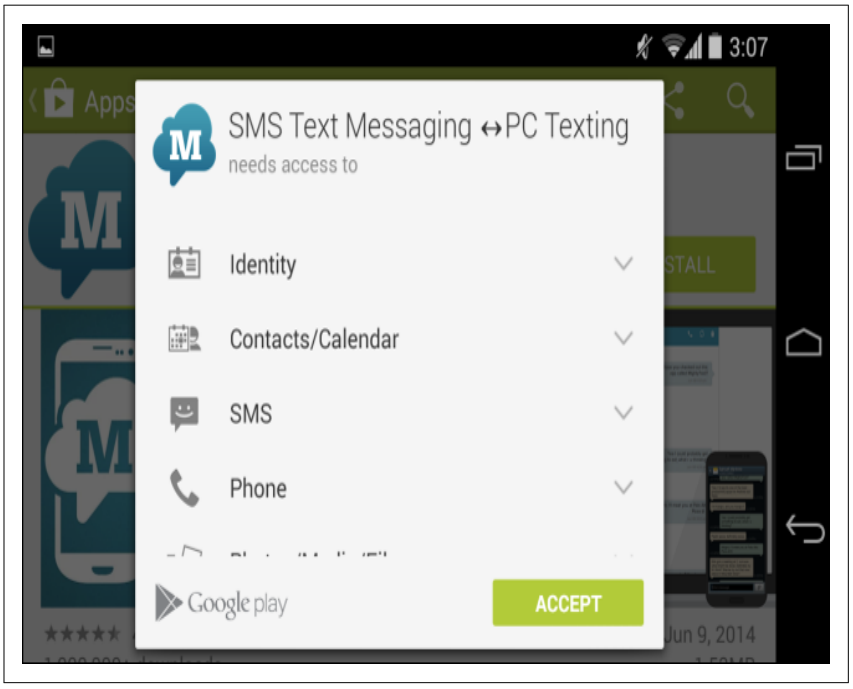


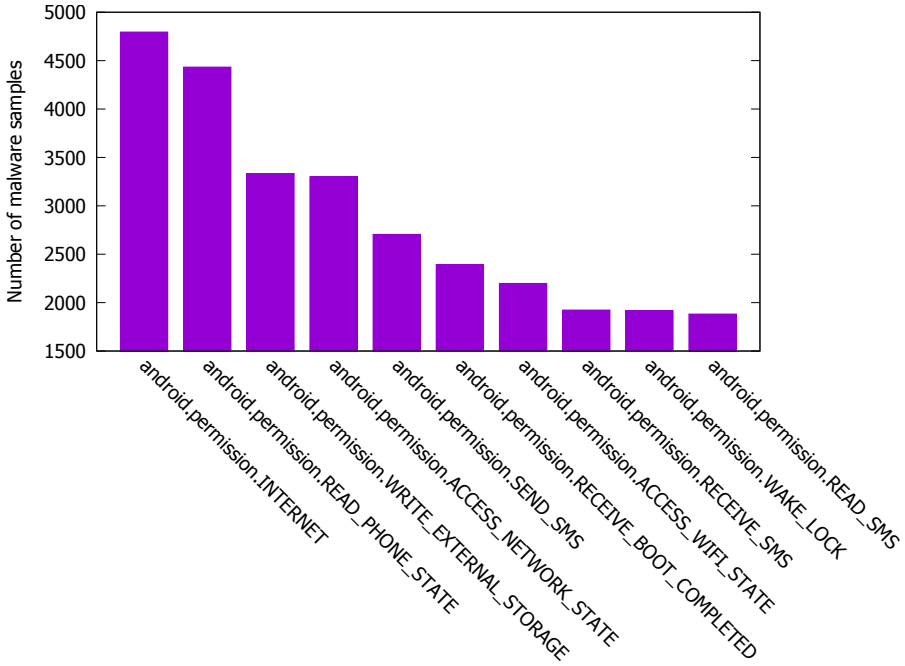
Figure 3.2 Installation screen showing the required permission for an application.

API calls

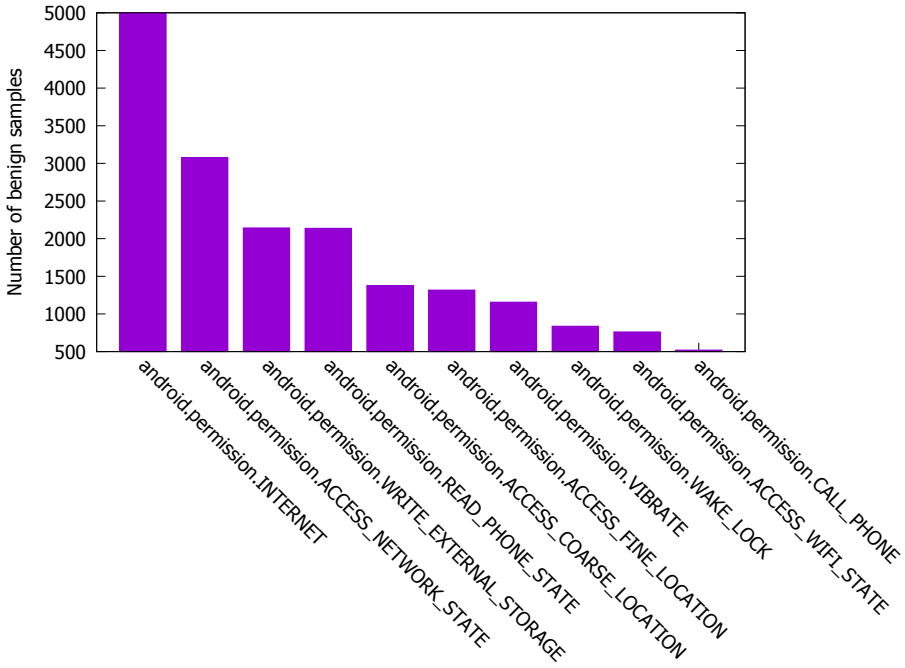
API calls are functions of Android java library that are available to application developers to request services from the operating system (OS) and to perform different operations. For example, to get the location of a device, the application will call the "TelephonyManager.getDeviceID()" function. Figure 3.4 shows the most commonly occurring API function calls in malware and benign applications. The figure shows that most malware applications use the "SmsManager.sendTextMessage()" API function for stealing money. "webkit.WebView()" is another function used by the malware apps to direct users to websites with malicious contents and scams.

ICC features

The Android OS allows the components of an app (describe in Sec. 2.2) to communicate with each other through ICC mechanism. This was made available to programmers to reduce the burden of coding similar components. Figure 3.5 shows

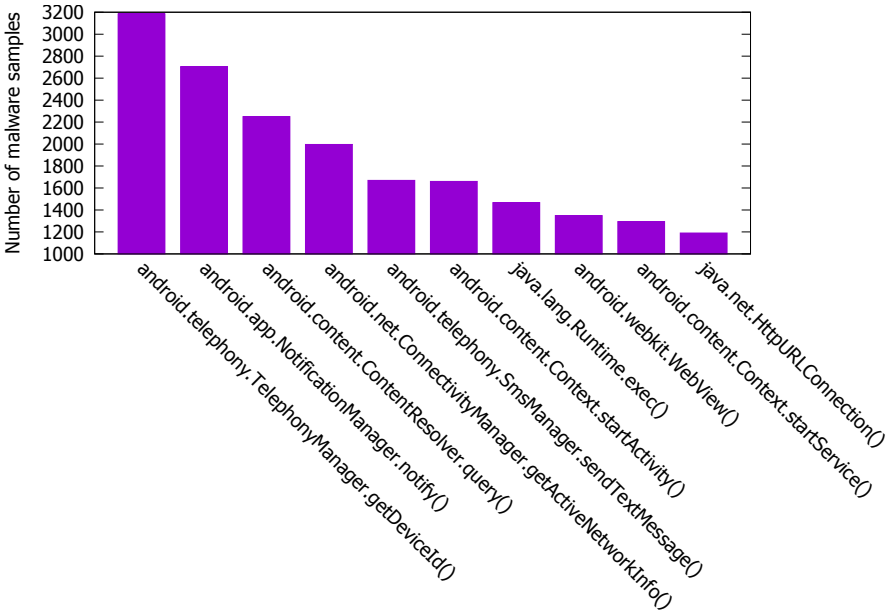


(a) Permission features in malware samples

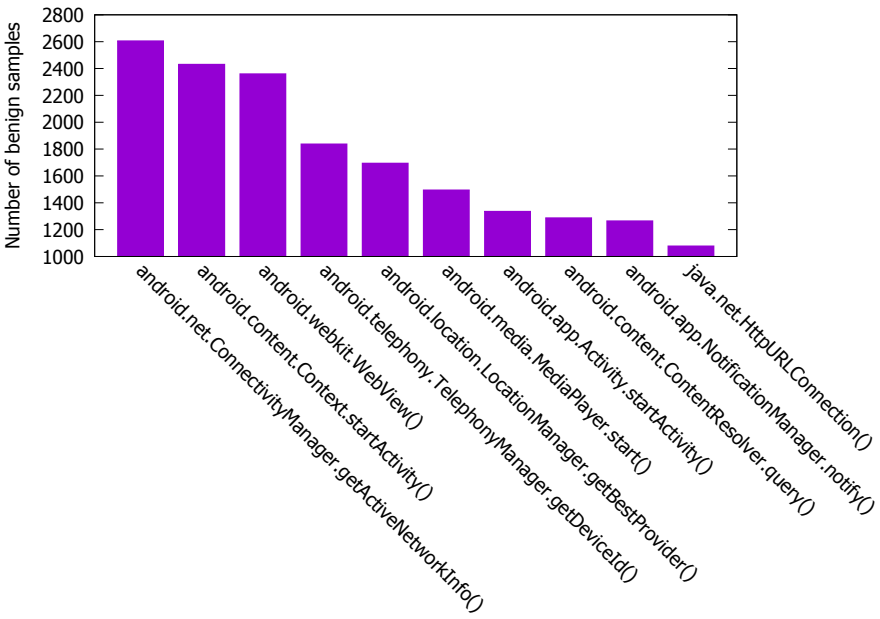


(b) Permission features in benign samples

Figure 3.3 Most commonly occurring permission features in malware and benign applications in our dataset.



(a) API calls in malware samples



(b) API calls in benign samples

Figure 3.4 Most commonly occurring API features in malware and benign applications in our dataset.

an example where ICC mechanism is useful. In this example, the Facebook app is trying to show the location of an address on a map. Using ICC, Facebook can transfer the address to a map application (e.g., Google maps) where the exact location of the address is displayed.

Malware developers can exploit this facility to construct collusion attacks. Figure 3.6 exemplifies one such attack scenario. The application on the left denoted by a red rectangle leaks user information by getting the device location and sending an SMS to the attacker. However, since these two functions are done by the same application, it is relatively easy to detect such a privacy leak attack. Conversely, two apps denoted by the green rectangles on the right side only perform one of the two actions, i.e., one application is only capable of getting the location and the other can only send SMSs. On their own, these two apps seem harmless but they can accomplish the same functionality with a collusion attack by communicating through ICC while bypassing detection.

Figure 3.7 shows the most commonly occurring ICC features in malware and benign applications. It is noteworthy that most of the benign and malicious applications contain “android.intent.action.MAIN" and “andoid.intent.category.LAUNCHER" as ICC features since they are used in basic execution process of an Android application. But we see many malicious applications having “android.intent.action.
BOOT_COMPLETED" and “android.intent.category.HOME" as ICC features that enable them to carry out secret activities in the background.

System calls

The Android system is built on top of Linux kernel and most of its operations are done through Linux system calls. The system calls made by an application can be traced during run-time, thereby dynamically revealing the malicious activities of an application. For example, some malware applications hide their malicious code to bypass detection when the program is launched. At a later period, they dynamically load

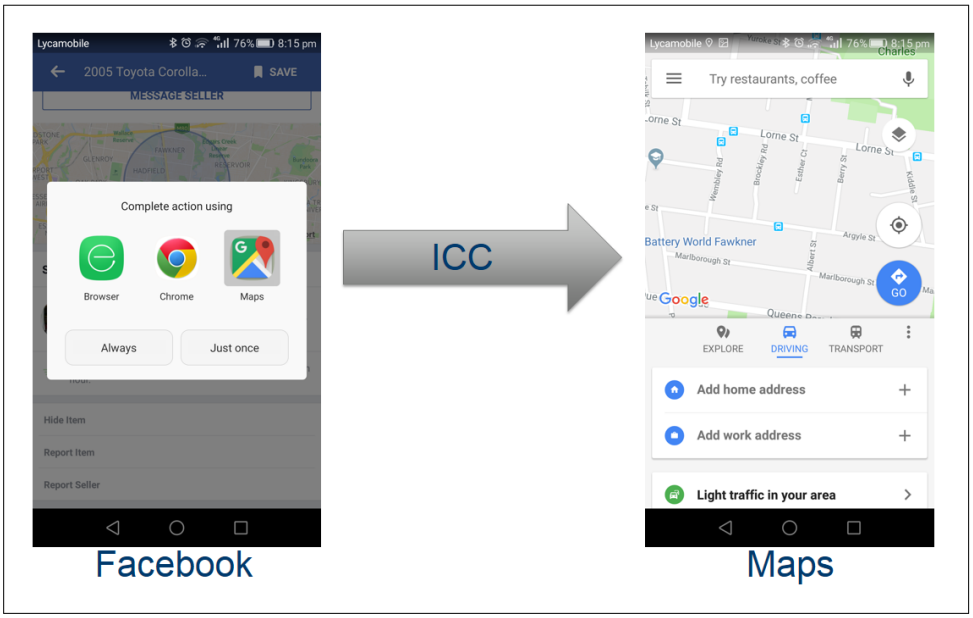


Figure 3.5 Apps communicating through ICC mechanism.

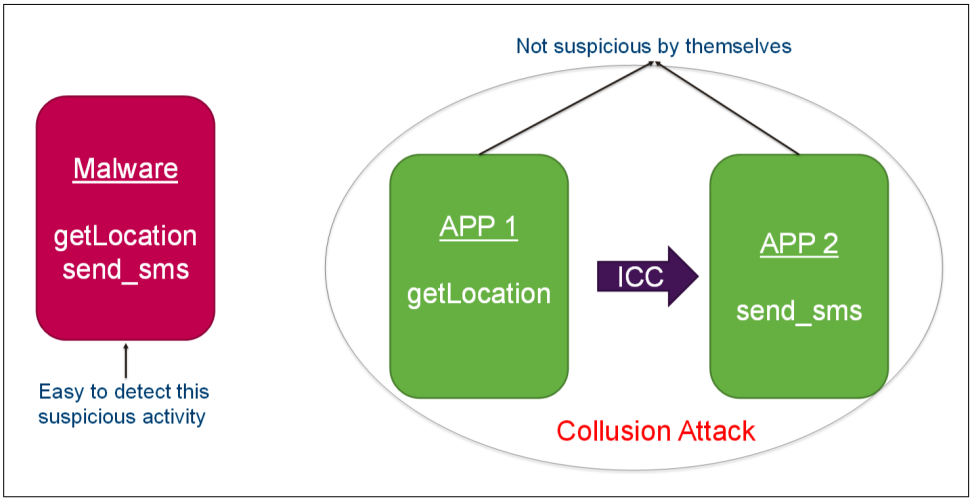
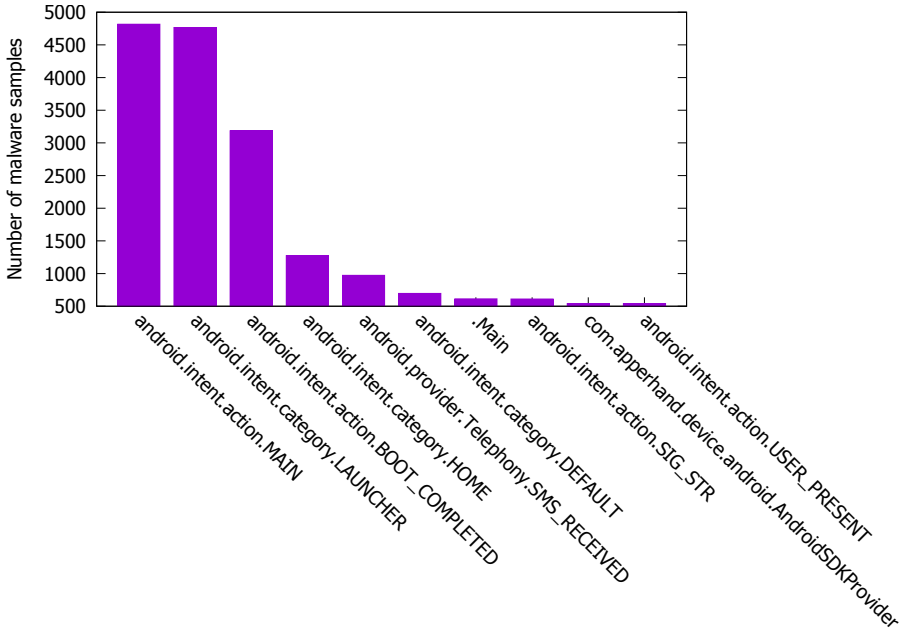
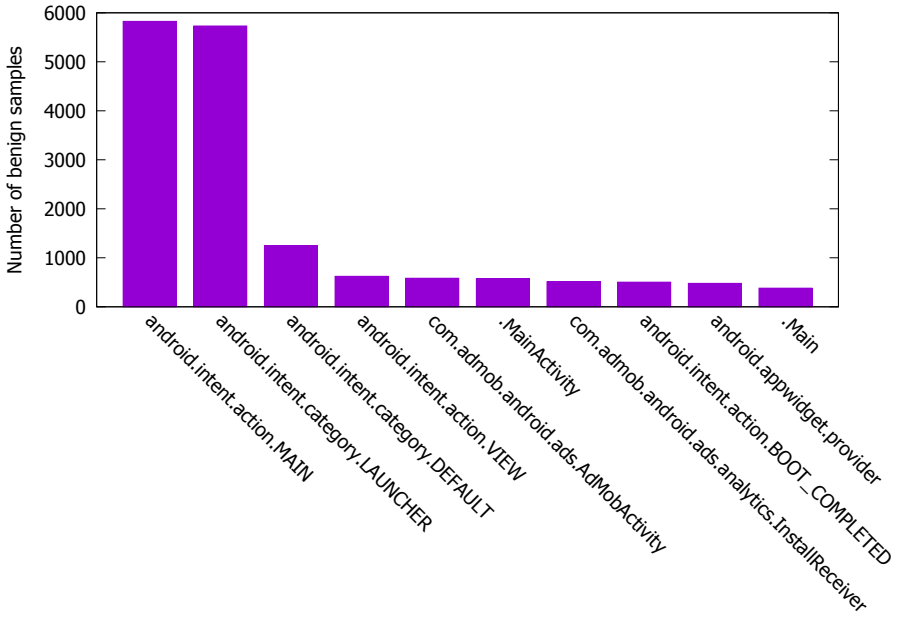


Figure 3.6 Collusion attack using ICC mechanism.

their malicious codes. This dynamic loading goes through a series of actions which includes making an “open()" system call for opening the target code and invoking “mmap()" to load it into the memory. These behaviors can be captured through dynamic analysis of system calls.



(a) Malware samples



(b) Benign samples

Figure 3.7 Most commonly occurring ICC features in malware and benign applications.

3.3 Encoding Contextual Information as Feature Value

The contextual information of Android API calls was briefly introduced in Sec. 2.9.1. This section discusses this in detail, as well as how we incorporate this information into feature vectors.

The execution procedure of the Android system makes the applications heavily dependent on “callback" methods. For example, when an application starts, the “onCreate" callback method of its “main activity" is executed. Or when a user clicks a button, the “onClickListener" callback method is called to carry out the actions associated with the button click. These callback methods are tied to some activation events, i.e., for the above examples, the “onCreate" method is tied to the event of starting the application and the “onClickListener" method is to tied to the event of clicking a button. These can be utilized to obtain additional information about an API call made by an Android application.

To illustrate the above, we take the example of the working principle of MoonSMS malware. A simplified call graph of the application is shown in Fig. 3.8. This malware appears as a benign application that sends greetings messages. In the process, it obtains the “`SEND_SMS`" permission so that it can call the “sendTextMessage()" function. As shown in Fig. 3.8 “SmsManager.sendTextMessage()" is a security-sensitive API call used to send SMSs which can be invoked by the application in several ways. First, the method is invoked when the user clicks the “send" button, which executes the “SendTextActivity.onClick()" callback method that consequently sends an SMS. This is the expected use case highlighted with green arrows in Fig. 3.8. Second, when the signal strength of the device is changed, Android OS generates a corresponding event. The application registers the “ActionReceiver.OnReceiver()" callback method to respond to this event. This callback method checks if the current time is between 11 pm and 5 am when the user is expected to be unaware of this covert action and

proceeds to send an SMS. Third, whenever the application is launched, its “SplashActivity.OnCreate()" callback method is called which checks if the last connect time is at least 12 hours ago. Expecting that one SMS per 12-hour interval will go unnoticed, the malware application invokes the sendTextMessage() function.

In the above example, the first use case is not malicious since the user is aware of the operation. The user can actually see the information on the screen and can modify the action if necessary. From an execution point of view, this is tied to the event of button click by the user. However, the last two scenarios are malicious since they are being carried out unnoticed by the user and used to send premium SMSs to a malicious server to steal money from the user. These are tied with “signal change" event and “application start" event respectively.

By analyzing the contextual information (i.e., the associated activation events), additional information about the usage of an API call can be obtained and, consequently, the behavior of an application more accurately predicted. Extracting contextual information has two components, namely, entry point identification and activation event resolution. For this, we first construct the call graph of an application using the Androguard [100] tool. Subsequently, the nodes in the call graph that contains a security-sensitive API call are identified. These nodes will represent the API features of an application. Thereafter the two components are executed consecutively. These are described bellow.

Entry point identification. To identify the entry points of a security-sensitive API call, the call graph of the application is traversed upward in a Depth First Search (DFS) [214] manner until a root node is reached. This process is carried out in an iterative fashion to identify every entry point from which the corresponding API can be reached. It noteworthy that an application call graph can have multiple root nodes or entry points since there is no main function in the Android application code and the operations are basically initiated through callback methods.

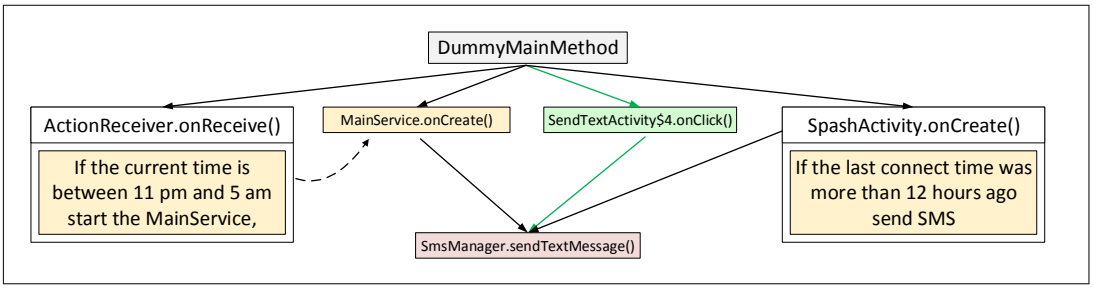


Figure 3.8 Call graph of Moon SMS malware

Activation event resolution. Activation events of an Android application can be of two types, user interface (UI) activation (user aware context) and background activation (user unaware context) [215]. To resolve which type of activation an entry point corresponds to, we check its overridden class. If an entry point overrides any of the three interfaces among "android.view.KeyEvent.Callback", "android.graphics.drawable.Drawable.Callback", and "android.view.accessibility.AccessibilityEventSource", it is resolved to a UI activation events since these are the top-level interfaces related to user interaction [215]. Otherwise, the entry point is resolved to a background event activation.

The example shown in Fig. 3.8 demonstrates that an API call initiated by user interaction is less likely to be a malicious case, whereas, an API call initiated by a background event has a greater potential of being malicious. Representing the API calls simply by a binary value as in [14, 54] cannot capture this contextual information which can lead to classifying both benign and malicious calls to the same class. Our initial experiment confirms that this way of representation produces more false negatives (i.e., misses malware). To encode the contextual information into a numeric value, we used the activation events to calculate a maliciousness indicator score Υ using the following equation

$$\Upsilon = \frac{W_{ui} v_{ui} + W_{be} v_{be}}{W_{ui} + W_{be}}. \tag{3.1}$$

here, W_{ui} and W_{be} are the weights associated with UI callback and background event

callback respectively. v_{ui} and v_{be} are binary values representing whether the API call is initiated by the respective activation event (1) or not(0). This score (Υ) is fed to the sigmoid function to obtain the feature value Γ:

$$\Gamma = sig(k\Upsilon). \tag{3.2}$$

The sigmoid function is defined as:

$$sig(kx) = \frac{1}{1 + \exp(-kx)} \tag{3.3}$$

here, k is the steepness parameter and x=Υ. The parameters in Eq. (3.1) and Eq. (3.2) need to be determined such that the computed feature values show a considerable distinction between the user aware and unaware API calls. This is formulated as the following optimization problem.

$$\underset{W_{ui}, W_{be}}{\text{argmax}} \quad \mathbb{K} \tag{3.4a}$$

$$s.t. \qquad W_{ui}, W_{be}, k = [W_1, W_2, W_3, W_4, \ldots] \tag{3.4b}$$

$$\mathbb{U} = [\Gamma, \text{ for } v_{ui} = 0, 1 \text{ and } v_{be} = 0, 1] \tag{3.4c}$$

$$\Psi = |\mathbb{U}_a - \mathbb{U}_b| \text{ for } a = 1, \ldots, (\|\mathbb{U}\| - 1) \quad b = a + 1, \ldots, \|\mathbb{U}\| \tag{3.4d}$$

$$\mathbb{K} = min(\Psi) \tag{3.4e}$$

where $|\mathbb{U}_a - \mathbb{U}_b|$ denotes the absolute difference between the a^{th} and b^{th} indices of $\mathbb{U}$, and $\|\mathbb{U}\|$ denotes the length of $\mathbb{U}$. We performed a grid search on the parameters W_{ui}, W_{be}, and k varying the range from 0.5 to 5 with an interval of 0.5. The optimal values were found to be 0.5, 4, and 1 for UI event activation (W_{ui}), background event activation (W_{be}), and k respectively.

3.4 Experiments and Results

The primary aim of this chapter and therefore the following experiments was to carefully examine the impact of different feature categories on mobile malware detection systems and their relative importance. Due to the promising performance of DNN (introduced in Sec. 2.5) reported in the literature, we also adopted a DNN based architecture for our first set of experiments. However, afterward we verified our findings with three different classifiers including SVM and Random forest. The first set of experiments was divided into two parts. First, the characteristics of individual feature categories were investigated, and second, all possible combinations of feature categories were taken and their effectiveness analyzed to find the best possible combination. Afterward, the contextual embedding described in Sec. 3.3 was included in the experiment set and its performance was recorded. A relevancy analysis for each feature category was then performed, followed by some additional experiments with the ICC feature category. Each of the experiments was run for 20 trials and the average of these trials is reported in the results below.

3.4.1 Characteristics of Individual Feature Categories and Their Effects

Table 3.1 shows the malware detection performance of a deep neural network using a single feature category. It can be seen that, as a single feature, permission gives the best result, attaining 96.69% accuracy. Since the first requirement an application must have is to get the necessary permissions to perform any sort of operation, "permission", as a single feature category, reveals the general behavior of an application more effectively than any other features. API calls and dynamic features provide comparable performance, but they capture the behavior of an application from different perspectives. API calls are statically extracted and represent the functionalities available in the application code. Dynamic features capture the run-time functionalities

Table 3.1 Malware detection performance with individual feature category. Deep learning model is used for classification.

Metric \ Features	Permission	API Calls	ICC	Dynamic features
TP	2831	2770	1690	2675
TN	5435	5352	5020	5263
FP	142	225	557	314
FN	141	202	1282	297
Accuracy	**96.69**	95.01	78.49	92.85
Precision	**0.952**	0.925	0.752	0.895
Recall	**0.953**	0.932	0.569	0.900
Specificity	**0.975**	0.960	0.900	0.944
G-mean	**0.963**	0.946	0.715	0.922
F1-score	**0.952**	0.928	0.648	0.898

while they are being executed. Most of the security-related operations can be traced back to some API calls which can be mapped to the corresponding system call. For example, to send an SMS, an application has to call "SmsManager.sendTextMessage()" API through the code which, during the execution, will invoke the "sendto()" system call from the OS. As a result, we found similar performances from these two categories. Dynamic features are useful to detect code obfuscation and dynamic code loading which are used by malware applications to bypass static detection systems. However, all-inclusive coverage of every possible execution sequence of an application is not feasible. Hence, aggregation of dynamic features with static analysis is more effective for malware detection. Later experiments further expand on this.

ICC features are found to be the lowest performing in this set of experiments. This is explainable in that the names of the components that communicate with each other are taken as ICC features. If different applications define different names for their components while carrying out basically the same functionality the resulting features

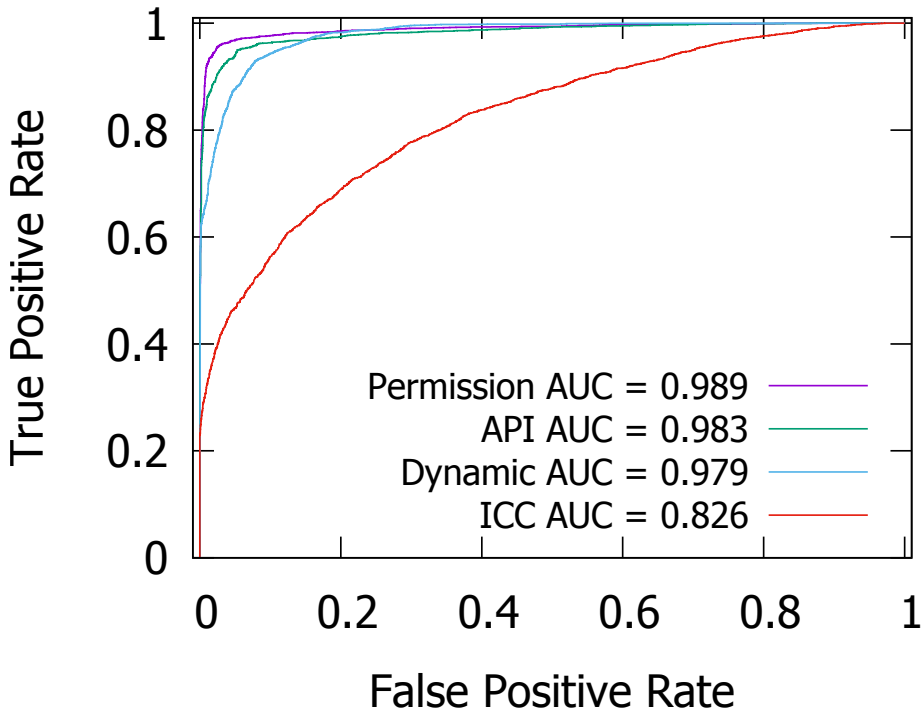


Figure 3.9 ROC curve and AUC for individual feature categories

will still be different. These behaviors will not be captured by the classifier except for the unlikely scenario where every application uses the same name for their components. As a result, a relatively lower performance by the ICC features is observed. Figure 3.9 also demonstrates that the lowest AUC value is obtained by ICC (0.826) while the best is obtained by permission features (0.989).

3.4.2 Malware Detection using Different Combinations of Feature Categories

This section describes malware detection capability with all possible combinations of feature categories. The first part presents the six combinations where only one category is combined with another which helps to analyze which feature categories complement each other well. The second part combined two or more categories resulting in another five combinations.

Table 3.2 Malware detection performance with combination of two feature categories.

Metric \ Features	Permission, API	Permission, ICC	Permission, Dynamic	API, ICC	API, Dynamic	ICC, Dynamic
TP	2910	2752	2833	2900	2840	2743
TN	5319	5431	5454	5363	5492	5343
FP	258	146	123	214	85	234
FN	62	220	139	72	132	229
Accuracy	96.26	95.72	96.94	96.65	**97.46**	94.58
Precision	0.919	0.950	0.958	0.931	**0.971**	0.921
Recall	**0.979**	0.926	0.953	0.976	0.956	0.923
Specificity	0.954	0.974	0.978	0.962	**0.985**	0.958
G-mean	0.966	0.950	0.966	0.969	**0.970**	0.940
F1-score	0.948	0.938	0.956	0.953	**0.963**	0.922

Table 3.2 shows the malware detection performance when only one feature category is combined with another. Drawing from the conclusion of the previous section, we see that dynamic features complement the static features most effectively when combined. The best performance in these experiments was obtained by combining “permission and dynamic features" and “API calls and dynamic features" achieving 96.94% and 97.46% accuracy respectively. Considering other performance metrics, overall, the later combination performed better.

Table 3.3 shows the malware detection performance when more than two feature categories are combined. It is noteworthy that the combination of permission, API calls, and dynamic features gives the best performance here with 97.78% accuracy and 0.968 F1-score. As reflected by the results of “all" feature category combination, further addition of ICC features does not improve the classification performance. Rather, in some cases, it slightly degrades performance, for example, the accuracy and F1-score falls to 97.6% and 0.966 respectively. Figure 3.10 further confirms this as the AUC value falls to 0.994 from 0.996. This is consistent with our findings from Sec. 3.4.1 where we experimented with individual feature categories and found ICC to be the least performing. Also, since the number of ICC features becomes very

Table 3.3 Malware detection performance with combination of more than two feature categories

Metric \ Features	Permission, API, ICC	Permission, API, Dynamic	Permission, ICC, Dynamic	API, ICC, Dynamic	All
TP	2905	2863	2910	2905	2909
TN	5422	5496	5291	5373	5435
FP	155	81	286	204	142
FN	67	109	62	67	63
Accuracy	97.40	**97.78**	95.93	96.83	97.60
Precision	0.949	**0.972**	0.911	0.934	0.953
Recall	0.977	0.963	0.979	0.977	0.979
Specificity	0.972	**0.985**	0.949	0.963	0.975
G-mean	0.975	0.974	0.964	0.970	0.977
F1-score	0.963	**0.968**	0.944	0.955	0.966

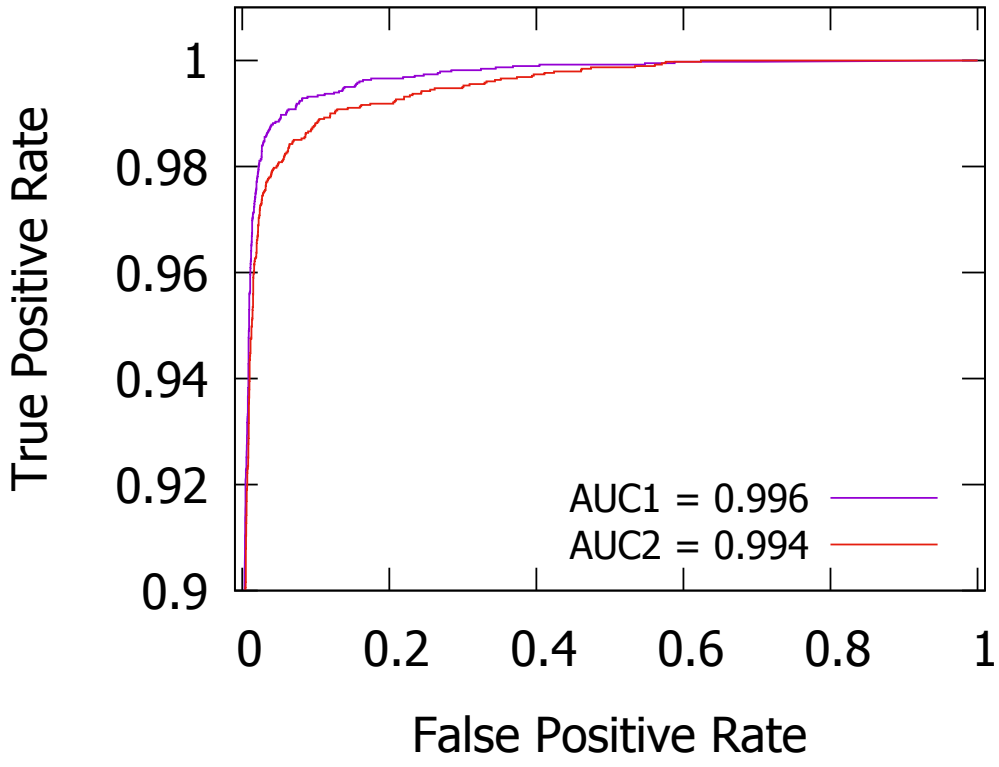


Figure 3.10 ROC curve and AUC for combination of feature categories. AUC1 = Permission, API, and Dynamic, AUC2 = All feature categories.

large (around 87,000) due to different components names, the feature space becomes very sparse. This results in over-fitting due to “The Curse of Dimensionality" [216]. Hence, it is important to make an informed decision, rather than blindly combining the feature categories, when designing a malware detection system.

3.4.3 Effect of Contextual Information

Table 3.4 shows the impact of embedding contextual information of API calls into the feature space. It shows that with this additional contextual information, API call outperforms permissions as a single feature category. This is because API calls can distinguish between malicious and benign behaviors more effectively than permission features when contextual information is provided. For example, sending an SMS by a benign application and malicious application is represented by the same permission, i.e., “android.permission.SEND_SMS" and therefore, is not distinguishable by the permission feature. On the other hand, with the contextual information (i.e., user aware or unaware use cases; described in Sec. 3.3), API features can differentiate between potential malicious and benign usages. Table 3.4 also shows that the combination of permission, API, and dynamic features gives the best result for all the performance measures, attaining 98.21% overall accuracy. Figure 3.11 shows the best AUC of 0.996 is obtained by the combination of permission, API, and dynamic features.

3.4.4 Validating with Other Widely Used Classifiers

This next set of experiments were conducted with various other classifiers to validate whether the relative performance with different combinations of feature categories for these classifiers stays consistent with our findings from DNN-based experiments. The result of these experiments is shown in Fig. 3.12. It can be seen that the actual performance of the individual classifier varies. However, the relative performance for different feature category combinations is consistent with our previous finding – the

Table 3.4 Malware detection performance using different categories of feature and their combination with embedded contextual information in relation to API calls.

Metric \ Features	API	Permission, API, Dynamic	ALL
TP	2867	2899	2876
TN	5502	5497	5496
FP	75	80	81
FN	105	73	96
Accuracy	97.89	**98.21**	97.93
Precision	0.973	**0.973**	0.973
Recall	0.965	**0.975**	0.968
Specificity	0.986	**0.986**	0.985
G-mean	0.976	**0.981**	0.977
F1-score	0.970	**0.974**	0.970

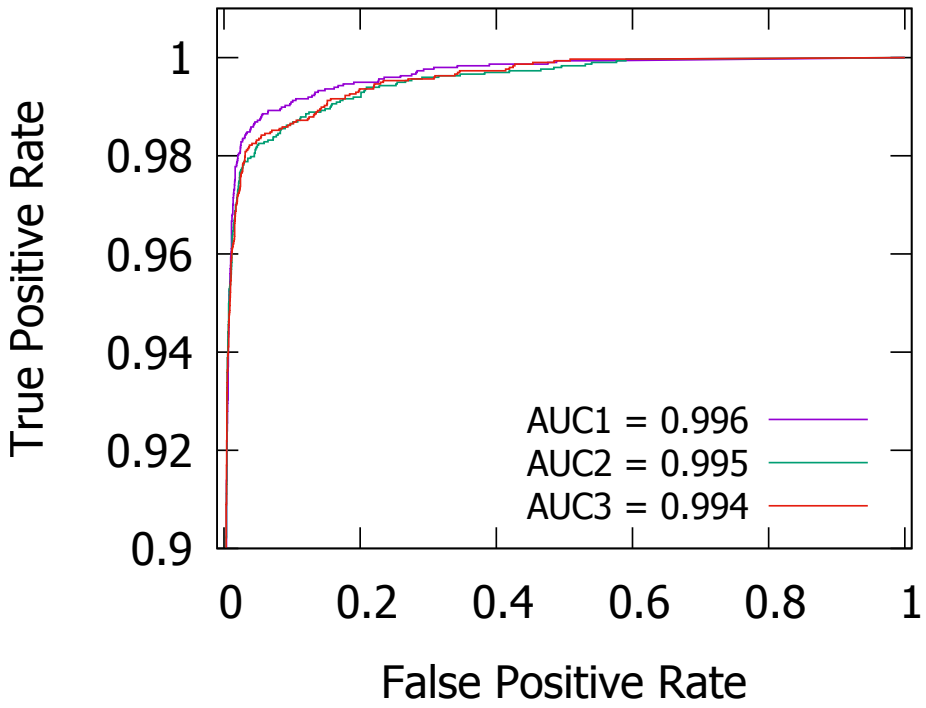
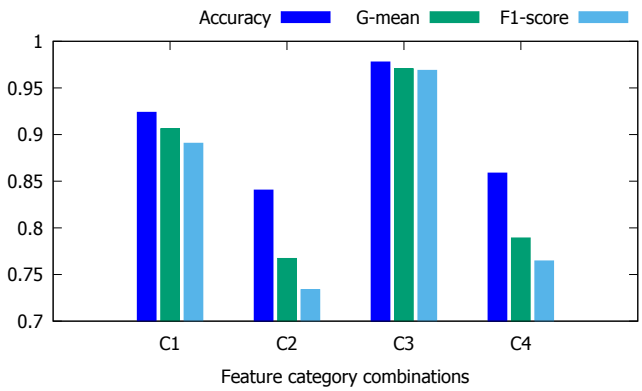
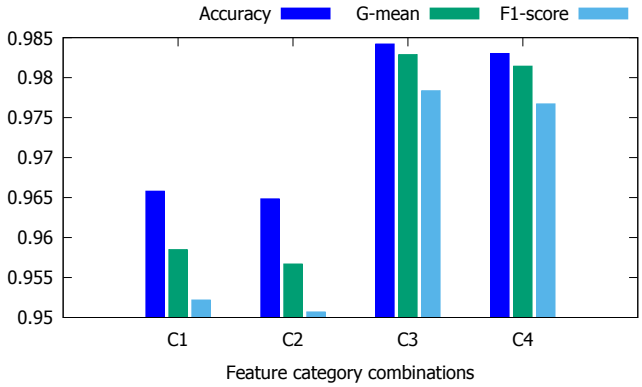


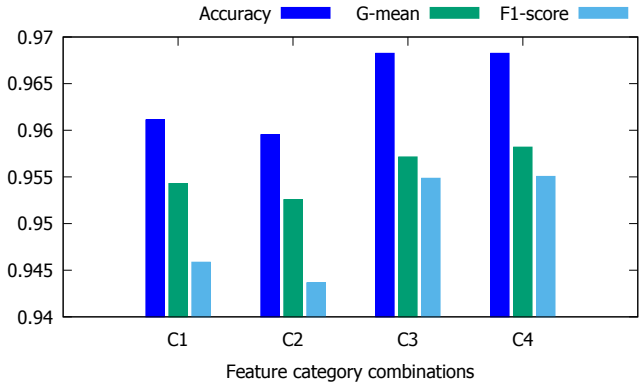
Figure 3.11 ROC curve and AUC with embedded contextual information. AUC1 = Permission, API, and Dynamic, AUC2 = All feature categories, and AUC3 = API features.



(a) SVM



(b) Random Forest



(c) Bayesian

Figure 3.12 Performance of different classifiers with various feature category combinations. Combination - *C1*: Permission, API, Dynamic, *C2*: All, **C3: Permission, API with context, Dynamic** and *C4*: All with context

combination of permission, API, and dynamic feature categories performs better than all other category combination as measured by SVM and random forest (Fig. 3.12a and Fig. 3.12b) and it gives a comparable performance as measured by Bayesian classifier (Fig. 3.12c). This again substantiates that using all feature categories does not necessarily provide any advantage in terms of detection performance.

3.4.5 Relevancy Analysis of Feature Categories

After observing that the ICC feature category has less impact on classifier performance and possibly introduces redundancy when combined with all other categories, we further studied the relevancy of each feature category. We used a relevancy measure based on entropy and information gain for this analysis. The entropy of a random variable X (e.g., a feature) is defined as:

$$E_n(X) = -\sum_i P(x_i) log_2(P(x_i)) \tag{3.5}$$

The entropy of X given another variable Y is:

$$E_n(X|Y) = -\sum_j P(y_j) \sum_i P(x_i|y_i) log_2(P(x_i|y_i)) \tag{3.6}$$

The *information gain* of X given Y can be calculated as [217]:

$$I_g(X|Y) = E_n(X) - E_n(X|Y). \tag{3.7}$$

We computed the information gain of each feature based on the class label and used it as the feature relevancy indicator. If this value was greater than a threshold ϑ, the feature was considered to be relevant. The value of ϑ is predefined manually. Figure 3.13 shows the number of relevant features with respect to the ϑ threshold for each category. Increasing the threshold indicates a more stringent condition for a feature to be selected as relevant which, in other words, means that only features that exhibit

higher influence on prediction outcome are selected at an increased threshold. Figure 3.13 shows that the number of relevant ICC features decreased sharply with increased threshold, while other feature categories showed a much slower decrement. Dynamic category showed the least decrement with increased threshold, suggesting that almost all of the feature set is highly relevant. However, we see that the number of relevant features for the ICC category drastically reduces with a slight increase in threshold value after $\vartheta = 1 \times 10^{-4}$. As a further step, we also inspected the individual ICC features deemed not relevant (i.e., redundant) at the threshold value 1×10^{-4}. At this threshold value, about 60000 ICC features were found to be redundant and, after closer inspection, we found that these features were absent in most of the apps. In fact, none of these features were present in more than 10 apps in the dataset. This suggests that this large number of ICC features do not provide any useful information about malware classification; rather, they introduce redundancy and mislead the classifier leading to decreased classification performance.

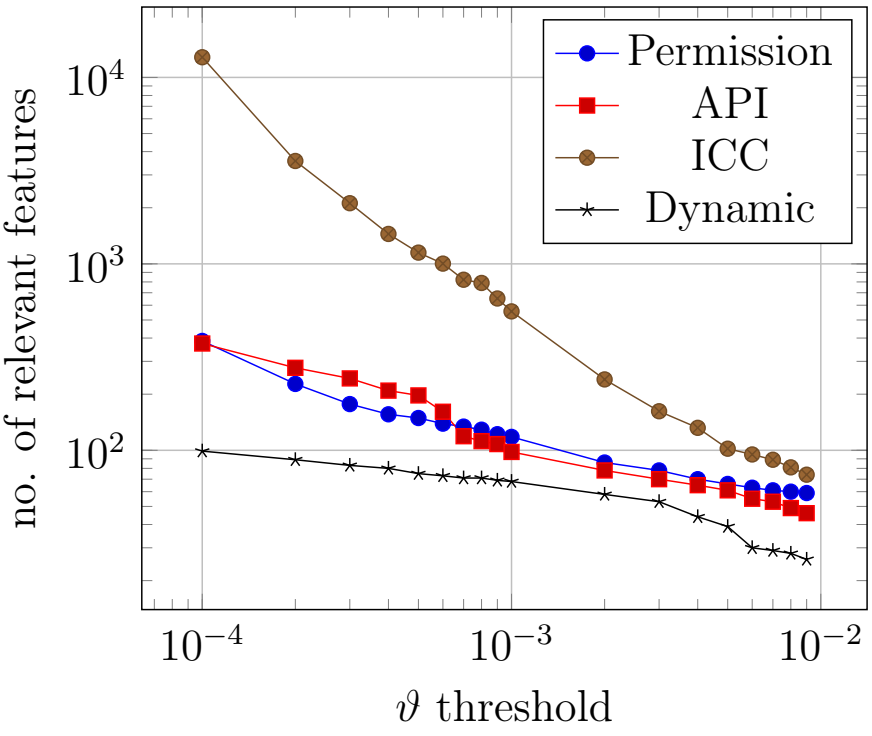


Figure 3.13 Number of relevant features

3.4.6 Further Analysis of ICC Feature Category

Previous experiments showed that the ICC feature category does not have a positive impact on malware classification models. However, this could also indicate that the dataset does not have a sufficient number of applications that perform collusion attacks, i.e., the type of attack ICC features can detect. Hence, to further analyze the efficacy of ICC features we augmented the dataset with malware samples that are specifically generated. For this purpose, we used the ACID tool [218] and generated 832 colluding apps that access one of the sensitive data listed in Table 3.5 and cooperate with another app to leak the information through ICC.

Table 3.5 Information accessed by generated colluding apps.

Function name	Accessed information
IMEI	Accesses the device IMEI number
Microphone	Records phone audio
Contacts	Reads phone contacts
History	Accesses the navigation history
Location	Accesses the device location
WiFi	Gets the list of WiFi SSIDs
Accounts	Reads the device account information

With this augmented dataset we performed additional experiments with the DNN classifier. Table 3.6 shows the results of these experiments. As the table shows, there is not much difference in the performance and only a slight improvement is observed, with the accuracy increasing from 98.39% to 98.66% and the AUC value (in Fig. 3.14) increasing from 0.982 to 0.997. However, this improvement comes with a huge computation overhead as depicted in Fig. 3.15 which shows that when ICC features are combined with other categories the model training time increases by a magnitude of ten. This can create serious issues especially when a faster deployment of the model is required. Also, increased training time hampers model retraining which is required for adaptation and for making the model robust against newer attacks and vulnerabilities.

Table 3.6 Malware detection performance with generated collusion attacks using ICC feature category. Note that the dataset used here is the augmented dataset comprising the previous samples and 832 colluding malware apps.

Metric \ Features	ICC	Permission, API, Dynamic	ALL
TP	2332	3746	3756
TN	5051	5484	5499
FP	526	93	78
FN	1472	58	48
Accuracy	78.70	98.39	98.66
Precision	0.816	0.976	0.980
Recall	0.613	0.985	0.987
Specificity	0.906	0.983	0.986
G-mean	0.745	0.984	0.987
F1-score	0.700	0.980	0.984

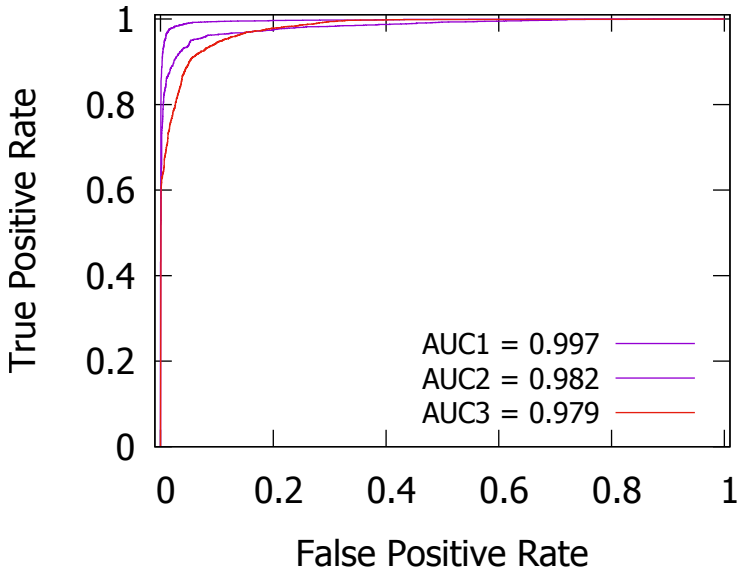


Figure 3.14 ROC curve and AUC with generated attacks using ICC feature category. AUC1 = All feature categories, AUC2 = Permission, API, and Dynamic, and AUC3 = ICC features.

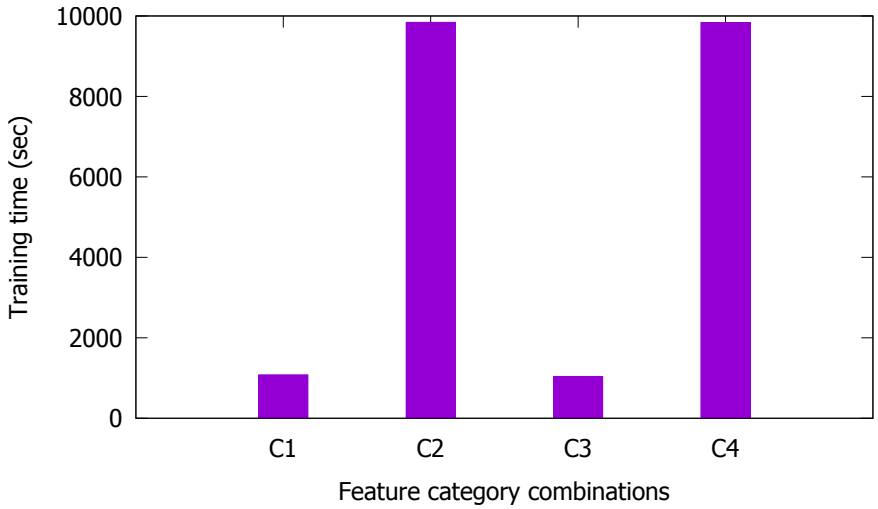


Figure 3.15 Training time for combination *C1*: Permission, API, Dynamic, *C2*: All, **C3: Permission, API with context, Dynamic** and *C4*: All with context

The aim of this study was to investigate the relative influence of different feature categories on mobile malware detection. Considering the performance and execution time of different feature category combinations, we recommend that the permission, API calls, and dynamic features be used for the mobile malware detection systems.

3.5 Conclusion

In this chapter, we analyzed the impact of different feature categories of mobile applications on machine learning-based malware detection systems. Through systematic investigation and extensive experiments, we showed that blind combination of features degrades classifier performance and some feature categories add redundancy to the feature space. In the process, feature categories that complement each other were determined and the combination of permission, API calls, and dynamic feature categories was found to yield the best performance. Additionally, a novel embedding technique for contextual information was devised in this chapter. This allows the contextual information to be embedded in the feature space which facilitates its combined usage with other feature categories. The classifier designed in this chapter provides

the basis for the experiments in the rest of the work. The next chapter discusses an evasive attack (known as adversarial attack) that breaks down the detection systems discussed in this chapter and presents an intelligent selective sampling technique to defend against such attacks.

Chapter Four

Robustness Against Adversarial Malware Attacks

4.1 Introduction

In the previous chapter, we conducted an extensive study on the feature categories for designing a mobile malware detection system and identified their best combination in terms of classification performance which provides the basis for the rest of our works. We recall from Sec. 2.10 that machine learning models are vulnerable to adversarial attacks, which is an evasive technique designed to bypass a detection system. An already detected malware can morph itself with carefully introduced perturbation and render a detection system ineffective. Retraining a model with self-crafted adversarial samples can be used to defend against such attacks. The quality of the retraining samples plays a crucial role in this regard since retraining with too many samples results in performance degradation due to over-fitting. Also, randomly selected sample subset, as suggested by some previous works, can lead to sub-optimal performance since good quality samples can be left out. No prior works in the literature addressed these issues. In this chapter, we design an intelligent technique for evaluating the sample quality and selectively choose the best possible sample set for adversarial retraining to make a malware detection system robust

against adversarial attacks.

4.2 Adversarial Examples

Adversarial attacks are conducted through adversarial samples crafted by introducing a small perturbation into correctly classified samples so that the classifier is misled and classifies them into incorrect classes. Formally, we take a malware detection model M (described in Sec. 2.5) and represent a correctly classified sample X as $M(X) = y$, where y is its original class label. An adversarial sample X' is crafted by introducing perturbation δ_x such that

$$\underset{\delta_x}{\operatorname{argmin}}\, M(X + \delta_x) = \{y' \mid y' \neq y\} \tag{4.1}$$

where y' denotes an incorrect class label and δ_x is the perturbation introduced to the sample X.

Figure 4.1 shows an illustrative example of an adversarial sample. The blue and green stripes represent the value '0' and '1' in the feature vector, respectively. The top part of the figure represents a malicious sample correctly classified by the detection system. The bottom part is an adversarially crafted sample with some of the features modified from the original malware sample. As shown in Fig. 4.1, the features were modified in two of the regions (indicated by two brackets at the bottom) and left unchanged in the other two (indicated by two brackets on the top). The unchanged regions represent the functionalities preserved from the original sample. Due to this careful modification, an adversarial sample can mislead the classifier. As long as the malicious functionalities of the application are preserved, an adversarially crafted sample can function as a malware application and go undetected by the detection system.

adversarial attacks can be divided into several types depending on the attacker's goal and knowledge about the detection system. These are described below.

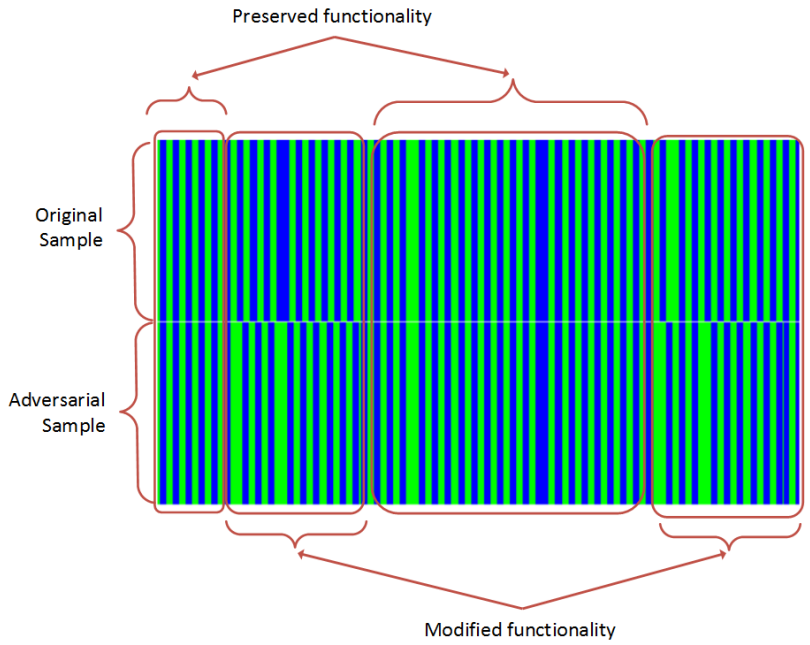


Figure 4.1 Illustrative example of an adversarial sample

Types of Adversarial Attack

Based on the attacker's knowledge about the detection system, adversarial attacks can be of three types:

- *White-box attack* assumes the attacker has a detailed knowledge about the detection system (i.e., knows the model architecture, hyper-parameters, training and testing data, and model weights).
- *Semi black-box attack* assumes the attacker has partial knowledge of the target system. For example, only the training and testing data are known without the internal detail of the model.
- *Back-box attack* assumes that the attacker does not have any knowledge of the target system regarding its data or parameters. The system is only accessible as a standard user, i.e., only the output of the model is available either in the form of class label or probability score.

Based on the attacker's goal, adversarial attacks can be of two types:

- *Targeted attack* specifies the target class of the adversarial sample. This is achieved by setting the condition to $y' = y_t$ in Eq. (4.1) where y_t is the intended target class.
- *Non-targeted attack* only tries to mislead the classifier while minimizing the perturbation. This could be achieved by launching several targeted attacks and taking a successful sample.

In a binary classification scenario (e.g., benign and malware), targeted attacks are equivalent to non-targeted attacks. It is noteworthy from the above discussion that an adversary is strongest in a white-box attack scenario where it has access to all the information about the target model. Most of the approaches in the literature are also white-box in nature [151, 154, 161, 219]. With a view to achieving robustness against the strongest possible attack, we also adopted a white-box attack model in our work. In the following section, the attack model and our proposed defense methods are detailed.

4.3 Methodology

In this section, we describe the methodology of our work. Figure 4.2 shows the workflow of adversarial retraining using our selective samples strategy (i.e., crafting and identifying the samples that, when used in retraining the detection model, will make the model more robust). A classifier trained with clean data (i.e., the dataset without adversarial samples) is used as a target for adversarial attacks. Afterward, the malware samples from the test data are perturbed to craft adversarial samples to evade detection by the classifier. The malicious samples that were correctly detected by classifier but evaded detection after perturbation were considered successful adversarial samples. Afterwards, a subset of adversarial samples was selected and combined with the original samples to retrain the same classifier. For this purpose, we propose two mechanisms to select adversarial samples. Section 4.3.1 describes the adversar-

ial sample crafting process adopted in our work. The detail of our sample selection technique is presented in Sec. 4.3.2, followed by the experiments and results in Sec. 4.4.

4.3.1 Crafting Adversarial Malware Samples

The technique of crafting adversarial samples described in this section can be applied to any differentiable classification function. However, the most common application of this is with DNN [151, 154, 161]. Our work also adopted a DNN to craft the adversarial samples. Note that once adversarial samples are crafted, they can also be used to test against other classification methods.

We begin with a *malware* sample $X \in \{0,1\}^n$ and note the prediction outcomes using the DNN. Using two neurons to represent two classes, the final layer of the trained DNN model M outputs two values $M(X) = [M_0(X), M_1(X)]$, indicating the probability of X being a benign application or a malware respectively. As the prediction, we choose the class that has the highest probability. For crafting an adversarial example, we want to find a small perturbation δ_x so that the output $M(X + \delta_x)$ is different from the original prediction and matches the attacker's goal. This can be done by solving Eq. (4.1) described in Sec. 4.2.

However, the complex functions learned by neural networks are generally non-linear and non-convex. As a result, finding a solution to this problem is difficult. Researchers have proposed various heuristic solutions including forward gradient-based [154] and saliency map-based solutions [219]. The most notable works in this regard are described in Sec. 2.10.2.

In our work, we adopt a saliency map-based adversarial sample crafting technique using the Jacobian matrix since it is more suitable for binary features [219]. The goal here is to craft adversarial malware samples so that they are classified as benign applications. The Jacobian matrix is defined as

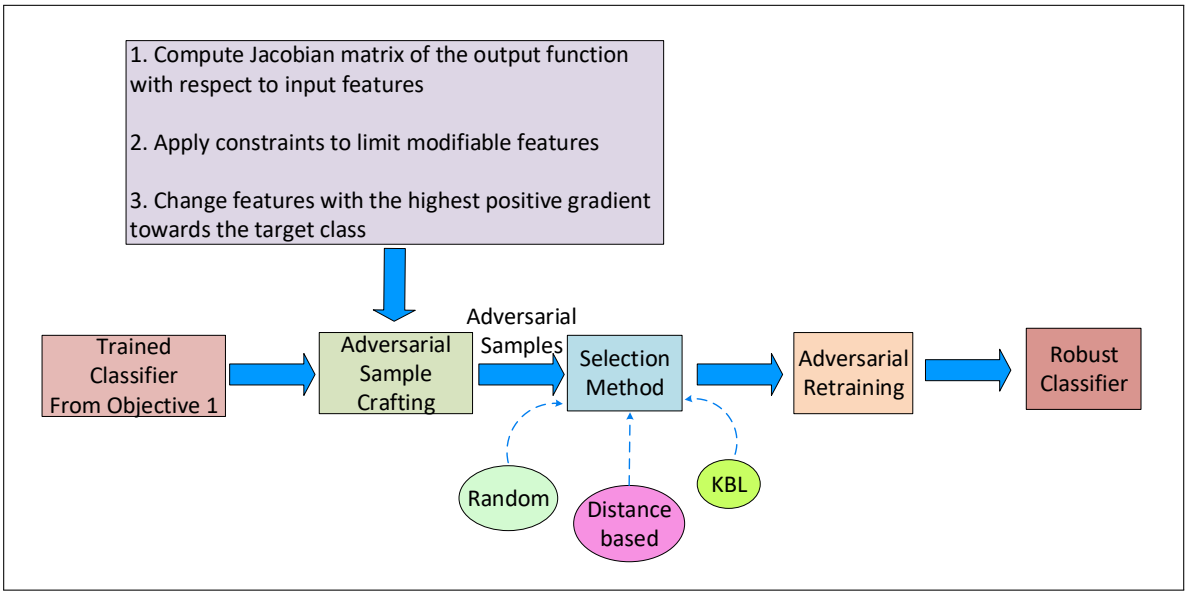


Figure 4.2 Work flow of selective adversarial retraining

$$J_M = \frac{\nabla M(X)}{\nabla X} = \begin{bmatrix} \frac{\nabla M_0(X)}{\nabla X_0} & \cdots & \frac{\nabla M_0(X)}{\nabla X_n} \\ \frac{\nabla M_1(X)}{\nabla X_0} & \cdots & \frac{\nabla M_1(X)}{\nabla X_n} \end{bmatrix} \tag{4.2}$$

here ∇ denotes the gradient of a function. After computing the Jacobian matrix (i.e., the derivatives of the cost function with respect to the input), we modify the feature that is most influential towards our target class. Since our target class is 0 (i.e., to fool the classifier to misclassify the adversarial sample as begin) we find the feature that has the highest value in $\nabla M_0(X)$ for modification. This process is iterated until the malicious application is misclassified as benign.

However, modifying the value of such features is not as straightforward. For example, the method was originally proposed for image processing where several constraints were imposed so that the modification of pixel values does not visually distort the image. An example of such a constraint can be allowing only a small change in pixel value (i.e., keeping the change under a certain threshold). However, unlike image processing, where the values of the image pixels are continuous, we only have 0 and 1

as feature values (except the contextual values which cannot be modified as we will see later). As a result, thresholding for feature values cannot be applied here. Further, an attacker will ensure that the malicious functionality of a malicious application is not hampered while adversarial samples are crafted and hence, arbitrary features cannot be modified. For this, we imposed a different set of constraints for crafting adversarial malware samples. These are described below.

Constraints for Adversarial Malware Crafting

As crafted malware samples need to preserve their malicious functionality, the features that could potentially hamper the maliciousness of an application cannot be modified. Taking the malicious behavior of a broad range of malware into account, we impose the following specific constraints on adversarial malware crafting to preserve its malicious functionality:

- Features are allowed to be added but not removed since removing a feature could potentially break down the corresponding code execution. This means we only allow modification to the features that have value 0.
- We allow the addition of only those features extracted from the AndroidManifest.xml file (discussed in Sec. 2.2) since adding features in this way does not necessitate modification of the original code, hence the original functionality of the code can be preserved.
- Modification of dynamic (i.e., system call) features is not allowed. This is because adding a dynamic feature (e.g., system call) requires modifying the code for executing the corresponding function which, in turn, may hamper the original functionality of the malware.

To implement the above constraints, we first define a vector $\nu \in \{0,1\}^n$, where $\nu_i = 1$ if feature i is allowed to be modified and $\nu_i = 0$ otherwise. After computing the derivatives from Eq. (4.2), we obtain the final score using the following equation

$$S(X) = \nabla M_0(X) \times neg(X) \times \nu \tag{4.3}$$

where $neg(X)$ is a negation function on vector X (i.e., for each index it changes the value from 0 to 1 and vice versa). After this, we choose the index i for modification using the following equation:

$$i = \underset{j}{\operatorname{argmax}}\, S(X_j) \tag{4.4}$$

4.3.2 Proposed Selection Methods for Adversarial Retraining

This section proposes two methods for selecting adversarial samples.

Method 1: Based on Distance from Cluster Center

In this method, adversarial samples are selected by computing the distance of these samples from the malware cluster center. For this, a distance score, $S_d(X)$, is first calculated for each adversarial samples X using the following equation:

$$S_d(X) = \min_{\forall k \in \mathbb{K}} Dist_k(X) \tag{4.5}$$

where $\mathbb{K}$ is the set of malware clusters, and $Dist_k(X)$ is a distance measure between sample X and cluster center k. Note that malware samples may form multiple clusters depending on the types of malware in the dataset. Any clustering algorithm (e.g., K-means clustering) can be used for this purpose. To investigate the impact of distance measure types on the proposed selection strategy, we experimented with various distance measures such as:

$$\text{The Euclidean distance} : \sqrt{\sum_{i=1} (a_i - b_i)^2} \tag{4.6}$$

$$\text{Hamming distance} : \sum_{i=1} (a_i \neq b_i) \tag{4.7}$$

$$\text{L1-norm} : \sum_{i=1} |a_i - b_i| \tag{4.8}$$

Thereafter, the samples were sorted based on the distance score and the first n number of samples with the least score were selected for retraining. Intuitively, these are the samples that should have been correctly classified since they are closer to the cluster center, yet they were able to fool the classifier.

Method 2: Based on Probability Derived from Kernel-based Learning (KBL)

In this method, adversarial samples are selected based on the probability derived from a kernel based learning. We implement this by first training a SVM kernel on the dataset free of any adversarial sample. The decision function $f(x)$ is computed such that $sign(f(x))$ is used to predict the label of a sample. Afterward, in order to compute the class probability $Pr(y = 1|x)$–that is, the probability that a sample is a malware–we consider the following approximation based on Platt [220]:

$$Pr(y = 1|x) \approx P_{A,B}(f) \equiv \frac{1}{1 + exp(Af + B)} \tag{4.9}$$

where $f = f(x)$. Assuming f_i is an estimate of $f(x_i)$, the best value of A, B is determined by minimizing the negative log likelihood of the training data:

$$\min_{A,B} \quad - \left[\sum_i t_i log(p_i) + (1 - t_i) log(1 - p_i) \right] \tag{4.10}$$

where $p_i = P_{A,B}(f_i)$ (Eq. (4.9)) and t_i's are the target label for the optimization problem defined as:

$$t_i = \begin{cases} \frac{N_+ + 1}{N_+ + 2}, & \text{if } y_i = +1. \\ \frac{1}{N_- + 2}, & \text{if } y_i = -1. \end{cases} \tag{4.11}$$

where N_+ and N_- are the number of positive and negative samples respectively. Using Eq. (4.9) - (4.11), the probability of adversarial samples to be a malware is computed. We chose the first n adversarial samples that have the least probability of being a malware. Showing a lower probability while maintaining the malware functionalities means these samples are more likely to fool a classifier.

4.4 Experiments and Results

The following section describes the experimental results. The first set of experiments were conducted with DNN followed by three other classifiers used to verify performance improvements.

4.4.1 Resilience by Retraining with Deep Neural Network

In this section, we present the performance evaluation of adversarial retraining with DNNs when the samples are chosen using our proposed selection strategy and compared with the prior works in the literature where retraining samples were randomly selected. We first trained a DNN on the clean dataset with thee hidden layers consisting of 2000, 1000, and 500 neurons respectively. The network achieved 98.3% accuracy. Applying the method and satisfying the constraints outlined in Sec. 4.3.1, 293 adversarial samples were generated by modifying the malware samples used in the test dataset. The classification accuracy dropped to 71% when tested on the adversarial data. Thereafter, a certain number (n) of adversarial samples were selected for retraining the classifier either randomly or using our proposed strategies.

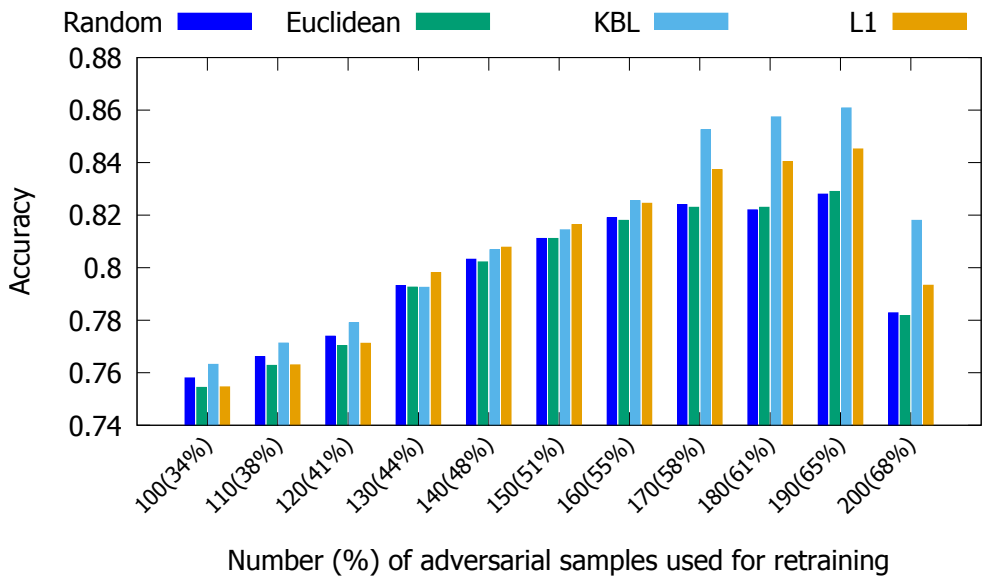
For our first proposed method, based on the distance from the malware cluster center, we considered the malware samples as a single cluster for implementation simplicity.

However, it can easily be extended for multiple clusters using Eq. (4.5) and for any clustering algorithm. The rest of the adversarial samples were combined with the clean test data for evaluation. Once the adversarial samples were selected, the classifier was retrained using a new training set consisting of original training samples and the selected adversarial samples as additional malicious samples. We incremented the value of n from 100 with an interval of 10. Each of these experiments was run for 20 trials varying random initialization of DNN weights and biases, and the average of these trials is reported in the results presented below.

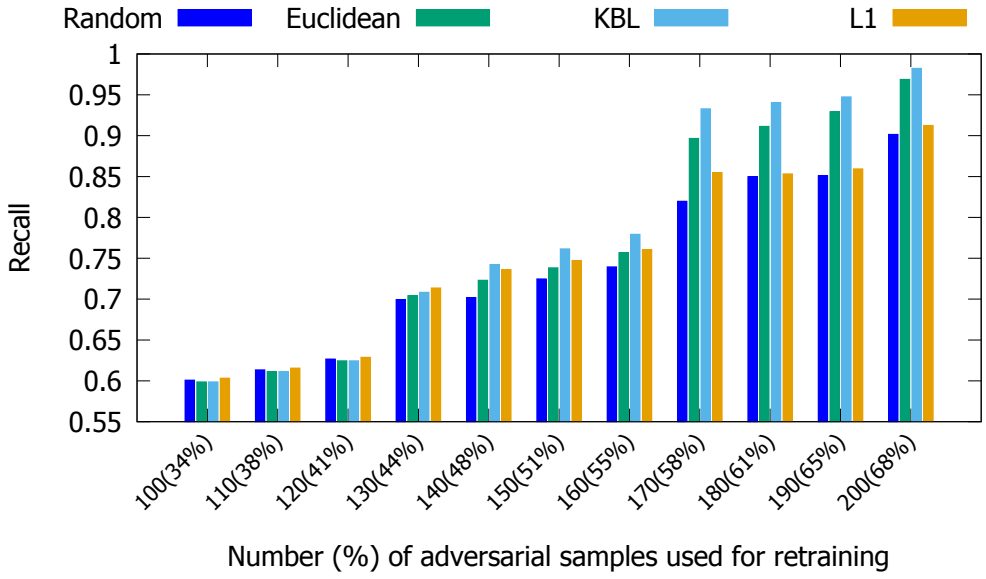
Figure 4.3 shows the comparison of accuracy and recall values between random selection [161] and our selection based on the Euclidean distance, L1 norm or KBL after a DNN is retrained with adversarial samples. Results show that our proposed selection methods outperform the random selection method. It is also noteworthy that the KBL approach performs better than other selective strategies.

We postulate that, the probability function f in Eq. (4.9) used in the KBL method essentially reflects a distance from the hyper-plane, separating the samples in the transformed kernel space. Arguably, the better separation between samples, as learned by the kernel for the transformed feature space, also captures better information from the samples, leading to better performance.

A comparison of random selection with our other distance-based selection methods showed only a few cases of random selection performing slightly better. This occurred mostly in cases where the number of samples for retraining is small (e.g., 100 and 110 samples) where the probability of randomly selecting better samples is higher. However, the performance difference is not significant in these cases and this effect diminishes as more samples are chosen for retraining. Figure 4.3 shows the results of KBL using the *RBF* kernel. We also explored the possibility of using a *linear* kernel, however, it did not result in any performance gain. It has already been shown in the literature that the linear kernel is a degenerate case of RBF kernel and, hence, a



(a) Accuracy



(b) Recall

Figure 4.3 a) Accuracy and b) recall values of deep neural network after adversarial retraining when adversarial samples are selected randomly or selectively based on the Euclidean distance, L1 norm or KBL

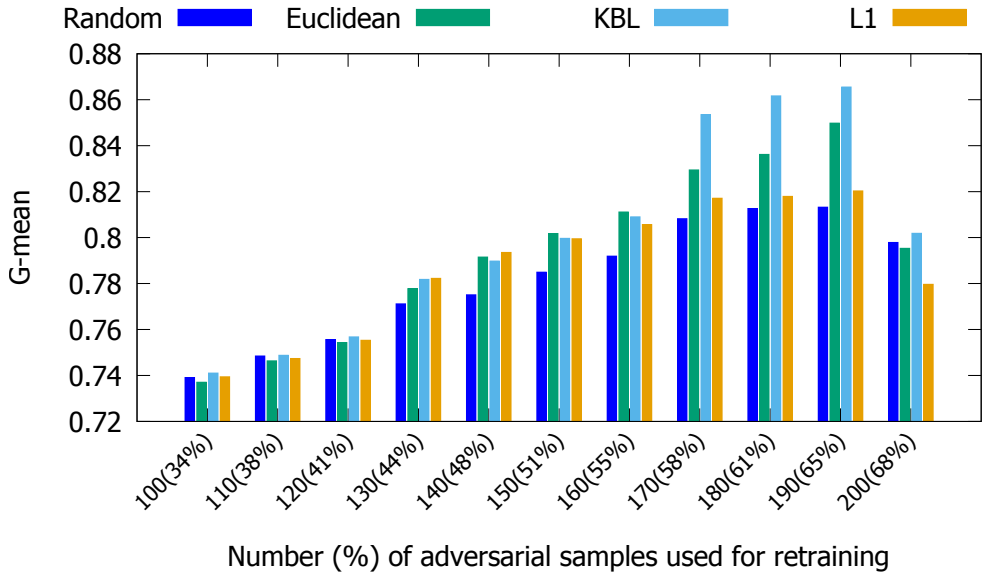
properly tuned RBF kernel gives similar or better performance in most cases [221].

Figure 4.3a further shows that a selection of 58–65% adversarial samples for retraining achieves better performance for all methods. This is especially evident when compared to random selection; KBL achieves a 6% accuracy improvement when trained with 65% of adversarial samples. Figure 4.3b also shows that KBL attains better recall value, which is more prominent when more than 50% of adversarial samples are selected for retraining. Higher recall value indicates the misclassification of fewer malicious samples which is a desired property since the cost of misclassifying a malicious application is significantly higher than that of misclassifying a benign application.

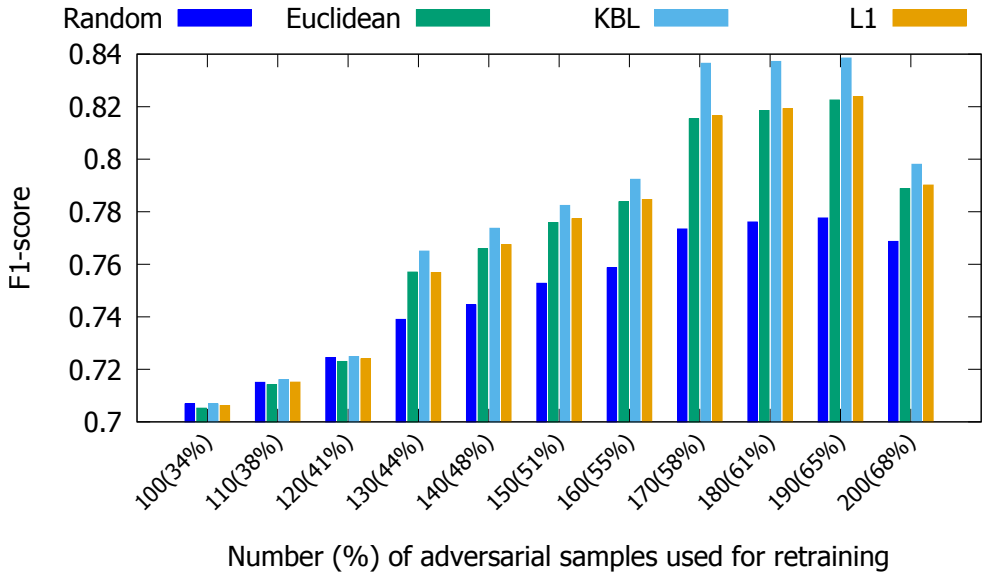
We further recorded G-mean and F1-score that represent a balance between the false positives and false negatives and the results are shown in Fig. 4.4. The figure shows that in most of the cases the selective sampling, especially KBL, performs better than the random selection strategy, and the highest value for G-mean (Fig. 4.4a) and F1-score (Fig. 4.4b) were obtained when the DNN model was retrained with 65% of the adversarial samples.

However, when more than 65% of the adversarial samples are selected for retraining, the performance degrades with the increasing number of samples. If too many adversarial samples are used for retraining, the classifier starts to over-fit towards malware and starts misclassifying benign samples at a higher rate. This can further be observed in Fig. 4.3b which shows that the recall value keeps increasing as more adversarial samples are used for retraining, indicating that more malicious applications are being correctly classified (i.e., the number of false negative is decreasing). This occurs at the expense of increased misclassification of benign samples when there are too many adversarial samples, such as above 65%. As a result, a gradual decrease of precision values was also observed as more adversarial samples were used for retraining.

A Wilcoxon Signed-Rank Test [21] comparing random selection and KBL based selective methods yielded a p-value of 4.4×10^{-4}, indicating that the performance



(a) G-mean



(b) F1-score

Figure 4.4 a) G-mean and b) F1-score of DNN after adversarial retraining with random and selective samples

Table 4.1 Accuracy and G-mean of adversarial retraining with different deep neural network architectures using random, the Euclidean distance based, L1 norm based and KBL selection. 65% adversarial samples were used for retraining.

# of hidden layers (# of neurons)	Accuracy (%)				G-mean			
	Random	Euclidean	L1	KBL	Random	Euclidean	L1	KBL
2([500, 500])	80.27	83.83	82.45	82.55	0.781	0.8218	0.8056	0.809
2([1000, 500])	80.27	83.83	83.73	85.25	0.781	0.8229	0.8171	0.8535
3([1000, 1000, 500])	79.85	83.61	84.04	85.74	0.7777	0.8211	0.8179	0.8616
3([2000, 1000, 500])	**80.69**	**84.39**	**84.52**	**86.08**	**0.7843**	**0.8262**	**0.8202**	**0.8655**
4([2000, 1000, 500, 500])	80.17	83.24	79.33	81.8	0.7802	0.8148	0.7796	0.8018
4([2000, 1000, 750, 500])	78.40	79.65	79.65	80.58	0.7763	0.7898	0.7898	0.7997

improvement of KBL, as compared to random selection, is statistically significant.

We further evaluated the impact of network architecture on the performance of adversarial retraining. Table 4.1 shows the accuracy and G-mean values of different adversarial retraining methods with varying network architecture. The table shows a slight variation in terms of performance depending on the number of layers and neurons in each layer of the network. Since networks with less than two hidden layers are considered shallow, we varied the number of hidden layers from two to four with a different number of neurons in each layer. Results show that our proposed selection mechanisms yield better performance than blind random selection, such as 80.69% and 0.7843 (random) vs 86.08% and 0.8655 (KBL) in terms of accuracy and G-mean respectively.

A plot of ROC curve and the AUC using the selection methods is shown in Fig. 4.5. The figure shows our selection methods achieve better AUC than random selection, and the best AUC of 0.958 is achieved by the KBL method vs 0.884 in random selection.

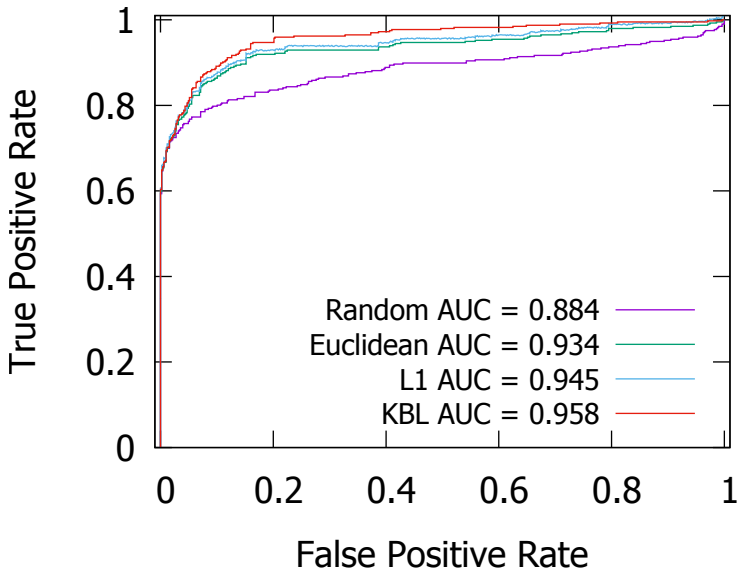


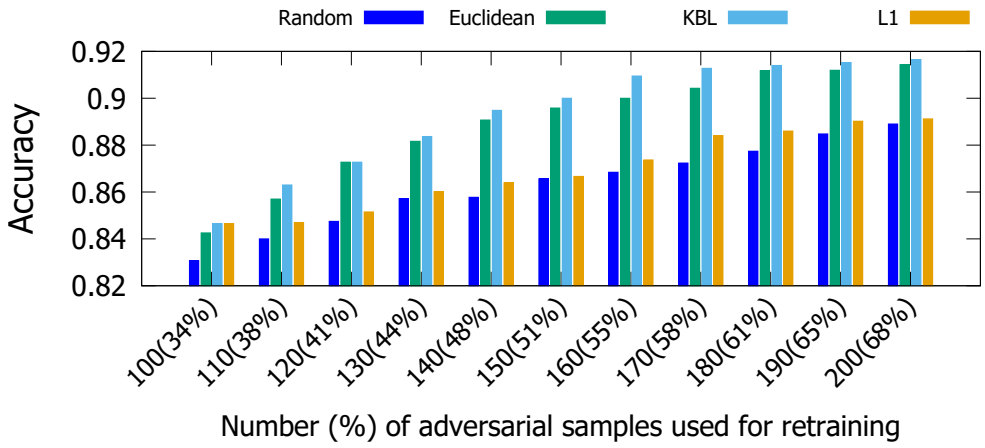
Figure 4.5 ROC curve and AUC for adversarial retraining using random, the Euclidean distance-based and KBL-based selection

4.4.2 Resilience by retraining with Other Classifiers

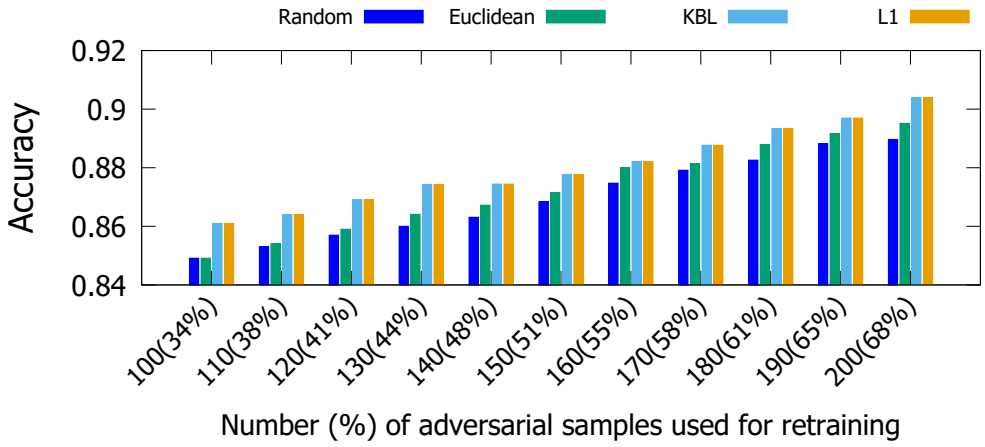
We further evaluated the selective strategies of adversarial retraining with three other widely used classifiers, namely, Random forest, SVM and Bayesian classifier. We first experimented with how effectively the adversarial samples crafted using DNNs can mislead those classifiers. For this, we first trained a classifier with original samples and tested it with clean test samples ensure its performance was satisfactory. Afterward, we mixed adversarial samples with the clean samples and evaluated the classifier's performance. Table 4.2 shows the accuracy of different classifiers when tested with clean samples and samples mixed with adversarial ones. The table shows a significant drop in classifiers' performance when tested with adversarial samples. This suggests that adversarial samples crafted using DNNs are similarly effective in misleading other classifiers as well.

We subsequently retrained these classifiers with the adversarial samples crafted using DNN and compared the efficacy of our selective methods to that of random selection. Figure 4.6 shows the comparison of malware detection accuracy after adversarial retraining with the classifiers as measured by a) SVM, b) Random forest, and c) Bayesian. It is observed that the KBL- and the Euclidean distance-based selection methods clearly outperform the random selection method for adversarial retraining in all the classifiers while L1 norm-based selection performs better than random selection for SVM and Random forest classifier. Calculating G-mean and F1-score also showed improved performance by our KBL method. For example, when using the SVM classifier at 65% samples, G-mean and F1-scores are 0.936 and 0.942 respectively for KBL selection, and 0.885 and 0.902 for random selection. The G-mean and F1-score values are shown in Figures 4.7 and 4.8 respectively.

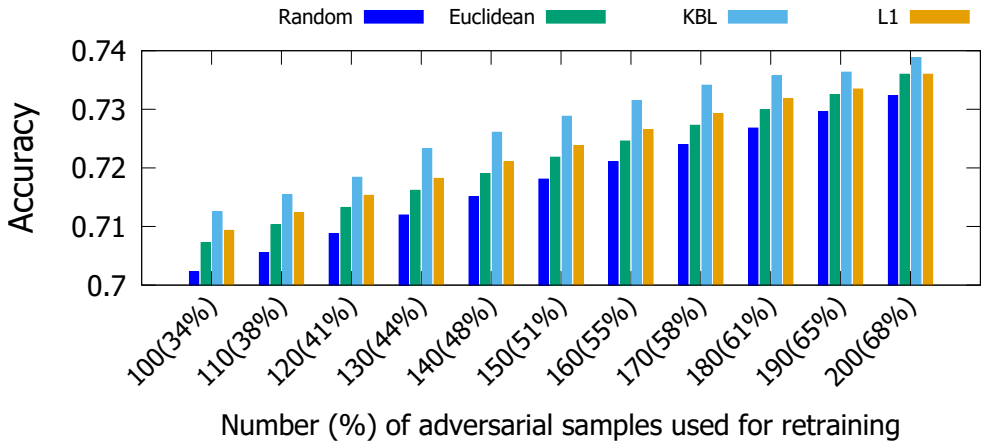
For these classifiers, a Wilcoxon Signed-Rank Test comparing KBL with random selection method yielded a p-value of $< 3.8 \times 10^{-4}$, validating the performance improvement as being statistically significant.



(a) SVM

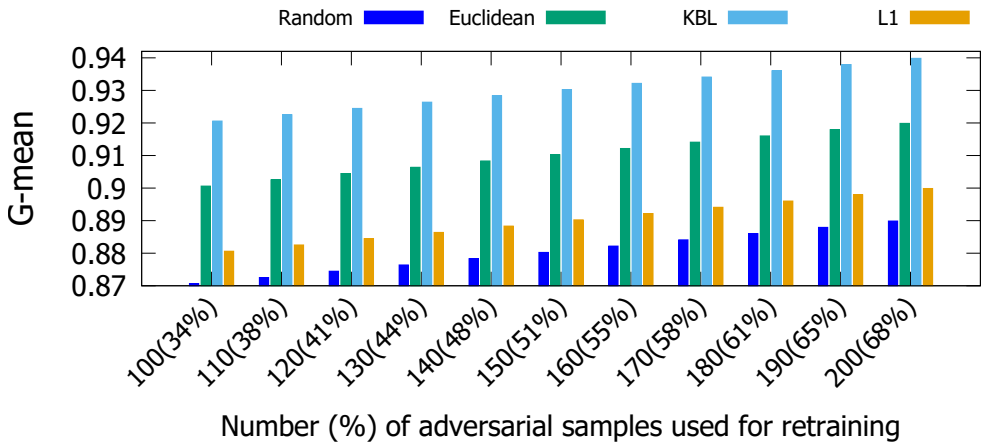


(b) Random Forest

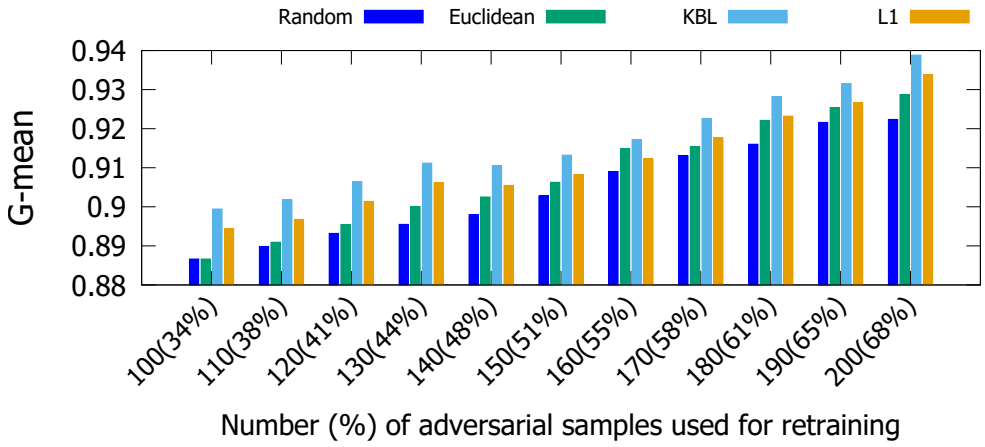


(c) Bayesian

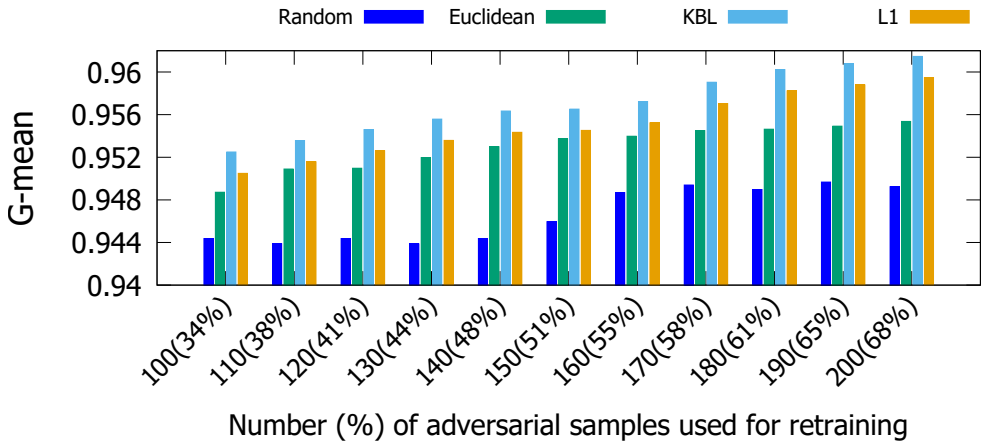
Figure 4.6 Malware detection accuracy of adversarial training for a) SVM b) Random forest and c) Bayesian classifier.



(a) SVM

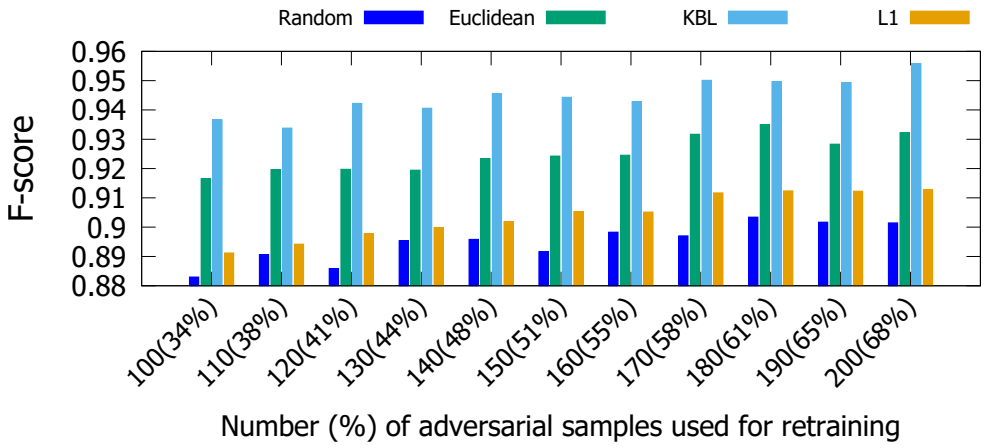


(b) Random Forest

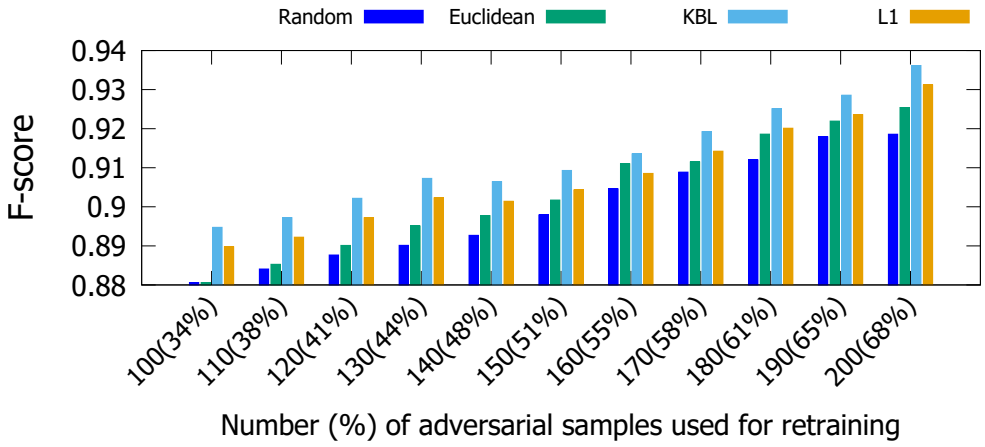


(c) Bayesian

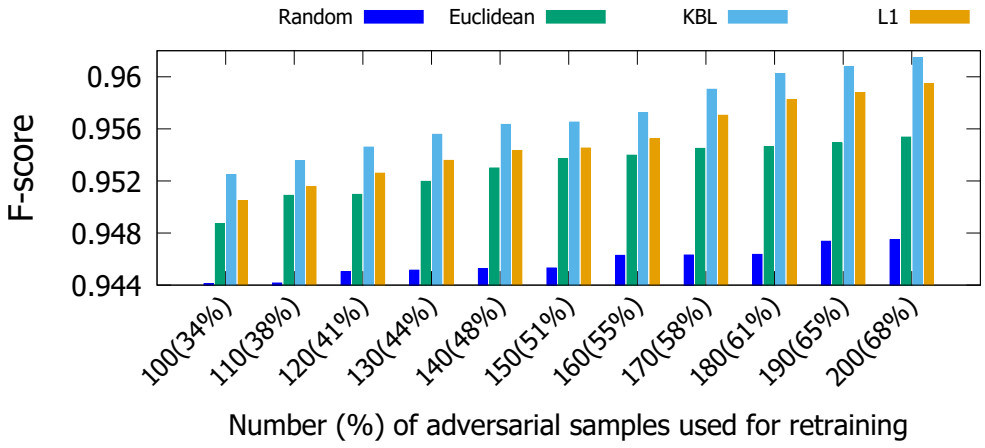
Figure 4.7 G-mean value of adversarial training for a) SVM b) Random forest and c) Bayesian classifier.



(a) SVM



(b) Random Forest



(c) Bayesian

Figure 4.8 F1-score value of adversarial training for a) SVM b) Random forest and c) Bayesian classifier.

Table 4.2 Accuracy of different classifiers when tested on clean samples and samples mixed with adversarial samples.

Classifier	Accuracy (%)	
	Clean Samples	Mixed with adveresarial samples
SVM	98.5	73.19
Random forest	97.75	73.83
Bayesian	97.25	72.82

4.5 Conclusion

In this chapter, we studied the vulnerability of malware detection systems against adversarial attacks. Rather than selecting random adversarial samples for retraining a classifier, we proposed two techniques for selecting adversarial samples. Experimental results reveal that, compared to the random selection strategy proposed in previous works, our selective sample selection strategy leads to better performance. Our KBL selection method achieved a 6% performance improvement in accuracy over randomized selection. While prior works are mostly based on deep neural networks, we show that other well-known classifiers are also vulnerable against adversarial samples in malware detection. Our selection methods yielded similar performance improvement for those classifiers as well. In the next chapter, we deal with an important data characteristic known as imbalanced data problem which can make a malware detection system biased and degrade its performance in detecting malicious applications. We propose a novel technique to balance the data and develop a hybrid system that effectively deals with imbalanced data problem in mobile malware detection.

Chapter Five

Tackling Data Imbalance in Model Training

5.1 Introduction

In the last chapter, we investigated adversarial attacks on malware detection systems. We showed that adversarial samples can successfully evade malware detection by carefully changing their features. We developed a selective strategy as a defense mechanism against such attacks that retrains a classifier with an optimal adversarial sample set. However, machine learning-based classifiers are designed for balanced data distribution, that is, the number of samples from each class (i.e., malware and benign) are expected to be similar. Proposed detection systems in the literature also suffer from this inherent assumption. In real-world malware detection problems, the data distribution is highly imbalanced since benign applications greatly outnumber malware applications. This imbalance data problem contributes to the degraded performance of machine learning models for malware detection. We address this gap in the literature. In the following sections, we present a hybrid approach using a novel synthetic oversampling strategy and an adaptive cost schema to handle imbalanced data problem in mobile malware detection.

5.2 Methodology

Figure 5.1 shows the workflow of our proposed approaches for handling imbalanced data in mobile malware detection. We first propose a new synthetic over-sampling technique for mobile malware detection and combine this with our first approach, which is based on the fuzzy synthetic minority over-sampling technique (Fuzzy-SMOTE [16]). This approach is labelled "Proposed Method 1" in Fig. 5.1. Afterward, we propose a dynamic minority class weight scheme combined with the synthetic sampling technique, labelled as "Proposed Method 2" in Fig. 5.1. Our proposed approaches are detailed in following sections.

5.2.1 Proposed Technique for Synthetic Malware Oversampling

Data imbalance in mobile malware detection is characterized by a large number of benign applications compared to the number of malicious applications. We define the *imbalance ratio,* ρ, as follows

$$\rho = \frac{no.\ of\ malware}{total\ no.\ of\ apps} \times 100\% \tag{5.1}$$

To ensure balanced data for training a classifier it is necessary to either increase the number of samples of the minority class (over-sampling) or decrease the number of samples of the majority class (under-sampling). However, over-sampling and under-sampling have their own drawbacks. Over-sampling the minority class by simply replicating the samples often results in redundant information for the classifier resulting in over-fitting. On the other hand, under-sampling by removing minority samples leads to valuable information loss resulting in poor classification performance, especially for the majority classes. Many works in the literature are, thus, in favor of over-sampling since it preserves the available information. Synthetic over-sampling

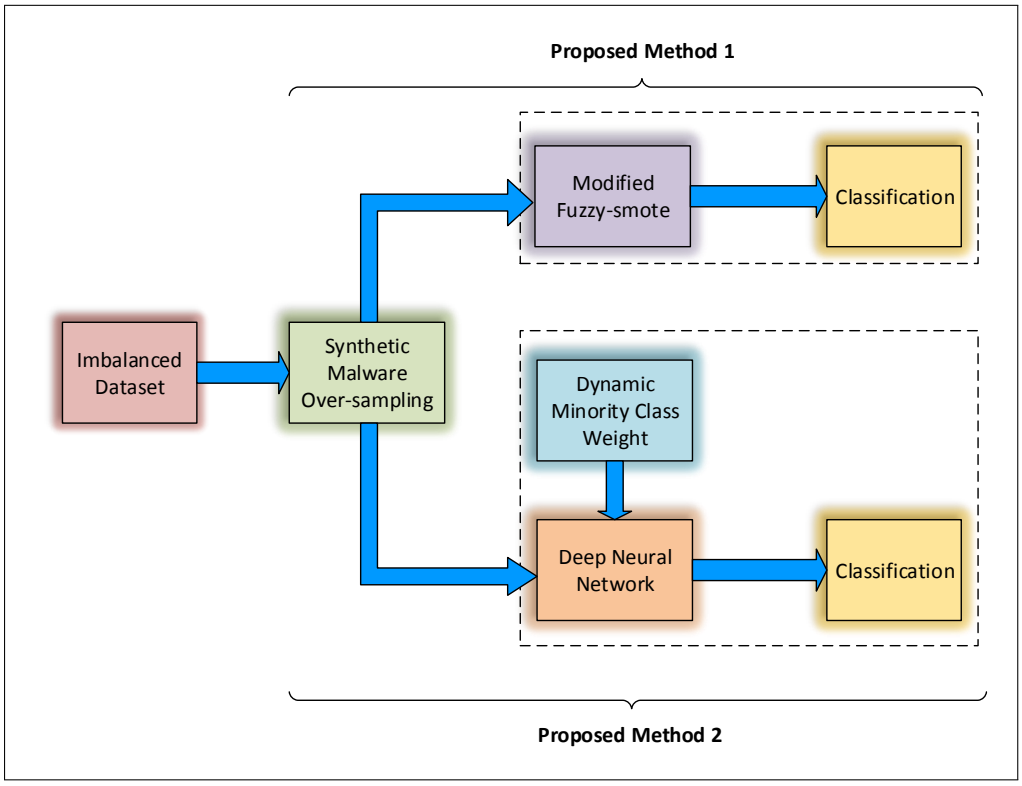


Figure 5.1 Work flow of proposed imbalanced data handling approaches

mitigates these two problems by generating artificial samples of the minority class, thus preserving all the available information of the majority class. Hence, creating synthetic samples of the minority class is generally a better choice for handling imbalanced data.

Section 2.11 described the most notable works in the literature for data balancing with synthetic samples. However, these techniques cannot be used in mobile malware detection for a few reasons. First, the techniques to generate synthetic samples are primarily for features having continuous values but the features in mobile malware detection (except the features with contextual information) in general are binary. Also, the general idea of interpolating artificial samples between two original samples by calculating the distance between them is not applicable here. Synthetic samples created in this way often do not represent a valid application since we cannot just randomly remove or add features to an application. A sample generated in this way

may represent an invalid application with broken code sequences. To illustrate this, we take the example of the MoonSMS malware described in Sec. 3.3 which sends premium messages to steal money from the user. It requires the "`SEND_SMS`" permission to function properly and calls the "sendTextMessage()" function for sending messages. This means that the feature vector of this application will have 1 in the index corresponding to "`SEND_SMS`" permission, and a specific value in the index representing the "sendTextMessage()" function depending on the contextual use of that function. If the synthetic sample generation process arbitrarily modifies these values, the corresponding application code will no longer have the functionality to send an SMS. Further, an interpolated value (i.e., a fractional number) cannot be used for a permission feature since it cannot have real values. This means this process of synthetic sampling may end up generating additional noise rather than useful data points.

We propose a technique for over-sampling malicious applications with the following conditions:

- The samples should not be copies of the original application, which results in data redundancy, rather, samples should be generated synthetically to provide additional information for the classifier.
- The generated samples should represent a valid application (i.e., the corresponding code should not have any broken sequences).
- The synthetic malware samples should have their malicious functionality preserved.

Our goal is to generate synthetic malware samples until a balance between the two classes is reached. For this, we took the original malware samples and modified their features to generate new samples. However, during the synthetic malware sampling process, we imposed the constraints described in Sec. 4.3.1 so that the above conditions are met and valid samples are generated.

Algorithm 7 Synthetic Malware Oversampling

Input: X_{mal}, N_{mal}, N_{ben}, Idx_m, ρ (candidate malware feature matrix, no. of candidate malware, no. of benign apps, indices of the features that are allowed to change, target malware to benign ratio)
Output: X_{mal} // augmented with synthetic malware

1: $\eta \leftarrow \frac{\rho \times N_{ben}}{1-\rho} - N_{mal}$ // required no. of malware
2: **while** $\eta > 0$ **do**
3: **if** $\eta > N_{mal}$ **then**
4: $S \leftarrow$ randomly choose N_{mal} no. of malware from X_{mal}
5: $\eta \leftarrow \eta - N_{mal}$
6: **else**
7: $S \leftarrow$ randomly choose η no. of malware of from X_{mal}
8: $\eta \leftarrow 0$
9: **end if**
10: $Syn \leftarrow \text{SynMal}(S, Idx_m)$ // generate synth. malware
11: $X_{mal} \leftarrow X_{mal} \cup Syn$
12: **end while**
13: **function** SynMal($Vectors$, $Idx\_mutable$)
14: **for each** V in $Vectors$ **do**
15: randomly choose index i such that, $V[i] = 0$ & $i \in Idx\_mutable$
16: $V[i] \leftarrow 1$
17: **end for**
18: **return** $Vectors$
19: **end function**

Algorithm 7 shows the pseudo-code of our proposed approach for synthetic malware oversampling. It uses a candidate malware feature matrix (X_{mal}), the number of candidate malware (N_{mal}) and benign applications (N_{ben}), indices of mutable features (Idx_m), and the desired ratio between malware and benign applications (ρ) as inputs. The desired ratio allows a user the flexibility to change the level of balance between malware and benign applications, oversampling can thus be undertaken to reach both perfect balance ($\rho = 0.5$) and moderate imbalance ($\rho \neq 0.5$). For our experiments, we consider ρ=0.5.

Line 1 of the algorithm, depending on N_{mal}, N_{ben} and ρ, calculates the number of malware necessary to reach the target malware-to-benign ratio. An iterative process then follows. At each iteration except the last, we select N_{mal} samples from the

current malware set, modify them to generate synthetic malware samples, and add them to the current set of malware samples (lines 2 to 12). This way of random choosing of malware, mutation of features, and augmenting the malware set before sampling in the next iteration reduces the probability that the same copy of synthetic malware is generated too many times. Thus, our approach reduces redundancy and also allows broader coverage of feature space. For the last iteration, only the number of malware samples required to reach the desired ratio (ρ) is generated and added to X_{mal}. Lines 13 to 19 define the function that modifies the malware feature vector to generate synthetic samples. It is noteworthy that our model is not affected by any of the limitations posed by distance measures (e.g., the Euclidean distance) since we are permuting the features rather than interpolating them.

5.2.2 Proposed Approach 1: Modified Fuzzy-SMOTE

This approach is based on Fuzzy-SMOTE [16] that over-samples the minority class samples depending on their membership degree to the minority class based on fuzzy set theory. The fuzzy set theory states that minority samples that have a lower degree of membership to the minority class are more likely to be misclassified. By over-sampling these samples, the decision region of the minority class can be broadened so that the classification bias is removed. To illustrate this, we consider two classes C_b and C_m denoting classes of benign and malicious applications respectively. Let $\mathbf{v}_b$ and $\mathbf{v}_m$ denote the centroids of the classes respectively which are defined as follows

$$\mathbf{v}_b = \frac{1}{N_b} \sum_{n=1}^{N_b} \mathbf{x}_{bn} \tag{5.2}$$

$$\mathbf{v}_m = \frac{1}{N_m} \sum_{n=1}^{N_m} \mathbf{x}_{mn} \tag{5.3}$$

where N_b and N_m denotes the number of samples in benign and malware class respectively and x_{bn} and x_{mn} denotes the n^{th} sample from benign and malware class

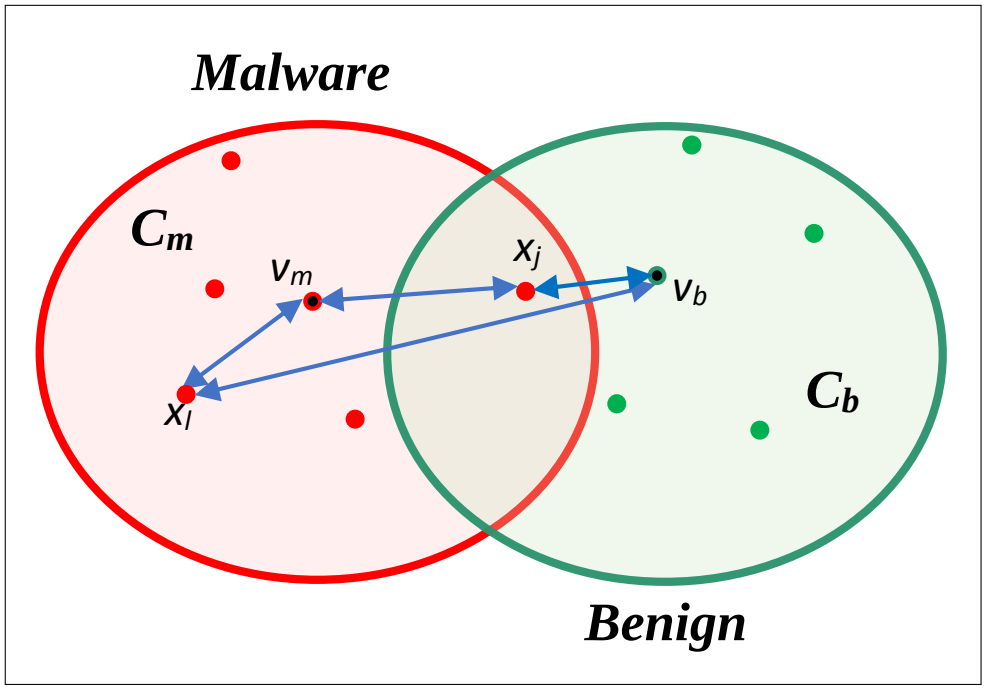


Figure 5.2 Illustrating membership degree for sample x_i and x_j to class C_b and C_m

respectively. The membership degree of a sample $\mathbf{x}$ to a class C_i, $M_{C_i}(\mathbf{x})$, is then defined as follows:

$$M_{C_i}(\mathbf{x}) = \left[\sum_{k=b}^{m} \left(\frac{\|\mathbf{x} - \mathbf{v}_i\|^2}{\|\mathbf{x} - \mathbf{v}_k\|^2} \right)^{1/(d-1)} \right]^{-1} \tag{5.4}$$

where k is set to b and m for benign and malware class respectively, $\|\mathbf{x} - \mathbf{v}_i\|$ denotes the Euclidean distance of sample $\mathbf{x}$ from centroid $\mathbf{v}_i$, and d denotes the fuzziness of membership to each class and is set to 2 as in the original work [16]. A higher value of $M_{C_i}(\mathbf{x})$ indicates that the sample $\mathbf{x}$ has a higher membership degree to class C_i and is likely to be classified as a class C_i sample.

Figure 5.2 illustrates membership determination. Let x_l and x_j be two malware class samples. Because x_l is in a region distinctly characterized by malware samples, it will have a higher membership degree to class C_m than to class C_b, hence it is likely to be classified to class C_m. But sample $\mathbf{x}_j$ lies in the fuzzy region. Thus, if a sample $\mathbf{x}_j$

has a membership degree such that $M_{C_m}(\mathbf{x}_j) \leq 0.5$ and $M_{C_b}(\mathbf{x}_j) > 0.5$, the sample is closer to class C_b than to C_m and is more likely to be classified to class C_b. To balance the data, the minority class samples (i.e., malware) whose membership degree to the minority class satisfy $M_{C_m}(.) \leq 0.5$ are identified first. These samples in the fuzzy region are those that Fuzzy-SMOTE over-samples using an interpolation technique.

As our first proposed approach, we consider a modification of Fuzzy-SMOTE. More precisely, similar to [16], we also consider the cluster centers of malware (minority) and benign (majority) class samples and consider the same membership function as in Eq. (5.4). Based on the membership function, we also identify malware samples that fall within the fuzzy regions. However, contrary to [16], whose interpolation technique leads to continuous variables and therefore invalid malware samples, we apply the over-sampling technique proposed in Sec. 5.2.1 to balance the data. This ensures that our approach considers only the synthetic samples that are valid and preserve malware functionalities. The candidate malware feature matrix passed as input for the first approach are only the samples that the fuzzy membership function in Eq. (5.4) characterizes as having a higher likelihood of misclassification.

5.2.3 Proposed Approach 2: Modified Loss Function with Dynamic Minority Class Weight

Here, we propose a hybrid of data level and algorithm level technique to deal with imbalanced data. We use the technique proposed in Sec. 5.2.1 as the data level technique. Additionally, at the algorithm level, we propose a modified loss function with a weight dynamically adjusted for the minority class. Since the cost of misclassifying a malware is greater than that of a benign application, our cost function initially gives more weight (penalty) to a malicious application. However, as the training progresses keeping a high penalty can further lead to model instability [222]. Hence, we adjust this weight dynamically based on the performance of the model on the validation set

during the training phase of the model.

For this we use the cross entropy loss and the usual cross entropy loss is defined as

$$-(ylog(p) + (1-y)log(1-p)) \tag{5.5}$$

where y is the original label and p is the probability predicted by the model. We define the weighted cross-entropy loss as

$$-(ylog(p) + P_w(1-y)log(1-p)) \tag{5.6}$$

where P_w is the minority class weight. We initialize this weight as follows

$$P_{w0} = \frac{N_m - N_b}{\gamma \times N_b} \tag{5.7}$$

$$P_w = 1 + P_{w0} \tag{5.8}$$

where γ is a parameter that controls the scale of the minority class weight. Our empirical study shows that, depending on the imbalance ratio, the optimal value for γ is between 20 and 90. The weight, P_w, is dynamically adjusted based on the model's performance during training. We calculate the F1-score of the model on the validation set after each epoch and then adjust the weight as follows

$$P_w = 1 + P_{w0}(1 + log(1 - F1\text{-}score)) \tag{5.9}$$

where P_{w0} is defined in Eq. (5.6). The lower bound of this weight is set to 1, i.e., when the majority and minority class weights become the same. Here, "F1-score" measure was used for weight adjustment instead of "accuracy" since accuracy can provide misleading information about classifier performance when dealing with imbalanced data (detailed in later sections). The characteristic curve of this dynamic weight is shown in Fig. 5.3 with varying initial weights. It shows that the minority class weight remains high where the model is suffering from class imbalance and its F1-score is

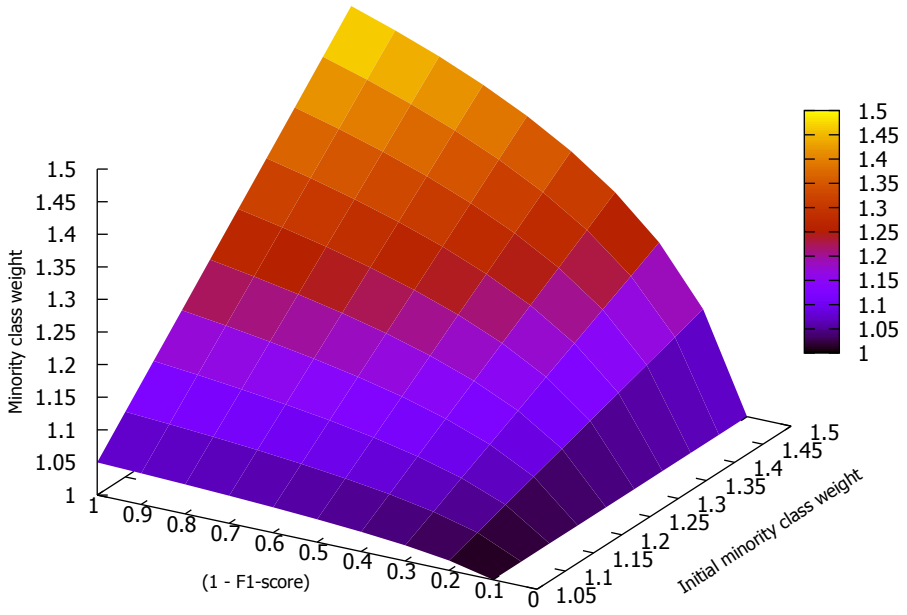


Figure 5.3 Minority class weight characteristic curve

poor. The minority class weight starts to decrease towards the region where the model starts to adapt and the F1-score improves.

5.3 Experiments and Results

5.3.1 Dataset Preparation

We took 50,000 samples from the datasets described in Sec. 2.6. Five hundred malware samples were randomly selected as minority class samples and the number of benign samples was varied from 4500 to 49500 to formulate several datasets with various imbalance ratios. A total of 10 datasets were produced in the process with an imbalance ratio varying from 10% to 1%. Stratified 10-fold cross-validation was used for each of the datasets. We took 20 trials of the experiments for each dataset and the average of these trials is reported in the results shown below.

5.3.2 Discussion

Impact of Data Imbalance on Detection Performance

Figure 5.4 shows the impact of data imbalance on the classifier performance. While the classification accuracy is above 97% for all the datasets, the figure clearly shows that the model suffers greatly from data imbalance in terms of recall rate and F1-score. It also shows that as the data imbalance increases, the classifier becomes more biased towards the majority class (benign applications) as evidenced through increased overall accuracy but decreasing recall value and F1-score. Such models with low recall and F1-score are useless as far as malware detection is concerned.

Performance Comparison

Figure 5.5 shows the performance comparison among the over-sampling, under-sampling, and our two proposed approaches in terms of accuracy, precision, recall, F1-score, and G-mean. The figure shows that both of our proposed approaches outperform other techniques for all the imbalanced ratios. The performance improvement is especially noticeable when the imbalanced ratio is lower than 5% for precision and F1-score (Figures 5.5a and 5.5c respectively). For example, with the dataset of 1% imbalance ratio, our two approaches achieve 0.87 and 0.89 F1-score respectively while over-sampling and under-sampling achieve 0.64 and 0.17 respectively.

We noticed different trends in terms of recall and F1-score between over-sampling and under-sampling. In the case of over-sampling, duplicates of the malware samples are created to balance the dataset. Even though the classifier suffers less from data imbalance, the duplication leads to over-fitting, and thus, it does not perform as well on the unseen malware samples as the under-sampling technique. As a result, the recall value for over-sampling is much lower than that of the under-sampling technique. However, since no information is lost from the benign samples, the classifier still performs well on benign samples which is reflected in the better F1-score values.

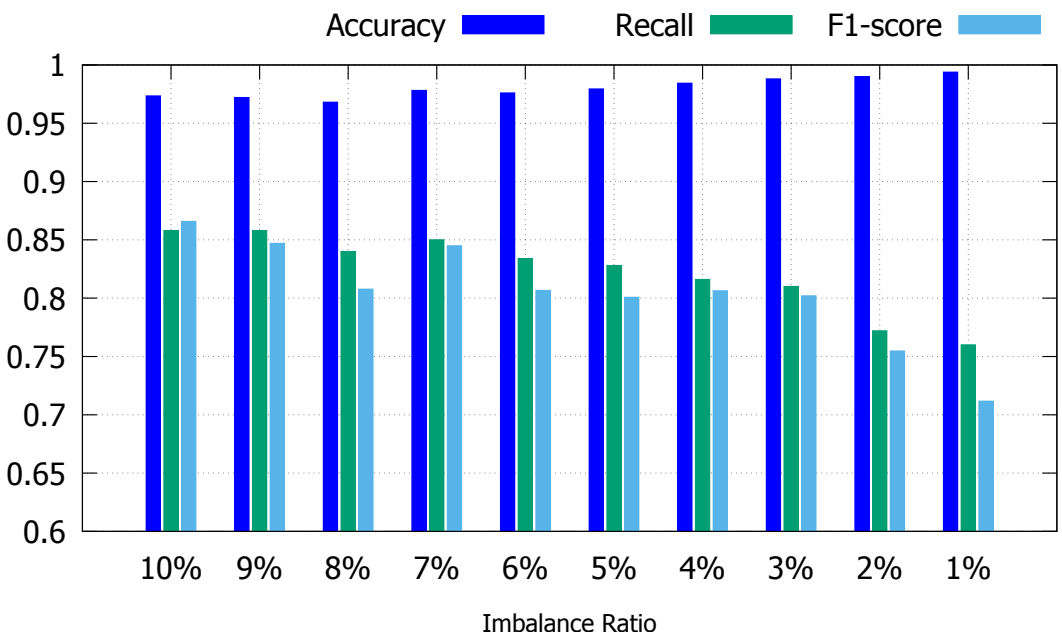


Figure 5.4 Effect of imbalance on malware detection (Imbalanced ratio of 10% indicates only 10% of the total samples are malware)

This can further be observed in precision values which are better for the over-sampling technique compared to under-sampling.

On the other hand, under-sampling removes samples from the benign class to balance the data. This removes the bias towards the majority class, resulting in better recall value. However, since a lot of information from the majority class is lost, the detection of benign samples suffers greatly. This is reflected by the drop in F1-score with increasing level of imbalance, for example, 0.52 and 0.27 for 5% and 2% imbalance ratio respectively (Fig. 5.5c).

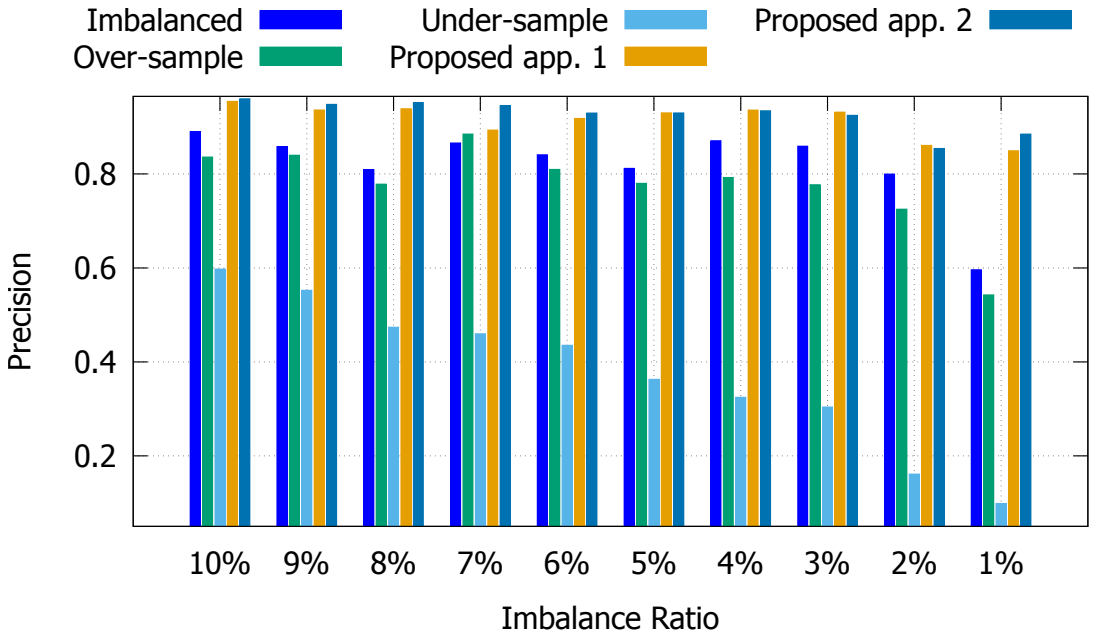
Our proposed approaches outperform all these methods. Since our approaches do not remove samples from the majority class they do not suffer from information loss. Further, unlike under-sampling, they do not create mere copies of the minority class sample; rather, they create synthetic samples by modifying feature values. Hence, they do not suffer from over-fitting. Moreover, our sample generation technique (described in Sec. 5.2.1) permutes the features in the binary domain rather than

interpolating their values. This enables it to explore valid sample space to reduce data imbalance. Consequently, we see better performance by our approaches.

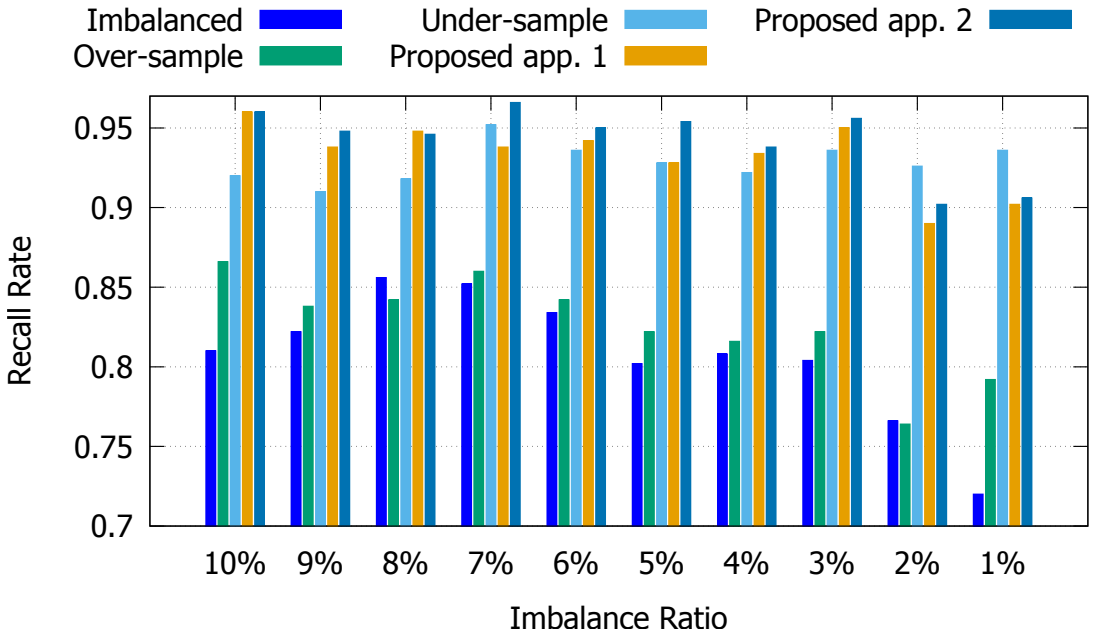
Further, in our second approach (Sec. 5.2.3), we devised a loss function that assigns dynamic weights to the minority class to reduce classification bias towards the majority class. This allows this approach to perform better on minority class. This effect is demonstrated in Fig. 5.5b and 5.5d which shows our second approach achieves higher recall and G-mean value.

In Fig. 5.5b, we see that in a very few cases, and only in terms of recall value, under-sampling performs slightly better than our proposed methods. This is observed when the imbalance ratio goes lower than 3%. Our detailed inspection of the results shows that when data imbalance is this high (i.e., imbalance ratio is low), under-sampling removes most of the benign samples from the training data. This allows the classifier to learn the malware class while ignoring most of the information from the benign class, resulting in slightly better recall value. However, this comes with a significant compromise in the overall performance which is evident through very low precision value and F1-score of the under-sampling method (Figures 5.5a and 5.5c). In contrast, our methods do not suffer from such limitations, as seen in Fig. 5.5a and Fig. 5.5c which show our methods outperform under-sampling by a significant margin in terms of both precision and F1-score.

Table 5.1 shows a detailed performance comparison of our methods with existing methods for 10% and 5% imbalance ratio. The table shows that under-sampling reduces more false negatives (FN) than over-sampling; however, since it removes a significant number of majority samples, the false positive (FP) increases significantly. On the other hand, over-sampling reduces FP but they perform poorer than our methods due to over-fitting. Our proposed approaches outperform other methods while our second approach achieves the best true positive (TP) count (482 for 10% and 477 for 5%). It is also noteworthy that while the two proposed approaches

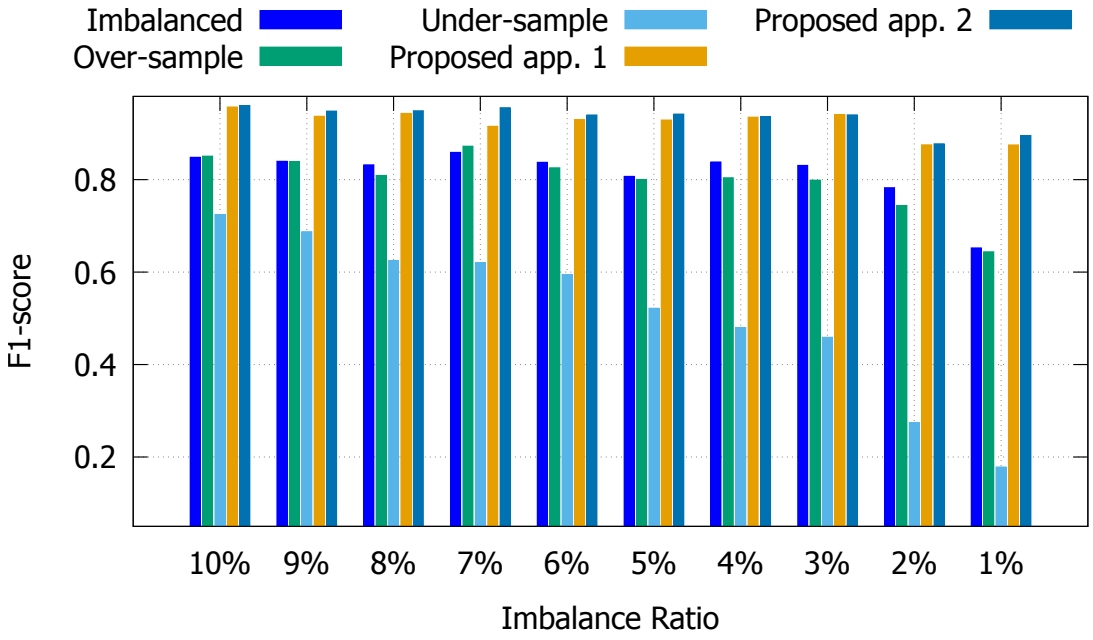


(a) Precision

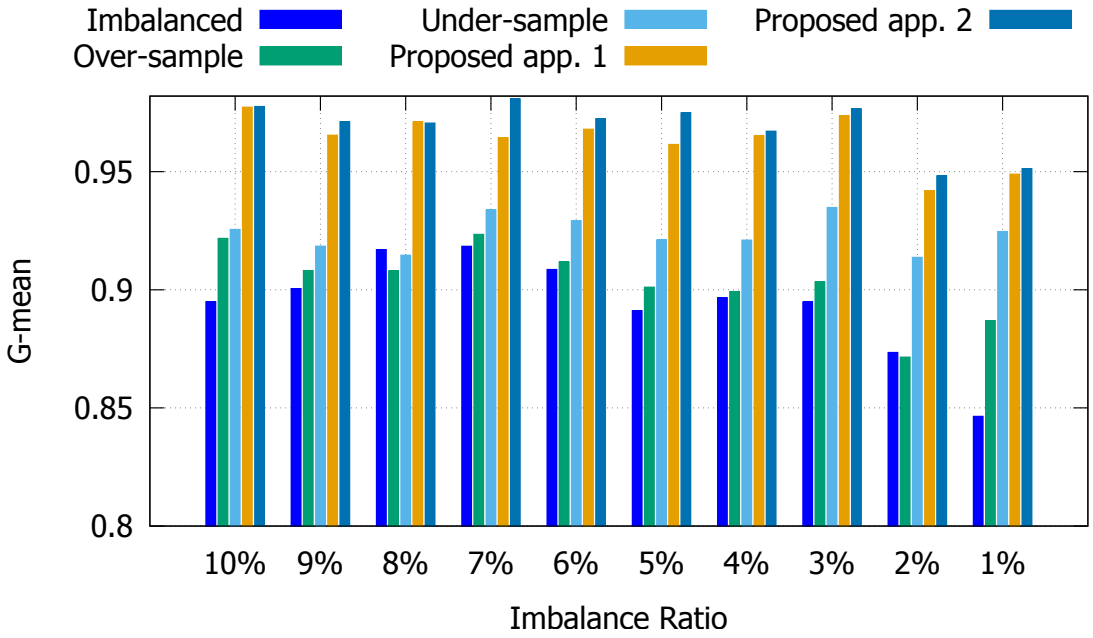


(b) Recall

Figure 5.5 Performance comparison of the proposed approaches with over-sampling and under-sampling techniques and when no imbalance-specific technique is used.



(c) F1-score



(d) G-mean

Figure 5.5 (cont.) performance comparison of the proposed approaches with over-sampling and under-sampling techniques and when no imbalance-specific technique is used.

Table 5.1 Detailed performance comparison for 10% and 5% imbalanced ratio.

Imbalance Ratio	Method	# of malware	# of benign	TP	TN	FP	FN	Accuracy (%)	G-mean	F1-score
10%	Imbalanced	500	4500	405	4450	50	95	97.10	0.895	0.848
	Oversample	500	4500	433	4415	85	67	96.96	0.922	0.851
	Undersample	500	4500	460	4190	310	40	93.00	0.926	0.724
	Fuzzy-SMOTE	500	4500	475	4452	48	25	98.54	0.969	0.929
	Proposed Approach 1	500	4500	480	4477	23	20	99.14	0.977	0.957
	Proposed Approach 2	500	4500	**482**	**4480**	**20**	**18**	**99.24**	**0.980**	**0.962**
5%	Imbalanced	500	9500	401	9407	93	99	98.08	0.891	0.807
	Oversample	500	9500	411	9384	116	89	97.95	0.901	0.800
	Undersample	500	9500	464	8685	815	36	91.49	0.921	0.522
	Fuzzy-SMOTE	500	9500	450	9450	50	50	99.00	0.946	0.900
	Proposed Approach 1	500	9500	464	9465	**35**	36	99.29	0.962	0.929
	Proposed Approach 2	500	9500	**477**	**9464**	36	**23**	**99.41**	**0.975**	**0.942**

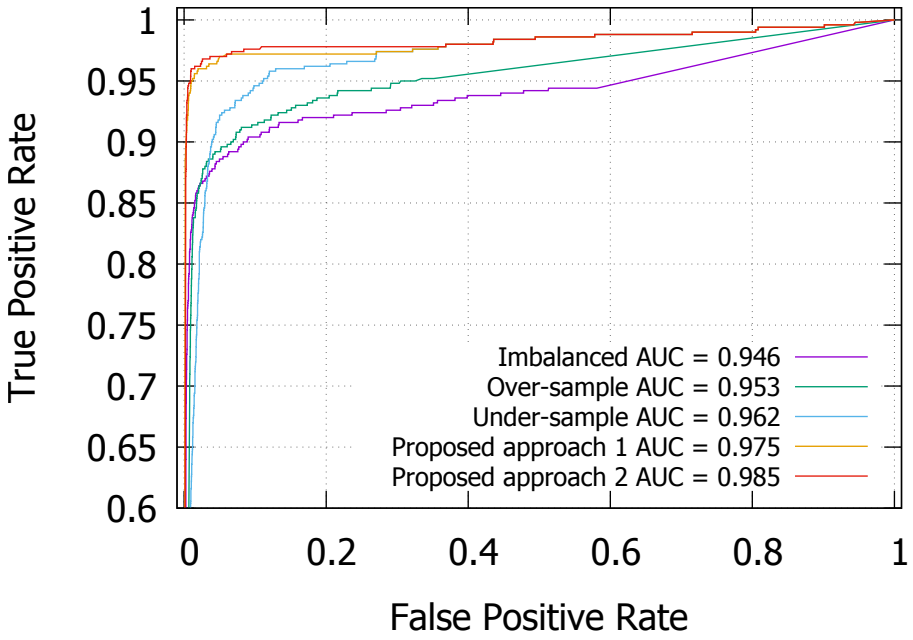
perform similarly in terms of accuracy, G-mean, and F1-score, the second approach achieves slight improvement in terms of FN compared to the first approach (18 for 10% and 23 for 5%). This is because the second approach assigns dynamic weight to the minority class which allows it to further reduce the classification bias towards the majority class.

Figures 5.6a and 5.6b show the ROC curve along with the AUC for 5% and 10% imbalanced ratio respectively. The figures show that our proposed techniques achieve better AUC values than the existing techniques for all the imbalanced ratios.

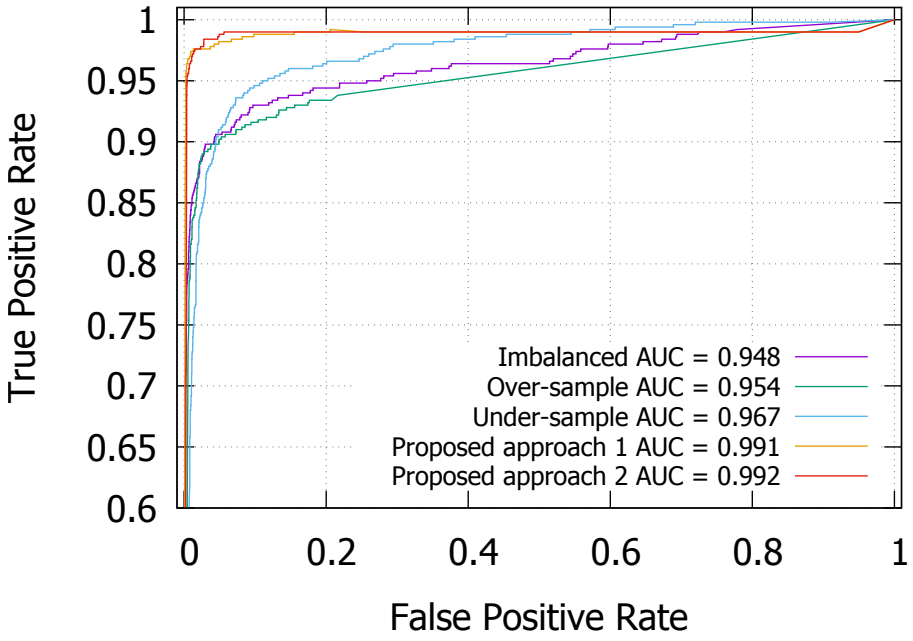
A Wilcoxon Signed-Rank Test [21] comparing our techniques with existing techniques shows that the p-value is always less than 5.1×10^{-3}, indicating that the performance improvement of our techniques is statistically significant.

Further Analysis of the Proposed Approaches

To further analyze the characteristics of our proposed approaches we conducted experiments by varying the range of fuzzy region (described in Sec. 5.2.2). Figure 5.7 shows the performance variation of the first proposed approach with various fuzzy regions. The model's performance is highly dependent on the chosen range of the fuzzy region. We see that different imbalance ratio of the dataset requires different fuzzy region for optimal performance (e.g., region 0.6 for 7% imbalance ratio, and 0.5 and 0.8 for 5% and 3% imbalance ratio respectively for F1-score measure). Also, for different performance measures, different fuzzy region yields optimal performance (e.g., in case of 3% imbalance ratio, the best performance is found when the fuzzy region is 0.8 for F1-score, 0.6 for G-mean, and 0.4 for specificity). Hence, deciding the fuzzy region for the best performance can prove to be a difficult task. On the other hand, the dynamic minority class weight in the second proposed approach is automatically adjusted based on the model performance during the training time using Eq. (5.9). As a result, no manual parameter adjustment is needed and the approach does not suffer from such drawbacks. Thus, of the two proposed approaches, the

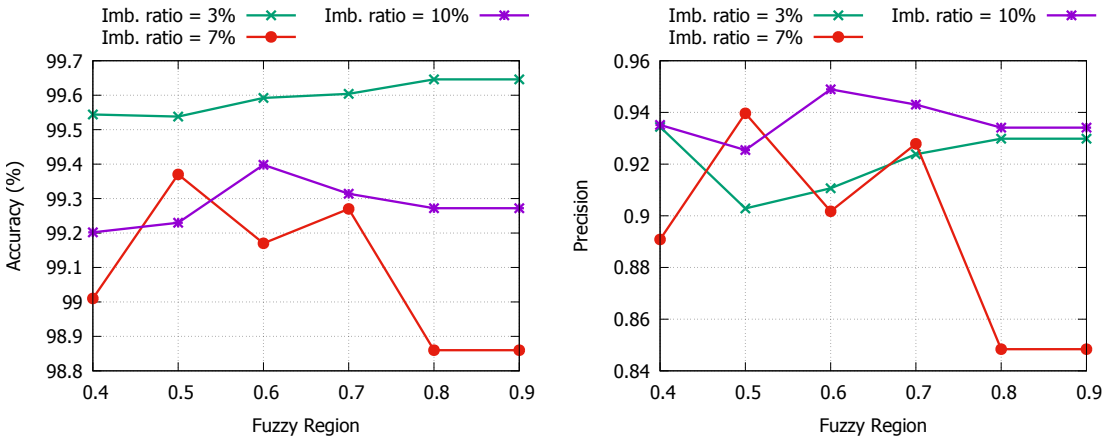


(a) Imbalance ratio = 5%



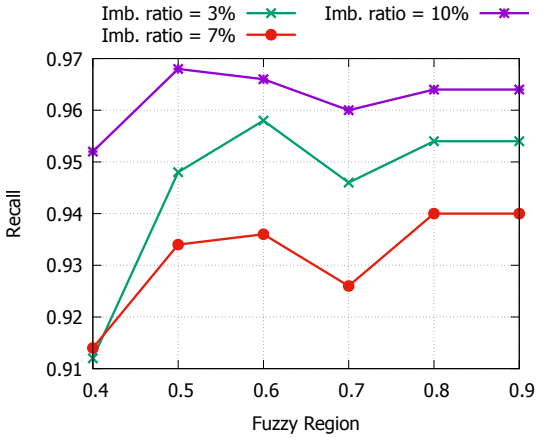
(b) Imbalance ratio = 10%

Figure 5.6 ROC curve and AUC of the proposed approaches, over-sampling and under-sampling techniques, and when no imbalance-specific technique is used.

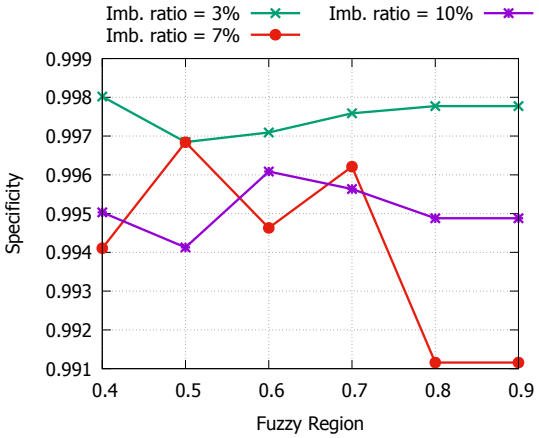


(a) Accuracy

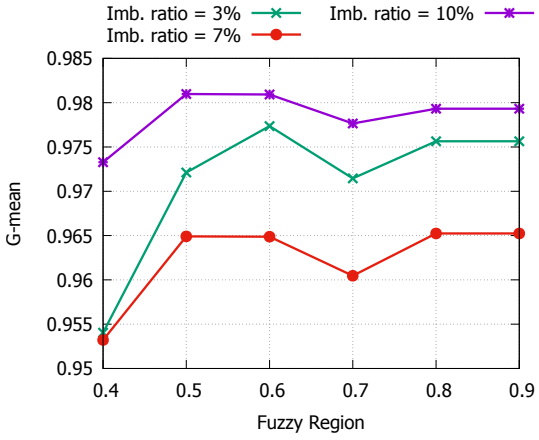
(b) Precision



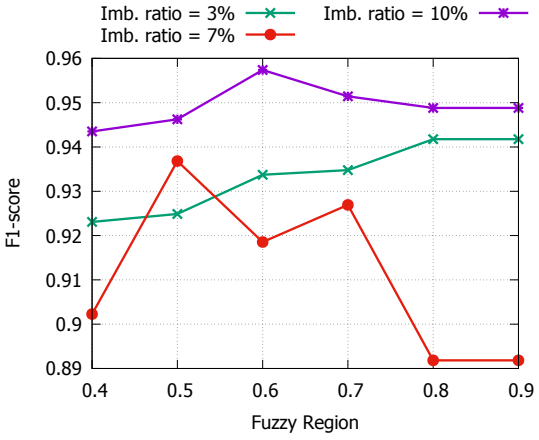
(c) Recall



(d) Specificity



(e) G-mean



(f) F1-score

Figure 5.7 Performance of the proposed approach 1 with varying fuzzy region.

second approach has higher potentials for application in mobile malware detection with imbalanced data.

5.4 Conclusion

In this chapter, we addressed the real-world data imbalance characteristics in mobile malware detection. We showed that traditional over-sampling and under-sampling do not work well since they suffer from over-fitting and information loss. The existing synthetic sampling technique also suffers from invalid sample generation. We proposed a novel synthetic malware generation technique that generates mobile malware samples with valid functionalities. Additionally, we proposed a hybrid technique that utilizes a loss function assigning weights to the minority class (i.e., malware) dynamically during the training process. Our experimental results show that our proposed approaches do not suffer from the above-mentioned drawbacks and the hybrid approach improves F1-score by 9%. In the next chapter, we conclude this thesis by presenting conclusive remarks and some directions for further extensions of this work.

Chapter Six

Conclusion and Future Works

In Chapter 1 we formulated three major research objectives that we wanted to address in this thesis. In the previous three chapters, those problems have been addressed in detail. In this chapter, we make concluding remarks based on the findings of the research that addressed those objectives. First, this work systematically investigated the impact of different feature categories on malware detection performance. Although several works in the literature have performed feature selection in a randomly chosen feature category set, none have systematically analyzed the effect of individual feature categories and their different combinations on malware detection performance. This study filled this gap. Additionally, this work proposed a technique to embed contextual information into the feature space allowing this information to be used in combination with other feature categories. Extensive experiments were conducted to analyze the performance impact of different feature categories on mobile malware detection. The experiments show that as a single feature category, “permission" provides the best performance with 96.26% accuracy without contextual information, and “API calls" provides the best performance with contextual information obtaining 97.89% accuracy. However, when features were combined, the combination of permission, API calls, and dynamic features provided the best performance attaining 98.21% accuracy and outperforming the combination of all categories (97.93% accuracy).

This work then proceeded to make the detection system robust against adversarial attacks. Tackling adversarial attacks is especially important in malware detection domain since they can take down a well-performing detection system by carefully crafting malware samples. The work proposed two approaches to select the best sample set from self-crafted adversarial samples for retraining classifiers because retraining with too many samples further degrades detection performance. Randomly selecting a sample set, as done in prior studies, results in sub-optimal performance. The KBL method proposed in this work achieves a 6% improvement in terms of detection accuracy. The Wilcoxon Signed-Rank test indicates that the performance improvement of our proposed approach is statistically significant with p-value $< 3.8 \times 10^{-4}$. One possible extension of our approach for further improvement could be to implement an ensemble learning system where different types of classifiers with different architectures are trained and used together for classification. However, this drastically increases the complexity of the system which may not be practical especially when dealing with large amounts of data and when fast deployment is required.

Finally, to address the real-world malware data characteristics, this thesis considered imbalanced class distribution in the training data set. Since existing data balancing techniques generate invalid malware samples, this work proposed a novel synthetic malware sample generation technique that preserves the malicious functionality of the original samples. To further counter the majority class bias, a dynamic cost schema was devised that automatically assigns minority class weight during model training. This hybrid technique improved F1-score by 9%.

The techniques developed in this thesis have great practical aspects that can provide assistance to system planners especially in designing security components. The methods proposed in this thesis can be used as a standalone component and integrated into the system for malware detection making use of its feature extraction and robust classification mechanism. Also, the individual contributions of this thesis such as protection against adversarial attacks, and imbalanced data handling mechanism can

be used separately to improve the robustness of a malware detection system.

Future Works

Considering that a malware detection system is required to be robust and continuously adapt itself to newfound vulnerabilities and exploits, we present some directions for extending the work presented in this thesis:

- Since attackers are constantly looking for new vulnerabilities in the system and attempting to exploit them, the problem of mobile malware detection is always evolving. Handling such issues (generally known as concept drift) in this regard is very important to ensure that the system is able to detect new generations of malware. The imbalanced data handling technique in this work can be utilized to address this. The method for handling imbalanced data deals with differences in data distribution in terms of sample size. It can be extended for concept drift that changes the data distribution from a different perspective since newer features get added to the feature space or certain features become stronger relative to others over time. In addition to being able to learn newer attacks, it is also important to form a mechanism to forget obsolete information. For example, Sec. 2.6 mentioned some attacks becoming invalidated or outdated due to OS upgrades. The information particularly associated with such attacks need to be forgotten by the model for better generalization and overall performance of the detection system. Identifying the point where concept drift starts to occur is also another major challenge in this regard and which can be addressed with time-stamped data and periodically evaluating the classifier looking for significant performance drop.

- The adversarial sample crafting technique in Chapter 4 can be extended to provide explainable decisions. For example, as demonstrated in this thesis, it is possible to identify the minimum number of features that need to be modified in

order to mislead a classifier. These features can be taken as the most influential features and analyzing their semantics and corresponding codes can provide the reasoning behind a particular prediction. This can be valuable in designing a better detection system, especially for resolving the false negatives since it can explain why a particular malicious behavior went undetected.

- The robustness mechanisms against adversarial attacks can be extended to achieve resilience against zero-day attacks. Zero-day attacks refer to the exploitation of newfound vulnerabilities in the system that have not been patched yet. Similar to adversarial retraining, this requires a proactive strategy. Continual research on current systems must be conducted and whenever a new vulnerability is found, the associated feature set should be identified. Malware samples utilizing the features corresponding with newfound vulnerabilities can then be crafted and a classifier can be retrained with these samples using the adversarial retraining strategies.

- The models designed in this work can be utilized to develop a *federated machine learning* system where a central machine learning system resides in a remote server and local detection models are trained in individual mobile devices. Each device trains their model on local data and periodically sends updated information about the model to the server. The server aggregates the information and updates the central model which is then propagated back to the individual devices. This enables a model to generalize better and learn from a wide range of data. The major challenge here is to design a proper system that can effectively aggregate the information obtained from the individual devices that can update the central model. An efficient protocol that reduces the number of communication rounds and optimally transfers useful information between the server and the devices also needs to be designed.

References

[1] Natarajan Sundaram, Cherian Thomas, and Loganathan Agilandeeswari. "A Review: Customers Online Security on Usage of Banking Technologies in Smartphones and Computers." In: *Pertanika Journal of Science & Technology* 27.1 (2019).

[2] MF Brunette et al. "Use of smartphones, computers and social media among people with SMI: opportunity for intervention". In: *Community mental health journal* 55.6 (2019), pp. 973–978.

[3] Elia Abi-Jaoude, Karline Treurnicht Naylor, and Antonio Pignatiello. "Smartphones, social media use and youth mental health". In: *CMAJ* 192.6 (2020), E136–E141.

[4] Yongliang Li and Linli Wu. "Research on Marketing and Profit Mode Innovation of Intelligent Mobile E-commerce". In: (2019).

[5] *Smartphone users worldwide*. https://www.statista.com/statistics/330695/. Accessed: April 23, 2020.

[6] Chao Yang, Jialong Zhang, and Guofei Gu. "Understanding the Market-level and Network-level Behaviors of the Android Malware Ecosystem". In: *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE. 2017, pp. 2452–2457.

[7] *Cyber security report*.
(https://www.cyber.gov.au/report). Accessed: Jun 27, 2020. July 2019.

[8] Kimberly Tam. "The Analysis and Classification of Android Malware". In: (2016).

[9] Li Chen et al. "Semi-supervised classification for dynamic Android malware detection". In: *arXiv preprint arXiv:1704.05948* (2017).

[10] Aiman A Abu Samra, Kangbin Yim, and Osama A Ghanem. "Analysis of clustering technique in android malware detection". In: *2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. IEEE. 2013, pp. 729–733.

[11] Suleiman Y Yerima, Sakir Sezer, and Gavin McWilliams. "Analysis of Bayesian classification-based approaches for Android malware detection". In: *IET Information Security* 8.1 (2014), pp. 25–36.

[12] Veelasha Moonsamy, Jia Rong, and Shaowu Liu. "Mining permission patterns for contrasting clean and malicious android applications". In: *Future Generation Computer Systems* 36 (2014), pp. 122–132.

[13] Qi Li and Xiaoyu Li. "Android malware detection based on static analysis of characteristic tree". In: *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2015 International Conference on*. IEEE. 2015, pp. 84–91.

[14] Daniel Arp et al. "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket." In: *Ndss*. Vol. 14. 2014, pp. 23–26.

[15] Ke Xu, Yingjiu Li, and Robert H Deng. "ICCDetector: ICC-based malware detection on Android". In: *IEEE Transactions on Information Forensics and Security* 11.6 (2016), pp. 1252–1264.

[16] Yanping Xu et al. "Fuzzy–synthetic minority oversampling technique: Oversampling based on fuzzy set theory for Android malware detection in imbalanced datasets". In: *International Journal of Distributed Sensor Networks* 13.4 (2017), p. 1550147717703116.

[17] Nitesh V Chawla et al. "SMOTE: synthetic minority over-sampling technique". In: *Journal of artificial intelligence research* 16 (2002), pp. 321–357.

[18] Yajin Zhou and Xuxian Jiang. "Dissecting android malware: Characterization and evolution". In: *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE. 2012, pp. 95–109.

[19] Kevin Allix et al. "Androzoo: Collecting millions of android apps for the research community". In: *2016 IEEE/ACM 13th Working Conf. on MSR*. IEEE. 2016, pp. 468–471.

[20] Ron Kohavi et al. "A study of cross-validation and bootstrap for accuracy estimation and model selection". In: *Ijcai*. Vol. 14. 2. Montreal, Canada. 1995, pp. 1137–1145.

[21] Frank Wilcoxon. "Individual comparisons by ranking methods". In: *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[22] Mahbub E Khoda et al. "Mobile Malware Detection: An Analysis of Deep Learning Model." In: *ICIT*. 2019, pp. 1161–1166.

[23] Mahbub E Khoda et al. “Mobile Malware Detection-An Analysis of the Impact of Feature Categories”. In: *International Conference on Neural Information Processing*. Springer. 2018, pp. 486–498.

[24] Mahbub E Khoda et al. “Robust Malware Defense in Industrial IoT Applications using Machine Learning with Selective Adversarial Samples”. In: *IEEE Transactions on Industry Applications* (2019).

[25] Mahbub Khoda et al. “Selective Adversarial Learning for Mobile Malware”. In: *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE. 2019, pp. 272–279.

[26] Mahbub Khoda et al. “Mobile Malware Detection with Imbalanced Data using a Novel Synthetic Oversampling Strategy and Deep Learning”. In: *2020 16th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE. 2020.

[27] Upkar Varshney, Ronald J Vetter, and Ravi Kalakota. “Mobile commerce: A new frontier”. In: *Computer* 33.10 (2000), pp. 32–38.

[28] Eric WT Ngai and Angappa Gunasekaran. “A review for mobile commerce research and applications”. In: *Decision support systems* 43.1 (2007), pp. 3–15.

[29] Viswanath Venkatesh, Venkataraman Ramesh, and Anne P Massey. “Understanding usability in mobile commerce”. In: *Communications of the ACM* 46.12 (2003), pp. 53–56.

[30] Upkar Varshney and Ron Vetter. “Mobile commerce: framework, applications and networking support”. In: *Mobile networks and Applications* 7.3 (2002), pp. 185–198.

[31] Keng Siau, Ee-Peng Lim, and Zixing Shen. “Mobile commerce: Promises, challenges and research agenda”. In: *Journal of Database Management (JDM)* 12.3 (2001), pp. 4–13.

[32] Ellen Opsahl Vinbæk et al. “On Online Banking Authentication for All: A Comparison of BankID Login Efficiency Using Smartphones Versus Code Generators”. In: *International Conference on Human-Computer Interaction*. Springer. 2019, pp. 365–374.

[33] Oktay Yildiz and Ibrahim Alper Doğru. “Permission-based android malware detection system using feature selection with genetic algorithm”. In: *International Journal of Software Engineering and Knowledge Engineering* 29.02 (2019), pp. 245–262.

[34] Harshverdhan Shukla. "A Survey Paper on Android Operating System". In: *Journal of the Gujarat Research Society* 21.5 (2019), pp. 299–305.

[35] Pantawee Pantaweesak, Phongpat Sontamino, and Danupon Tonnayopas. "Alternative Software for Evaluating Preliminary Rock Stability of Tunnel using Rock Mass Rating (RMR) and Rock Mass Quality (Q) on Android Smartphone". In: *Engineering Journal* 23.1 (2019), pp. 95–108.

[36] Ming Fan. "Modeling of android software behavior feature and its applications in malicious program analysis". PhD thesis. The Hong Kong Polytechnic University, 2019.

[37] Jie Ling, Xuejing Wang, and Yu Sun. "Research of Android Malware Detection based on ACO Optimized Xgboost Parameters Approach". In: *3rd International Conference on Mechatronics Engineering and Information Technology (ICMEIT 2019)*. Atlantis Press. 2019.

[38] *Number of android devices*. https://www.macrumors.com/2017/05/17/2-billion-active-android-devices/. Accessed: May 7, 2020.

[39] Mohammed K Alzaylaee, Suleiman Y Yerima, and Sakir Sezer. "Emulator vs real phone: Android malware detection using machine learning". In: *Proceedings of the 3rd ACM on International Workshop on Security And Privacy Analytics*. ACM. 2017, pp. 65–72.

[40] *Android OS architecture*. Accessed: June 30, 2020. URL: https://en.wikipedia.org/wiki/File:Android-System-Architecture.svg.

[41] Fauzia Idrees Abro. "Investigating Android permissions and intents for malware detection". PhD thesis. City, Universtiy of London, 2018.

[42] Attia Qamar, Ahmad Karim, and Victor Chang. "Mobile malware attacks: Review, taxonomy & future directions". In: *Future Generation Computer Systems* 97 (2019), pp. 887–909.

[43] Shreya Khemani, Darshil Jain, and Gaurav Prasad. "Android malware detection techniques". In: *Emerging Research in Computing, Information, Communication and Applications*. Springer, 2019, pp. 449–457.

[44] Darell JJ Tan, Tong-Wei Chua, and Vrizlynn LL Thing. "Securing android: a survey, taxonomy, and challenges". In: *ACM Computing Surveys (CSUR)* 47.4 (2015), pp. 1–45.

[45] Moses Aprofin Ashawa and Sarah Morris. "Analysis of Android malware detection techniques: a systematic review". In: (2019).

[46] Md Russel and Omar Faruque Khan. "AndroShow: Pattern Identification of Obfuscated Android Malware Application". In: (2019).

[47] Chenglin Li et al. "Android malware detection based on factorization machine". In: *IEEE Access* 7 (2019), pp. 184008–184019.

[48] William Enck et al. "TaintDroid: an information-flow tracking system for real-time privacy monitoring on smartphones". In: *ACM Transactions on Computer Systems (TOCS)* 32.2 (2014), p. 5.

[49] Chao Yang et al. "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications". In: *European symposium on research in computer security*. Springer. 2014, pp. 163–182.

[50] Guillermo Suarez-Tangil et al. "Dendroid: A text mining approach to analyzing and classifying code structures in android malware families". In: *Expert Systems with Applications* 41.4 (2014), pp. 1104–1117.

[51] Sen Chen et al. "GUI-Squatting Attack: Automated Generation of Android Phishing Apps". In: *IEEE Transactions on Dependable and Secure Computing* (2019).

[52] Mahdi Moodi and Mahdieh Ghazvini. "A new method for assigning appropriate labels to create a 28 Standard Android Botnet Dataset (28-SABD)". In: *Journal of Ambient Intelligence and Humanized Computing* 10.11 (2019), pp. 4579–4593.

[53] Yu Feng et al. "Apposcopy: Semantics-based detection of android malware through static analysis". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 576–587.

[54] Xin Su et al. "A deep learning approach to android malware feature learning and detection". In: *Trustcom/BigDataSE/ISPA, 2016 IEEE*. IEEE. 2016, pp. 244–251.

[55] Zhenlong Yuan et al. "Droid-Sec: deep learning in android malware detection". In: *ACM SIGCOMM Computer Communication Review*. Vol. 44. 4. ACM. 2014, pp. 371–372.

[56] Zi Wang et al. "DroidDeepLearner: Identifying Android malware using deep learning". In: *Sarnoff Symposium, 2016 IEEE 37th*. IEEE. 2016, pp. 160–165.

[57] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444.

[58] Yoshiko Ariji et al. "Automatic detection and classification of radiolucent lesions in the mandible on panoramic radiographs using a deep learning object detection technique". In: *Oral surgery, oral medicine, oral pathology and oral radiology* 128.4 (2019), pp. 424–430.

[59] Zhong-Qiu Zhao et al. "Object detection with deep learning: A review". In: *IEEE transactions on neural networks and learning systems* 30.11 (2019), pp. 3212–3232.

[60] Barret Zoph et al. "Learning data augmentation strategies for object detection". In: *arXiv preprint arXiv:1906.11172* (2019).

[61] Seijoon Kim et al. "Spiking-yolo: Spiking neural network for real-time object detection". In: *arXiv preprint arXiv:1903.06530* (2019).

[62] Longlong Jing and Yingli Tian. "Self-supervised visual feature learning with deep neural networks: A survey". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2020).

[63] Iván González-Díaz et al. "Perceptually-guided deep neural networks for ego-action prediction: Object grasping". In: *Pattern Recognition* 88 (2019), pp. 223–235.

[64] David P Williams. "Demystifying deep convolutional neural networks for sonar image classification". In: (2019).

[65] Yian Seo and Kyung-shik Shin. "Hierarchical convolutional neural networks for fashion image classification". In: *Expert Systems with Applications* 116 (2019), pp. 328–339.

[66] David Balderas, Pedro Ponce, and Arturo Molina. "Convolutional long short term memory deep neural networks for image sequence prediction". In: *Expert Systems with Applications* 122 (2019), pp. 152–162.

[67] Yunyang Xiong, Hyunwoo J Kim, and Varsha Hedau. "Antnets: Mobile convolutional neural networks for resource efficient image classification". In: *arXiv preprint arXiv:1904.03775* (2019).

[68] Michał Grochowski, Arkadiusz Kwasigroch, and Agnieszka Mikołajczyk. "Selected technical issues of deep neural networks for image classification purposes". In: *Bulletin of the Polish Academy of Sciences. Technical Sciences* 67.2 (2019).

[69] Francisco Erivaldo Fernandes Junior and Gary G Yen. "Particle swarm optimization of deep neural networks architectures for image classification". In: *Swarm and Evolutionary Computation* 49 (2019), pp. 62–74.

[70] Yanan Sun et al. "Evolving deep convolutional neural networks for image classification". In: *IEEE Transactions on Evolutionary Computation* (2019).

[71] Sabato Marco Siniscalchi et al. "Exploiting deep neural networks for detection-based speech recognition". In: *Neurocomputing* 106 (2013), pp. 148–157.

[72] Yanghao Li et al. “Scale-aware trident networks for object detection”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 6054–6063.

[73] Saining Xie et al. “Exploring randomly wired neural networks for image recognition”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 1284–1293.

[74] Yuanwei Wu, Ziming Zhang, and Guanghui Wang. “Unsupervised deep feature transfer for low resolution image classification”. In: *Proceedings of the IEEE International Conference on Computer Vision Workshops*. 2019.

[75] Bohan Zhuang et al. “Structured binary neural networks for accurate image classification and semantic segmentation”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 413–422.

[76] Tong He et al. “Bag of tricks for image classification with convolutional neural networks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 558–567.

[77] Weijie Chen et al. “All you need is a few shifts: Designing efficient convolutional neural networks for image classification”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 7241–7250.

[78] Shuyang Dou et al. “Deep learning’s impact on pattern matching for design based metrology and design based inspection”. In: *Metrology, Inspection, and Process Control for Microlithography XXXIII*. Vol. 10959. International Society for Optics and Photonics. 2019, 109592G.

[79] Dmitri Roussinov and Nadezhda Puchnina. “Combining neural networks and pattern matching for ontology mining-a meta learning inspired approach”. In: *2019 IEEE 13th International Conference on Semantic Computing (ICSC)*. IEEE. 2019, pp. 63–70.

[80] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. “A fast learning algorithm for deep belief nets”. In: *Neural computation* 18.7 (2006), pp. 1527–1554.

[81] Yoshua Bengio et al. “Greedy layer-wise training of deep networks”. In: *Advances in neural information processing systems*. 2007, pp. 153–160.

[82] Vinod Nair and Geoffrey E Hinton. “Rectified linear units improve restricted boltzmann machines”. In: *ICML*. 2010.

[83] Geoffrey E Hinton. “Training products of experts by minimizing contrastive divergence”. In: *Neural computation* 14.8 (2002), pp. 1771–1800.

[84] Vladimir Vapnik. *Statistical learning theory. 1998*. Vol. 3. Wiley, New York, 1998.

[85] Leif E Peterson. "K-nearest neighbor". In: *Scholarpedia* 4.2 (2009), p. 1883.

[86] Leo Breiman. "Random forests". In: *Machine learning* 45.1 (2001), pp. 5–32.

[87] Harald Binder and Martin Schumacher. "Adapting prediction error estimates for biased complexity selection in high-dimensional bootstrap samples". In: *Statistical Applications in Genetics and Molecular Biology* 7.1 (2008).

[88] Mila Parkour. *Contagio malware database*. 2013.

[89] *VirusTotal*. URL: https://www.virustotal.com/.

[90] Meenu Ganesh et al. "Cnn-based android malware detection". In: *2017 International Conference on Software Security and Assurance (ICSSA)*. IEEE. 2017, pp. 60–65.

[91] Wenjia Li et al. "An Android malware detection approach using weight-adjusted deep learning". In: *2018 International Conference on Computing, Networking and Communications (ICNC)*. IEEE. 2018, pp. 437–441.

[92] Luo Shiqi et al. "Android malicious code Classification using Deep Belief Network." In: *KSII Transactions on Internet & Information Systems* 12.1 (2018).

[93] Yi Zhang, Yuexiang Yang, and Xiaolei Wang. "A novel android malware detection approach based on convolutional neural network". In: *Proceedings of the 2nd International Conference on Cryptography, Security and Privacy*. 2018, pp. 144–149.

[94] ElMouatez Billah Karbab et al. "MalDozer: Automatic framework for android malware detection using deep learning". In: *Digital Investigation* 24 (2018), S48–S59.

[95] Dongfang Li, Zhaoguo Wang, and Yibo Xue. "Fine-grained android malware detection based on deep learning". In: *2018 IEEE Conference on Communications and Network Security (CNS)*. IEEE. 2018, pp. 1–2.

[96] Chihiro Hasegawa and Hitoshi Iyatomi. "One-dimensional convolutional neural networks for Android malware detection". In: *2018 IEEE 14th International Colloquium on Signal Processing & Its Applications (CSPA)*. IEEE. 2018, pp. 99–102.

[97] Fabio Martinelli, Fiammetta Marulli, and Francesco Mercaldo. "Evaluating convolutional neural network for effective mobile malware detection". In: *Procedia Computer Science* 112 (2017), pp. 2372–2381.

[98] Hani Alshahrani et al. "DDefender: Android application threat detection using static and dynamic analysis". In: *2018 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE. 2018, pp. 1–6.

[99] R Winsniewski. *Android–apktool: A tool for reverse engineering android apk files*. 2012.

[100] Androguard. *androguard/androguard*. Nov. 2017. URL: https://github.com/androguard/androguard.

[101] *Monkey tool*. https://cuckoo-droid.readthedocs.io/en/latest/. Accessed: May 12, 2020.

[102] *Genymotion*. Accessed: May 31, 2020. URL: https://www.genymotion.com/.

[103] *Monkey tool*. https://developer.android.com/studio/test/monkey.html. Accessed: February 17, 2020.

[104] Caren Marzban. "The ROC curve and the area under it as performance measures". In: *Weather and Forecasting* 19.6 (2004), pp. 1106–1114.

[105] Andrew P Bradley. "The use of the area under the ROC curve in the evaluation of machine learning algorithms". In: *Pattern recognition* 30.7 (1997), pp. 1145–1159.

[106] Jerome Fan, Suneel Upadhye, and Andrew Worster. "Understanding receiver operating characteristic (ROC) curves". In: *Canadian Journal of Emergency Medicine* 8.1 (2006), pp. 19–20.

[107] Jean Bergeron et al. "Static detection of malicious code in executable programs". In: *Int. J. of Req. Eng* 2001.184-189 (2001), p. 79.

[108] David Barrera et al. "A Methodology for Empirical Analysis of Permission-based Security Models and Its Application to Android". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. Chicago, Illinois, USA: ACM, 2010, pp. 73–84. ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866317. URL: http://doi.acm.org/10.1145/1866307.1866317.

[109] 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)*. IEEE. 2016, pp. 1–5.

[110] Laura Gheorghe et al. "Smart malware detection on Android". In: *Security and Communication Networks* 8.18 (2015), pp. 4254–4272. DOI: 10.1002/sec.1340. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sec.1340. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1340.

[111] Lok Kwong Yan and Heng Yin. "Droidscope: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis". In: *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*. 2012, pp. 569–584.

[112] Wei Yang et al. "AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 303–313. ISBN: 978-1-4799-1934-5. URL: http://dl.acm.org/citation.cfm?id=2818754.2818793.

[113] Annamalai Narayanan et al. "Context-aware, Adaptive and Scalable Android Malware Detection through Online Learning (extended version)". In: *CoRR* abs/1706.00947 (2017). arXiv: 1706.00947. URL: http://arxiv.org/abs/1706.00947.

[114] Annamalai Narayanan et al. "A multi-view context-aware approach to Android malware detection and malicious code localization". In: *Empirical Software Engineering* (Aug. 2017). ISSN: 1573-7616. DOI: 10.1007/s10664-017-9539-8. URL: https://doi.org/10.1007/s10664-017-9539-8.

[115] Erika Chin et al. "Analyzing inter-application communication in Android". In: *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM. 2011, pp. 239–252.

[116] Lucas Davi et al. "Privilege escalation attacks on android". In: *International Conference on Information Security*. Springer. 2010, pp. 346–360.

[117] William Enck, Machigar Ongtang, and Patrick McDaniel. "Understanding android security". In: *IEEE security & privacy* 7.1 (2009), pp. 50–57.

[118] Adrienne Porter Felt et al. "Permission Re-Delegation: Attacks and Defenses." In: *USENIX Security Symposium*. Vol. 30. 2011.

[119] Roman Schlegel et al. "Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones." In: *NDSS*. Vol. 11. 2011, pp. 17–33.

[120] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pp. 1329–1341.

[121] Damien Octeau et al. "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis". In: *Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis* (2013).

[122] Damien Octeau et al. “Composite constant propagation: Application to android inter-component communication analysis”. In: *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press. 2015, pp. 77–88.

[123] Li Li et al. “Iccta: Detecting inter-component privacy leaks in android apps”. In: *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press. 2015, pp. 280–291.

[124] Ali Feizollah et al. “AndroDialysis: analysis of android intent effectiveness in malware detection”. In: *computers & security* 65 (2017), pp. 121–134.

[125] Martin Szydlowski et al. “Challenges for dynamic analysis of ios applications”. In: *Open Problems in Network Security*. Springer, 2012, pp. 65–77.

[126] Sangeeta Rani and Kanwalvir Singh Dhindsa. “Android Malware Detection in Official and Third Party Application Stores”. In: *International Journal of Advanced Networking and Applications* 9.4 (2018), pp. 3506–3509.

[127] Yongkai Fan et al. “Software Malicious Behavior Analysis Model based on System Call and Function Interface”. In: *2019 IEEE 9th Annual International Conference on CYBER Technology in Automation, Control, and Intelligent Systems (CYBER)*. IEEE. 2019, pp. 59–64.

[128] Sungtaek Oh, Woong Go, and Taejin Lee. “A Study on The behavior-based Malware Detection Signature”. In: *International Conference on Broadband and Wireless Computing, Communication and Applications*. Springer. 2016, pp. 663–670.

[129] Hiromu Yakura et al. “Malware analysis of imaged binary samples by convolutional neural network with attention mechanism”. In: *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. 2018, pp. 127–134.

[130] Dewashish Upadhyay et al. “Detecting Malicious Behavior of Android Applications”. In: *International Journal of Science Technology & Engineering* 2.10 (2016), pp. 663–668.

[131] Laura Gheorghe et al. “Smart malware detection on Android”. In: *Security and Communication Networks* 8.18 (2015), pp. 4254–4272.

[132] Xi Xiao et al. “Android malware detection based on system call sequences and LSTM”. In: *Multimedia Tools and Applications* 78.4 (2019), pp. 3979–3999.

[133] Vitor Monte Afonso et al. “Identifying Android malware using dynamically obtained features”. In: *Journal of Computer Virology and Hacking Techniques* 11.1 (2015), pp. 9–17.

[134] Marko Dimjašević et al. "Evaluation of android malware detection based on system calls". In: *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*. ACM. 2016, pp. 1–8.

[135] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. "Crowdroid: Behavior-based Malware Detection System for Android". In: *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM '11. Chicago, Illinois, USA: ACM, 2011, pp. 15–26. ISBN: 978-1-4503-1000-0. DOI: 10.1145/2046614.2046619. URL: http://doi.acm.org/10.1145/2046614.2046619.

[136] Deepa K. et al. "Identification of Android malware using refined system calls". In: *Concurrency and Computation: Practice and Experience* 31.20 (2019), e5311. DOI: 10.1002/cpe.5311. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5311.

[137] Juan Ramos et al. "Using tf-idf to determine word relevance in document queries". In: *Proceedings of the first instructional conference on machine learning*. Vol. 242. Piscataway, NJ. 2003, pp. 133–142.

[138] Abada Abderrahmane et al. "Android Malware Detection Based on System Calls Analysis and CNN Classification". In: *2019 IEEE Wireless Communications and Networking Conference Workshop (WCNCW)*. IEEE. 2019, pp. 1–6.

[139] Taniya Bhatia and Rishabh Kaushal. "Malware detection in android based on dynamic analysis". In: *2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security)*. IEEE. 2017, pp. 1–6.

[140] Jae-wook Jang et al. "Andro-dumpsys: anti-malware system based on the similarity of malware creator and malware centric information". In: *computers & security* 58 (2016), pp. 125–138.

[141] Gerardo Canfora, Francesco Mercaldo, and Corrado Aaron Visaggio. "An hmm and structural entropy based detector for android malware: An empirical study". In: *Computers & Security* 61 (2016), pp. 1–18.

[142] Shifu Hou et al. "Deep neural networks for automatic android malware detection". In: *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*. 2017, pp. 803–810.

[143] Robin Nix and Jian Zhang. "Classification of android apps and malware using deep neural networks". In: *2017 International joint conference on neural networks (IJCNN)*. IEEE. 2017, pp. 1871–1878.

[144] Wei Wang, Mengxue Zhao, and Jigang Wang. "Effective android malware detection with a hybrid model based on deep autoencoder and convolutional

neural network". In: *Journal of Ambient Intelligence and Humanized Computing* 10.8 (2019), pp. 3035–3043.

[145] Xu Jiang et al. "Android malware detection using fine-grained features". In: *Scientific Programming* 2020 (2020).

[146] 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 (ISI)*. IEEE. 2017, pp. 140–142.

[147] 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.

[148] R Vinayakumar et al. "Detecting Android malware using long short-term memory (LSTM)". In: *Journal of Intelligent & Fuzzy Systems* 34.3 (2018), pp. 1277–1288.

[149] Fei Tong and Zheng Yan. "A hybrid approach of mobile malware detection in Android". In: *Journal of Parallel and Distributed Computing* 103 (2017), pp. 22–31.

[150] Ankita Kapratwar. "Static and Dynamic Analysis for Android Malware Detection". In: (2016).

[151] Christian Szegedy et al. "Intriguing properties of neural networks". In: *CoRR* abs/1312.6199 (2014).

[152] Christoph Molnar. *Interpretable machine learning*. Lulu. com, 2019.

[153] Dong C Liu and Jorge Nocedal. "On the limited memory BFGS method for large scale optimization". In: *Mathematical programming* 45.1-3 (1989), pp. 503–528.

[154] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. "Explaining and Harnessing Adversarial Examples". In: *CoRR* abs/1412.6572 (2015).

[155] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. "One pixel attack for fooling deep neural networks". In: *IEEE Transactions on Evolutionary Computation* 23.5 (2019), pp. 828–841.

[156] TB Brown et al. "Adversarial patch. arxiv e-prints (dec. 2017)". In: *arXiv preprint cs.CV/1712.09665* 1.2 (2017), p. 4.

[157] Anish Athalye et al. "Synthesizing robust adversarial examples". In: *arXiv preprint arXiv:1707.07397* (2017).

[158] Hyrum S Anderson et al. “Evading machine learning malware detection”. In: *Black Hat* (2017).

[159] Hung Dang, Yue Huang, and Ee-Chien Chang. “Evading classifiers by morphing in the dark”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 119–133.

[160] Wei Yang et al. “Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps”. In: *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACM. 2017, pp. 288–302.

[161] Kathrin Grosse et al. “Adversarial examples for malware detection”. In: *European Symposium on Research in Computer Security*. Springer. 2017, pp. 62–79.

[162] Nicolas Papernot et al. “Distillation as a defense to adversarial perturbations against deep neural networks”. In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 582–597.

[163] Kathrin Grosse et al. “On the (statistical) detection of adversarial examples”. In: *arXiv preprint arXiv:1702.06280* (2017).

[164] Zhitao Gong, Wenlu Wang, and Wei-Shinn Ku. “Adversarial and clean data are not twins”. In: *arXiv preprint arXiv:1704.04960* (2017).

[165] Seyda Ertekin et al. “Learning on the border: active learning in imbalanced data classification”. In: *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*. ACM. 2007, pp. 127–136.

[166] Julio Hernandez, Jesús Ariel Carrasco-Ochoa, and José Francisco Martínez-Trinidad. “An empirical study of oversampling and undersampling for instance selection methods on imbalance datasets”. In: *Iberoamerican Congress on Pattern Recognition*. Springer. 2013, pp. 262–269.

[167] Tom Fawcett and Foster J Provost. “Combining Data Mining and Machine Learning for Effective User Profiling.” In: *KDD*. 1996, pp. 8–13.

[168] Kazuo J Ezawa, Moninder Singh, and Steven W Norton. “Learning goal oriented Bayesian networks for telecommunications risk management”. In: *ICML*. 1996, pp. 139–147.

[169] David D Lewis and Jason Catlett. “Heterogeneous uncertainty sampling for supervised learning”. In: *Proceedings of the eleventh international conference on machine learning*. 1994, pp. 148–156.

[170] Susan Dumais et al. “Inductive learning algorithms and representations for text categorization”. In: *Proceedings of the seventh international conference on Information and knowledge management*. ACM. 1998, pp. 148–155.

[171] Marko Grobelnik. "Feature selection for unbalanced class distribution and naive bayes". In: ICML. 1999.

[172] David D Lewis and Marc Ringuette. "A comparison of two learning algorithms for text categorization". In: *Third annual symposium on document analysis and information retrieval*. Vol. 33. 1994, pp. 81–93.

[173] William W Cohen. "Learning to classify English text with ILP methods". In: *Advances in inductive logic programming* 32 (1995), pp. 124–143.

[174] Miroslav Kubat, Robert C Holte, and Stan Matwin. "Machine learning for the detection of oil spills in satellite radar images". In: *Machine learning* 30.2-3 (1998), pp. 195–215.

[175] Jerzy Błaszczyński and Jerzy Stefanowski. "Neighbourhood sampling in bagging for imbalanced data". In: *Neurocomputing* 150 (2015), pp. 529–542.

[176] Bartosz Krawczyk. "Learning from imbalanced data: open challenges and future directions". In: *Progress in Artificial Intelligence* 5.4 (2016), pp. 221–232.

[177] Michael Pazzani et al. "Reducing misclassification costs". In: *Proceedings of the Eleventh International Conference on Machine Learning*. 1994, pp. 217–225.

[178] Pedro Domingos. "Metacost: A general method for making classifiers cost-sensitive". In: *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 1999, pp. 155–164.

[179] Nathalie Japkowicz. "The class imbalance problem: Significance and strategies". In: *Proc. of the Int'l Conf. on Artificial Intelligence*. 2000.

[180] Charles X Ling and Chenghui Li. "Data mining for direct marketing: Problems and solutions." In: *KDD*. Vol. 98. 1998, pp. 73–79.

[181] Edward Raff and Charles Nicholas. "Malware Classification and Class Imbalance via Stochastic Hashed LZJD". In: *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*. ACM. 2017, pp. 111–120.

[182] Jason Van Hulse, Taghi M Khoshgoftaar, and Amri Napolitano. "Experimental perspectives on learning from imbalanced data". In: *Proceedings of the 24th international conference on Machine learning*. ACM. 2007, pp. 935–942.

[183] Rehan Akbani, Stephen Kwek, and Nathalie Japkowicz. "Applying support vector machines to imbalanced datasets". In: *Machine learning: ECML 2004* (2004), pp. 39–50.

[184] Miroslav Kubat, Stan Matwin, et al. "Addressing the curse of imbalanced training sets: one-sided selection". In: *Icml*. Vol. 97. Nashville, USA. 1997, pp. 179–186.

[185] Hui Han, Wen-Yuan Wang, and Bing-Huan Mao. "Borderline-SMOTE: a new over-sampling method in imbalanced data sets learning". In: *International conference on intelligent computing*. Springer. 2005, pp. 878–887.

[186] Chumphol Bunkhumpornpat, Krung Sinapiromsaran, and Chidchanok Lursinsap. "Safe-level-smote: Safe-level-synthetic minority over-sampling technique for handling the class imbalanced problem". In: *Pacific-Asia conference on knowledge discovery and data mining*. Springer. 2009, pp. 475–482.

[187] Taeho Jo and Nathalie Japkowicz. "Class imbalances versus small disjuncts". In: *ACM Sigkdd Explorations Newsletter* 6.1 (2004), pp. 40–49.

[188] Hansang Lee, Minseok Park, and Junmo Kim. "Plankton classification on imbalanced large scale database via convolutional neural networks with transfer learning". In: *2016 IEEE international conference on image processing (ICIP)*. IEEE. 2016, pp. 3713–3717.

[189] Inderjeet Mani and I Zhang. "kNN approach to unbalanced data distributions: a case study involving information extraction". In: *Proceedings of workshop on learning from imbalanced datasets*. Vol. 126. 2003.

[190] Ricardo Barandela et al. "The imbalanced training sample problem: Under or over sampling?" In: *Joint IAPR international workshops on statistical techniques in pattern recognition (SPR) and structural and syntactic pattern recognition (SSPR)*. Springer. 2004, pp. 806–814.

[191] Dennis L Wilson. "Asymptotic properties of nearest neighbor rules using edited data". In: *IEEE Transactions on Systems, Man, and Cybernetics* 3 (1972), pp. 408–421.

[192] Shoujin Wang et al. "Training deep neural networks on imbalanced data sets". In: *2016 international joint conference on neural networks (IJCNN)*. IEEE. 2016, pp. 4368–4374.

[193] Justin M Johnson and Taghi M Khoshgoftaar. "Survey on deep learning with class imbalance". In: *Journal of Big Data* 6.1 (2019), p. 27.

[194] Tsung-Yi Lin et al. "Focal loss for dense object detection". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2980–2988.

[195] Keisuke Nemoto et al. "Classification of rare building change using cnn with multi-class focal loss". In: *IGARSS 2018-2018 IEEE International Geoscience and Remote Sensing Symposium*. IEEE. 2018, pp. 4663–4666.

[196] Haishuai Wang et al. "Predicting hospital readmission via cost-sensitive deep learning". In: *IEEE/ACM transactions on computational biology and bioinformatics* 15.6 (2018), pp. 1968–1978.

[197] Yulu Zhang et al. "Image classification with category centers in class imbalance situation". In: *2018 33rd Youth Academic annual conference of Chinese Association of Automation (YAC)*. IEEE. 2018, pp. 359–363.

[198] Wan Ding et al. "Facial action recognition using very deep networks for highly imbalanced class distribution". In: *2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*. IEEE. 2017, pp. 1368–1372.

[199] Chen Huang et al. "Learning deep representation for imbalanced classification". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 5375–5384.

[200] Ziwei Liu et al. "Large-scale celebfaces attributes (celeba) dataset". In: *Retrieved August* 15 (2018), p. 2018.

[201] Qi Dong, Shaogang Gong, and Xiatian Zhu. "Imbalanced deep learning by minority class incremental rectification". In: *IEEE transactions on pattern analysis and machine intelligence* 41.6 (2018), pp. 1367–1381.

[202] Lei Cen et al. "A probabilistic discriminative model for android malware detection with decompiled source code". In: *IEEE Transactions on Dependable and Secure Computing* 12.4 (2015), pp. 400–412.

[203] Borja Sanz et al. "Puma: Permission usage to detect malware in android". In: *International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions*. Springer. 2013, pp. 289–298.

[204] Rajvardhan Oak et al. "Malware Detection on Highly Imbalanced Data through Sequence Modeling". In: *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*. 2019, pp. 37–48.

[205] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[206] *Palo Alto Networks. 2019*. Accessed: May 31, 2020. URL: https://www.paloaltonetworks.com/products/secure-the-network/wildfire.

[207] Songqing Yue. "Imbalanced malware images classification: a CNN based approach". In: *arXiv preprint arXiv:1708.08042* (2017).

[208] Zhenxiang Chen et al. "Machine learning based mobile malware detection using highly imbalanced network traffic". In: *Information Sciences* 433 (2018), pp. 346–364.

[209] Vasileios Kouliaridis et al. "A survey on mobile malware detection techniques". In: *IEICE Transactions on Information and Systems* 103.2 (2020), pp. 204–211.

[210] *Number of android applications*. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. Accessed: February 17, 2020.

[211] Kathy Wain Yee Au et al. "Pscout: analyzing the android permission specification". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 217–228.

[212] *Android debug bridge*. https://developer.android.com/studio/command-line/adb.html. Accessed: April 16, 2020.

[213] *strace command*. https://linux.die.net/man/1/strace. Accessed: May 17, 2020.

[214] Robert Tarjan. "Depth-first search and linear graph algorithms". In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.

[215] Mu Zhang et al. "Semantics-aware android malware classification using weighted contextual api dependency graphs". In: *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 2014, pp. 1105–1116.

[216] *Curse of dimensionality*. Accessed: July 14, 2020. URL: https://medium.com/@cxu24/dd60b5611543.

[217] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN: 1-55860-238-0.

[218] Jorge Blasco and Thomas M Chen. "Automated generation of colluding apps for experimental research". In: *Journal of Computer Virology and Hacking Techniques* 14.2 (2018), pp. 127–138.

[219] Nicolas Papernot et al. "The limitations of deep learning in adversarial settings". In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, pp. 372–387.

[220] John C. Platt. "Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods". In: *ADVANCES IN LARGE MARGIN CLASSIFIERS*. MIT Press, 1999, pp. 61–74.

[221] S Sathiya Keerthi and Chih-Jen Lin. "Asymptotic behaviors of support vector machines with Gaussian kernel". In: *Neural computation* 15.7 (2003), pp. 1667–1689.

[222] Salman H Khan et al. "Cost-sensitive learning of deep feature representations from imbalanced data". In: *IEEE transactions on neural networks and learning systems* 29.8 (2017), pp. 3573–3587.