Northumbria University
NEWCASTLE

ANDROID MALWARE DETECTION USING MACHINE LEARNING TO MITIGATE ADVERSARIAL EVASION ATTACKS

HUSNAIN RAFIQ

PhD

2022

ANDROID MALWARE DETECTION USING MACHINE LEARNING TO MITIGATE ADVERSARIAL EVASION ATTACKS

HUSNAIN RAFIQ

A thesis submitted in partial fulfilment of the requirements of the University of Northumbria at Newcastle for the degree of Doctor of Philosophy

Department of Computer & Information Sciences

December 2022

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original. I also declare that this work has not been submitted in whole or in part for consideration for any degree or qualification, to this or any other university. This dissertation is my work and to the best of my knowledge, does not contain content which breaks any law of copyrights, except as specified in the text and acknowledgments.

Name: Husnain Rafiq

Date: 7 December 2022

Dedications

I want to dedicate this thesis to my parents Muhammad Rafiq and Mussarat Rafiq, my wife Sana Husnain, siblings, and my dear friend Irfan Ayub who always supported me during this journey.

Acknowledgements

I would like to thank Prof Dr Nauman Aslam and Dr Biju Issac for their valuable guidance, support and encouragement at every step. Without their help, this research work would not have been possible. I would also like to thank Prof Dr Muhammad Aleem, who motivated me to pursue PhD after completion of my master's degree.

I would like to extend my gratitude to Rizwan Hamid Randhawa and Dr Muhammad Khalid for their support during the paper submission procedures. I would also like to thank my colleagues Rana Faisal Shahzad and Mohammad AlJaidi for their constant support and valuable discussions.

Abstract

In the current digital era, smartphones have become indispensable. Over the past few years, the exponential growth of Android users has made this operating system (OS) a prime target for smartphone malware. Consequently, the arms race between Android security personnel and malware developers seems enduring. Considering Machine Learning (ML) as the core component, various techniques are proposed in the literature to counter Android malware, however, the problem of adversarial evasion attacks on ML-based malware classifiers is understated. ML-based techniques are vulnerable to adversarial evasion attacks. The malware authors constantly try to craft adversarial examples to elude existing malware detection systems. This research presents the fragility of ML-based Android malware classifiers in adversarial environments and proposes novel techniques to counter adversarial evasion attacks on ML-based Android malware classifiers.

First, we start our analysis by introducing the problem of Android malware detection in adversarial environments and provide a comprehensive overview of the domain. Second, we highlight the problem of malware clones in popular Android malware repositories. The malware clones in the datasets can potentially lead to biased results and computational overhead. Although many strategies are proposed in the literature to detect repackaged Android malware, these techniques require burdensome code inspection. Consequently, we employ a lightweight and novel strategy based on package names reusing to identify repackaged Android malware and build a clones-free Android malware dataset. Furthermore, we investigate the impact of repacked Android malware on various ML-based classifiers by training them on a clones-free training set and testing on a set of benign, non-repacked malware and all the malware clones in the dataset. Although trained on a reduced train set, we achieved up to 98.7% F1 score. Third, we propose Cure-Droid, an Android malware classification model trained on hybrid features and optimized using a tree-based pipeline optimization technique (TPoT). Fourth, to present the fragility of Cure-Droid model in adversarial environments, we formulate multiple adversarial evasion attacks to elude the model. Fifth, to counter adversarial evasion attacks on ML-based Android malware detectors, we propose CureDroid*, a novel and adversarially aware Android malware classification model. CureDroid* is based on an ensemble of ML-based models trained on distinct set of features where each model has the individual capability to detect Android malware. The CureDroid* model employs an ensemble of five ML-based models where each model is selected and optimized using TPoT. Our experimental results demonstrate that CureDroid* achieves up to 99.2% accuracy in non-adversarial settings and can detect up to 30 fabricated input features in the best case. Finally, we propose TrickDroid, a novel cumulative adversarial training framework based on Oracle and GAN-based adversarial data. Our experimental results present the efficacy of TrickDroid with up to 99.46% evasion detection.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Overview

As smartphones grow in usage, they are becoming an essential part of our daily lives. A recent report revealed that US cell phone users spend over four hours and twenty minutes on their phones each day [1]. The reason being, smartphones are crucial in a number of ways, including communication, navigation, access to the internet, capturing experiences, storing data or information and carrying out financial transactions. Smartphones have equipped users with various applications and generate a vast amount of sensitive data. Therefore, the security of these devices is of prime concern.

Android dominates the mobile OS industry with more than 70% market share and over 2.8 billion active users which is significantly more than iPhone Operating System (iOS) or any other mobile OS [2]. Apart from official Google play store, Android users can download apps from third party app stores such as Samsung, Amazon and F-Droid. According to a recent report [3], Google play store hosts more than 2.56 million Android applications (apps) with 142.9 billion downloads in 2020 alone. Before an app is published on Google play stores, it is screened for potential malware threats with a dynamic emulator environment called Google bouncer. In spite of Google play store security, malware developers still find ways to manipulate it by exploiting the vulnerabilities in the system. Furthermore, the open source nature of Android also makes it possible for malware developers to upload malicious apps to the third party app stores. Consequently, with the rise in Android OS usage, it has become popular amongst the smartphone malware authors. Malware developers employ various techniques such as encryption [4], code obfuscation [5], dynamic code upload [6] and application repackaging [7] to evade the existing Android security mechanism.

Android security analysts are constantly working to address the issue of Android malware propagation. Most of the existing anti-virus systems rely on signature-based detection [8], [9], [10]. A typical signature-based detection model analyses multiple segments of Android apps based on network traffic, strings, byte code sequences and file hashes to locate patterns similar to known malware samples. However, signature based detection is very fragile against evasion attacks. An attacker can easily bypass a signature based detection technique by minimal effort such as code encryption and application re-packaging [11]. On the other hand, behaviour-based detection techniques tend to be more robust than signature-based detection models. Techniques based on behaviour-based detection employ ML and deep learning (DL) to detect malware in Android apps. Over the past decade, ML and DL has been widely used by researchers to classify malware and benign apps to mitigate Android malware attacks [12], [13], [14]. The ML and DL models are trained on various features extracted from Android apps such as permissions, Application Programming Interfaces (APIs), intents, network traffic, power usage and software components to classify malware and benign apps. Numerous studies in the current literature have demonstrated that advanced ML/DL-based defenses are not only capable of classifying large amounts of malware and benign apps, but are also capable of detecting novel malware samples [15].

1.2 Problem Statement

A recent survey by *AV-Test* reports that the Android malware development peaked from 2015 to 2018, followed by a decreasing trend since 2018 [16]. Nonetheless, the amount of Android malware is still tremendous with over 2.64 million new Android malware developed in 2021 [16]. The phenomenal growth of Android malware is due to ease of malware cloning techniques such as application repackaging. Machine learning has been utilized extensively in Android malware classification tasks; however, the reliability of such techniques in adversarial environments is of key concern. Interestingly, ML-based malware classification models can be easily manipulated by using adversarial examples during training or testing phase [17], [18], [19]. The adversarial attacks performed during training are known as poisoning attacks, whereas the adversarial attacks performed during the testing phase refer to evasion attacks. To perform poisoning attacks, the adversary corrupts the training data in order to compromise the training process. On the other hand, evasion attacks are performed to evade the underlying ML model by carefully crafting a malicious sample such that the model miss-classifies it during the testing phase. The evasive samples are usually created by code obfuscation and injecting or removing discriminating features from the Android app [20].

Many ML-based solutions are proposed in the literature to defend against Android malware. Although most of the techniques proposed in the literature report remarkable results, the security of ML-based malware detectors is of concern. Since the critical assumption behind ML is that the data used for the training phase represents the problem domain and deliberate modifications of data do not occur [21]; therefore, models built using ML are vulnerable to adversarial attacks. Consequently, an ideal malware detector must be able to classify the adversarial examples correctly. The other significant aspect discussed in this research is emphasising the problem of repacked malware in the Android malware datasets. Repacked/cloned samples in malware repositories can potentially lead to biased classification results and computational expense in terms of the feature extraction process [22] [23]. This research is an effort towards developing

ML-based accurate and adversarially-aware Android malware detection techniques trained on clones-free datasets.

1.3 Research Questions

Inspired to address the challenges in developing accurate and adversarially robust Android malware detection techniques, the following research questions are formulated:

- **RQ1:** How often do malware samples in benchmark Android malware repositories re-use package names, and can we consider them as clones/repacked versions of known malware?
- **RQ2:** What is the impact of repacked malware on ML-based Android malware classifiers?
- **RQ3:** How fragile are ML-based Android malware classifiers against adversarial evasion attacks?
- **RQ4:** How can we harden the security of ML-based Android malware classifiers against adversarial evasion attacks?

1.4 Research Contributions

The novel research contributions of this research are as follows:

- Research questions RQ1 and RQ2 are addressed. We first quantify the potential clones of known malware within three benchmark Android malware datasets (Drebin [24], AMD [25] and Androzoo [26]). We employ a simple yet effective method of comparing the package names of the samples under observation with known malicious package names. Furthermore, we propose the *AndroMalPack* model to investigate the impact of repacked apps with identical package names on various machine learning models. *AndroMalPack* extracts permissions, APIs and Intent-based features from Android apps in the datasets to train the machine learning models. *AndroMalPack* removes all the repacked malware samples (based on package name reusing) from the training sets however, retains the repacked malware in the test sets to measure the effectiveness of ML models. Although detecting repacked malware based on package names is a lightweight approach and can be easily evaded (by renaming the package), our target in this work is to quantify existing clones in the benchmark datasets rather than detecting novel clones. *AndroMalPack* selects the best performing ML model (Random Forrest) on reduced datasets and further tunes the hyper-parameters by employing nature inspired algorithms (NIAs) to achieve improved results. Finally, we publish a comprehensive dataset of repacked apps based on the identical package names in the Drebin, AMD, and AndroZoo datasets to foster the research on the analysis of repacked Android malware.

- Research questions RQ3 and RQ4 are addressed. The primary concern is to highlight and address the vulnerability of ML-based Android malware classifiers against adversarial evasion attacks. First, we propose *CureDroid*, an Android malware classifier trained on hybrid features and optimised using Tree-based pipeline optimization technique (TPoT). Although *CureDroid* achieves a remarkable classification accuracy, we present the fragility of the proposed model in adversarial settings. We design and develop mimicry attacks, feature removal attacks, and feature injection in conjunction with feature elimination attacks to evade *CureDroid*. Furthermore, to counter evasion attacks, we propose *CureDroid**, a novel and adversarially robust extension of *CureDroid*. *CureDroid** employs multiple representations of discriminating feature subsets from an extensive array of various features rather than using one large feature set. The primary motivation for using the single feature vector source partitioned into various subsets (APIs, intents, system calls and permissions-based subsets of features) is to enable each representation to contribute to training on separate classifiers. Consequently, multiple classifiers trained on different Android app feature representations makes the attacker's job challenging to evade the malware detection model.

- Research question RQ4 is further addressed by exploring the effectiveness of adversarial training as a defence strategy to counter adversarial evasion attacks on Android malware detectors. First a case study is developed to evade multiple ML-based classifiers trained on API calls-based features of Android apps. We employ mimicry attacks based on Oracle and Generative Adversarial Networks (GANs)-based adversarial data to evade various ML-based classifiers. Furthermore, we propose a novel and robust adversarial training scheme called *TrickDroid* based on cumulative adversarial training of various ML-based classifiers on Oracle and GAN based adversarial data to improve evasion detection.

1.5 Thesis Structure

- **Chapter 2** provides a discussion on the background of the Android malware analysis and detection domain in terms of the related works. First, we analyse the ML-based Android malware detection techniques based on static, dynamic and hybrid analysis. Secondly, we categorize the adversarial attacks based on target classifiers/techniques in Android malware detection domain. We then investigate the defence proposed in literature against adversarial attacks on Android malware detectors.

- **Chapter 3** proposes *AndroMalPack*. Section 3.2 presents a brief background about Android application repackaging. Section 3.3 provides the motivation for the study, Section 3.4 presents the details of datasets used and the statistics about potential repacked malware based on package names reusing. Section 3.5 presents *AndroMalPack*, an Android malware classifier trained on clones free train sets and optimised using nature-inspired algorithms. The experimental results are demonstrated in Section 3.6 and comparison with related work is discussed in Section 3.7. Section 3.8 presents details about the published dataset, and Section 3.9 concludes the chapter.

- **Chapter 4** proposes *CureDroid**, an ensemble classifiers-based Android malware classification model to mitigate adversarial evasion attacks. Section 4.2 discuses the motivation for the study and Section 4.3 presents the details of hybrid featured dataset and feature extraction process. Section 4.4 provides the details of the proposed CureDroid model and evaluation results, section 4.5 gives a description of proposed adversarial evasion attacks on CureDroid. Section 4.6 presents the CureDroid* model and an analysis of the results of CureDroid*. Section 4.7 discusses the related work and Section 4.8 concludes the chapter.

- **Chapter 5** proposes *TrickDroid*, a robust cumulative scheme for adversarial training. Section 5.1 provides a comprehensive background of adversarial training techniques in machine learning domain. Section 5.2 presents the proposed attacks methodology and experimental results are discussed in Section 5.3. Section 5.4 concludes the chapter.
- **Chapter 6** concludes this thesis by discussing potential future work and open challenges.

1.6 Research Publications

Journal Papers:

[1] Husnain Rafiq, Nauman Aslam, Usman Ahmed, and Jerry Chun-Wei Lin. ”Mitigating Malicious Adversaries Evasion Attacks in Industrial Internet of Things,” in IEEE Transactions on Industrial Informatics, vol. 19, no. 1, pp. 960-968, Jan. 2023, doi: 10.1109/TII.2022.3189046.

[2] Husnain Rafiq, Nauman Aslam, Muhammad Aleem, Biju Issac and Rizwan Hamid Randhawa. ”AndroMalPack: enhancing the ML-based malware classification by detection and removal of repacked apps for Android systems”. Sci Rep 12, 19534 (2022). https://doi.org/10.1038/s41598-022-23766-w

Conference Papers:

[1] Husnain Rafiq, Nauman Aslam, Biju Issac, and Rizwan Hamid Randhawa. ”An Investigation on Fragility of Machine Learning Classifiers in Android Malware Detection,” IEEE INFOCOM 2022 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS),

2022, pp. 1-6, doi: 10.1109/INFOCOMWKSHPS54753.2022.9798161.

[2] Husnain Rafiq, Nauman Aslam, Biju Issac and Rizwan Hamid Randhawa. ”On Impact of Adversarial Evasion Attacks on ML-based Android Malware Classifier Trained on Hybrid Features.” In 14th International Conference on Software, Knowledge, Information Management and Applications (SKIMA), 2022. (Accepted)

Chapter 2

Background

2.1 Introduction

The application of machine learning has proven to be very effective in malware detection due to remarkable classification accuracy and the ability to identify novel samples. In terms of Android malware detection, numerous characteristics such as permission, intents, system calls, API calls, and strings are extracted from Android applications and incorporated into feature vectors. Furthermore, ML-based algorithms are trained on these feature vectors to predict the labels of unknown data. A generic model representing the operation of any typical ML-based Android malware classification system is depicted in Figure 2.1. Although ML-based malware detectors can accurately categorize the malware and benign apps more than even a malware security analyst, these techniques are vulnerable to maliciously crafted inputs. Therefore, the adversary can purposefully fabricate malicious input in motivation to evade the ML-based classification model. The strategies employed to deliberately mislead ML-based classification models are called adversarial machine learning.

Due to the numerous applications of ML in various fields, a plethora of attack mechanisms are proposed by researchers in order to demonstrate the fragility of ML-based classifiers [27],

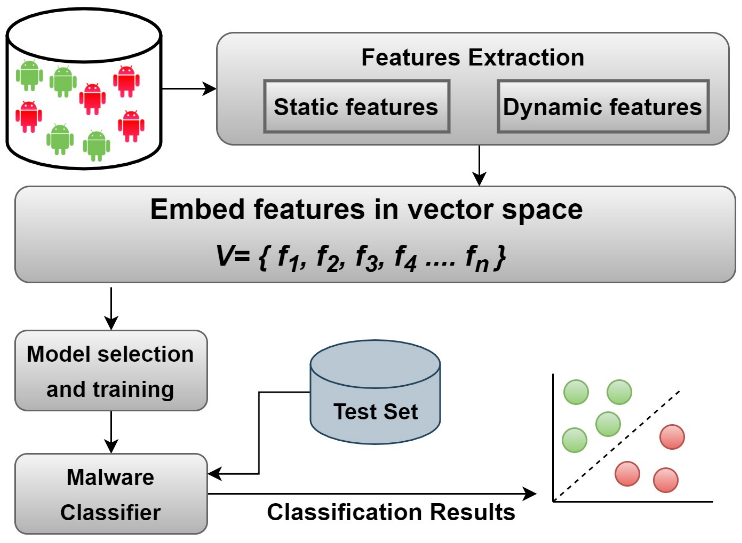


Figure 2.1: Generic model of ML-based Android malware classification

[28]. For example, in the field on computer vision, the authors in [29] demonstrated that a medical image of a benign tumour could be easily manipulated to be classified as a malignant tumour by injecting carefully crafted distortion. Likewise, the authors in [30] argue that medical image-based classifiers are easier to evade than natural image-based classifiers. The reason is that medical images generally have significant discriminating regions regarding class distribution. Consequently, a small perturbation in high attention segments can lead to misclassification. Similarly, in the case of image processing in autonomous vehicles, an adversary can craft a sample such that it may appear to be a stop sign to the human eye. In contrast, the ML model can be misled into classifying it as a passing sign [31]. Furthermore, the authors in [32] and [33] demonstrated the adversarial attacks on learning-based PDF malware detectors to present the vulnerabilities in ML-based learning models. In the case of the Windows operating system, a recent survey by [28] presented an extensive review of practical adversarial attacks on ML-based Windows malware detectors.

This work mainly focuses on state-of-the-art techniques for ML-based Android malware detection. We explore various malware detection techniques, adversarial attacks and potential defence mechanisms proposed in the literature related Android malware domain. In summary following are the main contributions of this chapter:

- An in-depth analysis of the Android operating system and the structure of the Android Application Package (APK) files is presented.
- In order to develop a taxonomy of the methodologies, we investigate the state-of-the-art approaches relevant to the Android malware detection domain.
- A comprehensive review of the characteristics of Android applications used for ML-based Android malware detection is provided.
- We investigate the reliability of ML-based Android malware detection in adversarial environments, considering the level of expertise possessed by the adversary.
- We categorize the strategies used for adversarial attacks against ML-based Android malware classifiers.
- We survey and categorize the adversarial defences proposed in the literature to counter adversarial evasion attacks on Android malware classifiers.

2.2 Overview of Android Ecosystem

This section provides a brief background information on Android OS architecture, application structure. naming conventions and APK reverse engineering process.

2.2.1 Android OS Architecture

Android is an open-source operating system based on the Linux kernel and is compatible with a wide range of smart devices such as smartphones, television sets, smartwatches and tablets. Figure 2.2 presents the standard architecture of Android OS. Since Android OS is based on the open-source Linux kernel, developers can add new features, employ custom optimization, and perform extreme configurations. Unlike the Linux kernel, which only allows system calls, the

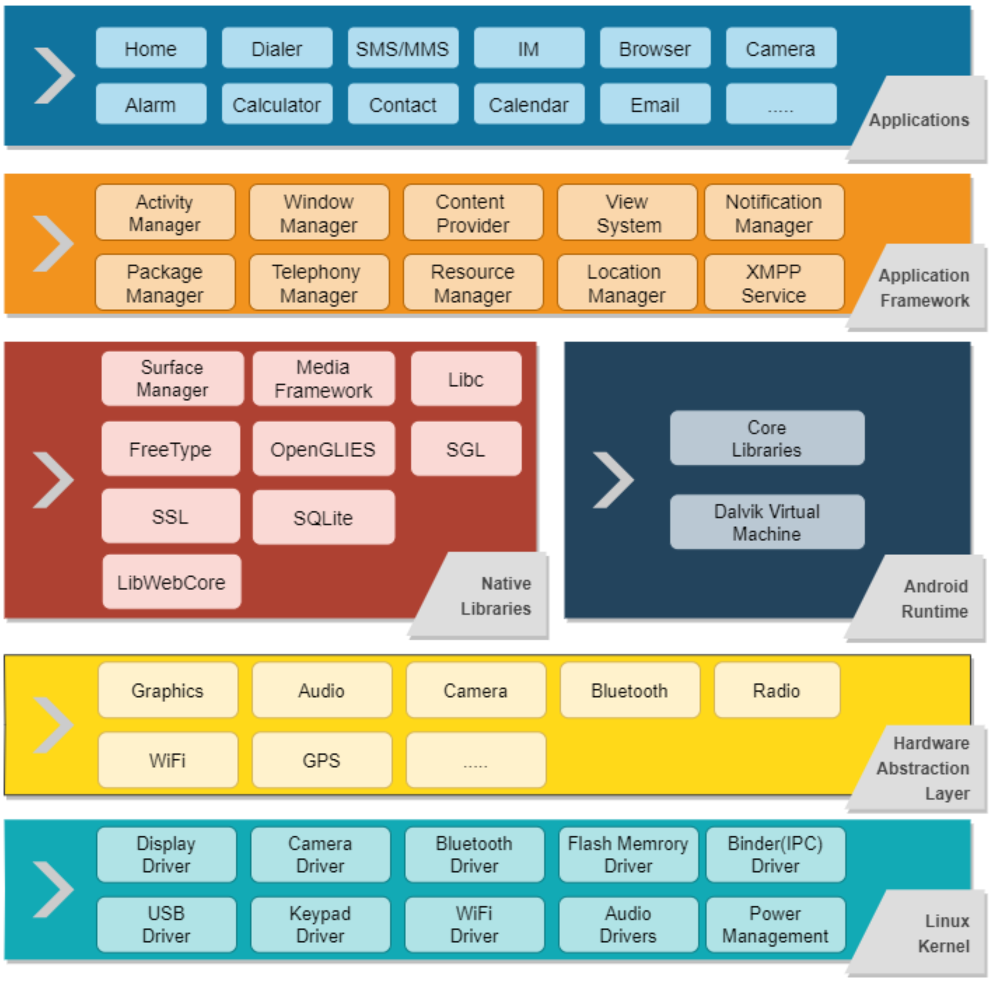


Figure 2.2: Android OS Architecture

Android framework communicates with underlying hardware through APIs. Consequently, a Hardware abstraction layer (HAL) is implemented over the Linux kernel in order to establish an interface for hardware and software communication in Andriod OS design. Android runtime and native libraries are introduced on top of the HAL layer in Android OS architecture. Furthermore, the Android runtime environment includes Dalvik Virtual Machine (DVM) and essential libraries. DVM leverages core features of the Linux kernel, such as memory management and threading, to ensure that many instances of Android apps run effectively. The native libraries layer provides a set of libraries that enable access to the core features of Android OS, such as media, SSL, and SQLlite. The next layer in the Android OS architecture stack is the Android application framework which provides Android developers with access to a wide range of powerful Android APIs. Android APIs are a collection of classes and packages that allow developers to communicate with Android OS functions such as SMS management, telephony, and Bluetooth. Finally, the top layer of the Android OS stack includes specific pre-installed system applications such as email, alarm, browser and clock. Moreover, any additional third-party apps installed through the official Google play store or third-party app stores are also included in the application layer.

2.2.2 Android Application Structure

Android applications are generally written in Java programming language and published in a compressed format known as an Android application package (APK). Smartphone users based on Android OS can directly download and install apps from the official app store (Google play store) or other third-party app stores such as Amazon and AppChina. Generally, Android APK is made up of the following components:

- **Dalvik byte code:** Android applications are developed in Java and then compiled as *.class* files (Dalvik byte code). The *.class* files are compressed into a single Dalvik executable file named *classes.dex*, which is ultimately executed on the DVM.
- **Manifest file:** Android application consists of an *AndroidManifest.xml* file that contains important information about the app's components and structure. The *AndroidManifest.xml* file contains information about the primary package name, permissions required by the app, hardware components accessed by the app, activities, broadcast receivers, services, intent filers, and software features required by the app.
- **Resources:** It contains all of the necessary resources required by the app such as pictures, animations, layouts, and user interface. At build time, all of the needed resources are compiled into the app.

- **Libraries:** This folder contains all compiled external libraries that are utilised in the application.
- **Signatures:** All Android apps have to be digitally signed before they can be published on an app store. A digital signatures refers to a cryptographic hash which represents the app developer.

2.2.3 Application Naming Conventions

Every Android app on the app store must have an app name and a unique package name. App name refers to the app's title that appears on the app store. It is not requisite for an app to have a unique name as multiple apps on the Android official app store can be found sharing the same name. On the other hand, the Android package name is the unique identity of an app which is defined in *AndroidManifest.xml* file. Usually, the package name is the name of the base package, which is created when the app is developed. The base package can have further sub-packages containing java classes and activities. No two apps installed on the same device can have identical package names [34]. If two apps with identical package names are installed on the same device, the latter will override the previous one as an updated version. Malware authors frequently upload cloned apps with the same package names and slight modifications to trick antivirus systems which rely on hash-based detection.

2.2.4 Reverse Engineering

Android application package (APK) is a zipped archive that contains *classes.dex* file, *Android-Manifest.xml* file, resources and libraries in compressed form. The zip archive contents are not human-readable as the java classes are compressed into a single Dalvik executable (*.dex*) file. However, it is possible to reverse engineer an APK file to extract java source code and corresponding files using several tools. Figure 2.3 explains the steps required to reverse engineer an APK. APK tool is used to unzip the APK file to extract *classes.dex* file. The *classes.dex* file is then decompiled in form of java-archive *.jar* file by using *dex2jar* tool. The *.jar* produced by *dex2jar* tool contain java byte code in form of *.class* files which are still not human-readable. Finally, a java decompiler tool such as JAD is used to decompile the *.class* files in the form of java source code.

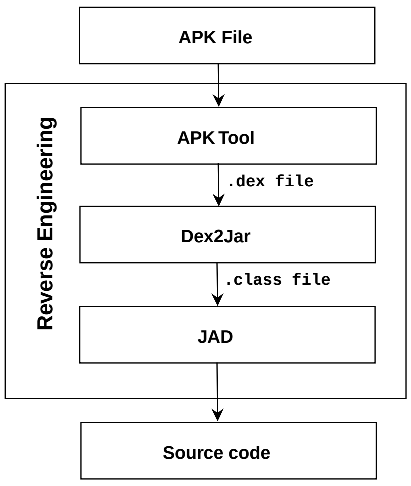


Figure 2.3: Android Application Package (APK) Reverse Engineering

Reverse engineering of an Android application is typically carried out while performing static analysis of an Android application. After the app has been reverse engineered, the malware analyst can access the app's source code and the *manifest.xml* file to retrieve various app characteristics. An Android application's source code is typically parsed to obtain data and features like strings, network addresses, API calls, and function call graphs. In contrast, it is possible to extract static features like permissions, filtered intents, package names, services, receivers and hardware components from the *manifest.xml* file of an Android app.

2.3 Android Malware Analysis Techniques

Malware analysis is broadly classified into three major types 1) static analysis, 2) dynamic analysis, and 3) hybrid analysis. This section presents a taxonomy of Android malware analysis techniques (Figure 2.4) with a brief description of each technique. Furthermore, we list the Android application features employed for malware detection based on static and dynamic malware analysis techniques. The orange highlighted parts in Figure 2.4 refer to the techniques employed in this work for Android malware analysis and detection.

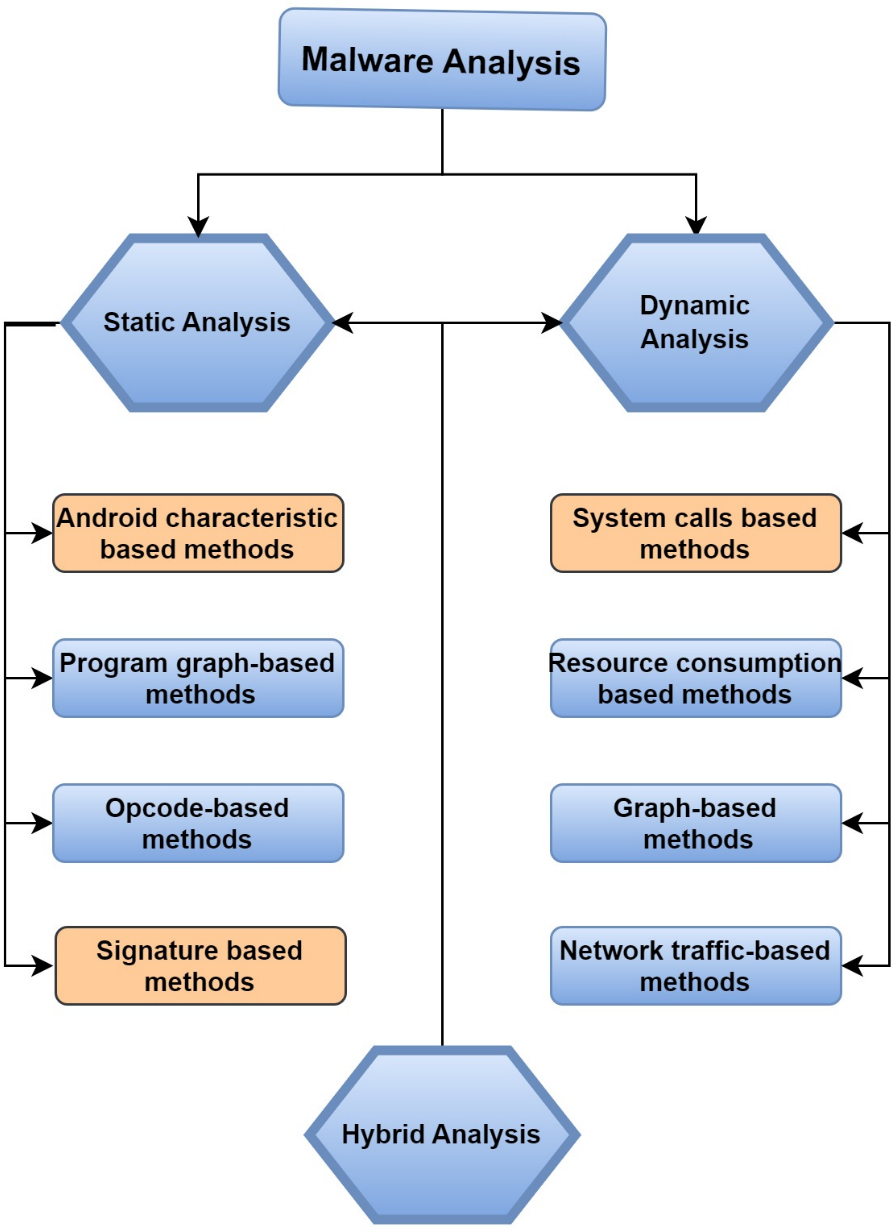

Malware Analysis
Static Analysis
Dynamic Analysis
Android characteristic based methods
Program graph-based methods
Opcode-based methods
Signature based methods
System calls based methods
Resource consumption based methods
Graph-based methods
Network traffic-based methods
Hybrid Analysis

Figure 2.4: Malware Analysis Taxonomy

2.3.1 Static Analysis

Static analysis is the technique of analyzing malicious files without executing the application. Static analysis involves analyzing an application's source code or binary code to detect potentially malicious activity. It allows an analyst to evaluate the static attributes of a malicious file, such as strings, hashes, signatures, and metadata, to find patterns similar to known malware. Since static analysis does not require to execute an application, it is inexpensive in terms of computational cost and time. Static analysis is widely used in the ML-based Android malware detection domain. Various discriminating features are extracted from malicious and benign Android apps, and are employed for training and testing ML-based algorithms. Table 2.1 presents the list of various features that can be extracted through static analysis to perform Android malware detection. Furthermore, in this sub-section, we briefly describe popular Android malware detection techniques based on static analysis.

Table 2.1: Overview of prominent dynamic features used in existing approaches

Features	**Source**	**Description**	**Ref.**
Permissions	*AndroidManifest.xml*	Special privileges which an app requires for execution.	[24], [35], [36], [37], [38], [39]
API calls	Source/byte code	API calls invoked by app extracted through static analysis.	[35], [40], [24], [36], [41], [42]
URLs	Source code	URLs embedded as strings in source code of an app.	[43], [24], [44]
Intents	*AndroidManifest.xml*	Intent objects define specific actions to be performed.	[45], [46], [24], [47]
Opcodes	Byte code	An instruction which define the operation to be performed.	[48], [49], [50], [51], [52], [53]
Strings	Source Code	Strings such as names of functions, classes, objects and variables.	[54], [55], [56], [57]
Package Name	*AndroidManifest.xml*	Name of the main package used to identify a specific Android application.	[58], [59], [60]
App components	*AndroidManifest.xml*	list of activities, services, receivers and providers and for each app component.	[24], [61], [62], [63]

2.3.1.1 Android characteristic-based method

In this method, multiple characteristics of Android apps are extracted using static analysis, and ML-based models are trained to categorize Android malware and benign apps. Characteristics of Android apps refer to features extracted from *AndroidManifest.xml* file such as permissions, intents and hardware components and features from source code such as API calls, n-grams and URLs. Drebin [24], one of the most cited works in the Android malware detection domain, employed a characteristics-based method for Android malware detection. Drebin performed static analysis to extract multiple features from Android apps such as APIs, permissions, intents and hardware components to train a linear Support Vector Machine (SVM) model in order to categorize Android malware and benign apps. The evaluation results of Drebin report 94% malware detection accuracy with a meagre false positive rate. Likewise authors in [35], [40], [54], [36], [37], [38], [39] and [64] identified Android malware by using characteristic-based methods.

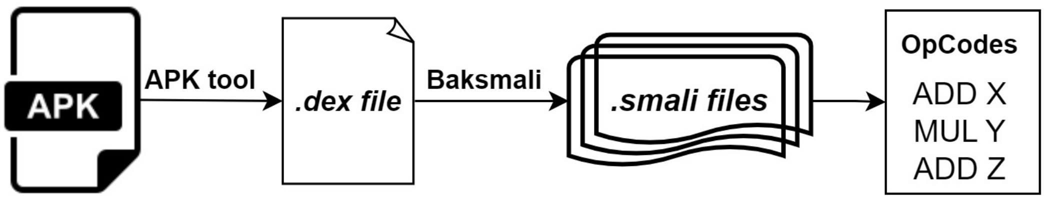


Figure 2.5: Opcodes Extraction from APK

2.3.1.2 Opcode-based method

Opcodes-based methods for Android malware detection use a sequence of opcodes extracted from Android apps to classify malicious and benign apps. The overview of opcode extraction from Android apps is presented in Figure 2.5. Android application is deployed as a compressed file (APK) which contains *AndroidManifest.xml* file, Dalvik executable (*.dex*) file and resources. The *.dex* file contains application byte code and can be disassembled to extract multiple *smali* files. Each *smali* file corresponds to a class in the form of assembly code generated by Dalvik virtual machine (DVM). The *smali* code consists of human-readable instructions where each instruction has opcodes and operands. The authors in [48] created a feature set by extracting n-grams from opcode sequences from Android apps and applied ML-based algorithms for malware detection. Yan et. al. [65] used Long Short-Term Memory (LSTM) technique to learn the contextual semantics from raw opcode sequences extracted from Android apps in order to identify malware. Furthermore, a recent study [53] demonstrated that opcode-based techniques for malware detection can be utilized to detect obfuscated Android malware.

2.3.1.3 Program graph-based method

Graph-based methods for Android malware detection are used in both static and dynamic analysis techniques, based on the graph extraction method. In static analysis, Control Flow Graphs (CFGs) are extracted from the source code of the Android app in order to detect malware. CFGs represent a program's hierarchical flow where a node refers to an instruction/statement, and an edge represents the directed flow from one node to another. Graph-based methods are widely employed in the Android malware detection domain as they can capture more semantic features from the apps than characteristic and opcode-based methods. Fan et al. [66] employed a graph-based method to construct frequent sub-graphs based on API calls to identify common behaviour between the same Android malware families. Similarly, [67] extracted application similarity graphs based on function calls and combined ML-based algorithms to detect Android malware. Consequently, they achieved up to 95.5% accuracy on various Android malware datasets. Yang et al. [68] proposed DGCNDroid, a deep graph convolutional network-based model by using API call sequence extracted through static analysis. The model was evaluated using 11,120 apps and achieved up to 98.2% malware detection accuracy.

2.3.1.4 Signature based methods:

Signature-based malware detection techniques consider footprints of different components of an app, such as methods and classes in order to compare them with signatures of known malware. Although signature-based techniques have a very low false positive rate, these methods are not robust against obfuscated malware, require regular updates and cannot detect zero-day malware samples. AndroSimilar [58] considered a similarity hashing-based method to generate signatures of Android apps in order to detect obfuscated malware samples; however, it only achieved 63% accuracy. Zheng et al. proposed DroidAnalytics [60], a multi-level signature generation technique to detect repacked Android malware. DroidAnalytics had significant benefits compared to traditional hash-based techniques (MD5, SHA1, SHA256) and could detect 2,475 malware samples belonging to 102 different families. Ngamwitroj et al. [69] employed a string sequence of permissions and broadcast receivers of an Android app as a signature to detect Android malware. They achieved 86.56% accuracy when tested on a 525 malicious and 122 benign Android apps dataset.

2.3.2 Dynamic Analysis

Dynamic analysis involves executing an application in order to detect potential malicious activity. Since executing a malicious application may harm the host device, it is generally performed in an isolated virtual environment [70] [71]. The dynamic analysis allows the analyst to study the behaviour of an app by analyzing the registry, file system, resource consumption, processes and network traffic in safe mode. Furthermore, dynamic analysis can capture the significant features of an Android app, such as instruction sequences, API calls sequence and system calls. These characteristics are further embedded into feature vectors to train ML-based classifiers. Table 2.2 presents the list of noticeable dynamic features used in various studies to detect Android malware. Since dynamic analysis requires the execution of an app in a controlled environment, it is more expensive than static analysis in terms of resource and time consumption. Following is a brief description of popular Android malware detection techniques based on dynamic analysis.

2.3.2.1 System calls based methods

As discussed in Section 2.2.1, all the apps in Android run on the application layer, while Android OS is based on the Linux kernel. Whenever an app requires access to the core functionality of the Linux kernel, such as power management or network connection, system calls are used

Table 2.2: Overview of prominent dynamic features used in existing approaches

Feature	**Description**	**Ref.**
Used permissions and API calls	Requested permissions and API calls traces extracted through dynamic analysis.	[72], [73], [74], [75]
Network traffic	These features include the data extracted from network traffic logs, HTTP communications and data from network packets.	[76], [77], [78], [79], [80]
System calls	System calls are invoked in order to access the core functionality of kernel. System call traces can be extracted through dynamic analysis.	[81], [82], [83], [84], [64], [85]
Battery life	Battery consumption of apps are profiled at runtime in order to find malicious patterns	[86], [87], [88]
CPU utilization	This feature refers to the CPU utilization of an app at runtime.	[89], [88], [90], [91]
Memory usage	This feature captures the memory usage of a particular app in order to find malicious patterns.	[88], [90], [89]

to shift the control from the application layer to the Linux kernel [81]. Likewise, control is returned to the application layer from the kernel mode once the required task has been completed. Canfora et al. [81] performed dynamic analysis to extract the sequence of system calls from the Android apps for malware detection. They employed ML-based models to learn the associations between system calls and achieved up to 97% malware detection accuracy on a dataset of 1000 benign and 1000 malicious Android apps. Similarly, Vidal et al. [82] employed a sequence of boot system calls to develop a pattern recognition system in order to detect Android malware. Consequently, they achieved up to 95.6% detection accuracy by evaluating Drebin [24] and Genome [92] Android malware datasets.

2.3.2.2 Graph-based methods

In dynamic analysis, graph-based techniques are used to generate graphs from Android applications during execution time in order to perform malware detection. The extracted graphs are further embedded into feature vectors, and ML algorithms are trained to categorize Android malware and benign apps. Lin et al. [93] employed dependency graphs based on the dynamic behaviour of an app as a feature to classify Android malware and benign apps. They used function calls and the data dependency between function calls to construct dependency graphs. They transformed the graphs into feature vectors in order to train a model based on SVM and correctly classified 213 out of 225 malware samples in their dataset. Pektaş et al. [94] generated API call graphs from Android apps by performing dynamic analysis and transformed the graphs into feature vectors to train a deep neural networks-based model. They evaluated the model on a balanced dataset of 58,139 Android malware and benign apps and obtained up to 98.65% accuracy. Surendran et al. [95] proposed GsDroid, a technique to represent Android apps as a directed graph of sequenced system calls and combined ML-based algorithms to learn from malicious patterns. GsDroid obtained up to 99% malware detection accuracy on various Android malware datasets.

2.3.2.3 Resource consumption-based methods

A compromised device may exhibit specific patterns in terms of power consumption, CPU utilization and memory usage; therefore, resource consumption-based methods consider these as a feature to detect potential malware. Hongyu and Tang [87] considered the power consumption of Android applications as a feature to detect malware. They profiled the power consumption of different categories of apps, where each app was monitored for 5 minutes. Based on the profiled data, they were able to detect 79 out of 100 malicious Android apps in the test set. Milosevic et al. [89] extracted execution traces of CPU utilization and memory usage of Android apps in order to detect malware and achieved 84% malware detection accuracy when evaluated on Genome malware dataset[92]. Similarly, Shehu et al. [88] employed CPU utilization, memory usage, battery and network consumption-based features of Android apps to generate fingerprints. They evaluated the proposed technique on a limited dataset of obfuscated versions of known malware and were able to detect all the obfuscated malware samples in the test set.

2.3.2.4 Network traffic-based methods

Network traffic-based methods capture the network data to detect an Android app's malicious behaviour. Malik et al. proposed CREDROID [76], a pattern-based malware detection method derived from network traffic logs. CREDROID analyzed the Android genome malware dataset [92] and found that 63% of the malware samples in the dataset generate network traffic. Wang et al. [77] considered the HTTP communication flows of Android apps in the form of text and applied Natural language processing ((NLP) on text to develop an Android malware detection model. They evaluated the model on a 31,706 benign and 5258 malicious Android apps dataset and achieved 99.15% malware detection accuracy. A subsequent study by Wang et al. [78] considered the URL addresses in the HTTP traffic of Android apps for malware detection. They employed multi-view neural networks trained on URL data to formulate a malware detection model and achieved 98% accuracy on the test set.

2.3.3 Hybrid Analysis

The hybrid analysis incorporates a combination of static and dynamic analysis techniques to extract various features from Android apps in order to detect malware. Static analysis is lightweight and effective in detecting novel and repacked malware; however, it may fail in case of code obfuscation and dynamic code loading. On the other hand, dynamic analysis captures the behaviour of an app at run time. It is more effective in detecting evasive malware but requires a controlled environment and is a time-intensive task. Consequently, hybrid analysis is employed to overcome the shortcomings of static and dynamic malware detection techniques. Kouliaridis et al. proposed Androtomist [96], an automated tool based on hybrid analysis to monitor the behaviour of Android apps. Androtomist employed ensemble classifiers trained on a combination of static and dynamic features of benign and malicious Android apps. When evaluated on three different Android malware datasets, Androtomist outperformed various existing approaches for Android malware detection.

Arshad et al. proposed SAMADroid [97] a three-level Android malware detection technique based on hybrid analysis. SAMADroid employed neural networks trained on a combination of static and dynamic features and claimed malware detection accuracy of 99.07% on the Drebin dataset [24]. Similarly, Tuan et al. proposed eGSDroid [98], a two-level Android malware detection method based on hybrid analysis. The first level considered static analysis to indicate sensitive components of Android apps, whereas the second stage employed dynamic analysis to detect potential data leakage from sensitive components of apps. Although eDSDroid achieved promising results, the evaluation was performed on a limited dataset of toy apps instead of real-world applications. Maryam et al. proposed cHybriDroid [99], an Android malware classifier based on the conjunction of static and dynamic features of Android apps. They employed tree-based pipeline optimization technique (TPOT) [100] to formulate malware detection model and achieved up to 96% malware detection accuracy on Drebin dataset [24].

2.4 Android Malware Repackaging

Application repackaging refers to reverse engineering an app, injecting custom functionality, and re-assembling the app into deployable form. Malware developers commonly use application repacking to inject malicious payloads into cloned versions of popular apps on the Android platform. The process of Android application repackaging is shown in Figure 2.6. Following are the main steps involved in the repackaging of an Android application:

- Download trending premium or free app from the mainstream app stores.
- Reverse engineer the app with disassemblers (as discussed in Section 2.2.4).
- Inject malicious payload into the app and update the *AndroidManifest.xml* and resources file.
- Use *ApkTool* to reassemble and compile the app in the form of a deployable APK.
- Deploy the app in various third-party app stores.

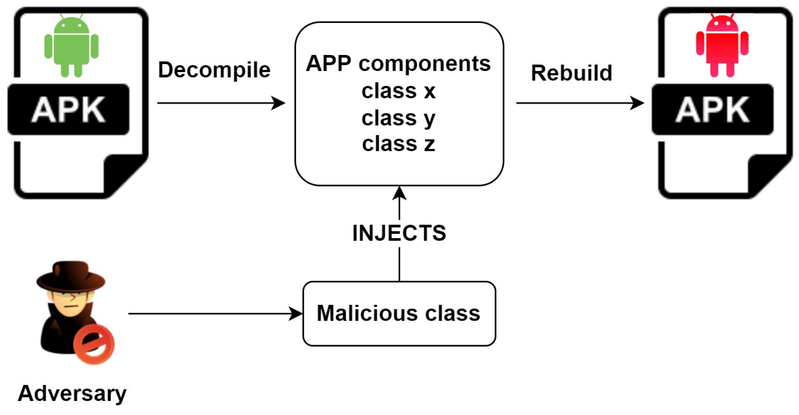


Figure 2.6: APK Repackaging

Apart from injecting malicious payloads in benign apps, malware developers often repack existing malware to evade antivirus systems. Most antivirus systems depend on known malware signatures for malware detection [101]. The malicious signatures databases of the antivirus systems are updated regularly. In the case of Android, a simple unpack and recompilation of the application without any modifications results in a change of the entire signature [11]. Reassembling the app changes the organization of contents like classes, methods and variables in the *classes.dex* file, which eventually affects the signature of the app. Consequently, attackers regularly use the practice of simple recompilation to create exact clones of known malware to deceive antivirus systems. Apart from malicious code injection and simple repackaging, attackers also repack premium apps with custom advertisements and distribute them for free to generate revenue.

Android malware repackaging has become a significant concern for security analysts over the past few years. Currently, most antivirus systems rely on the signature-based detection [102, 8, 9]. In contrast, application repackaging or creating clones of Android malware have become a common practice by attackers to evade such techniques. During the past few years, the research community have shown prevalent interest in the detection of repacked and cloned malware by employing alternative techniques [7]. Zhou et al. presented one of the preliminary studies on repacked malware in the Android malware domain and claimed that more than 80% of the existing Android malware is repacked [92]. Likewise, DNADroid [103] was proposed to detect potential clones of Android apps by using dependency graphs based on methods in the Android app. Zheng et al. proposed DroidAnalytics [60], an Android malware detector based on a multi-level signature generation technique with the ability to determine malware clones. ImageStruct [104] and a similar work DroidEagle [105] leverage the similarity of images and UI layout to detect potential clones and repacked malware in Android apps. DroidClone [106] rely on the structure and reusing of code segments to detect repackaged apps and clones of Android malware. Singh et al. employ a multi-view machine learning-based technique to detect repacked Android malware [107] and report up to 97.46% accuracy using 15,297 malware samples.

Glanz et al. proposed CodeMatch [108], a technique based on advanced library detection and fuzzy hashing to detect repacked Android apps. They applied the CodeMatch tool on various Android app stores and revealed that 15% of the apps in the commercial app stores are repacked versions of known apps. Ishii et al. [109] proposed Appraiser to perform a large-scale analysis of cloned apps in Android app repositories. They evaluated 1.3 million apps from various Android app stores and found that around 13% of the apps in third-party app stores are clones of existing apps. Furthermore, they revealed that up to 70% of the cloned apps in third-party app stores are repacked versions of known malware. Gaofeng et al. [110] proposed a technique to detect repacked Android malware based on mobile edge computing. They employed the Density Peak Cluster method on network traffic data to find the similarities between Android apps. As a result, they detected up to 92% of the repacked apps in the dataset.

Alam et al. [111] proposed DroidClone to address the problem of clones in Android malware. DroidClone employs *MAIL*, a novel language to identify control flow patterns in the program. When evaluated on a dataset of 2050 malware and 2130 benign Android apps, DroidClone achieved a detection rate of up to 94.2%. A recent study by Roopak Surendran [112] investigated the impact of semantically similar Android malware apps on various ML models. Surendran employed an opcode subsequence-based clustering technique to identify malware clones in the Drebin dataset. The results show that the malware detection rate drops from 95% to 91% when malware clones are removed from the dataset.

2.5 Adversarial Machine Learning

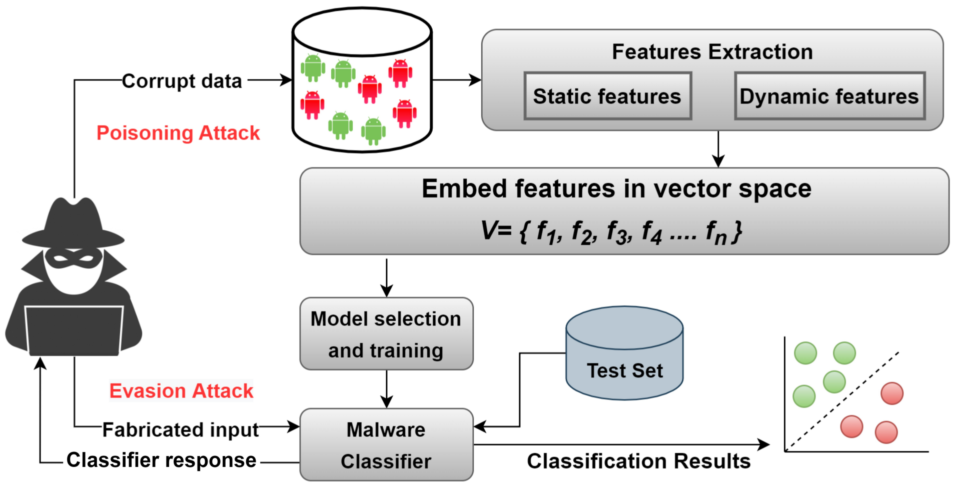


Figure 2.7: Types of Adversarial Attacks

As discussed earlier, researchers increasingly use ML-based techniques to develop countermeasures for Android malware. Since the key assumption behind ML is that the data used for the training phase represents the problem domain and deliberate modifications of data do not occur [21]; therefore, models built using ML are vulnerable to adversarial attacks. An attacker can deliberately formulate adversarial samples in order to evade the ML-based model [17], [18], [19]. Figure 2.7 presents a generalized model of adversarial attacks on a typical ML-based malware detection system. Generally, adversarial Attacks on ML are classified into evasion attacks and poisoning attacks. Evasion attacks are performed on trained models, whereas poisoning attacks refer to corrupting the training data to compromise the training process. Since this work focuses on adversarial evasion attacks, most techniques discussed in the literature review are related to adversarial evasion attacks and defences on Android malware detectors.

A critical component of adversarial attacks is the level of the attacker's knowledge about the target system. Srndic and Laskov [113] describe training data, feature set and classification algorithm as the key components of any learning-based system. Consequently, knowledge about these attributes plays a significant role for an adversary where obtaining all three is the best scenario (Figure 2.8). Similarly, Biggio and Roli [114] defined the following three levels of the attacker's knowledge:

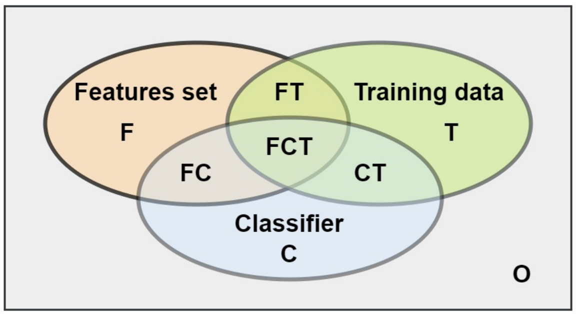


Figure 2.8: Level of Attacker's knowledge defined by Srndic and Laskov [113] where O represents that adversary has no knowledge about the target system.

Perfect knowledge (PK): this is the best case scenario for an attacker where the adversary has the knowledge about the training data, the learning algorithm, features set and the hyper-parameters setting of the classifier. The attacks performed with PK can also be referred to as white box attacks as the attacker has complete knowledge about the underlying system.

Limited knowledge (LK): in this case, the attacker has partial knowledge about the target system. The adversary may know about the learning algorithm but is unaware of the system's internals, such as hyper-parameters settings of the classifiers and the training data. The adversary might have partial or full knowledge about the dataset and features set in terms of the dataset. In the worst case, the attacker can have knowledge of the particular feature subset considered by the learning algorithm. The attacks performed with LK can also be referred to as grey box attacks as the adversary has partial knowledge about the underlying system.

Zero knowledge (ZK): this is considered the worst-case scenario for an attacker as the adversary does not know the system, such as the learning algorithm, hyper-parameters setting and the training data. The attacks performed with ZK are also referred to as black box attacks, where the adversary probes the defence mechanism with random inputs in order to observe the output.

2.5.1 Evasion Attacks on Android Malware Classifiers

In this section, we survey and categorise various evasion attacks proposed in the literature to elude Android malware detectors.

Features manipulation: In this type of attack, the adversary manipulates the features of the malicious app by either injecting or removing features to evade the malware detection model. Abaid et al. applied a feature manipulation attack on Drebin [24], a mainstream Android malware classifier. They applied feature injection and removal in Android apps and reported an evasion rate of up to 100% by manipulating a few features in the feature vector of malicious apps. Chen et al. [115] employed feature manipulation to present the fragility of linear classifiers trained on Android malware and benign apps against evasion attacks. As a result, they achieved up to 100% evasion by modifying up to 25 features. Grosse et al. [116] evaluated the impact of feature manipulation on a neural network-based Android malware classifier. They aimed to preserve the malicious semantics of samples while performing modifications in the feature vector. They evaluated the model on Drebin [24] dataset and achieved up to 63% evasion rate. Calleja et al. [117] proposed LagoDroid, a framework to trick the RevealDroid [118] classifier into miss-classifying the malware family. LagoDroid used a genetic algorithm to generate known malware variants by adding several new features to the existing malware feature set while preserving the actual semantics of the malware. As a result, LagoDroid achieved up to 98% evasion rate.

Transformation Attacks: These attacks aim to obfuscate the malicious behaviour of Android apps to evade commercial antivirus tools. Rastogi et al. [11] proposed DroidChameleon to perform three different types of transformation attacks on Android apps. The attacks include trivial transformations, transformations detectable through static analysis and transformations, which are non-detectable through static analysis. They evaluated the transformed apps on ten commercial antivirus tools and demonstrated that antivirus systems are vulnerable to transformation attacks. Meng et al. [119] demonstrated the vulnerability of 57 antivirus tools by applying 12 different transformation attacks on Android apps. Similarly, Faruki et al. [120] also demonstrated the low resilience of commercial antivirus tools by performing code obfuscation-based transformations in Android apps.

Gradient-based Attacks: These attacks are generally applied to DL-based classification models. They compute the gradient of the network loss function in relation to the input and perturb the input to evade the classifier with minimal modifications. Grosse et al. [121] applied gradient attack on neural network-based Android malware classifier trained on Drebin dataset [24]. As a result, they achieved up to 80% evasion rate.

Mimicry Attacks : In this type of attack, the adversary highlights the discriminating features of benign apps and injects those features into malicious apps to trick the classification algorithm into producing false labels. Demontis et al. [122] employed mimicry attacks to evade SVM trained on Drebin [24] dataset. As a result, they achieved up to 90% evasion rate. Similarly, Cara et al. [20] formulated mimicry attacks using API calls-based data. They trained Multi-layer perceptron (MLP) on API calls based on data extracted from benign and malicious apps. The MLP model was then evaluated by providing adversarial examples generated through mimicry attacks. The experimental results demonstrated that 80% of the adversarial samples evaded the MLP model.

Reinforcement Learning-based Attacks: refers to adversarial attacks that employ reinforcement learning to constantly add perturbations in malware samples to evade the classification model. Rathore et al. [123] proposed black-box attacks based on Reinforcement learning (RL) to modify malware samples to be classified as benign. They evaluated the ten ML-based models against evasive malware and achieved up to 58% evasion rate. Similarly, Rathore et al. employed

RL to add limited perturbations in Android malware while preserving the malicious semantics of the malware. They evaluated the proposed evasion attacks on eleven permissions-based Android malware detectors and achieved a 46% evasion rate. They also performed the RL-based evasion attacks on eleven intents-based Android malware detectors and reported an evasion rate of up to 95%.

GANs-based Attacks: Generative Adversarial Networks (GANs) could be used to automatically generate adversarial examples to trick ML classifiers. Zhang et al. [124] proposed AndrOpGAN, a technique to generate adversarial examples of Android malware that achieved an evasion rate of 99% against four malware detectors. Furthermore, Li et al. [125] proposed a technique based on bi-objective GANs to generate a novel adversarial examples attack method against Android malware classifiers. Salman et al. [126] used GANs to harden the security of Android malware detectors through intents-based features.

2.5.2 Adversarial Defences on Android Malware Classifiers

In this section, we survey and categorise various defences proposed in the literature to counter adversarial evasion attacks on Android malware classifiers.

Adversarial training: refers to training the learning algorithm with adversarial examples. Generating adversarial samples to train learning algorithms is an area of research that has gained dominant interest in the research community [127]. Grosse et al. [121] adversarially trained neural networks on adversarial data and significantly reduced evasion rates. Taheri et al. [128] used five different evasion attack models on Android malware classifiers and used GANs to formulate an adversarial training-based countermeasure against evasion attacks. The authors in [128] claim that GAN-based adversarial training methods improve the evasion detection of Android malware by up to 50%.

Robust feature selection: Adversarial attacks against machine learning-based classifiers are carried out by modifying the attributes of the original application. Generally, the features with high frequency in a given class are perturbed to elude the classification model [129]. Adversarial feature selection techniques contemplate removing easily manipulable characteristics from feature vectors to make the evasion process more difficult for the adversary. Chen et al. [115] proposed SecCLS, an adversarially robust feature selection technique. SecCLS selected features of the Android app which are difficult to manipulate and eventually made the attacker's job harder to evade the classifier. Demontis et al. [122] proposed Secure SVM (SecSVM), an adversarially robust extension of SVM. SecSVM employed more evenly distributed feature weights instead of employing high-weighted features. The experimental results demonstrated that SecSVM significantly increases the adversarial robustness of the classifier against evasion attacks.

Ensemble classifiers: Generally, ensemble classifiers-based systems are employed to enhance classification accuracy in non-adversarial settings. Ensemble classifiers generate a collaborative decision (output label) based on criteria such as voting, bagging or individual decision. Calleja et al. [117] proposed RevealDroid*, an ensemble of classifiers-based techniques to counter adversarial evasion attacks. RevealDroid* incorporated multiple tree-based classifiers trained on random feature sets extracted from the Android app. The experimental results of RevealDroid* demonstrated a significant reduction in evasion rate.

Table 2.3: A Comparative Analysis of Techniques Related to Adversarial Attacks and Defences on Android Malware Detectors

Technique	**Year**	**Dataset Source**	**Features type**	**Adversary Knowledge**	**Attack Type**	**Target**	**Evasion**	**Proposed Countermeasure**
[130]	2015	Drebin	N/A	Black-box	Transformation	Commercial Antivirus systems	up to 99%	N/A
[121]	2016	Drebin	Static	White-box	Gradient-based	Neural networks	80%	Adversarial training
[131]	2016	Drebin	Static	Black-box	Mimicry	Random Forrest	up to 20%	N/A
[132]	2017	Drebin	Static	Black-box	Feature manipulation	SVM	up to 100%	N/A
[133]	2017	Malware genome	Static	Black-box	Transformation	Commercial Antivirus systems	up to 100%	N/A
[115]	2017	Comodo Cloud	Static	Black-box	Feature manipulation	SVM	up to 98%	Secure SVM
[116]	2017	Drebin	Static	Black-box	Feature manipulation	Neural networks	up to 69%	Defensive distillation
[134]	2017	Comodo Cloud	Static	Black-box	Mimicry	SVM	83.97	Secure learning
[122]	2017	Drebin	Static	White-box	Mimicry	SVM	up to 90%	Secure learning
[135]	2018	Comodo Cloud	Static	White-box	Feature manipulation	Linear-SVM	82.12%	Features engineering
[136]	2018	Drebin	Static	Black-Box	Transformation	Sec-SVM	up to 90%	Enhanced Sec-SVM
[137]	2018	Drebin	Static	Black-box	Transformation	Comercial Antivirus systems	51.31%	N/A
[117]	2018	Drebin	Static	Black-box	Feature manipulation	RevealDroid	99.10%	Ensemble classifiers
[138]	2018	App stores	Dynamic	Black-box	Label Flipping	Random forest	up to 100%	Adversarial training
[139]	2018	Drebin	Static	Black-box	Gradient-based	RF and SVM	up to 99%	N/A
[125]	2019	Androzoo	Static	Black-box	GAN-based	Firewall	up to 95%	N/A
[140]	2019	MamaDroid/Drebin	Static	Black-box	Gradient-based	Nueral Networks	up to 60%	Ensemble learning
[141]	2020	Androzoo/VirusTotal	Static	White-box	Feature manipulation	SVM/Sec-SVM	up to 100%	N/A
[20]	2020	Drebin/Contagio/HelDroid	Static	Black-box	Mimicry	Multi-layer Perceptron	up to 80%	N/A
[142]	2020	Drebin/AMD/VirusTotal	Static	Black-box	Mimicry	SVM	up to 80%	N/A
[124]	2020	N/A	Static	Black-box	GAN-based	Antivirus systems	up to 44%	N/A
[143]	2021	Drebin	Static	Black-box	Feature manipulation	DT/RF/MLP	up to 60%	N/A
[144]	2021	Androzoo	Static	Black-box	Transformation	Drebin/Sec-SVM	81.07%	N/A
[145]	2021	Androzoo	Static	Black-box	Feature manipulation	Drebin/Sec-SVM	up to 97%	Attack efficient classifier
[146]	2021	Androzoo	Static	Black-box	GAN-based/Mimicry	DroidDetector/DeepclasssifyDroid	up to 50%	image-based classifier
[147]	2022	Androzoo/Amd	Static	Black-box	Feature manipulation	SVM/LR/RF	up to 100%	N/A
[148]	2022	N/A	Static	White-box	Gradient-based	Nueral Networks/RF/XGB	up to 75%	Adversarial training
[149]	2022	Static	Drebin	Black-box	Feature manipulation	DT/ND/LR/SVM/RF/DNNs	up to 95.31%	Adversarial training

,

2.5.3 Discussion

Table 2.3 presents an overview of techniques related to adversarial attacks and defences on Android malware classifiers. Most of the techniques mentioned in Table 2.3 use the Drebin dataset [24] to conduct the experiments. Although Drebin is one of the most used datasets in the Android malware detection domain [150], the apps collected in the Drebin dataset are from a period of 2010 to 2012. Since the Android apps from 2010 till 2012 are compatible with the obsolete versions of Android OS, the credibility of malware classifiers trained and tested on the Drebin dataset is of concern. Therefore in this work, in addition to Drebin, we employ more recent datasets such as AMD, KronoDroid and Androzoo. These datasets contain apps ranging from 2010 to 2022.

Furthermore, most of the approaches presented in Table 2.3 employ static features to train the corresponding malware classifiers and consider black-box attacks. Since this research focuses on evading ML-based Android malware classifiers at test time, black-box attacks are considered. Interestingly, it has been observed that a fair amount of the techniques mentioned in Table 2.3 target linear classifiers (particularly SVM) for evasion attacks. The reason being, linear classifiers are more vulnerable to adversarial evasion attacks as compared to tree-based classifiers and neural networks [132]. In this work, we also focus on evading the non-linear (XGB, RF and DT) in addition to linear classifiers (SVM, LR and PT).

As shown in Table 2.3, apart from [132], [134], [138] [141], [124], [143] and [146], all of the other techniques report more than 70% evasion rate on the target classifiers or Antivirus systems. Consequently, these reported evasion rates highlight the grave concern of ML-based Android malware classifiers in adversarial settings and provide a strong motivation to build evasion-aware Android malware detectors. Apart from evading the target classifiers, half of the techniques presented in Table 2.3 do not provide a countermeasure against evasion attacks. Furthermore, four of the techniques that have proposed countermeasures ([121], [138], [148], and [149]) employ adversarial training as a countermeasure against evasion attacks. Although adversarial training can provide security against one type of adversarial perturbations, it does not guarantee adversarial defence against other types of adversarial attacks [151] [152]. A robust adversarial defence should have the potential to detect diverse perturbations rather than focusing on a particular type of modification in the adversarial sample. This work aims to build adversarially aware Android malware detection systems that can detect different types of evasive malicious Android malware samples.

2.6 Summary

This chapter has provided a detailed overview of Android malware detection techniques. We provided a comprehensive overview of the Android Ecosystem, emphasising Android OS architecture, APK structure and reverse engineering process. We then presented a taxonomy of Android malware detection techniques based on the state-of-the-art approaches proposed in the literature. We identified and discussed the discriminating features of Android applications frequently used for training ML-based Android malware detectors. Furthermore, we emphasised the problem of adversarial evasion attacks on ML-based Android malware classifiers. We categorised and discussed various adversarial evasion attacks and defences proposed in the literature on Android malware detection.

In this thesis, we focus on developing accurate and adversarially aware ML-based Android malware detection techniques. We emphasise the problem of repacked malware in benchmark Android malware repositories and its impact on ML-based Android malware classifiers. Furthermore, we demonstrate the fragility of various ML-based malware classifiers in adversarial settings. Finally, we propose adversarial defences based on ensemble classifiers and adversarial training to harden the security of ML-based Android malware classifiers against evasion attacks.

Chapter 3

Enhancing the ML-based Malware Classification by Detection and Removal of Repacked Apps for Android Systems

3.1 Introduction

A torrent of Android malware attacks (over 12 million) has emerged in the past few years [153]. Most of the time, attackers produce clones by repacking existing legitimate or malicious apps to achieve the desired malevolent objectives [102]. According to some previous studies [92], 80% of the mobile malware is repackaged. Since the Android apps are available to download from public app stores such as the Google play store and other third-party app stores, an attacker can easily retrieve the legitimate app, reverse engineer the app and inject malicious code. Then the attacker can publish the modified version of the original app on public app stores [154]. This kind of attack refers to a repackaging attack. The motivation behind application repackaging is not always malicious. It has been observed that some developers get access to the source code of premium apps, repack the apps and distribute the cloned version for free. This refers to application plagiarism. The plagiarized version of premium apps is further used as a source of income by incorporating paid advertisements and in-app purchases.

Numerous techniques have been proposed in literature [7] to detect Android repackaged malware. The machine learning (ML) being the core element of Android malware detection, most of the techniques discussed in [7] focus on detecting the clones. However, to the best of our knowledge, no such study has been conducted to investigate the affects of removing the repackaged apps from training datasets. As discussed before, many apps in the current repositories are clones of existing malware. The classification results of ML-based algorithms highly depend on the quality of the data used for the training process. In contrast, pre-processing the training data is a burdensome and time-consuming task. In the case of Android, the apps need to be reverse engineered to extract features. Various tools are used to reverse engineer the Android apps [155] whereas the time required for the reverse engineering process is dependent on the size of the app. Since 2015, Google has increased the size limitation of Android apps from 50MB to 100MB [156] and with the growth of the size of apps, the cost of reverse engineering increased even more. Moreover, the training time and optimization time required for ML algorithms are also dependent on the size of the training dataset. Consequently, the repackaged apps in the training sets of ML-based algorithms result in increased costs.

This chapter first highlights the problem of repackaged malware by finding the potential clones of existing malware in 3 benchmark Android malware datasets. In order to quantify the occurrence of repacked malware in the datasets, we use a simple yet powerful strategy by matching package names of samples under observation with known malicious package names. Then, we investigate the impact of cloned apps based on the same package names on multiple machine learning models. In order to do so, we propose the AndroMalPack model. AndroMalPack extracts permissions, APIs and Intent-based features from the apps in the datasets to train the machine learning models. AndroMalPack removes all the repacked malware samples (based on package name reusing) from the training set. However, AndroMalPack retains the repacked malware in the test sets to measure the effectiveness of ML models. AndroMalPack employs seven different machine learning models (support vector machines (SVM), linear regression (LR), decision trees (DT), random forests (RF), xgboost (XGB), AdaBoost (AB) and k-nearest neighbours (KNN)) with default hyper-parameters trained on the clones free train-sets. Furthermore, AndroMalPack selects the best performing ML model on reduced datasets and further tunes the hyper-parameters by employing nature-inspired algorithms (NIAs) to achieve even better results. Three nature-inspired algorithms (bat algorithm, firefly algorithm and grey wolf optimizer) are used to optimize the hyper-parameters of best performing classification algorithm. Finally, we publish a comprehensive dataset of cloned apps based on the same package names in Drebin, AMD and AndoZoo datasets to support further research in repacked malware analysis.

The key contributions of this chapter can be summarized as follows:

1. We quantify the potential clones of known malware based on package names reusing in 3 benchmark Android malware datasets (Drebin, AMD and Androzoo).

2. We propose AndroMalPack, a classifier trained on clones free datasets and optimized using nature inspired algorithms by using permissions, APIs and intents-based features. Contrary to traditional 80/20 train and test splits, AndroMalPack filters outs the repacked malware (based on package name reusing) from training sets, whereas test sets contain all repacked malware in addition to non-repacked and benign samples. Consequently, AndroMalPack significantly reduces the training set size yet retains high classification accuracy. Although trained on reduced train sets, AndroMalPack outperforms multiple state of the art techniques in terms of classification results.

3. We publish a hash dataset of 389,995 repackaged apps based on package names reusing in Drebin, AMD and Androzoo repositories to foster future research in repacked Android malware analysis domain.

Table 3.1: Malware Samples in Drebin from Top 5 Families

Malware Family	**Samples**
FakeInstaller	821
OpFake	363
BaseBridge	330
Kmin	147
FakeDoc	132

3.2 Motivation

Our preliminary study on repacked malware started with investigating malware samples from the Drebin dataset [24]. Drebin contains 5560 malware samples belonging to 117 different malware families. We selected 1793 malware samples from the top 5 families to detect repacked malware based on the number of samples in each family (Table 3.1). Furthermore, we reverse-engineered the selected apps to extract multiple features like permissions, intents, hardware components, the network address and package names. Interestingly, we found a massive repetition of the apps' package names under analysis. Our findings reveal that 48.68% of the apps in the selected dataset share some frequently used package names. Consequently, we churned out the apps which share the same package names for further analysis. As discussed earlier, simple re-compilation of Android apps (re-construction of *classes.dex* file) results in a significant change in the app's signature. Therefore, all the apps that share the same package names still have different hash values, so a more robust signature generation technique is needed. Our target at this stage was to develop a novel signature generation technique such that all samples with the same package names should have identical signatures. Subsequently, instead of relying on calculating the hash value of *classes.dex* file, we considered the hash generation for all the extracted source codes of the apps. In an Android app, all the source code is present in the main package of the app. Therefore, once the app is reverse-engineered, the hash of the main package is calculated.

Further analysis of apps sharing the same package names revealed that most share the same source code with minor changes. Traditional hash generation algorithms like SHA1 [157] and MD5 [158] take input from an arbitrary file and produce a fixed-length cryptographic hash as an output. Calculating the SHA1 or MD5 hash of two identical files will always produce the same output. Most antivirus systems maintain contemporary databases of MD5 and SHA1 hashes of know malware. However, a minor change in the original malware results in a significant SHA1 or MD5 hash change. Therefore, instead of calculating SHA1 or MD5 hashes of the source codes of the apps sharing the same package names, we considered using a more robust hashing technique called SSDeep [159].

SSDeep is based on a Context-Triggered Piece-wise Hashing (CTPH) technique known as fuzzy hashing. As shown in Figure 3.1, contrary to traditional hashing, CTPH segemnts a given input in multiple blocks. Furthermore, the hash of each block is calculated and then concatenated to form a final hash. Hence a slight change in the original file will affect some parts of the hash; however, the other segments will remain the same. Consequently, given the fuzzy hashes of two identical files, i.e. the original file and a file with some minor changes, the SSdeep algorithm can provide the similarity score between two hashes. In order to calculate the similarity between two hashes, SSDeep employs edit distance metric. Given two strings, the edit distance between them corresponds to the minimum number of mutations required to change the first string into the second, where a mutation refers to either updating, inserting, or removing a character [159]. In contrast, SHA1 and MD5 hashes cannot compare the similarity between two hashes. Therefore, we considered using fuzzy hashes. If there are minor changes in the cloned malware, we can still get a similarity score by comparing it with known malware hashes.

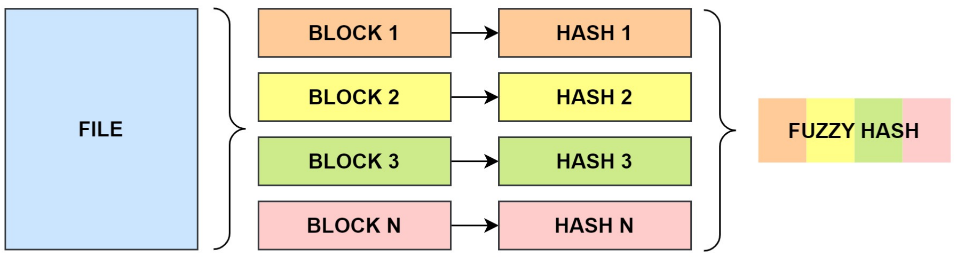


Figure 3.1: Generation of Fuzzy Hash

Algorithm 1 presents our fuzzy hash-based methodology to detect repacked malware. Let *D* be the dataset of the top 5 families from the Drebin dataset. We reverse engineer all the apps in *D* to extract a set of distinct package names as *DPN*=$\{Pn_1, Pn_2, Pn_3, \ldots Pn_n\}$. Furthermore, we randomly select one app from *D* for each package name in *DPN*, calculate its fuzzy hash using the SSDeep algorithm and place it in a set *FH*. The set of fuzzy hashes *FH* and an APK from *D* are provided as input to the Algorithm 1, whereas a similarity score is produced as an output. We calculate the fuzzy hash of the source code of the given APK as the first step *(Algorithm 1, line 1)*. The hash of the APK is then compared with all the hashes in *FH* by using the hash comparison utility of SSDeep algorithm *(Algorithm 1, line 3)*. If the similarity score at any point is greater than the threshold value, the APK is declared as repacked malware, and the similarity score is returned *(Algorithm 1, lines 4-6)*. The algorithm returns 0 if none of the hashes in *FH* has a similarity score above the threshold. According to some previous studies, if two files are almost similar, their SSdeep hashes have a similarity score ranging from 40% to 60% [160, 161, 162, 163]. Furthermore, we performed a case study by comparing the hashes of known repacked apps and found that, in most cases, the SSdeep hash similarity value was above 70%. Therefore, in this study, the threshold value for experiments was set at 70% similarity score.

```
Algorithm 1: Repacked Malware Detection using Fuzzy hash
Input: $FH = \{h_1, h_2, h_3 \ldots\ldots h_n\}$ and $APK$
Output: $SimilarityScore$
1: $hash \Leftarrow SSDeepHash(APK)$
2: for $all\ i \in FH$ do
3:     $Similarity \Leftarrow SSDeepSim(i, hash)$
4:     if $Similarity > threshold$ then
5:         Return $Similarity$
6:     end if
7: end for
8: Return 0
```

Table 3.2 summarizes the results of repacked malware detection using fuzzy hashes. We used 873 malware samples from 5 families for experiments and found six frequently reused packages. Furthermore, we randomly selected one sample from each set of apps sharing the same

Table 3.2: Fuzzy Hash-based Similarity Results

Package Name	Family	Samples	Average Similarity Score
com.software.application	FakeInstaller	234	10.6%
com.software.appinstaller	FakeInstaller	193	66.8%
com.keji.danti	BaseBridge	164	63.4%
com.extend.battery	FakeDoc	120	44.1%
com.km.installer	Kmin	65	72.3%
ad.notify1	Opfake	97	96.9%

package name and calculated its fuzzy hash. The fuzzy hash is then compared with hashes of all the remaining samples, which share the same package names. The app is declared repacked malware if its fuzzy hash has a 70% similarity score with any of the hashes in *FH*. As reported in Table 3.2, the average detection rate based on fuzzy hashes of malware samples sharing the same package name is 58.81%. Although the results from fuzzy hash-based detection are not promising, however provided us with solid motivation for further analysis of repacked malware based on package name reusing. Nonetheless, the results from fuzzy hash-based repacked malware detection support our premises that samples sharing the same package name are potential clones of already known malware. Further in this chapter, instead of focusing on signature-based detection, we employ machine learning-based algorithms to detect repacked malware based on package name-based similarity. We further extend the scope of our work by employing another two Android malware datasets to investigate malicious apps sharing identical package names. We aim to develop an Android malware classifier that is trained on reduced datasets (clones free) while preserving high detection accuracy.

3.3 Datasets

This section discusses the details of the datasets we explore for the presence of malware clones based on the same package names. We use three well known Android malware datasets Drebin [24], AMD [25] and Androzoo [26] for analysis. Table 3.3 presents the summary of the selected malware datasets. Furthermore, this section quantifies repacked malware based on package names reused in each dataset.

3.3.1 Drebin

The Drebin dataset was released in 2014 to foster research in Android malware detection domain. The Drebin dataset is publicly available and is one of the most cited works in the Android malware domain [150]. Drebin contains 123,453 benign and 5560 malicious apps, including all the apps from Android malware genome project [92] (one of the pioneer Android malware datasets)

3.3.2 AMD

Android malware dataset (AMD) was released in 2017 and contains 24,553 Android malware apps belonging to 71 different malware families. AMD consists of malware samples collected from 2010 to 2016 and is one of the most extensive publicly available Android malware datasets.

3.3.3 Androzoo

Androzoo is a publicly available, regularly updated and most popular Android apps dataset currently being used in recent studies [164, 165]. Androzoo was released in 2016 with more than 3 million Android apps and is constantly being updated. By the end of the second quarter of 2021, Androzoo contains more than 15 million Android apps. The Android apps in Androzoo are collected from several platforms like the Google app store, third-party Android app stores and *VirusShare*. The Androzoo dataset's apps are scanned and labelled for potential malware using more than 60 antivirus tools. Androzoo provides meta-data for Android apps like size, upload date, signatures and package name in the form of an excel file which is regularly updated. We considered 695,470 malware apps from the Androozoo dataset to analyse repacked malware based on reusing package names. Our criteria for app selection from Androzoo was that each app must be labelled as malware by at least ten antivirus tools.

Table 3.3: Summary of selected Malware Datasets

Dataset	Families	Samples	Date
Drebin	117	5560	2014
AMD	71	24,553	2017
Androzoo	1969	695,470	2016

3.3.4 Malware clones in datasets

As discussed in Section 3.2, our preliminary study on the Drebin dataset revealed the presence of frequently reused package names amongst malware samples. Further investigation on samples sharing the same package names showed that most share almost the same source code. This motivation led us to further explore multiple well-known Android malware datasets and quantify the samples sharing the same or similar package names. Although detecting repacked malware based on package names is a lightweight approach and can be easily evaded, our target in this work is to quantify existing clones in the dataset rather than detecting novel clones. The reason is that the selected datasets are very popular amongst the research community, so the presence of clones must be considered in future works to avoid biased results [22]. Furthermore, we investigate our claim's credibility that samples having the same package name are clones of known malware. Based on the results of our initial attempt using fuzzy hashes has provided us with reasonable ground to further investigate by incorporating ML-based algorithms.

Algorithm 2: Quantifying Repacked Malware in Datasets using Package Names

```
Input: $Dataset = \{Apk_1, Apk_2, Apk_3 \ldots\ldots Apk_n\}$
Output: $Count(Repacked\_Malware)$
1: $P_{names} = \{\}$
2: for $all\ i \in Dataset$ do
3:     $Pack\_name \Leftarrow AndroGuard(Apk_i)$
4:     if $Pack\_name \notin P_{names}$ then
5:         $P_{names}.Append(Pack\_name)$
6:     end if
7: end for
8: $Repacked\_Malware \Leftarrow Size(Dataset) - len(P_{names})$
9: Return $Repacked\_Malware$
```

Table 3.4: System Specifications

Features	**Specifications**
Processor	Intel(R) Corei7, 2.60GHz, 6 Cores
GPU	NVIDIA GeForce GTX 1650 Ti 4GB GDDR6
Cache size	12MB
RAM	16 GB DDR4-2933MH
Platform	Windows 10

Algorithm 2 presents our methodology to quantify repacked malware in Drebin, AMD and Androzoo datasets based on package names reusing. A dataset is provided as an input to the Algorithm 2, and the number of repacked malware based on reused package names is provided as an output. We take an empty set P_{names} *(Algorithm 2, line 1)* which is populated with the distinct package names in the given dataset. Furthermore, we extract the package names of all the apps in the given dataset using the Androguard tool *(Algorithm 2, line 3)*. AndroGuard [166] is a python-based tool which can extract multiple features from *AndroidManifest.xml* file of a given APK. The extracted package name is then appended in the Package names list P_{names} if not already present in it *(Algorithm 2, lines 4-6)*. Consequently, the list P_{names} is populated with all the distinct package names within the dataset. The algorithm then returns the number of samples that reuse existing malicious package names.

We used the AndroGuard tool to extract package names of samples from Drebin and AMD datasets. In contrast, Androzoo already provides information about the package names. The metadata provided by Androzoo saved a fair amount of time as the Androguard tool performs reverse engineering of an APK to extract features. The time required to reverse engineer an app depends on the size of the app. It took 2.5 seconds on average to reverse engineer apps from Drebin and AMD datasets to extract package names using the Androguard tool (System specification shown in Table 3.4). Our experiments to find repacked malware samples based on package names reused in Drebin, AMD and Androzoo datasets are shown in Figure 3.2. 52.3% of the samples in Drebin and 29.4% of samples in the AMD dataset contain reused package names. Compared to Drebin and AMD, the Androzoo dataset contains far more samples and interestingly, 42.3% of them contain reused package names. Table 3.5 outlines the statistics about each dataset's top 10 most reused package names in malware samples.

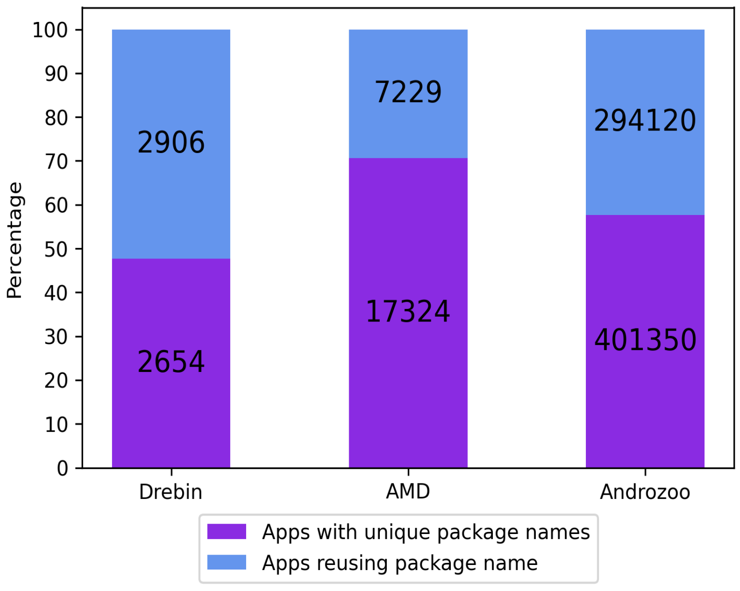


Figure 3.2: Quantity of Repacked malware in Datasets based on package names reusing

Table 3.5: Top 10 most reused packages in Datasets

Drebin		AMD		Androzoo	
Package Name	Sample	Package Name	Samples	Package Name	Samples
com.software.application	234	com.soft.android.appinstaller	548	com.software.application	2114
com.soft.android.appinstaller	193	tk.jianmo.study	384	com.xgbuy.xg	1183
Jk7H.PwcD	117	com.software.application	274	com.soft.android.appinstaller	769
com.extend.battery	110	edu.raj.sphincter	255	ad.notify1	727
ad.notify1	97	jp.bravo.honda	150	com.qihoo.appstore	676
com.convertoman.proin	92	com.android.app	143	ch.nth.android.contentabo_l01_sim_univ	535
vbkoxh.cswnpr	83	org.slempo.service	143	com.nemo.vidmate	475
com.depositmobi	71	fl.affectionate	114	com.qiyi.video	416
com.software.app	54	de.granulocyte	101	nang.dv	408
com.km.launcher	52	org.zxformat	98	tk.jianmo.study	384

3.4 AndroMalPack

As discussed in Section 3.2, signature-based malware detection is very fragile against a simple mutation in original malware. Consequently, malware authors often repack existing malware with minimal modifications to trick antivirus systems relying on signature-based detection.

Therefore, we employ ML-based algorithms to create a more robust solution for repacked malware detection. The motivation to use ML algorithms is to support our claim that malware samples sharing the same package names in popular Android malware datasets are clones of known malware. In this section, we propose AndroMalPack (Figure 3.3), an ML-based Android malware classifier trained on clones free train sets and optimized using nature-inspired algorithms (NIAs).

3.4.1 Data pre-processing

As shown in Figure 3.3, AndroMalPack is provided with a malicious Android apps dataset. Instead of splitting the dataset into random train and test sets (the traditional approach), AndroMalPack extracts the apps' package names to build the train and test sets. All the apps which have reused package names are directly assigned to the test set, whereas 70% of the apps with unique package names are assigned to the train set, and 30% are allocated to the test set. Consequently, train and test set distribution by AndroMalPack confirms the exclusion of malware samples sharing the same package names from the training set and eventually retains diversity and perceptible reduction of training set size. Furthermore, the benign apps dataset apps are randomly distributed 70% in the train set and 30% in the test set.

3.4.2 Features set modeling

After train and testing set splits, AndroMalPack extracts the features from the Android apps. We extract three types of features by using static analysis on Android apps. Android permissions and intent filters-based features are extracted from *AndroidManifest.xml* file, whereas API calls-based features are extracted from the source code of the apps. Following is a brief description of the extracted features:

Android Permissions: Android protects the privacy of the user by employing the permissions model. Android programmers must declare the sensitive permissions required by the app in the *AndroidManifest.xml* file. Once the app is installed on an Android device, it notifies the user

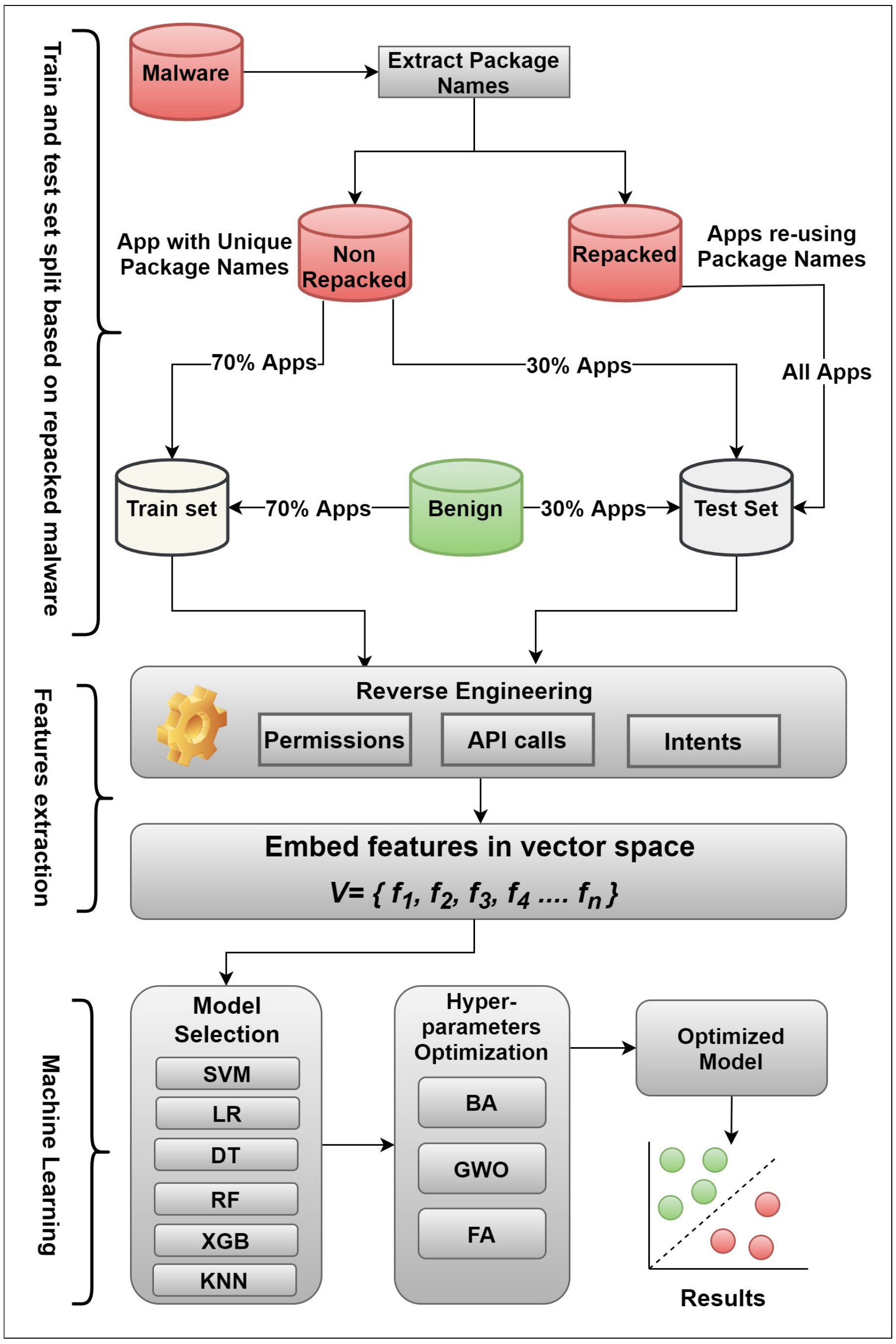


Figure 3.3: Block Diagram of AndroMalPack

about sensitive permission that the app requires, such as access to contacts, camera or microphone. The pattern of permissions required by an Android app can employed to detect malware by employing ML-based algorithms. Numerous techniques in literature use the Android permissions model to detect potential malware in Android apps [39, 41, 37]

Intent Filters: define the communication mechanism between different components of an Android app. Intents are simple message objects that transfer the information between different modules such as activities, content providers, services and broadcast receivers of an Android app. The intent filters of an Android app are declared in *AndroidManifest.xml* file and can be employed as a feature set to train ML algorithms in order to detect Android malware. Many techniques in literature employ intent filters in addition to other features from *AndroidManifest.xml* file for malware detection [46, 167, 168].

API calls: Android application programmable interfaces (APIs) are a set of specifications and protocols that are used to build and integrate Android applications. API calls are invoked in apps at run-time to perform different tasks like sending SMS and getting network information. API calls-based features are efficient in malware detection and are used by many existing malware detection techniques [169, 170].

The aforementioned features are employed to construct feature vectors from samples in the datasets. We construct a binary encoded feature vector for each APK such that the presence of a particular feature in the APK is marked as 1 in the feature vector whereas absence is marked as 0. Algorithm 3 explains our methodology for feature set modeling. The Algorithm 3 takes an APK, a list of unique permissions, a list of unique API calls and a list of unique intent filters. APK tool is used to extract *AndroidManifest.xml* file of the given APK *(Algorithm 3, line 1)*. Permissions and intent filters-based features are then extracted from the *AndroidManifest.xml* *(Algorithm 3, line 2-3)*. Furthermore, we use the Androguard tool to extract all the API calls from the given APK *(Algorithm 3, line 4)*. Then we compare each permission in the unique permissions list, and if a particular permission in the unique permissions list is present in the extracted permissions set, the corresponding permissions vector bit is set to 1; otherwise, the bit is assigned 0 value *(Algorithm 3, line 5-11)*. The same process is applied to construct the intents

Algorithm 3: Feature Set Modeling

```
Input: APK, $Permission_{(list)}$ , $API_{(list)}$, $Intents_{(list)}$
Output: Features Vector
1: $manifest.xml \Leftarrow Apk\_Tool(APK)$
2: $Permissions_{(set)} \Leftarrow manifest.xml$
3: $Intents_{(set)} \Leftarrow manifest.xml$
4: $API\_calls_{(set)} \Leftarrow AndroGuard(APK)$
5: for $each\ permission \in Permissions_{(list)}$ do
6:     if $permission \in Permission_{(set)}$ then
7:         $Vector_{(Perm)} \Leftarrow 1$
8:     else
9:         $Vector_{(Perm)} \Leftarrow 0$
10:    end if
11: end for
12: for $each\ intent \in Intents_{(list)}$ do
13:    if $intent \in Intents_{(set)}$ then
14:        $Vector_{(Int)} \Leftarrow 1$
15:    else
16:        $Vector_{(Int)} \Leftarrow 0$
17:    end if
18: end for
19: for $each\ api \in API_{(list)}$ do
20:    if $api \in API\_calls_{(set)}$ then
21:        $Vector_{(api)} \Leftarrow 1$
22:    else
23:        $Vector_{(api)} \Leftarrow 0$
24:    end if
25: end for
26: $FV \Leftarrow Concat(Vector_{(Perm)}, Vector_{(Int)}, Vector_{(api)})$
27: Return $FV$
```

vector *(Algorithm 3, line 12-18)* and the API calls vector *(Algorithm 3, line 19-25)*. Finally, the three vectors (Permissions, intent filters and API calls) are concatenated and returned by the algorithm *(Algorithm 3, line 26-27)*.

3.4.3 Learning Phase

AndroMalPack considers SVM, LR, DT, RF, XGB, AB and KNN to train ML models. Furthermore, based on the classification results, the best performing model is selected and further tuned using NIAs. We consider Bat algorithm (BA) [171], Grey wolf optimizer (GWO) [172] and Firefly algorithm (FA) [173] to optimize the best performing model in motivation to achieve even better classification results. Finally, the results obtained by AndroMalPack are compared with classifiers trained on datasets without considering repacked malware to present the efficacy of AndroMalPack.

3.5 Experimental Results

In this section, we report the evaluation results of AndroMalPack. Prompt from the analysis performed in Section 2, contrary to traditional 80/20 train test splits of datasets, AndroMalPack considers training the classifiers on reduced train sets. The reduced training set of each dataset confirms the exclusion of malware samples sharing the same package names from the training set and eventually retains diversity and perceptible reduction of training set size. Table 3.6 presents the distribution of samples in train and test sets based on package names from Drebin, AMD and Androzoo datasets. We considered all the samples from Drebin and AMD datasets; however, we contemplated 25116 samples from the Androzoo dataset. As shown in Figure 3.2, the Androzoo dataset contains 294,120 potential repacked malware samples, whereas the process of reverse engineering to extract features from all these apps is expensive in terms of time and memory. Therefore, we selected 14,939 samples with unique package names and 10,177 with reused package names from the Androzoo dataset to reduce samples.

Table 3.6: Train and test set splits for classifiers trained on clones free train sets

Malware Dataset	**Total Malware Samples**	**Malware Samples in Train set**	**Benign Samples in Train set**	**Malware Samples in Test set**	**Benign Samples in Test set**
Drebin	5560	2704	4200	2856	1800
AMD	24553	15157	4200	9396	1800
Androzoo	25116	13039	4200	12077	1800

We evaluate the results based on the outcome of the confusion matrix. Confusion matrix summaries the results of machine learning classifiers based on correct and incorrect predictions by using the following metrics:

- **True Positive (TP):** signifies the number of malicious apps correctly classified by the ML classifiers.
- **False Positive (FP):** signifies the number of benign apps classified as malware by the ML classifier.
- **True Negative (TN):** signifies the number of malicious apps classified as benign by the ML classifier.
- **False Negative (FN):** signifies the number of benign apps correctly classified by the ML classifier

The performance metrics which we consider are accuracy (Eq.3.1), recall (Eq.3.2), precision (Eq.3.3) and F1-score (Eq.3.4) derived from the confusion matrix.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{3.1}$$

$$Recall = \frac{TP}{TP + FN} \tag{3.2}$$

$$Precision = \frac{TP}{TP + FP} \tag{3.3}$$

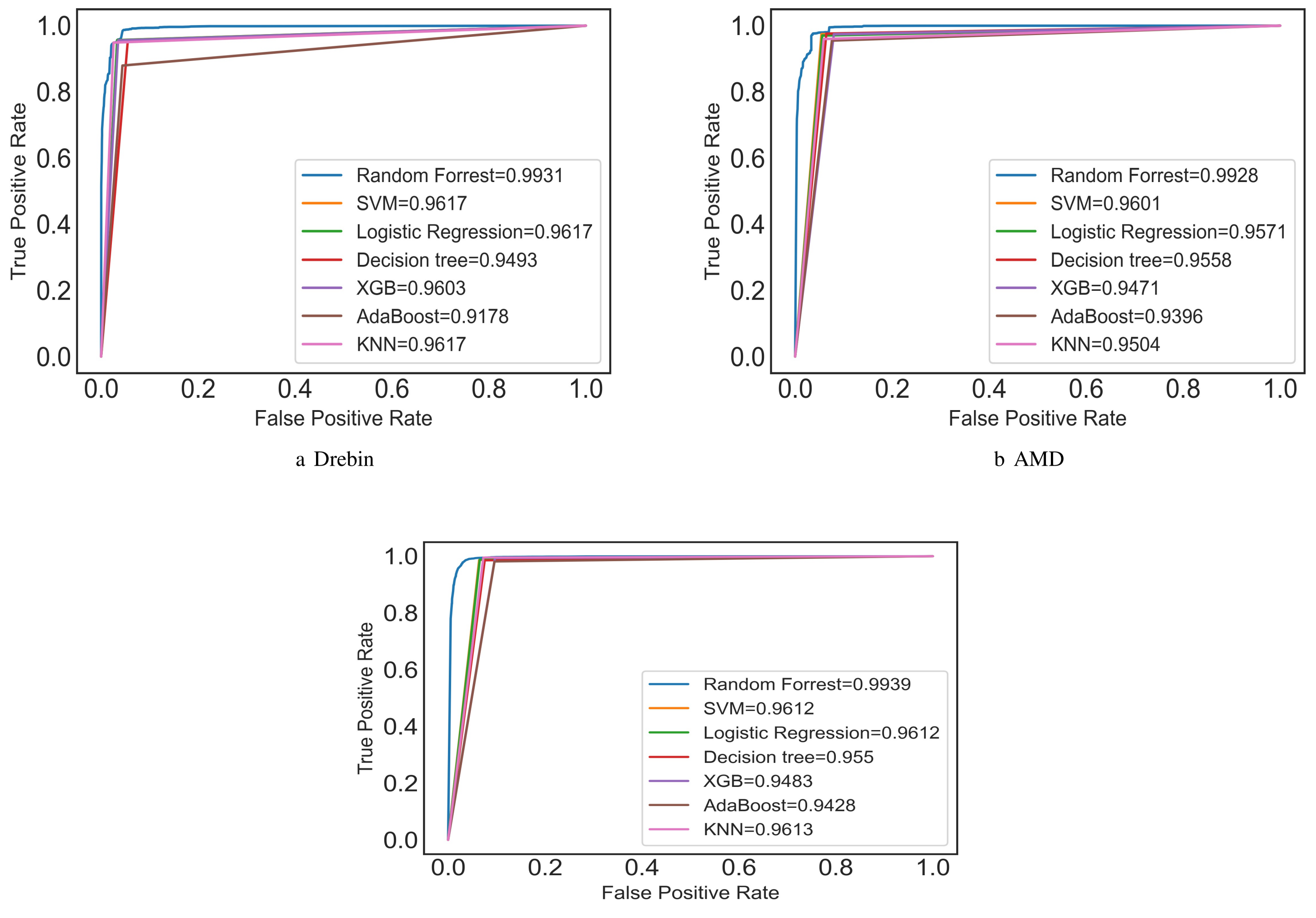
$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{3.4}$$

Table 3.7 presents the results of classifiers trained on reduced train sets with default hyper-parameters settings. Apart from the performance on the Drebin dataset, RF outperforms SVM, LR, DT, AB, XGB and KNN in terms of classification results. Although the classifiers are trained on reduced train sets, whereas test sets contain all the repacked malware samples and non-clone malware and benign apps, RF achieves high precision and recall scores. Similarly,

Table 3.7: Results of Classifiers Trained on Reduced Train sets

		SVM	**LR**	**DT**	**RF**	**XGB**	**AB**	**KNN**
Drebin	Accuracy	96.28	96.24	94.88	96.02	96.09	92.27	96.33
	Recall	95.6	95.6	95.4	96.1	95.5	87.9	94.9
	Precision	95.7	95.7	93.1	94.8	95.5	93.9	96.6
	F-measure	95.7	95.7	94.2	95.5	95.5	90.8	95.8
AMD	Accuracy	96.61	96.26	96.43	96.89	95.85	94.61	95.43
	Recall	97.4	97	97.6	97.9	97.4	95.5	96
	Precision	97.7	97.7	97.4	97.8	96.8	96.9	97.6
	F-measure	97.6	97.4	97.5	97.8	97.1	96.2	96.8
Androzoo	Accuracy	97	97.05	96.55	97.53	96.39	95.61	97.31
	Recall	98.7	98.8	98.5	99.5	99.3	98.1	99.5
	Precision	96.9	96.9	96.4	97	97.4	95.5	98
	F-measure	97.8	97.8	97.5	98.2	97.4	96.8	98

Figure 3.4 depicts the receiver operating characteristic (ROC) curves derived from classifiers trained on reduced train sets from Drebin, AMD and Androzoo datasets. The ROC curves plot the false positive rate (FPR) on the x-axis, whereas the true positive rate (Recall) is plotted on the y-axis. The ROC curves show remarkable results where RF yields the best results compared to SVM, LR, DT, AB, XGB and KNN. Subsequently, to further enhance the performance of AndroMalPack, we employ NIAs to determine the optimal hyper-parameters settings of the best-performing classifier (RF). We consider BA [171] (see Appendix A), FA [173] (see Appendix B) and GWO [172] (see Appendix C) for hyper-parameters tuning of RF.



a Drebin

b AMD

c Androzoo

Figure 3.4: ROC curves of classifiers trained on reduced train sets from Androzoo

Table 3.8: Hyper-parameters for RF proposed by NIAs

	n_estimators			***max_depth***			***min_sample_split***			***max_features***		
	BA	**FA**	**GWO**	**BA**	**FA**	**GWO**	**BA**	**FA**	**GWO**	**BA**	**FA**	**GWO**
Drebin	60	80	80	34	28	28	2	2	2	*auto*	*sqrt*	*auto*
AMD	60	80	80	36	38	38	2	2	2	*sqrt*	*sqrt*	*sqrt*
Androzoo	40	80	80	32	32	32	2	2	2	*sqrt*	*sqrt*	*auto*

Table 3.9: Results of AndroMalPack

	Bat Algorithm				**Firefly Algorithm**				**Grey Wolf Optimzer**			
	Acc	**Recall**	**Pre**	**F1**	**Acc**	**Recall**	**Pre**	**F1**	**Acc**	**Recall**	**Pre**	**F1**
Drebin	98.29	98.7	97.7	98.1	98.22	98.5	97.4	98	98.22	98.5	97.4	98
AMD	98.21	99.4	98.1	98.7	98.17	99.4	98.1	98.7	98.17	99.4	98.1	98.7
Androzoo	97.94	99.8	97.2	98.5	97.9	99.8	97.2	98.5	97.94	99.8	97.2	98.5

Table 3.8 present the optimal hyper-parameters setting for RF classifiers determined by NIAs (BA, FA and GWO) based on Drebin, AMD and Androzoo datasets. Furthermore, Table 3.9 presents the classification results achieved by AndroMalPack, an Android malware classifier based on RF and optimized using NIAs. Compared to the results of the RF classifier in Table 3.7, AndroMalPack remarkably strengthens the performance by employing NIAs to determine the optimal setting of hyper-parameters. Furthermore, as shown in Table 3.9, in the case of each dataset, RF optimized using BA performs slightly better than FA and GWO, whereas the result obtained from FA and GWO are almost similar with a marginal difference. However, in addition to classification results, we also consider the time complexity of NIAs as a performance metric for AndroMalPack. Figure 3.5 depicts the time each NIA (BA, FA and GWO) takes to optimize the RF hyper-parameters based on Drebin, AMD and Androozoo datasets. The population size for each NIA was initialized with 50, and max iterations were set to 100. Subsequently, BA outperforms FA and GWO in terms of time complexity and classification results. Nevertheless, the performance of FA and GWO is also convincing in terms of classification results; however, compared to BA, FA and GWO take a significant amount of time to find the optimal hyper-parameters in the case of each dataset (Figure 3.5). Therefore, AndroMalPack prefers BA compared to FA and GWO for hyper-parameters optimization to enhance the performance of RF for Android malware classification.

Table 3.10: Results of classifiers on Datasets using Random 80/20 Train and Test Splits

		SVM	**LR**	**DT**	**RF**	**XGB**	**AB**	**KNN**
Drebin	Accuracy	96.03	96.25	95.74	97.09	95.78	94.98	97.6
	Recall	94.5	94.7	95.9	96.8	94.2	92.2	96.8
	Precision	95.5	95.8	93.6	95.9	95.1	95.1	97.1
	F-measure	95	95.3	94.7	96.4	94.7	93.6	97
AMD	Accuracy	97.71	97.81	97.59	98.27	97.56	95.86	98.04
	Recall	98.6	98.7	98.5	99.5	99.1	97.3	99.3
	Precision	98.4	98.3	98.3	98.2	97.7	97.2	98.1
	F-measure	98.5	98.5	98.4	98.8	98.4	97.2	98.7
Androzoo	Accuracy	97.38	97.38	97.29	98.26	96.75	95.83	97.8
	Recall	98.8	98.7	98.6	99.6	98.9	97.8	99.2
	Precision	97.7	97.7	97.7	98.1	96.7	96.6	97.8
	F-measure	98.2	98.2	98.2	98.8	97.8	97.2	98.5

Furthermore, we consider the traditional 80/20 random train test split regardless of repacked malware in the datasets to compare the results with AndroMalPack. Table 3.10 shows the classification results obtained from classifiers based on 80/20 train and test sets split with 10-fold cross-validation. Apart from the classification results from the Drebin dataset, the RF classifier outperforms all the other classifiers in terms of accuracy, recall, precision and F1 score. Furthermore, compared to the classification results of AndroMalPack, the results obtained by classifiers based on 80/20 random train test splits are subtle, with a marginal difference. Consequently, we can conclude that removing repacked malware based on the same package names from training sets does not significantly affect the classification results.

3.5.1 Discussion

The experimental results of AndroMalPack revealed that removing the malware samples sharing the same package names from the training sets does not significantly impact classification results. Similarly, to prove that malware samples sharing the same package names are repacked versions of known malware, AndroMalPack assigns all samples with reused package names to the test set in addition to benign apps and non-repacked malware. Interesting, AndroMalPack

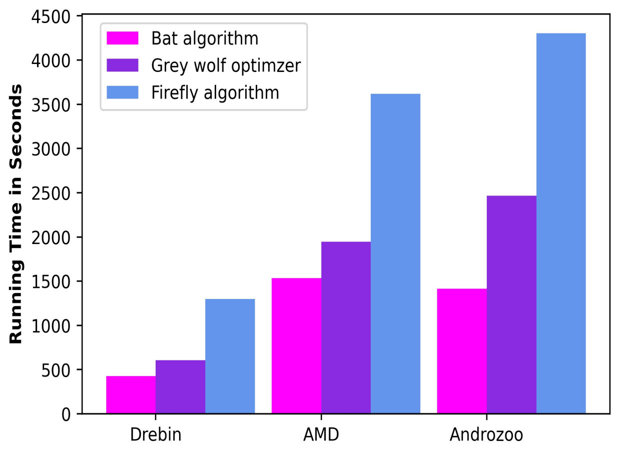


Figure 3.5: Running time comparison of NIAs

achieves up to 98% accuracy with the aforementioned train and test set distribution. Consequently, the results reflect our claim that malware samples sharing the same package names are clones of existing malware. Our Analysis of three benchmark Android malware datasets (Drebin, AMD and Androzoo) revealed that a significant amount of malware samples in these repositories are repacked (based on package name reusing). We emphasize that while performing Android malware analysis, repacked malware should be of concern. Repacked malware creates an overhead in terms of time and computational expenses. Consequently, removing the repacked malware can save a fair amount of time in the reverse engineering process to extract features from Android apps.

In order to present the effectiveness of removing repacked malware from the datasets, we profile the reverse engineering time to extract features based on two scenarios. In scenario 1, we consider reverse-engineering the full dataset regardless of repacked malware, whereas, in scenario 2, we remove the repacked malware and profile the reverse engineering time. As shown in Figure 3.6, the removal of repacked apps in Drebin, AMD and Androzoo datasets results in a significant reduction in processing time. It took 2.5 seconds to extract APIs, intents, and permissions-based features from an APK by employing the Androguard tool's static analysis. On the other hand,

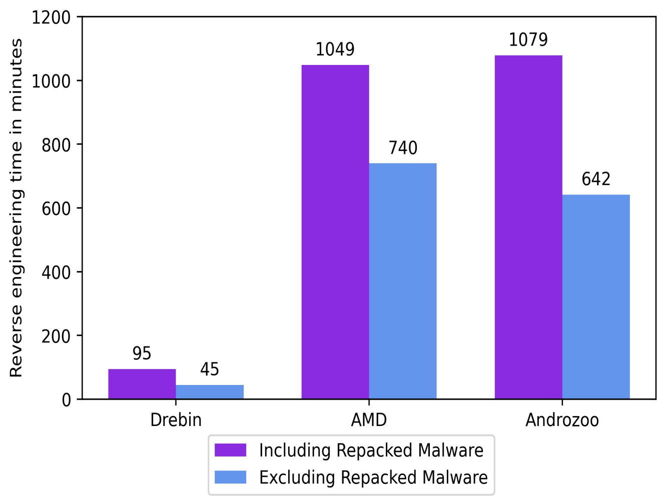


Figure 3.6: Features extraction time comparison

dynamic analysis can take anywhere between 60 seconds and 10 minutes per APK to extract features [174, 175, 75, 176]. Nevertheless, the evaluation results of AndroMalPack prove that removing the repacked malware from training sets does not have a significant impact on classification results. Furthermore, as discussed in [177] and [22], the duplicates in datasets can cause adverse effects on ML-based models by producing biased results. Consequently, we encourage fellow researchers to consider repacked malware in Android malware datasets while performing ML-based malware detection to train classifiers on reduced yet diverse data.

3.6 Comparison

This work focuses on a simple yet powerful strategy for repacked malware detection by using package name-based similarity. We demonstrated that many apps in popular Android malware repositories share common package names. Our further analysis revealed that apps sharing the same package names are repackaged versions of existing malware. Similarly, most of the existing techniques focus on the detection of repacked and cloned malware using various techniques and report that plethora of malware is repacked instead of being novel [7]. However, in the

Android malware domain, apart from the study proposed by Zhao et al [22], no extensive study has been conducted on the impact of duplicates on ML-based classifiers. Zhao et al. considered duplicates based on three distinct features (*.dex* code similarity, op-code sequence and API calls). They evaluated them using four different datasets (Genome, Drebin, AMD and RmvDroid [178]). Compared to Zhao et al., we considered a novel and more lightweight strategy (package names based on similarity). Interestingly, in the case of the Drebin dataset, package names-based similarity (52.3%) outperforms, *.dex* code similarity (35.9%) and op-code sequence (48.6%) in [22] to detect malware clones, whereas API based similarity is almost similar to our approach (52.4%). However, in the case of the AMD dataset, apart from *.dex* code similarity (21.8%), Op-code (47.6%) and API calls based similarity (52.2%) outperforms package based similarity (29.4%). Likewise, Irolla et al. use op-code similarity to quantify duplicates in Drebin dataset [23]. Irolla et al. claim that 49.35% samples in the Drebin dataset are repackaged and question the biased results of existing ML-based classifiers trained on the Drebin dataset. As compared to [23], package name based repackaged malware detection is more lightweight and outperforms Irolla et al. technique by finding 52.3% repacked malware in the Drebin dataset.

Furthermore, we propose AndroMalPack, an Android malware classifier trained on clones free training sets and optimized using NIAs. The training sets of AndroMalPack exclude all the apps which share common package names and consequently reduce the size of training data yet preserve high classification results. Table 3.11 presents the detailed comparison of AndroMalPack with recent Android malware detection techniques in literature.

3.7 AndroMalPack Dataset

In order to foster the research in the domain of repackaged Android malware analysis, we publish a cryptographic hash-based dataset of repacked Android apps having the same package names (AndroMalPack dataset[1]). AndroMalPack dataset is distributed into three comma-

[1]https://github.com/hasnainrafique/AndroMalPack-Dataset

Table 3.11: Comparison of AndroMalPack with related work

Technique	**ML-Model**	**Features**	**Dataset**	**Accuracy**	**Recall**	**Precision**	**F1 score**
[179]	Ensemble	Permissions and source code	M0Droid	95.6	95.7	95.8	95.6
[43]	Random Forrest	Permissions, APIs and system calls	Custom	88.26	88.40	88.16	N/A
[180]	Random Forrest	Permissions, APIs, intents and hardware	Drebin	97.24	97	98	97
[181]	Random Forrest	API calls	Drebin AMD	96 98	95 98	97 99	96 98
[182]	CatBoost	Permissions and op-code sequence	Drebin and Custom	97.40	96.77	98.0	97.38
[183]	Random Forrest	Permissions based features	Androzoo	81.53	81.53	82.59	84.18
[42]	Profile hidden Markov model	API calls	Drebin	94.5	N/A	93.0	93.9
[184]	SVM	Permission, intents and byte-code	AMD	95.09	95.09	95.11	95.10
AndroMalPack	Random Forrest	Permission, API calls and Intents	**Drebin** **AMD** **Androzoo**	**98.29** **98.21** **97.14**	**98.70** **99.40** **99.8**	**97.7** **98.1** **97.2**	**98.1** **97.7** **97.5**

separated (*.csv*) files where each file contains cryptographic hashes of repacked apps from Drebin, AMD and Androzoo datasets, respectively. Each file in the AndroMalPack dataset contains two columns where the first column contains the hash of the app and the second column contains the corresponding package name. The files are sorted in descending order based on the number of frequently reused package names in each dataset. Since the access to Drebin, AMD and Androzoo are protected by the owners, we do not provide the APK files. Access to the datasets (Drebin, AMD and Androzoo) can be requested through an authorized source, and our dataset of hash values can be employed to churn out repackaged apps based on package name reusing. Drebin and Androzoo datasets label each app with a SHA256 hash, whereas AMD datasets label apps using MD5 hashes. Likewise, the AndroMalPack dataset uses SHA256 hashes for Drebin and Androzoo, whereas MD5 hashes for the AMD dataset to represent repackaged apps based on package name reusing.

3.8 Summary

Malware authors often repack existing malware to deceive antivirus systems. Consequently, many apps in popular Android malware datasets are clones of existing malware. This chapter emphasizes the problem of repackaged Android malware in 3 benchmark Android malware datasets. We leverage package names based on similarity to quantify repackaged malware in the datasets and reveal 52.3% malware samples in Drebin, 29.8% of malware samples in AMD and 42.3% malware samples in Androzoo dataset reuse existing package names. Furthermore, we investigate the impact of malware samples sharing the same package names on six ML-based classifiers (SVM, LR, DT, RF, XGB, AB and KNN). Interestingly, our experimental results demonstrate that removing malware samples sharing the same package names from training sets of ML-classifiers does not significantly impact classification results. Consequently, we propose AndroMalPack, an Android malware classifier based on RF, trained on clones free train sets and optimized using nature-inspired algorithms. Although AndroMalPack is trained on reduced train sets, it preserves remarkable classification results. Finally, we publish an AndroMalPack dataset to foster the research on repackaged Android malware based on package names reusing. AndroMalPack dataset contains 389,995 cryptographic hashes of samples sharing the same package names in the Drebin, AMD and Androzoo datasets.

Chapter 4

Evasion-aware Android malware detection model based on multiple classifiers system

4.1 Introduction

ML-based malware detection techniques have garnered a significant interest among malware researchers due to their ability to identify novel samples. Various features are extracted from malicious and benign Android applications (apps) using either static, dynamic or hybrid analysis. Moreover, these features are transformed into feature vector sets and ML-based algorithms are trained on these vectors to predict malware and benign applications. Although ML-based malware detection techniques have demonstrated promising results, they are vulnerable to adversarial evasion attacks [17, 18, 19].

This chapter focuses on highlighting the fragility of ML-based Android malware detectors in adversarial settings and propose a countermeasure against evasion attacks. First, we present a motivating case study by evading Drebin [24], one of the mainstream Android malware classifiers, to demonstrate the fragility of ML-based Android malware classifiers. Secondly, we create a hybrid featured balanced of 18,000 Android malware and 18,000 benign apps. Third, we proposed CureDroid, an Android malware classifier trained on hybrid features and optimized using the tree-based pipeline optimization technique (TPoT). Fourth, we examine the effectiveness of CureDroid in adversarial settings by performing mimicry attacks (MA), feature removal attacks (FRA) and mimicry with feature removal attacks (MFRA). Fifth, we propose CureDroid*, an adversarially robust extesion of CureDroid. Contrary to training on a single feature vector source, CureDroid* employs multiple classifiers trained on distinct logical subsets of feature vectors extracted from Android apps. Finally, we investigate the performance of CureDroid* in the adversarial setting and prove the effectiveness of the proposed model.

We summarise the main contributions of this chapter in the following aspects:

1. We propose CureDroid, an Android malware classification model trained on hybrid features and optimized using tree-based pipeline optimization technique.

2. We evaluate CureDroid in adversarial settings. We perform mimicry attacks, feature removal attacks and mimicry attacks in conjuction with feature removal to present the fragility of ML-based Android malware detection models against adversarial evasion attacks.

3. We propose CureDroid*, an adversarially robust extension of CureDroid. CureDroid*, is a novel, scalable and adversarially aware Android malware classification model. CureDroid* is based on an ensemble of ML-based models trained on distinct set of features where each model has the individual capability to detect Android malware.

4.2 Motivation

As a preliminary study to investigate the fragility of Android malware detectors in adversarial settings, we evaluate a case study by applying adversarial evasion attacks on *Drebin* [24], one of the most cited works related to Android malware detection [150]. Drebin is a lightweight on-device malware detector that extracts Android applications' features through static analysis. Drebin employs a publicly accessible dataset containing 5560 malicious and 123,453 benign Android applications. Furthermore, Drebin extracts permissions, intents, API calls, network addresses and hardware components-based features from the apps and embeds them into a multi-dimensional feature vector space. Drebin trains Linear SVM on the extracted features to categorise Android malware and benign apps. According to the authors, Drebin achieved up to 94 percent malware detection accuracy with a very low false positive rate. Since Drebin is not open-source, we build a similar model by training a linear SVM on the Drebin dataset with the same feature set used in the Drebin classifier. Figure 4.1 uses ROC curve plots to depict the results of our experiments to replicate Drebin.

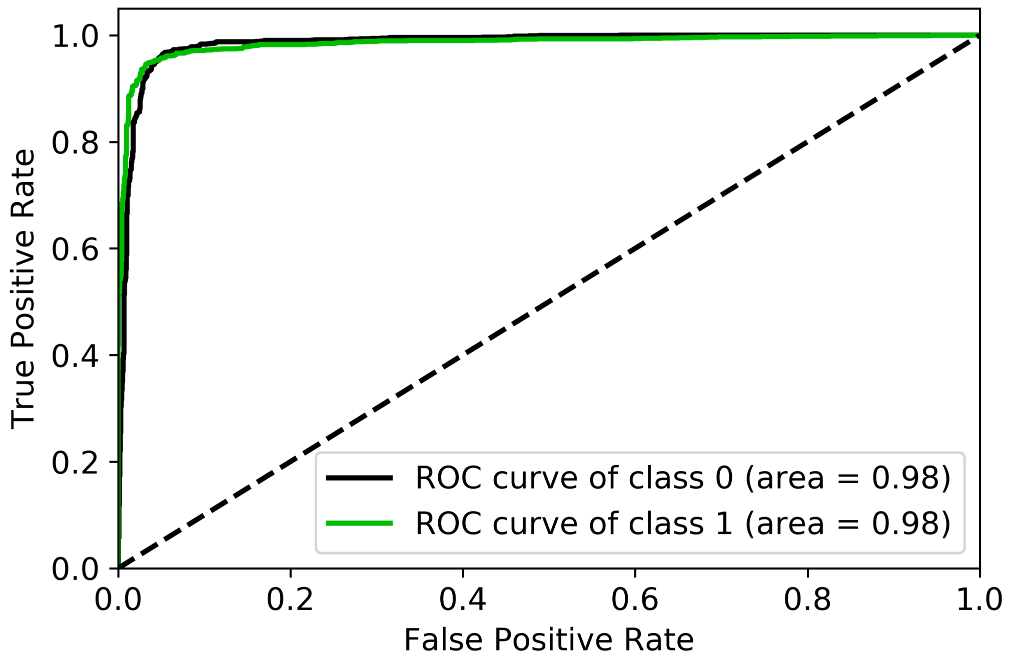


Figure 4.1: Drebin ROC Plot

The purpose of this case study is to evade the classifier; therefore, we perform features injection attack on the Drebin classifier. An overview of features injection attack is presented in Figure 4.2. As discussed in Section 2.5, once the adversary has the knowledge of the classifier and is aware of the data used to train it (the best-case scenario), an attacker can easily circumvent the classifier. The attacker can highlight top features from the training data based on a specific classifier (linear SVM in the case of Drebin) and carefully mutate them to achieve evasion. The attacker has the option of adding a new feature or removing one from the original feature set. Drebin employs a binary features set, where 1 denotes the presence of a feature in the application and 0 denotes its absence. Since removing a feature may alter the semantics of malware, we solely rely on injecting new features into the app (i.e., by changing 0s to 1's in the feature vector). We find the top 20 features for the benign class based on linear SVM classifier and then search for these top features in the feature vectors of malicious apps. If the feature is missing, i.e. 0 in the malicious samples, we change it to 1. The method of adding features is linear, i.e., we change the first top feature in all malicious samples from 0 to 1 and then test the data against the model to determine the evasion rate. Subsequently, the second top feature is altered and tested on the model, and the process is repeated for the remaining top 18 features.

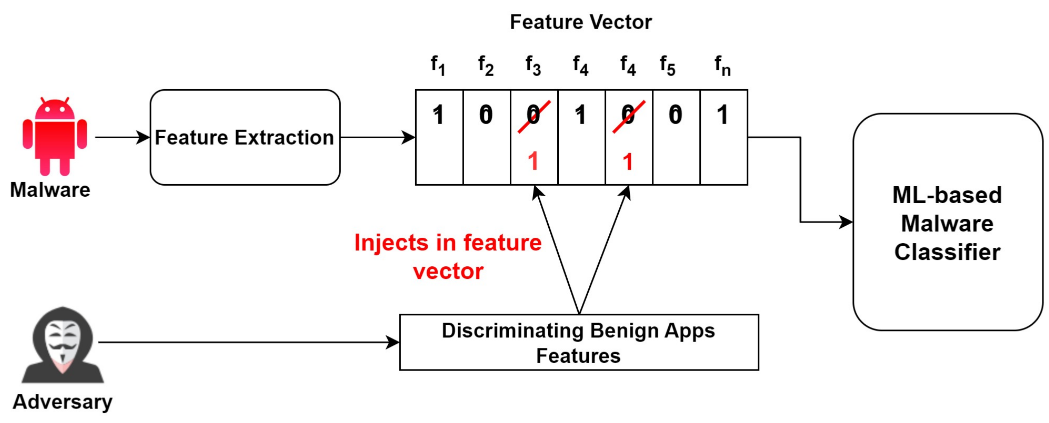

Feature Vector
f1 f2 f3 f4 f4 f5 fn
1 0 0 1 0 0 1
1 1
Malware
Feature Extraction
Injects in feature vector
Discriminating Benign Apps Features
Adversary
ML-based Malware Classifier

Figure 4.2: Feature Injection Attack

As shown in Figure 4.3, our experiments on a dataset of 5560 malicious apps reveal that 48%

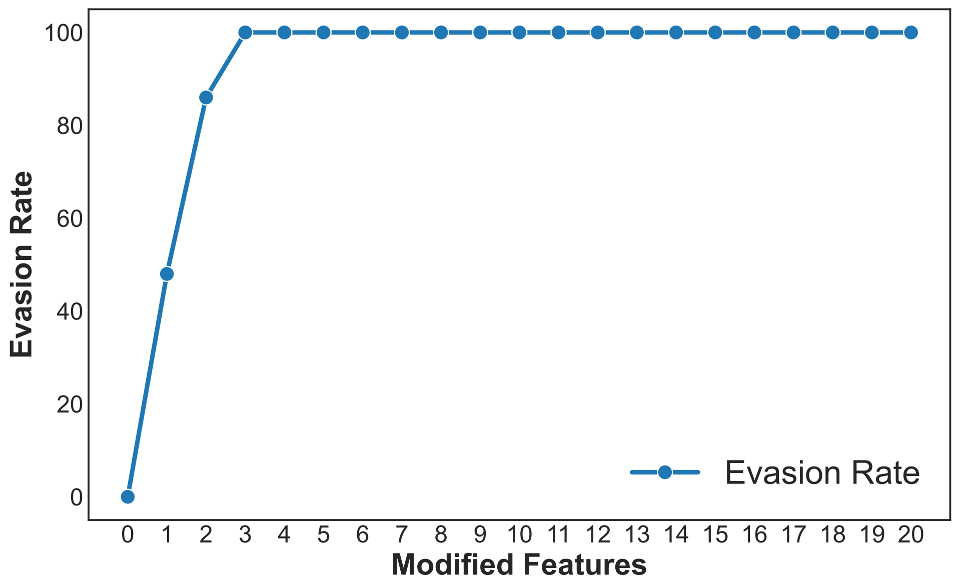


Figure 4.3: Evasion attack on Drebin

of all malicious samples are evaded by just mutating one binary feature in the feature vector. Furthermore, 86% of malware is evaded by just mutating two features, and interestingly, a 100% evasion rate is achieved by mutating only three features in the original feature vector. Consequently, this case study demonstrates the vulnerability of machine learning-based Android malware classifiers in a adversarial environment. Furthermore, the findings of this case study provide a tangible motivation for the development of an Android malware classifier that is not only accurate but also resilient to adversarial evasion attacks.

4.3 Dataset and Feature extraction

We leverage the KronoDroid dataset [185] to access hybrid features extracted from 18,000 Android malware and 18,000 benign apps. The dynamic features include the presence or absence of 288 distinct system calls in Android apps. On the other hand, static features consist of permissions and intents-based features. Since the KronoDroid data set does not contain API call-based

Table 4.1: Overview of Used feature

Feature sets	
Static features	Requested permissions API calls Filtered intents
Dynamic features	System calls

features, therefore in order to make a diverse feature vector, we extract API call-based features from real apps downloaded from the Androzoo dataset [26]. KronoDroid dataset labels each app with a unique SHA256 hash value; therefore, we use the same hash values to collect actual APK files from the AndroZoo dataset and append the extracted API calls with the feature sets obtained KronoDroid dataset. We performed reverse engineering on the Android apps in the dataset to extract API calls-based features. Android apps are programmed using Java and compiled in Dalvik byte code. However, it is possible to reverse engineer an APK in the form of java code using various tools. The process of reverse engineer an Android app is discussed in detail in Section 2.2.4. Table 4.1 lists the Android app features employed in this study.

A brief description of API calls, permissions and intents-based features is presented in Section 3.4.2. In order to build a diverse feature vector, we incorporate system calls in the feature vector in addition to API call, intents and permissions. Android OS is based on the Linux kernel and executes all the apps on the application layer. Whenever an app requires access to the core functionality of the Linux kernel, such as power management or network connection, system calls are used to shift the control from the application layer to the Linux kernel. Likewise, control is returned to the application layer from the kernel mode once the required task is completed. System calls traces can be extracted from Android apps by executing the app in a controlled environment.

4.3.1 Features set modeling

We construct a binary encoded feature vector for each APK in the dataset to train ML-based models. We mark a particular feature's presence in the APK as 1 in the feature vector, whereas absence is marked as 0. Algorithm 4 explains our methodology for feature set modeling. The Algorithm 4 takes as input an APK, a list of unique permissions, a list of unique API calls, a list of unique intent filters and a list of unique system calls. The next step in the algorithm is to extract the permissions, intents, API calls and system calls-based features from each app in the dataset and embed them in separate feature sets, respectively *(Algorithm 4, line 1-4)*. Then we compare each permission in the unique permissions list. If particular permission in the unique permissions list is present in the extracted permissions set, the corresponding permissions vector bit is set to 1; otherwise, the bit is assigned 0 value *(Algorithm 4, line 5-9)*. The same process is applied to construct the intents vector *(Algorithm 4, line 10-14)*, the API calls vector *(Algorithm 4, line 15-19)* and the system calls vector *(Algorithm 4, line 20-24)*. Finally, the four vectors (Permissions, intent filters, API calls and System calls) are concatenated and returned by the algorithm *(Algorithm 3, line 25-27)*.

4.4 CureDroid

This section proposes CureDroid, an Android malware classifier trained on hybrid features and optimised using the tree-based pipeline optimization technique (TPOT). Figure 4.4 presents the block diagram of CureDroid model. CureDroid takes as an input a dataset of malicious and benign Android apps and extracts permissions, intents, API calls and system calls-based features. The most tedious part of ML is to select the best performing algorithm and tune the corresponding hyperparameters. This process can be burdensome and time-intensive brute force search as there are many ML algorithms (24 ML-algorithms in SK-learn Python library), and each algorithm has numerous hyperparameters settings. Therefore, CureDroid employs TPoT [186], an automated ML (Auto ML) tool for this task. TPoT is a genetic programming-based AutoML system to optimize a series of features and ML models to produce maximum classification results for supervised learning. TPoT leverages Scikit-Learn [187], a python programming language library, to access ML algorithms. Furthermore, the operators of the TPoT library correspond to the

Algorithm 4: Feature Set Modeling

```
Input: $APK, Permission_{(list)}, API_{(list)}, Intents_{(list)}, SysCalls_{(list)}$
Output: $Features\_Vector$
1: $Permissions_{(set)} \leftarrow APK$
2: $Intents_{(set)} \leftarrow APK$
3: $API\_calls_{(set)} \leftarrow APK$
4: $SysCalls_{(set)} \leftarrow APK$
5: for $each\ permission \in Permissions_{(list)}$ do
6:     if $permission \in Permission_{(set)}$ then
7:         $Vector_{(Perm)} \leftarrow 1$
8:     else
9:         $Vector_{(Perm)} \leftarrow 0$
10:    end if
11: end for
12: for $each\ intent \in Intents_{(list)}$ do
13:     if $intent \in Intents_{(set)}$ then
14:         $Vector_{(Int)} \leftarrow 1$
15:     else
16:         $Vector_{(Int)} \leftarrow 0$
17:     end if
18: end for
19: for $each\ api \in API_{(list)}$ do
20:     if $api \in API\_calls_{(set)}$ then
21:         $Vector_{(api)} \leftarrow 1$
22:     else
23:         $Vector_{(api)} \leftarrow 0$
24:     end if
25: end for
26: for $each\ syscall \in SysCalls_{(list)}$ do
27:     if $syscall \in SysCalls_{(set)}$ then
28:         $Vector_{(S\_calls)} \leftarrow 1$
29:     else
30:         $Vector_{(S\_calls)} \leftarrow 0$
31:     end if
32: end for
33: $Features\_Vector \leftarrow Concat(Vector_{(Perm)},$
            $Vector_{(Int)}, Vector_{(api)}, Vector_{(S\_calls)})$
34: Return $Features\_Vector$
```

learning algorithm, features preprocessing and features selection algorithm. Gradient boosting (GB) classifier was determined to be the optimal model for categorising Android malware and

Table 4.2: Hyperparameters setting of Gradient Boosting Classifier for HybridDroid Model

Parameter	Value
learning_rate	0.1
max_depth	6
max_features	0.3
min_samples_leaf	11
n_estimators	100
min_samples_split	4
subsample	0.65

benign apps when the TPoT optimization was applied to the hybrid featured dataset. Table 4.2 depicts the information about the selected model and corresponding hyperparameters settings returned by the TPOT optimization technique in the CureDroid model.

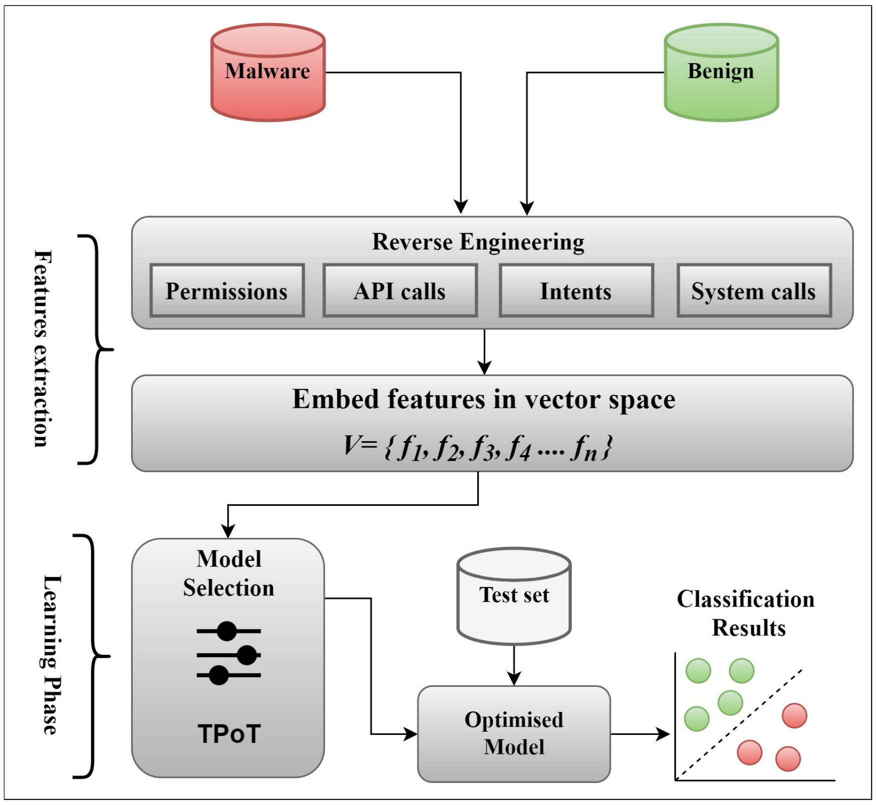


Figure 4.4: Block Diagram of CureDroid

Table 4.3 presents the performance of CureDroid, an Android malware classifier trained on

Table 4.3: Performance of CureDroid Model

Parameter	Value
Accuracy	99.2
Precision	98.9
Recall	99.5
F1-score	99.2

hybrid features and optimized using a tree-based pipeline optimization technique. As shown in Table 4.3, CureDroid achieves remarkable malware classification results (up to 99.2% accuracy). Although CureDroid achieves high performance in classifying Android malware and benign apps, our target is to develop an adversarial evasion aware classifier. Therefore in the next section, we investigate the performance of CurDroid in adversarial environments and present the fragility of high performing ML-based malware classifiers under evasion attacks.

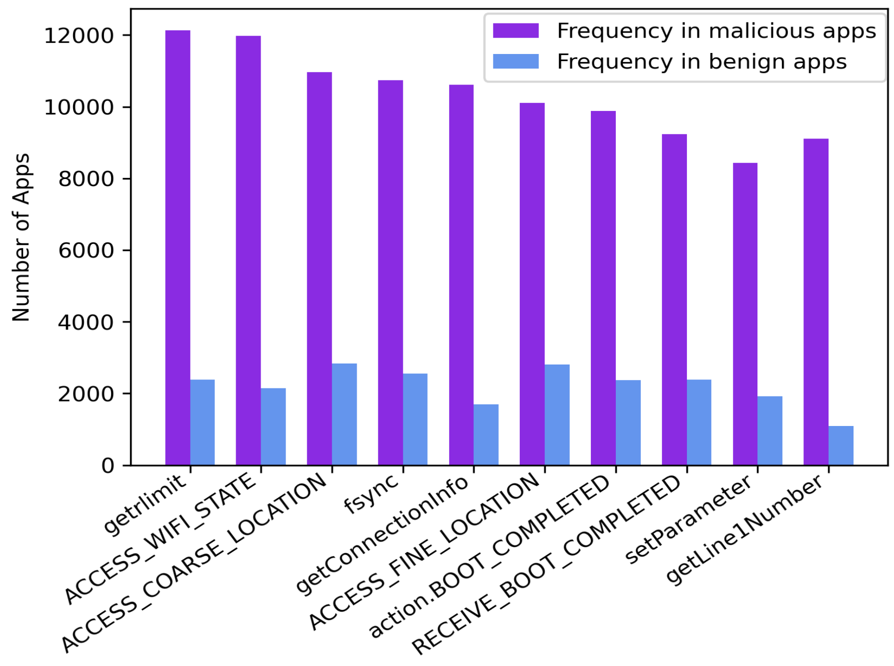
4.5 Adversarial Attacks

In order to present the fragility of ML-based Android malware classifiers in adversarial environments, we evaluate the proposed CureDroid model against adversarial evasion attacks in this section.

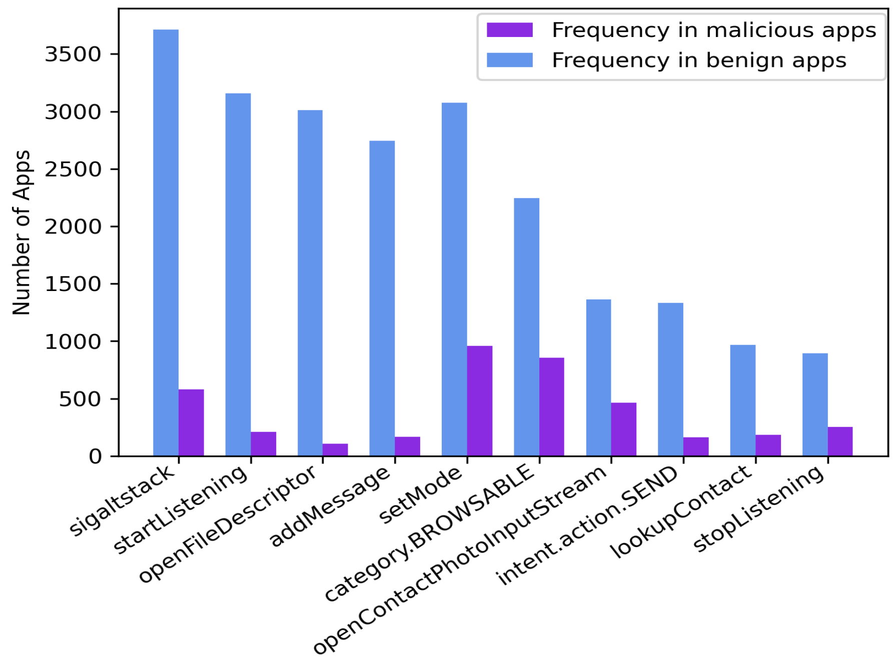
4.5.1 Adversarial Strategies

In order to carry out an evasion attack, attackers would alter the characteristics of malicious software to avoid detection. Algorithm 4 shows that an Android app may be represented as a binary feature vector following the feature extraction process. In order to represent the change made in the original application, the evasion attack typically involves adding or removing a binary from the vector [115]. In previous works such as [188], it has been suggested to randomly perturb binary feature vectors to evaluate classifiers against adversarial evasion attacks. In [188], the authors randomly fabricate up to 40 features in the binary vector and claim that the fabricated samples do not significantly affect the performance of the proposed model. However, our analysis suggests that contrary to the random perturbations in feature vectors, fabricating the discriminating benign or malicious features significantly eludes the classification model. We first identify the top discriminating features between malicious and benign apps within the dataset. We identify a feature as discriminatory based on the frequency with which it appears in malware and benign apps. Figure 4.5(a) presents the top ten discriminating features present in the malicious Android apps, whereas Figure 4.5(b) presents top ten discriminating features present in the benign apps within the dataset. Furthermore, using the discriminating features, we evaluate the proposed CureDroid model in three different adversarial settings:

Mimicry Attack: refers to the process of injecting malicious apps with discriminating characteristics of benign apps in order to force the classification system to generate invalid labels. The pseudo-code of our mimicry attack is presented in Algorithm 5. Dataset of malicious Android apps (M) and top 30 discriminating features from benign class (F_{Top}) are provided as input to the algorithm. Then, we check the presence and absence of the features in F_{Top} within each malicious app m in the dataset. Consequently, if a feature from F_{Top} is missing, i.e. 0 in the malicious sample, we change it to 1 *(Algorithm 5, lines 1-4)*. The process of adding the features is carried out linearly i.e. we mutate 1 top feature in all the malicious samples from 0 to 1 and test the samples on the model *(Algorithm 5, line 6)* to find out evasion rate. Subsequently, the second top feature is mutated and then tested on the model and the same process is applied for the top 30 discriminating features of benign apps.



a Malware Samples



b Benign Samples

Figure 4.5: Discriminating features in Android apps

Algorithm 5: Mimicry Attack

```
Input: $M = \{m_1, m_2, m_3 \ldots\ldots m_n\}$ and $F_{Top} = \{F_1, F_2, F_3 \ldots..F_{30}\}$
Output: $E_{Rate}$
 1: for $all\ i \in F$ do
 2:     for $all\ j \in M$ do
 3:         if $i \in j == 0$ then
 4:             $j[F[i]] \leftarrow 1$
 5:         end if
 6:         $M_{Evade} \leftarrow j$
 7:     end for
 8:     $E_{Rate} \leftarrow Classifier(M_{Evade})$
 9: end for
10: Return $E_{Rate}$
```

Feature Removal Attack (FRA): refers to the process of removing discriminating features of malware from malicious apps. The pseudo-code for feature removal attack remains similar to Algorithm 5 with some modifications. In the case of FRA, the input F_{Top} feature set consists of the top 30 discriminating features of malicious apps. Apart from that, *(Algorithm 5, line 3-4)*, the opposite scenario is considered in the case of FRA, i.e. if a particular feature is present in the malware, it is removed (change 1 to 0).

Mimicry with Feature Removal Attack (MFRA): refers to the process of simultaneously adding and removing discriminating features from malware. At each step of MFRA, one discriminating feature of benign apps is injected, and one discriminating feature of malicious apps is eliminated from the malware. The pseudo-code of our MFRA is presented in Algorithm 6. Dataset of malicious Android apps (*M*), top 30 discriminating features form benign class (F_{Benign}) and top 30 discriminating features form malware class ($F_{Malware}$) are provided as input to the algorithm. Then, we check the presence and absence of the features in F_{Benign} and $F_{Malware}$ within each malicious app *m* in the dataset. Consequently, if a feature from F_{Benign} is missing, i.e. 0 in the malicious sample, we change it to 1 and if a feature from $F_{Malware}$ is present, i.e. 1 in the malicious sample, we change it to 0 *(Algorithm 6, lines 5-6).* The process of injecting and eliminating the features in each app in the dataset is carried out linearly i.e. we mutate one top malware and benign feature simultaneously in all the malicious samples and test on the model *(Algorithm 6, line 8)* to find out evasion rate.

Algorithm 6: MFRA Algorithm

```
Input: $M = \{m_1, m_2, m_3 \ldots\ldots m_n\}$
$F_{Benign} = \{F_1, F_2, F_3 \ldots..F_{30}\}$
$F_{Malware} = \{F_1, F_2, F_3 \ldots..F_{30}\}$
Output: $E_{Rate}$
 1: for $i = 0\ to\ 30$ do
 2:     for $all\ j \in M$ do
 3:         if $F_{Benign}[i]\ in\ j\ is\ 0$ then
 4:             $j[F_{Benign}[i]] \leftarrow 1$
 5:         end if
 6:         if $F_{Malware}[i]\ in\ j\ is\ 1$ then
 7:             $j[F_{Malware}[i]] \leftarrow 0$
 8:         end if
 9:         $M_{Evade} \leftarrow j$
10:     end for
11:     $E_{Rate} \leftarrow Classifier(M_{Evade})$
12: end for
13: Return $E_{Rate}$
```

In an ideal case, regardless of mimicry attack, FRA or MFRA, the adversarial settings should preserve the semantics of the original malware. Compared to FRA and MFRA, mimicry attacks are practically simpler to perform and preserve the semantics of the malicious app. Mimicry attacks are performed by injecting discriminating features of benign apps into the malicious apps. Practically, this can be achieved by adding functions in the source code that are never called, adding source code after return statements, and including fake permissions in the *manifest.xml* file.

On the other hand, FRA and MFRA is only feasible if the adversary can eliminate specific features without compromising the malicious functionality of the app. Therefore, in practice FRA and MFRA is complex to implement and may limit the functionality of the app. Some features such as strings, services, receivers and providers are listed in manifest file and implemented as java source code. These features can be renamed in order formulate feature elimination. Furthermore, an attacker can encode API calls and decode them dynamically using reflection in order to imitate API calls removal. Nevertheless, in this study, we assume that the attacker has ability to remove feature and therefore we evaluate the model in under all three adversarial settings.

4.5.2 Attack on CureDroid

We evaluate the proposed CureDroid model against mimicry attack, FRA and MFRA to investigate the adversarial robustness of the proposed model in terms of evasion rate. *Evasion rate* is defined as the ratio of misclassified malware samples to the total malware samples in the test set. Figure 4.6 presents the results of adversarial evasion attacks on the CureDroid model. The results indicate that CureDroid is vulnerable to all three adversarial attacks. As shown in Figure 4.6, by injecting up to eight discriminating features of benign apps in malware, a mimicry attack can attain up to 40% evasion rate. Since we limit the mimicry attack to 30 perturbations, the maximum evasion rate achieved by mimicry attacks reached up to 59%.

In the case of FRA, the attack method achieved up to 55% evasion rate by removing ten features. Likewise, the evasion rate increased up to 75% by removing 14 features. Finally, FRA achieved a 100% evasion rate when 20 discriminating features were removed from the malware app. In the case of MFRA, the adversarial attack achieved up to 45% evasion rate by just adding and removing three features in the malicious apps. Furthermore, the evasion rate increased to 71% after six iterations of MFRA and reached over 90% after nine iterations. Our experimental results show that the MFRA is the most effective adversarial attack against CureDroid. The reason is that MFRA simultaneously injects discriminating features of benign apps and removes discriminating features of malicious apps from the malware at each iteration. Consequently, at each iteration, MFRA significantly changes the distribution probability of malware towards benign class, and as a result, the model misclassifies the sample.

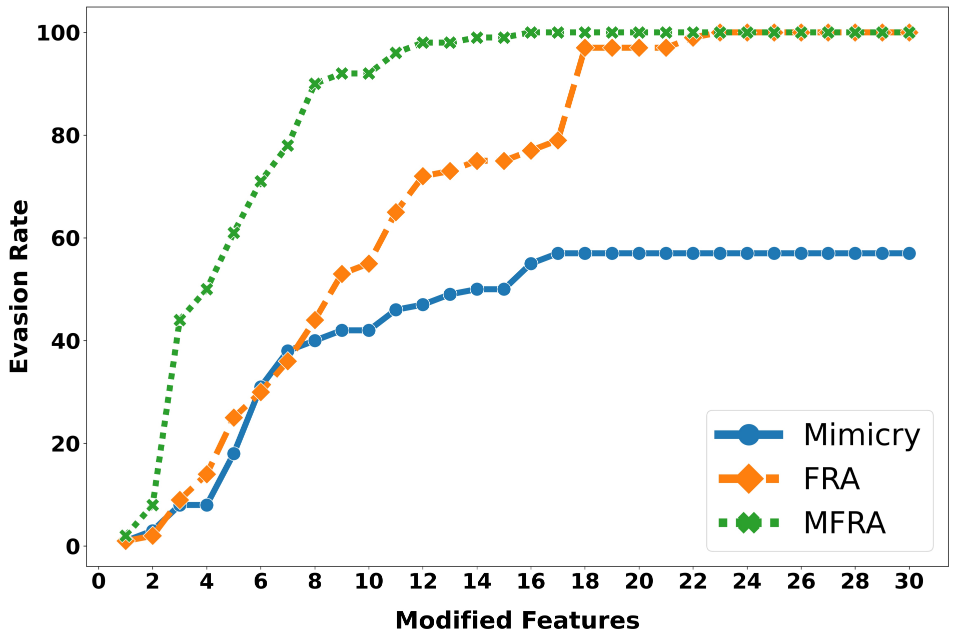


Figure 4.6: Adversarial Attacks on CureDroid

4.6 A Countermeasure: CureDroid*

In this section, we present CureDroid*, an adversarially robust extension of CureDroid*. In order to create an adversarially robust ML-based classification system, generally, three approaches are taken into consideration [114]: 1) adversarial training of the classifier; 2) combining several classifiers to create an ensemble model and 3) hardening target classifiers against evasion attacks. In this work, we focus on employing ensemble classifiers and making the attacker's job challenging to evade the malware detection model.

Figure 4.7 presents the block diagram of the CureDroid* model. Following is a step by step explanation of the CureDroid* model:

- A dataset of Android malware and benign apps in the form of APK files is taken as input.
- The APK files from the dataset are reverse engineering in order to extract API calls, in-

tents, permissions and system calls-based features.

- Instead of just considering one mixed feature vector, CureDroid* considers subsets of features (APIs, intents, permissions and system calls) and trains ML models on each subset of features separately. The selection of ML-model and the hyperparameters settings is determined by using TPoT.
- The final label generated by the CureDroid* is based on 'OR' operation on the labels produced by the each classifiers in the ensemble pipeline.

As discussed earlier, an attacker can perform mimicry, FRA and MFRA-based feature manipulations to elude an ML-based malware detection system. A slight feature vector mutation can result in the misclassification of a malicious app. CureDroid* model can be employed in order to counter such attacks. Since CureDroid* employs multiple models trained on distinct feature sets, therefore modifying a feature might affect the classification accuracy of a particular classifier in the ensemble (such as the intents classifier). However, the model will still be able to detect the malware using other classifiers in the ensemble model (such as system calls, APIs and permissions). Consequently, CureDroid* makes it challenging for the attacker to evade the model as the adversary will need to elude all the classifiers in the ensemble model.

4.6.1 Experimental Results

The process of model selection and optimization for CureDroid* is performed using the tree-based pipeline optimization technique (TPoT). The classifiers are trained on a dataset of 18,000 malware and 18,000 benign apps (discussed in Section 4.3).

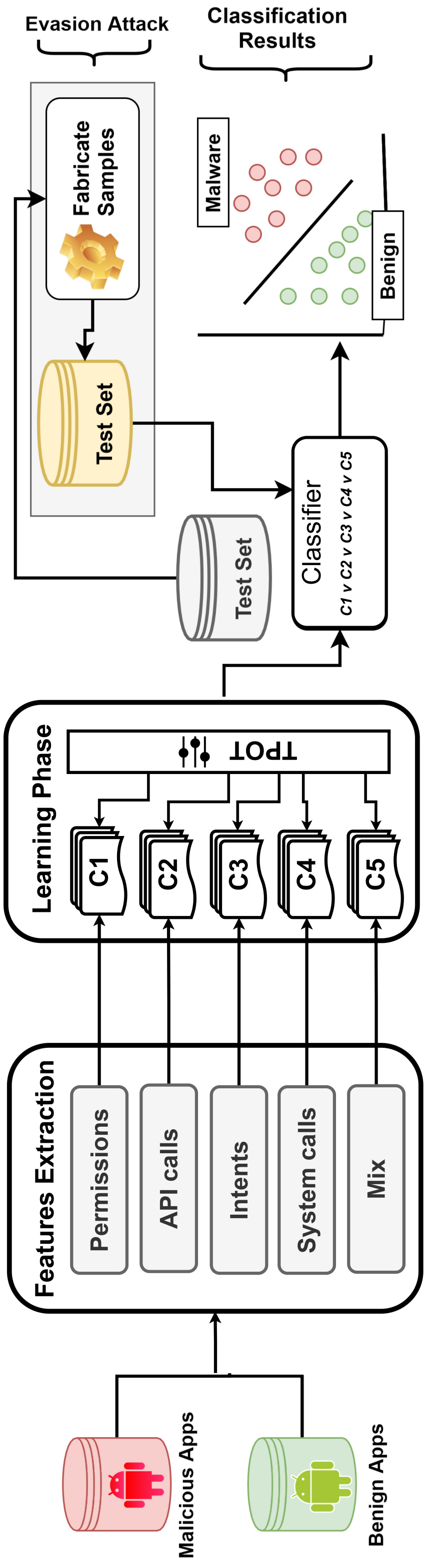


Figure 4.7: Block Diagram of CureDroid* Model

Table 4.4: TPOT model selection for Feature subsets

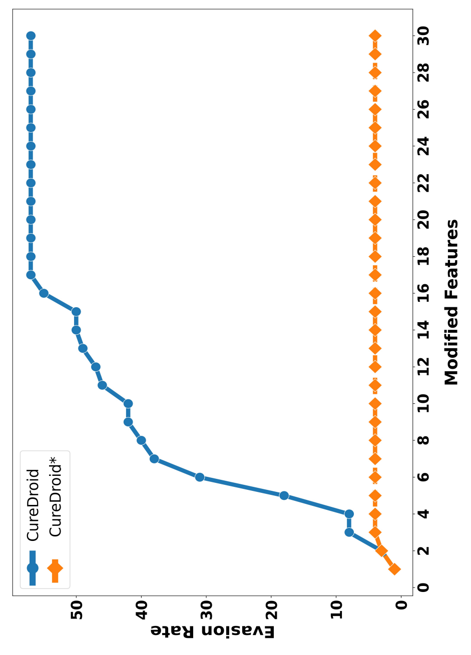
Features class	**Classifier**	**Hyper-parameters settings**					
System calls	Extra Trees	bootstrap	criterion	max_features	min_samples_split	min_samples_leaf	n_estimators
		false	*entropy*	0.65	4	1	100
API calls	Perceptron	alpha			learning_rate_init		
		0.001			0.001		
Intents		0.01			0.01		
Permissions	Gradient boosting	max_depth	max_features	min_samples_leaf	min_samples_split	subsample	n_estimators
		6	0.3	11	4	0.65	100
Mix features	XGB	9	0.75	2	7	0.35	100

Table 4.4 presents the details about selected classifiers and corresponding hyper-parameters setting using TPoT. The ensemble model comprises five classifiers trained on system calls, API calls, intents, permissions and mixed features independently. Table 4.5 presents the results of classifiers in ensemble model in non-adversarial settings. The results indicate that all the classifiers in the ensemble model have individual potential to classify Android malware and benign app with high accuracy. Since the final label generated by CureDroid* is based on OR operation on labels generated by the classifiers in the ensemble model, CureDroid* can achieve up to 99.2% malware classification accuracy in non-adversarial settings.

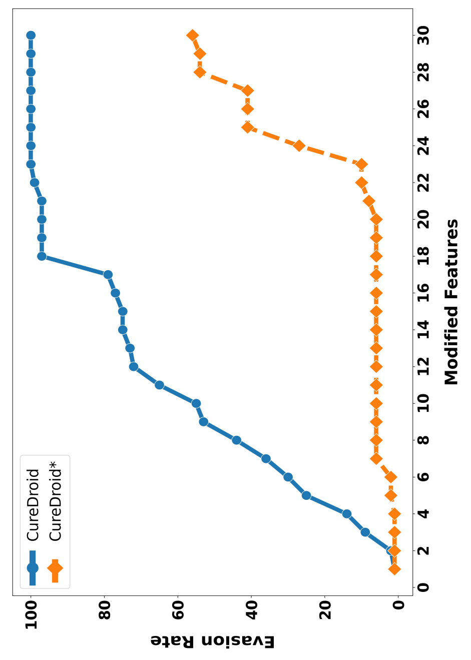
Table 4.5: Android malware classification results based on various feature subsets

Features set	**Accuracy**	**Recall**	**Precision**	**F-measure**
Permissions	96.1	96.2	96	96.1
API-calls	98.4	98.	98.9	98.4
Intents	86.6	79.5	92.2	85.4
System calls	87.5	87	87.5	87.3
Mix	99.2	99.5	98.9	99.2

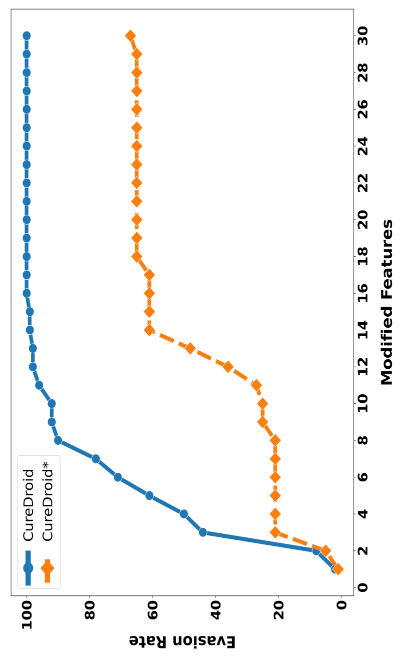
Figure 4.8 presents the performance of the CureDroid* model in adversarial environments. We perform mimicry attacks, FRA and MFRA on CureDroid* and compare its adversarial robustness with the CureDroid model. The results indicate that compared to CureDroid, CureDroid* significantly improves the security of the model against adversarial evasion attacks. The reason is that CureDroid employs multiple classifiers, and each classifier in the ensemble model is trained on distinct features. Therefore the adversary needs to elude all the classifiers in the ensemble model in order to evade the model. As shown in Figure 4.8(a), the maximum evasion rate achieved by mimicry attacks is 4% with up to 30 modifications in the malware features. In contrast, the mimicry attack achieved up to 60% evasion rate in the case of the CureDroid model.



a Mimicry Attack



b FRA



c MFRA

Figure 4.8: Performance of CureDroid* in Adversarial settings

Furthermore, Figure 4.8(b) depicts the performance of CureDroid* against FRA. The results indicate that the evasion rate is significantly reduced compared to the CureDroid. CureDroid* retains a 2% evasion rate when up to 6 features are modified and increases to 6% when up to 20 features are perturbed. FRA achieves a significant increase in evasion rate when more than 20 features are modified (up to 56%). Similarly, Figure 4.8(c) presents the results of MFRA on CureDroid*. Compared to mimicry and feature removal attacks, CureDroid* is less effective in the case of MFRA. Nevertheless, compared to CureDroid, CureDroid* significantly reduces the evasion rate against MFRA. It retains a 20% evasion rate when up to eight iterations of MFRA are completed. In comparison, 90% of the sample generated using MFRA evaded the CureDroid model when up to 8 modifications were performed.

4.7 Performance Comparison with state-of-the-art

This section briefly discusses the state-of-the-art techniques related to the CureDroid* model. We also perform a comparison of the related techniques with CureDroid*. Table 4.6 shows the comparison results in terms of technique, publication year, target classifier/technique, the dataset used, the technique's evasion rate, and finally, whether the said approach provides a countermeasure for evasion attack or not. As shown in the table 4.6, [140, 189, 190, 191] focuses only on evading the existing techniques. Although these techniques achieve high evasion rates, however authors have not provided any countermeasure to mitigate such attacks. Compared to these techniques, our proposed evasion algorithms achieved a remarkable upto 90% evasion rate on the target classifier. We also provide a countermeasure (CureDroid*) which can be employed to mitigate such evasion attacks.

The authors in [121] and [117] have evaded target classifiers and proposed countermeasure techniques to mitigate such attacks. Grosse et al. [121] performed evasion attacks on classifiers based on deep neural networks. They achieved an evasion rate of up to 63% on the Drebin dataset with an average of 20 perturbations in feature vectors. Grosse et al. also proposed two

Table 4.6: A comparison among different evasion techniques related to CureDroid*

References	Year	Target	Dataset	Evasion Rate	Countermeasure
Android HIV [140]	2019	Drebin (SVM)	Drebin	99%	No
TLAMD [189]	2019	Random Forest	Drebin	93%	No
Harel [190]	2020	Drebin (SVM)	Drebin	99%	No
Mystiqe-S [191]	2017	Antivirus	Custom	94%	No
Grosse [121]	2016	Deep Learning	Drebin	63%	Yes
LagoDroid [117]	2018	RevealDroid	Custom	97%	Yes
CureDroid*	***2022***	***CureDroid (GB)***	**Drebin/AMD**	***upto 90%***	***Yes***

countermeasure techniques for adversarial evasion attacks: distillation and classifier retraining. However, none of the proposed defensive mechanisms provided promising results against evasion attacks providing a maximum of 33% adversary detection in case of classifier retraining. Furthermore, LagoDroid [117] evaded a recent classifier called RevealDroid [118] with an evasion rate up to 97%. Authors in [117] also proposed a countermeasure called RevealDroid* to mitigate evasion attacks on RevealDroid [118]. Although RevealDroid* performs well against a small number of modifications, the performance of RevealDroid* declines if the number of modifications is high. Moreover, RevealDroid* requires many ensemble classifiers to detect potential evasion. The authors have employed an ensemble of 16 decision tree-based classifiers to perform experiments. In contrast, we used an ensemble of 5 ML-based classifiers and achieved high adversarial detection i.e. up to 30 modifications in case of MA, 21 modifications in case of FRA and 9 modifications in case of MFRA in the actual feature vector.

4.8 Summary

In this chapter, we proposed CureDroid, an Android malware classifier trained on hybrid features and optimized using a tree-based pipeline optimization technique. Although CureDroid achieves a remarkable malware detection accuracy (up to 99.2%), we present the fragility of the proposed technique in adversarial environments. We performed mimicry attacks, FRA and MFRA, to evade CureDroid. Moreover, we propose CureDroid*, a TPoT-based adversarially robust extension of CureDroid. CureDroid* consider multiple classifiers trained on the distinct feature sets extracted from Android apps. A single feature vector is distributed into multiple subsets such that an ML-based classifier trained on each distinct subset has the individual potential to detect Android malware and benign apps. We performed an empirical case study to evade CureDroid* using mimicry, FRA and MFRA and prove the effectiveness of CureDroid* in adversarial settings. Our experiments indicate that CureDroid* significantly reduces the evasion rate compared to CureDroid in adversarial settings.

Chapter 5

An Oracle and GAN-based Cumulative Adversarial Training Technique to improve Evasion detection for Android Malware

5.1 Introduction

The adversarial evasion attacks are primarily dependent on the attacker's insight to defender's feature set of training data [192]. The detection model makes a prediction based on ranked features that can be a piece of sensitive information for the attacker. The attacker can make a slight change into any of the top-ranked features to generate an adversarial sample [193], [121]. However, such attacks are based on the domain knowledge of the attacker. Although the Android malware detectors can hide the underlying model, however, there are many publicly available Android malware datasets that can help the attacker to get insights into the training data [190]. So there is a large gap to fill in research for adversarial evasion detection considering the publicly available datasets while designing a sophisticated Android malware detector.

We highlight the fragility of ML classifiers such as support vector machine (SVM), logistic regression (LR), perceptron (PT), decision tree (DT), random forest (RF) and xgboost (XGB) to compare their effective candidacies for the adversarial malware detection. We have performed Oracle and Generative Adversarial Network (GAN) based adversarial attacks against a practical dataset called *Drebin* that is publicly available [24]. We propose a technique to generate adversarial evasion examples that fool the classifiers mentioned above. It has been demonstrated that the linear classifiers such as SVM, LR, and Perceptron (PT) are least effective in contrast to their ensemble counterparts in the adversarial malware detection for Android. Since there is no silver bullet defence against evasion attacks, therefore, only proactively knowing the attacker's manipulations could be cardinal to a robust defence strategy [192]. This is where the concept of adversarial training could be exploited to form an effective and proactive defence [194]. We propose a robust adversarial training scheme called ***TrickDroid*** based on cumulative adversarial training of ensemble classifiers on Oracle and GAN based adversarial data to improve evasion detection. Finally, we compare our results with adversarial training of individual Oracle and GAN based attacks and adversarial training.

Following are the main contributions of this chapter:

1. We highlight the fragility of the ML classifiers against adversarial evasion attacks. We perform mimicry attacks based on Oracle and Generative Adversarial Network (GAN) against these classifiers using our proposed methodology.
2. We demonstrate by experiments that that among ML classifiers, the detection capability of linear classifiers can be reduced as low as 0% by perturbing only up to 4 out of 315 extracted API features.
3. As a countermeasure, we propose *TrickDroid*, a cumulative adversarial training scheme based on Oracle and GAN-based adversarial data to improve evasion detection.

5.2 Proposed Attacks Methodology

Our proposed methodology of evasion attacks is illustrated in Figure 5.1 which shows the key components of the system. In the feature extraction module, we reverse engineer the Android applications to extract API-based features. The extracted features are further used to train multiple ML classifier models. To evade the trained classifiers, we generate code injection and GAN-based adversarial data in the adversarial samples generation module. The adversarial samples are further tested on the existing pre-trained classifiers. Finally, we perform adversarial training to harden the security of Android malware classifiers against adversarial evasion attacks.

5.2.1 Dataset and Feature Extractor

In this study, we use Drebin [24] as a benchmark dataset. The dataset is composed of 5560 malicious and 213,453 benign applications. We randomly select 5600 benign applications to balance the dataset. Furthermore, we reverse engineer the Android application packages (APKs) in the dataset to extract java source code. APKs are decompiled in the form of *.dex* and then transformed into *.jar* files. The *.jar* files are then disassembled into java source code in order to extract features. Static analysis is applied on the reverse-engineered code to extract API-based features from the Android applications. API-based features tend to be strong behaviour-based features for malware classification [18, 195]. A total of 315 unique API calls were found from all of the applications in the dataset. Furthermore, each application in the dataset is transformed into a feature vector of length 315. Each cell of the feature vector contains a binary value where 1 represents the presence of a specific feature and 0 represents its absence.

5.2.2 ML Models Segment

We use SVM, LR, PT, DT, RF and XGB classifiers on API-based features of APKs. All of the classifiers use default hyper-parameters setting (provided in sklearn 1.0.1 python library), whereas 10-folds are used for cross-validation. We randomly distribute the dataset into 80%

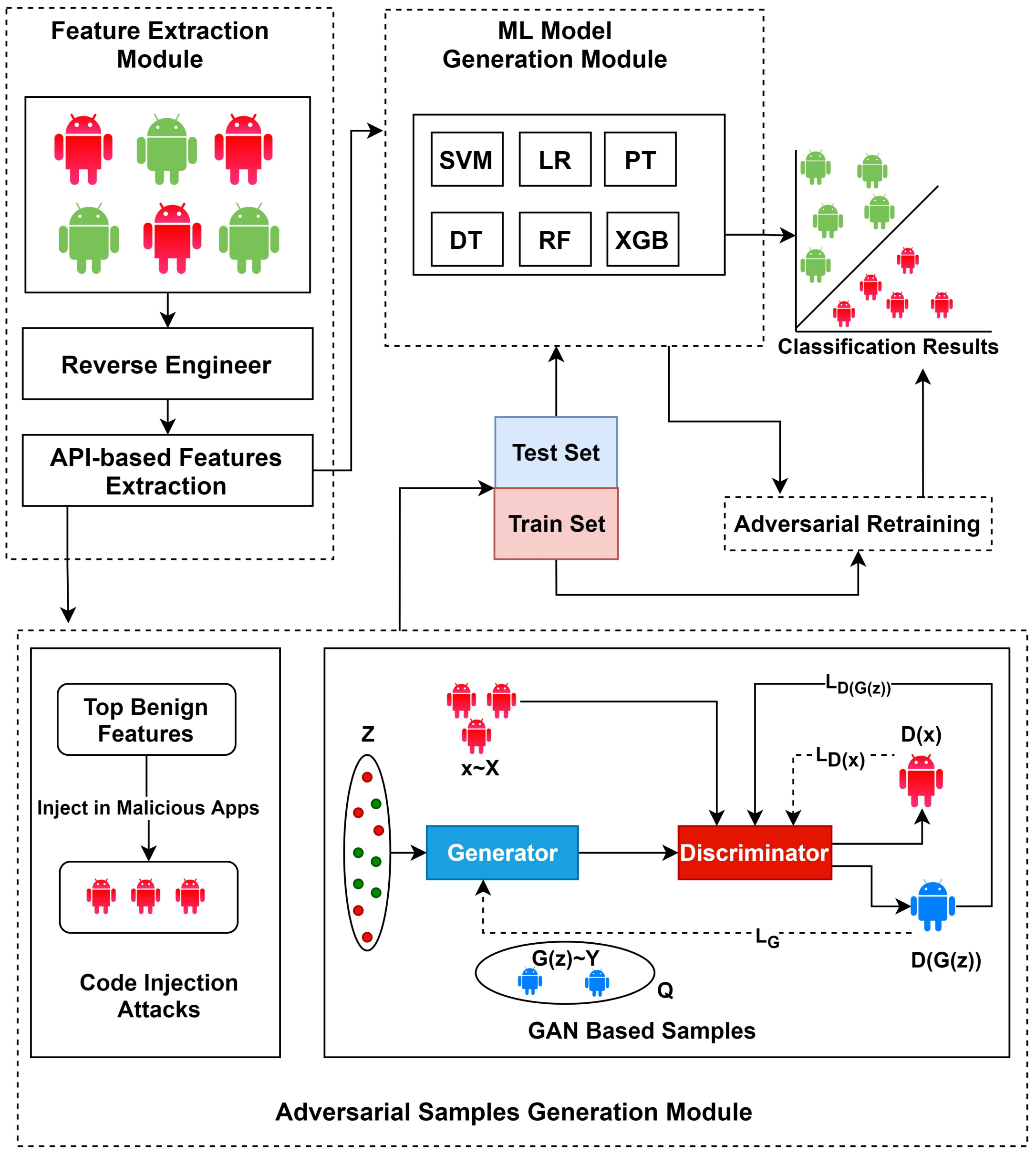


Figure 5.1: Evasion Attacks Methodology

training set and 20% testing set for each iteration for cross-validation. In order to present the fragility of the Android malware classifiers, code injection and GAN-based adversarial evasion attacks will be applied. Subsequently, we will perform adversarial training to harden the security of models against such attacks.

5.2.3 Evasion Attacks Generator

In this section, we discuss our evasion attack strategies against pre-trained Android malware classifiers. The first step is to train the ML classifiers on an API-based dataset. Once the classifiers are trained, we apply code injection and GAN based evasion attacks on the classifiers.

5.2.3.1 Code Injection Attacks (CIA)

To perform code injection attacks (CIA), we first find the top 20 most discriminating features from benign Android applications from the dataset and then inject those one by one in the malicious applications. It has been observed that many API-based features are frequently used and are overlapping in both malicious and benign applications e.g. *StartActvity()*, *GetDeviceId()*, *GetActiveNetworkInfo()* etc. However, some features are highly discriminating in the sense of defining the class of Android applications, e.g. *sendTextMessage()* API is present in 1903 malicious applications in the dataset, whereas only 42 benign applications call this API.

Algorithm 7 shows the pseudo-code of features injection attack. The dataset of malicious Android apps M and top 20 features of benign class from Drebin F_{Top} are provided as input to the algorithm. Once the top features are identified, we look for those in the feature vectors of malicious apps. If a feature is missing, i.e. 0 in the malicious samples, we change it to 1 *(Algorithm 7, lines 1-4)*. The process of adding the features is carried out linearly, i.e. we mutate 1 top feature in all the malicious samples from 0 to 1 and test the samples on the model *(Algorithm 7, line 6)* to find out the evasion rate. Subsequently, the second top feature is mutated and then tested on the model and the same process is applied for the top 20 features.

5.2.3.2 GAN Adversarial Examples Attacks (GAEA)

A GAN is a combination of two neural networks, among which one is called generator ($\mathcal{G}$) and the other is known as discriminator ($\mathcal{D}$). ($\mathcal{G}$) generates the data and ($\mathcal{D}$) evaluates this generated

Algorithm 7: CIA Algorithm

```
Input: $M = \{m_1, m_2, m_3 \ldots\ldots m_n\}$ and
$F_{Top} = \{F_1, F_2, F_3 \ldots\ldots F_{20}\}$
Output: $E_{Rate}$
 1: for $all\ i \in F$ do
 2:     for $all\ j \in M$ do
 3:         if $i \in j == 0$ then
 4:             $j[F[i]] \leftarrow 1$
 5:         end if
 6:         $M_{Evade} \leftarrow j$
 7:     end for
 8:     $E_{Rate} \leftarrow Classifier(M_{Evade})$
 9: end for
10: Return $E_{Rate}$
```

data. Both these networks are connected in a way that the loss of $\mathcal{D}$ is fed back to $\mathcal{G}$ while $\mathcal{D}$'s weights are not updated so that $\mathcal{G}$ can try to follow the real data probability distribution more efficiently and fool the $\mathcal{D}$. In this work, the primitive version of GAN, also called vanilla GAN was used, to keep the experiments simplistic for estimating GANs potential for Android API based data generation. There is a further research gap for the exploration of a suitable GAN for the generation of Android API data. We leave this as future work. The generator model $\mathcal{G}$ in original/vanilla GAN can be represented as $\mathcal{G}$:z$\rightarrow$ $\mathcal{X}$ where z is the normal distribution from noise space and $\mathcal{X}$ is the real data distribution. The discriminator $\mathcal{D}$:$\mathcal{X}$ $\rightarrow$[0,1] model is a classifier that outputs an estimate of probability how much the data coming from $\mathcal{G}$, is real or fake. The loss function of the combined model can be represented by Equation 5.1.

$$\begin{split}\min_{\mathcal{G}} \max_{\mathcal{D}} V(\mathcal{D},\mathcal{G}) = \mathbb{E}_{x\sim p_{data}(x)}[\log \mathcal{D}(x)] + \\ \mathbb{E}_{z\sim p_z(z)}[\log(1-\mathcal{D}(\mathcal{G}(z)))]\end{split} \tag{5.1}$$

Here, $\mathbb{E}$ stands for the probability estimation; x and z are the real and noise samples, respectively, while p_{data} and p_z represent the probability distributions of real and noise data. The goal in the mini-max game is to minimise the $\mathcal{G}$ loss in creating data similar to the real data. Since

Table 5.1: GAN Configuration

Parameter	**Value**
Network Type	Densely Connected Feed Forward
Number of Layers	$\mathcal{G}$: 5, $\mathcal{D}/C$: 4
Input Layer Activations	$\mathcal{G}$: relu , $\mathcal{D}$: relu
Output Layer Activations	$\mathcal{G}$: sigmoid , $\mathcal{D}$: sigmoid
Batch Size	128
Multiplier(n)	128
Neurons in Input Layer	$\mathcal{G}$: 128 , $\mathcal{D}$: 128
Neurons in Layer 1	$\mathcal{G} : n \times 1 = 128, \mathcal{D} : n \times 2 = 256$
Neurons in Layer 2	$\mathcal{G} : n \times 2 = 256, \mathcal{D} : n \times 1 = 128$
Neurons in Layer 3	$\mathcal{G} : n \times 3 = 384$
Neurons in Output Layer	$\mathcal{G}$: 315, $\mathcal{D}/C$: 1
Layer Regularization	$\mathcal{G}, \mathcal{D}$: $BatchNorm$
Optimizer	Adam (beta_1=0.5, beta_2=0.9)
Loss Function	binary cross entropy
Learning Rate	1e-5

the generator can not control the loss of $\mathcal{D}$ on real data but it can maximise the loss of $\mathcal{D}$ on generated data $\mathcal{G}(z)$. The loss function of $\mathcal{G}$ is given by Equation 5.2.

$$J^{\mathcal{G}}(\mathcal{G}) = \mathbb{E}_{z \sim p_z(z)}[\log(\mathcal{D}(\mathcal{G}(z)))] \tag{5.2}$$

Table 5.1 shows the hyperparameter settings for the GAN model. It can be observed from this table that we used 'sigmoid' in the output layer of $\mathcal{G}$ due to the reason that we wanted to generate the API data in which the values need to be between 0 and 1.

We propose a GAN based methodology inspired by [196] that could mimic and generate the API based APK feature set. We propose the GAN evaluation by tweaking the classifier two-sample test (C2ST) [197] for $\mathcal{G}$ performance evaluation. The C2ST is a quantitative metric to compare two different samples of data. In other words, if we have samples real_APK_API data ($\mathcal{X}_m$) and GAN_APK_API data ($G(z)$), then we can assess if both samples have similar probability

Algorithm 8: C2ST Algorithm

```
Input: $\mathcal{X}_m$ (real_APK_API samples), $G(z)$ (GAN_APK_API samples), Classifier
Output: $\mathcal{A}$(accuracy)
1: $t_r \leftarrow \mathcal{X}_m[0 : m(8/10)] \cup G(z)[0 : m(8/10)]$
2: $t_s \leftarrow \mathcal{X}_m[m(8/10) : m] \cup G(z)[m(8/10) : m]$
3: train ML_classifier on $t_r$
4: test ML_classifier on $t_s$
5: Return $\mathcal{A}$ = (TP + TN)/(TP + TN + FP + FN)
```

distributions. The more the distributions overlap, the more is the chance that GAN_APK_API samples are realistic. The C2ST method has been shown in Algorithm 8. Here, $\mathcal{A}$ denotes the accuracy after splitting the input m samples from ($\mathcal{X}$) i.e. ($\mathcal{X}_m$) into 80% train set t_r and 20% test set t_s. The accuracy $\mathcal{A}$ is computed as per the Equation 5.4.

The GAN evaluation used in GAEA is different from C2ST in the evaluation parameter. The intuition is that the metric in C2ST, i.e. 'accuracy', should be replaced with the evasion rate (e_{Rate}) if we want to reduce the false negatives in the classifier performance in post augmentation testing. The false negatives are the possible evasions that are already present in the test set, which the classifiers are not trained on. Hence, the C2ST has been tweaked so that the objective function becomes as given in the Equation 5.3. In Equation 5.3, $e_{Rate(\hat{argmax})}$ is the evasion rate on D_{test} which is test set, n_{test} is the total number of samples in test set, z_i are the samples in test set, l_i are the labels, $f(z_i)$ is the conditional probability distribution $p(l_i = 1|z_i)$ and $\mathbb{I}$ is the indicator function. The intuition is that if a GAN_APK_API data is very close in probability distribution with a real_APK_API samples, then the evasion rate in Equation 5.3 should remain close to 100%. This means that the classifier was totally evaded, or the sample was misclassified as real_APK_API data. So if we use the evasion rate as the metric instead of accuracy, then we can better minimise the false negatives due to the reason that accuracy includes the value for false positives (FP) and true negatives (TN) given by Equation 5.4. Since our objective function is to minimise false negatives in generator evaluation so we must choose the epochs in which the evasion was the highest instead of accuracy being the lowest.

$$e_{Rate(\hat{argmax})} = \frac{1}{n_{test}} \sum_{z_i, l_i \in D_{test}} \mathbb{I}[\mathbb{I}(f(z_i) > \frac{1}{2}) = l_i] \tag{5.3}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.4}$$

However, in the evasion rate, we only have true positives (TP) and false negatives (FN) as given by Equation 5.5.

$$Evasion_{Rate} = \frac{FN}{TP + FN} \tag{5.5}$$

In Algorithm 9, first of all, we need to extract the malicious real_APK_API samples $\mathcal{X}_m$ from the preprocessed train set $\mathcal{T}$. Then we create GAN models and start training for 150 epochs in which for the batch size of 128, in each batch iteration, x_i is taken as a random batch from $\mathcal{X}_m$. We use the normal distribution of mean = 0 and standard deviation = 1 of the same size as of batch for noise input to $\mathcal{G}$. The $\mathcal{G}$ and $\mathcal{D}$ compute their gradients and update in backpropagation. After each epoch, we generate data G_{z_i} of size equal to $\mathcal{X}_m$ and add in a set U. Now, we can perform the proposed method to evaluate the performance of $\mathcal{G}$ in terms of evasion rate e_{Rate}. We perform 10-fold train-test split with 70-30 ratio and compute the average evasion rate $e_{Rate_{avg}}$. After the training is complete, we use the weights of the $\mathcal{G}$ for the epoch in which the value of $e_{Rate_{avg}}$ was maximum and generate GAN_APK_API data $Gz_{\mathcal{I}_{argmax}(e_{Rate})}$. The GAEA algorithm then outputs the E_{Rate} illustrated in Figure 5.4 the details of which will be mentioned in section 5.3.

5.3 Experimental Results and Analysis

In this section, we evaluate the performance of different ML classifiers against code injection attacks (CIA) and GAN adversarial examples attacks (GAEA). Furthermore, we perform adversarial training of ML classifiers on CIA called CIA Adversarial Training AT or 'CIA AT' and GAEA Adversarial Training or 'GAEA AT' to improve the evasion detection of classical ML classifiers and evaluate against evasion attacks. We also perform GAEA on classifiers trained

Algorithm 9: GAEA Algorithm

Input: $\mathcal{T}$ (preprocessed train set in csv format), batch_size, epochs, batches, Classifier

Output: E_{Rate}

```
 1: $\mathcal{X}_m \sim \mathcal{T}$
 2: Create $\mathcal{G}$ and $\mathcal{D}$ models
 3: for $i \in epochs$ do
 4:     for $j \in batches$ do
 5:         $x_i \sim \mathcal{X}_m$
 6:         $z_j \longleftarrow \mathcal{N}_{\{mean=0,std=1,size=batch\_size\}}$
 7:         $g_{z_j} \longleftarrow \mathbb{E}_{z_i \sim p(z_j)}$
 8:         $\theta_{\mathcal{D}j} \longleftarrow \theta_{\mathcal{D}j} - \eta \nabla \theta_{\mathcal{D}j} \mathcal{L}(x_j)$
 9:         $\theta_{\mathcal{D}j} \longleftarrow \theta_{\mathcal{D}j} - \eta \nabla \theta_{\mathcal{D}j} \mathcal{L}(g_{z_j})$
10:         $\theta_{\mathcal{G}j} \longleftarrow \theta_{\mathcal{G}j} - \eta \nabla \theta_{\mathcal{G}j} \mathcal{L}(z_j)$
11:     end for
12:     $\theta_{\mathcal{G}i} \longleftarrow \theta_{\mathcal{G}j}$
13:     $z_i \longleftarrow \mathcal{N}_{\{mean=0,std=1,size=sizeof(\mathcal{X}_m)\}}$
14:     $G_{z_i} \longleftarrow \mathbb{E}_{z_i \sim p(z)}$
15:     U = $\mathcal{X}_m \cup G_{z_i}$
16:     for $k \in 10$ do
17:         split_pointer = k
18:         $t_r$ ← 80% of U
19:         $t_s$ ← 20% of U
20:         train Classifier on $t_r$
21:         test Classifier on $t_s$
22:         compute $e_{Rate}$
23:     end for
24:     Compute $e_{Rate_{avg}}$
25: end for
26: $z \longleftarrow \mathcal{N}_{\{mean=0,std=1,size=Normal-real\_APK\_APIsamples}$
27: $Gz_{\mathcal{I}_{argmax}(e_{Rate_{avg}})} \longleftarrow \mathbb{E}_{z \sim p(z)}$
28: $E_{Rate} \longleftarrow Classifier(Gz_{\mathcal{I}_{argmin}(E_{Rate})})$
29: Return $E_{Rate}$
```

Table 5.2: Classification Results

	Precision	**Recall**	**F1-measure**	**Accuracy**
SVM	0.898	0.841	0.868	0.876
Logistic regression	0.891	0.840	0.865	0.872
Perceptron	0.717	0.914	0.804	0.783
Decision Tree	0.924	0.870	0.896	0.902
Random forest	0.927	0.881	0.904	0.908
Xgboost	0.897	0.831	0.862	0.871

with CIA AT and CIA on classifiers trained with GAN AT. Finally, we perform evasion attacks on TrickDroid, a proposed adversarial training scheme on both CIA and GAEA based data and record the evasion rate in Figure 5.4. We use an API-based dataset which is composed of 5560 malicious and 5600 benign Android applications for the experiments. The experiments were performed on Dell G3 with 2.60GHz 6 core(s) processor, 16GB RAM and NVIDIA RTX 2060 GPU, running Windows 10.

In case of no adversarial attacks (NAT), we train the SVM, LR, PT, DT, RF and XGB classifiers on default hyper-parameters settings with 10-folds cross-validation with a distribution of 80% train set and 20% test set on each iteration. Table 5.2 presents the classification results obtained by the classifiers trained on API-based features. Amongst all the other classifiers, RF yields remarkable classification results with 90.8% accuracy. Figure 5.2 presents the results of CIA where the x-axis presents the number of features injected, and the y-axis represents the evasion rate. Consequently, linear classifiers SVM, LR and PT are affected the most with an evasion rate of 100%, which means all the adversarial samples in the test set were evaded. In comparison, DT was evaded the least with an evasion rate of 44.82. The evaluation of the CIA shows that linear classifiers are very fragile against the CIA.

The next attack we performed was the GAEA on NAT classifiers. As shown in Figure 5.3, similar to the CIA, in the case of GAEA, linear classifiers were affected the most with an evasion rate of more than 85% in all cases, whereas DT was least affected as compared to all the other classifiers with an evasion rate of 46.14%. As compared to CIA, GAEA have a slightly lower evasion rate

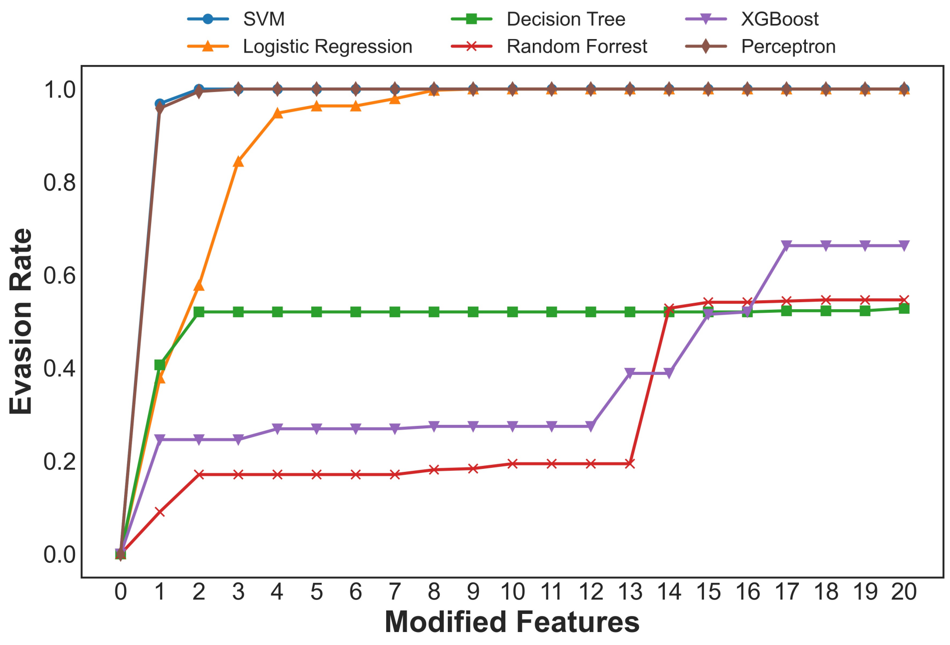


Figure 5.2: Code Injection Attack

with no classifier being evaded 100%. However, both of the attacks (CIA and GAEA) have proved to be significantly effective in evading pre-trained classifiers on the Android malware dataset.

As a countermeasure to mitigate the effects of evasion attacks, we retrain classifiers on adversarial data and then evaluate those against evasion attacks. Firstly, we retrain the classifiers on code injection attacks (CIA AT) and perform the evaluation. As shown in Algorithm 1, to perform CIA, we inject the top features of benign Android applications in the malicious apps and evaluate those against pre-trained classifiers. We do so by first injecting the first top discriminating feature of benign apps into the malicious apps in all of the test sets and evaluating it against the classifiers. Furthermore, in addition to the first top feature, we inject the second top discriminating feature of the benign app in a malicious test set and perform the evaluation. The same process is applied till the injection of the top 20 benign features in the malicious test set. As discussed earlier, the CIA proved to be very effective to evade multiple ML classifiers. To per-

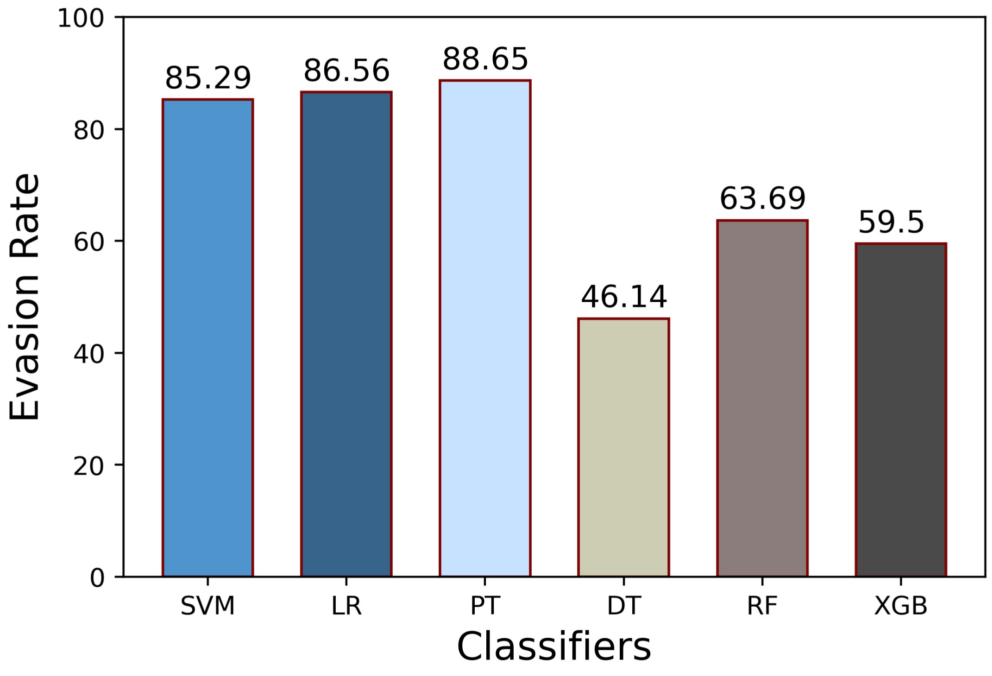


Figure 5.3: Results of GAEA

form retraining of classifiers on CIA, we generate an Oracle where for each malicious Android app, we added 20 new modified samples. The first sample has one top benign feature injected, the second sample has two benign top features injected and so on. Consequently, the size of the training set increased by 20 folds (i.e. 5560 to 111200). Although the size of the training set has dramatically increased, however, the CIA AT proved to be very effective. As a result of adversarial training of existing classifiers on CIA data (CIA AT in Figure 5.4), the most evaded classifier is XGB with only a 0.88% evasion rate.

In the next experiment, we perform GAEA Adversarial Training (GAEA AT) on the classifiers. We generate 5500 samples similar to the original malicious data using the method as mentioned in Section 5.2. We augmented the GAEA data with the original dataset. As shown in Figure 5.4, classifiers trained on GAN adversarial examples (GAEA AT) perform remarkably well against GAEA with a worst-case of 12.53% evasion rate achieved in the case of PT trained on GAEA. All the other classifiers retrained on GAEA have an evasion rate of less than 10%. As compared to CIA AT, it is worth mentioning here that GAN based adversarial sample attacks were minimised by just retraining the classifiers on 5500 adversarial samples; however, to avoid CIA, we retrained classifiers on an Oracle of 111200 new samples as mentioned previously.

Furthermore, we perform experiments by performing GAEA on CIA AT and CIA on GAEA AT to cross-validate the efficacy of the two adversarial training CIA AT and GAEA AT on the classifiers. As shown in Figure 5.4, in case of performing CIA on GAN based adversarial trained classifiers (CIA[GAEA AT]), all the linear classifiers (SVM, LR and PT) have been evaded more than 85% whereas DT, XGB and RF perform very well with a worst-case evasion rate of 17.85% in case of XGB. Consequently, by applying GAEA on classifiers trained on code injection attacks (GAEA[CIA AT]), surprisingly, the results were opposite to CIA[GAEA AT]. As shown in Figure 5.4, in the case of GAEA[CIA AT], all the linear classifiers performed remarkably well with a worst-case evasion rate of 11.15% in the case of PT. Whereas DT, RF and XGBoost were evaded more than 82% in all cases. As a final countermeasure, we train

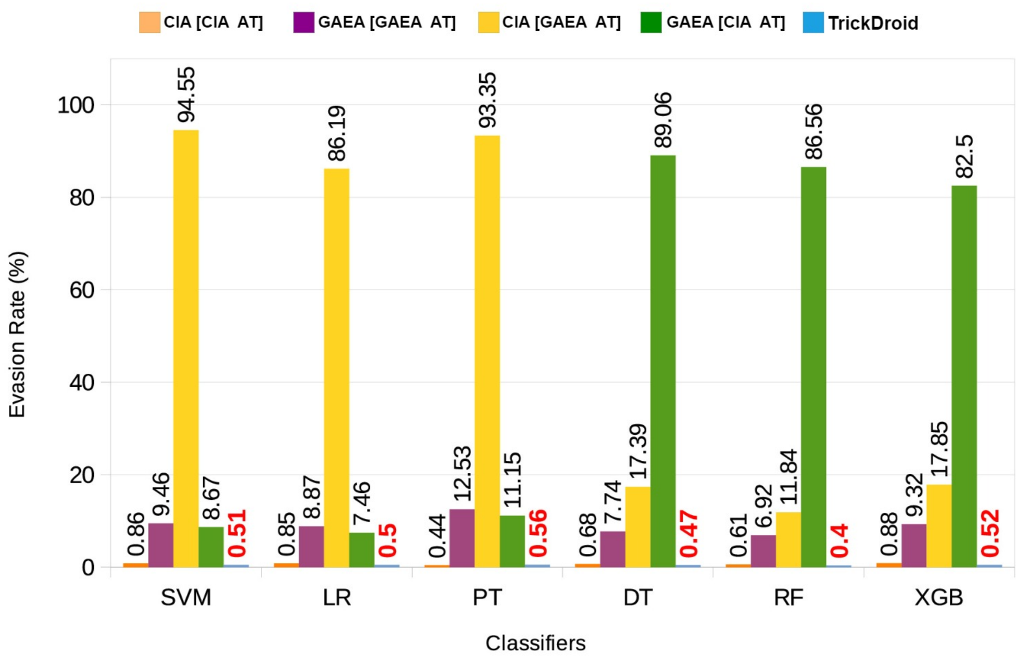


Figure 5.4: Evasion Rate

classifiers on both CIA AT and GAEA AT and call this adversarial training as ***TrickDroid***. As shown in Figure 5.4 (highlighted in the red colour text), TrickDroid remarkably works well against both CIA[GAEA AT] and GAEA[CIA AT] with an evasion rate of no more than 0.51 in the worst case.

5.4 Summary

The excessive use of Machine learning (ML) classifiers in Android malware detection demands a greater degree of inherent security due to the threats of adversarial evasion attacks. In this work, we highlight the fragility of classical ML classifiers against these types of attacks. After performing Oracle and GAN adversarial examples based attacks on different ML classifiers on a public Android dataset, we demonstrate an evasion rate of up to 100%. Our experiments reveal that the linear classifiers are less robust as compared to their ensemble counterparts both in Oracle and GAN based attacks. Furthermore, we present that despite adversarial training against one attack type, the classifiers are still vulnerable to other attacks. Hence, in order to further ruggedize the classifiers, we propose Trickdroid, a cumulative adversarial training technique and demonstrate its efficacy with up to 99.46% evasion detection.

Chapter 6

Conclusion and Future Work

6.1 Summary of Contributions

Nowadays, smartphones can accomplish nearly all functions traditionally associated with personal computers. Android OS dominates the smartphone OS industry with more than 70% market share. As a result of its widespread use, Android OS have become a prime target for smartphone malware developers. Therefore, serious initiatives have been taken by the industry and researchers to ensure the security of these smart devices. Security analysts have extensively researched and successfully demonstrated ML-based Android malware detection techniques to effectively mitigate the Android malware problem. Despite the impressive classification accuracy reported by ML-based techniques, they remain insufficient. The reason is that the design process of these techniques does not usually consider adversarial scenarios. Consequently, a carefully crafted perturbation in the malicious sample can result in misclassification.

In this thesis, the research begins by identifying the research gaps related to the design and development of accurate and adversarially aware ML-based Android malware detectors. Based on the research gaps, four research questions were identified. The RQ1 focused on the problem of repacked malware in benchmark Android malware repositories. The RQ2 focused on quantifying the impact of repacked malware on ML-based Android malware detectors. RQ3 stressed on investigating the fragility of high-performing ML-based Android malware classifiers against adversarial evasion attacks. Finally, the RQ4 was related to designing and developing accurate and adversarially robust ML-based Android malware classifiers.

In order to address the RQ1, Chapter 3 starts with quantifying the repacked malware in three benchmark Android malware datasets (Drebin, AMD and Androzoo). The analysis of these datasets revealed that 52.3% apps in Drebin, 29.8% apps in the AMD and 42.3% apps in the Androzoo dataset reuse existing package names. Further analysis of the apps sharing the same package names revealed that most of them share the same source code with minor modifications. A case study was then performed by employing fuzzy hashes to detect repacked Android malware in benchmark Android malware repositories. Prompted by the motivating results of the case study, ML-based techniques were further incorporated to build a more robust solution for detecting repacked Android malware. The RQ2 was then addressed by investigating the impact of repacked malware on ML-based Android malware detectors. In order to do so, we proposed AndroMalPack, an Android malware classifier trained on clones free datasets and optimized using NIAs. Although trained on repacked/clones free train set, AndroMalPack achieved up to 98.7% F1-score.

Furthermore, Chapter 4 focused on addressing RQ3 and RQ4. First, we proposed CureDroid, an Android malware classifier trained on hybrid features and optimized using a tree-based pipeline optimization. CureDroid achieved a remarkable malware detection accuracy of up to 99.2% on a dataset of 18,000 malicious and 18,000 benign Android apps. In order to address the RQ3, we formulated three types of evasion attacks (mimicry attacks, FRA and MFRA) to evade CureDroid. Consequently, it was demonstrated that although the CureDroid* classifier had high classification accuracy in a non-adversarial setting, it is vulnerable to adversarial evasion attacks (achieved up to 100% evasion rate). In order to build an adversarially aware classifier (to address RQ4), we proposed CureDroid*. CureDroid* is based on multiple classifiers trained on the distinct feature sets extracted from Android apps. The experimental results proved that the CureDroid* model mitigates the impact of adversarial attacks on ML-based malware classifiers. CureDroid* was able to detect up to 30 perturbations in feature vectors while retaining high malware classification accuracy.

Chapter 5 further addresses the RQ4 by exploring the effectiveness of adversarial training in order to enhance the adversarial robustness of ML-based Android malware classifiers. First, six different pre-trained ML-based classifiers were evaded by employing novel oracle and GAN-based adversarial evasion attacks. The experiments revealed that linear classifiers such as SVM, LR and PT are more vulnerable to adversarial attacks than their ensemble counterparts such as RF, DT and XGB. Furthermore, adversarial training was performed on ML-based classifiers in order to harden the security of ML-based Android malware classifiers against adversarial attacks. The experimental results revealed that despite training the classifiers on one type of attack, they were still vulnerable to the other type of attacks. Consequently, we proposed TrickDroid, a cumulative adversarial training technique and demonstrated its efficacy with up to 99.46% evasion detection.

6.2 Future Directions

There are twofold future directions that this work can inspire. The first type is related to designing automated tools, and the other is designing more adversarial robust Android malware classifiers. Following is a brief discussion about these two directions:

6.2.1 Automated Tools

Following are two types of automated tools that can be designed and implemented to foster research in mitigating adversarial attacks on Android malware classifiers.

6.2.1.1 Churn GAN Generated Synthetic Data

Generative adversarial networks (GANs) are employed to generate synthetic data which follows the actual distribution. In terms of generating Android malware data that mimic real applications, it is not always the case that the generated data preserves the malicious semantics. Therefore, there is a need for an automated tool that can help in churning the GAN-generated data. The tool can churn the data based on predefined rules such as checking if any discriminating features are removed from the vector or too many features are injected into the vector.

6.2.1.2 Modification of APK

In order to train Ml-based algorithms on malicious and benign Android apps, the apps are converted into the form of feature vectors. Generally, the feature vectors contain binary values where 1 represents the presence of a feature and 0 represents the absence of a particular feature. Furthermore, adversarial attacks are applied by perturbing the binary values in the feature vector instead of making a change in real applications. Therefore, an automated tool is needed to parse the changes in feature vectors and apply the modifications in the real APK. Once the app is modified, it can leveraged by the malware analysts by testing the modified apps on commercial antivirus tools on a large scale.

6.2.2 Adversarially Robust Classifiers

In this thesis, we have proposed techniques that can be employed to enhance the performance of Android malware classifiers and counter evasion attacks. Following are some future directions that can be further explored to ruggedise the adversarial robustness of Android malware detectors:

6.2.2.1 Scale CureDroid*

We proposed the CureDroid* model to mitigate adversarial evasion attacks on the Android malware classifiers. Although CureDroid* performs well against evasion attacks, the attacker can still elude the proposed system by evading all the classifiers in the ensemble model. The CureDroid* model can be further scaled by identifying more feature subsets that have the individual capability to classify malicious and benign Android applications. The novel feature subsets will add more classifiers to the CureDroid* model. Consequently, it will harden the security of the CureDroid* model as the attacker will need more effort to evade all the classifiers in the ensemble model.

6.2.2.2 Robust feature engineering

Adversarial attacks against machine learning-based classifiers are carried out by modifying the attributes of the original application. Generally, the features with high frequency in a given class are perturbed to elude the classification model. An interesting future direction would be to explore the effectiveness of removing easily manipulatable characteristics from feature vectors to make the evasion process more difficult for the adversary. Furthermore, it would be worth exploring the adversarial robustness of classifiers trained on random features from input data rather than following the uniform pattern at each iteration.

6.2.2.3 Image-based Adversarially Robust Classifiers

The features extracted from the Android application can be transformed into multi-dimensional feature vectors like those used for image classification tasks. An interesting future direction can be to explore the effectiveness of image-based classification algorithms using Android data. Numerous countermeasures are proposed in the literature to counter adversarial evasion attacks on image-based classifiers [198]. Consequently, these countermeasures can be employed on classifiers trained on images based on Android data, and adversarial effectiveness can be measured.

Glossary

AB	AdaBoost. 38, 50
APIs	Application Programming Interfaces. 2, 5, 12
APK	Android Application Package. 11, 24
apps	Applications. 1, 13
BA	Bat algorithm. 51, 53
CFGs	Control Flow Graphs. 19
CTPH	Context-Triggered Piece-wise Hashing. 40
DL	Deep learning. 2
DT	Decision trees. 38, 50
DVM	Dalvik Virtual Machine. 12, 13
FA	Firefly algorithm. 51, 53
FN	False Negative. 52
FP	False Positive. 52
FRA	Feature removal attacks. 63

GANs	Generative Adversarial Networks. 6
GB	Gradient boosting. 69
GWO	Grey wolf optimizer. 51, 53
HAL	Hardware abstraction layer. 12
iOS	IPhone Operating System. 1
KNN	K-nearest neighbours. 38, 50
LR	Linear regression. 38, 50
LSTM	Long Short-Term Memory. 19
MA	Mimicry attacks. 63
MFRA	Mimicry with feature removal attacks. 63
ML	Machine Learning. iv
MLP	Multi-layer perceptron. 30
NIAs	Nature inspired algorithms. 5, 51
NLP	Natural language processing (. 23
OS	Operating system. iv, 1, 2, 11
PT	Perceptron. 87, 96, 99
RF	Random forests. 38, 50
RL	Reinforcement learning. 30, 31

SVM	Support Vector Machine. 18, 22, 50
TN	True Negative. 52
TP	True Positive. 52
TPoT	Tree-based pipeline optimization technique. 5
XGB	Xgboost. 38, 50

References

[1] S. O'Dea. Daily time spent on mobile phones in the u.s. 2019-2023, 2021. URL https://www.statista.com/statistics/1045353/mobile-device-daily-usage-time-in-the-us/. Accessed on 2022-09-25.

[2] David Curry. Smartphones os global market share, 2021. URL https://www.businessofapps.com/data/android-statistics/. Accessed on 2022-09-25.

[3] Mansoor Iqbal. Apps download and usage statistics, 2021. URL https://www.businessofapps.com/data/app-statistics/. Accessed on 2022-09-25.

[4] Pavol Zavarsky, Dale Lindskog, et al. Experimental analysis of ransomware on windows and android platforms: Evolution and characterization. *Procedia Computer Science*, 94: 465–472, 2016.

[5] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.

[6] Stephen McLaughlin. On dynamic malware payloads aimed at programmable logic controllers. In *6th USENIX Workshop on Hot Topics in Security (HotSec 11)*, 2011.

[7] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *IEEE Transactions on Software Engineering*, 2019.

[8] Yanfang Ye, Tao Li, Donald Adjeroh, and S Sitharama Iyengar. A survey on malware detection using data mining techniques. *ACM Computing Surveys (CSUR)*, 50(3):1–40, 2017.

[9] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4):1–41, 2017.

[10] Hemant Rathore, Sanjay K Sahay, Piyush Nikam, and Mohit Sewak. Robust android malware detection system against adversarial attacks using q-learning. *Information Systems Frontiers*, 23(4):867–882, 2021.

[11] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security*, 9(1):99–108, 2013.

[12] Junyang Qiu, Jun Zhang, Wei Luo, Lei Pan, Surya Nepal, and Yang Xiang. A survey of android malware detection with deep neural models. *ACM Computing Surveys (CSUR)*, 53(6):1–36, 2020.

[13] Vasileios Kouliaridis and Georgios Kambourakis. A comprehensive survey on machine learning techniques for android malware detection. *Information*, 12(5):185, 2021.

[14] Asma Razgallah, Raphaël Khoury, Sylvain Hallé, and Kobra Khanmohammadi. A survey of malware detection in android apps: Recommendations and perspectives for future research. *Computer Science Review*, 39:100358, 2021.

[15] Stuart Millar, Niall McLaughlin, Jesus Martinez del Rincon, and Paul Miller. Multi-view deep learning for zero-day android malware detection. *Journal of Information Security and Applications*, 58:102718, 2021.

[16] AV-Test. Android malware statistics, 2021. URL https://www.av-test.org/en/statistics/malware/. Accessed on 2022-09-25.

[17] Abdullah Al-Dujaili, Alex Huang, Erik Hemberg, and Una-May O'Reilly. Adversarial deep learning for robust detection of binary encoded malware. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 76–82. IEEE, 2018.

[18] Shifu Hou, Yanfang Ye, Yangqiu Song, and Melih Abdulhayoglu. Make evasion harder: An intelligent android malware detection system. In *IJCAI*, pages 5279–5283, 2018.

[19] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 387–402. Springer, 2013.

[20] Fabrizio Cara, Michele Scalas, Giorgio Giacinto, and Davide Maiorca. On the feasibility of adversarial sample creation using the android system api. *Information*, 11(9):433, 2020.

[21] Pavel Laskov and Richard Lippmann. Machine learning in adversarial environments, 2010.

[22] Yanjie Zhao, Li Li, Haoyu Wang, Haipeng Cai, Tegawendé F Bissyandé, Jacques Klein, and John Grundy. On the impact of sample duplication in machine-learning-based android malware detection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3):1–38, 2021.

[23] Paul Irolla and Alexandre Dey. The duplication issue within the drebin dataset. *Journal of Computer Virology and Hacking Techniques*, 14(3):245–249, 2018.

[24] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[25] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground

truth analysis of current android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 252–276. Springer, 2017.

[26] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE, 2016.

[27] Nikolaos Pitropakis, Emmanouil Panaousis, Thanassis Giannetsos, Eleftherios Anastasiadis, and George Loukas. A taxonomy and survey of attacks against machine learning. *Computer Science Review*, 34:100199, 2019.

[28] Luca Demetrio, Scott E Coull, Battista Biggio, Giovanni Lagorio, Alessandro Armando, and Fabio Roli. Adversarial exemples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection. *ACM Transactions on Privacy and Security (TOPS)*, 24(4):1–31, 2021.

[29] Samuel G Finlayson, Hyung Won Chung, Isaac S Kohane, and Andrew L Beam. Adversarial attacks against medical deep learning systems. *arXiv preprint arXiv:1804.05296*, 2018.

[30] Xingjun Ma, Yuhao Niu, Lin Gu, Yisen Wang, Yitian Zhao, James Bailey, and Feng Lu. Understanding adversarial attacks on deep learning based medical image analysis systems. *Pattern Recognition*, 110:107332, 2021.

[31] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*, 2017.

[32] Davide Maiorca, Battista Biggio, and Giorgio Giacinto. Towards adversarial malware detection: Lessons learned from pdf-based attacks. *ACM Computing Surveys (CSUR)*, 52 (4):1–36, 2019.

[33] Yuanzhang Li, Yaxiao Wang, Ye Wang, Lishan Ke, and Yu-an Tan. A feature-vector generative adversarial network for evading pdf malware classifiers. *Information Sciences*, 523:38–48, 2020.

[34] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. Andrubis–1,000,000 apps later: A view on current android malware behaviors. In *2014 third international workshop on building analysis datasets and gathering experience returns for security (BADGERS)*, pages 3–17. IEEE, 2014.

[35] Suleiman Y Yerima, Sakir Sezer, and Igor Muttik. High accuracy android malware detection using ensemble learning. *IET Information Security*, 9(6):313–320, 2015.

[36] Abhishek Kumar Singh, CD Jaidhar, and MA Ajay Kumara. Experimental analysis of android malware detection based on combinations of permissions and api-calls. *Journal of Computer Virology and Hacking Techniques*, 15(3):209–218, 2019.

[37] Akshay Mathur, Laxmi Mounika Podila, Keyur Kulkarni, Quamar Niyaz, and Ahmad Y Javaid. Naticusdroid: A malware detection framework for android using native and custom permissions. *Journal of Information Security and Applications*, 58:102696, 2021.

[38] Jin Li, Lichao Sun, Qiben Yan, Zhiqiang Li, Witawas Srisa-An, and Heng Ye. Significant permission identification for machine-learning-based android malware detection. *IEEE Transactions on Industrial Informatics*, 14(7):3216–3225, 2018.

[39] Luiz C Navarro, Alexandre KW Navarro, André Grégio, Anderson Rocha, and Ricardo Dahab. Leveraging ontologies and machine-learning techniques for malware analysis into android permissions ecosystems. *Computers & Security*, 78:429–453, 2018.

[40] Junfeng Yu, Qingfeng Huang, and CheeHoo Yian. Droidscreening: a practical framework for real-world android malware analysis. *Security and Communication Networks*, 9(11): 1435–1449, 2016.

[41] Moutaz Alazab, Mamoun Alazab, Andrii Shalaginov, Abdelwadood Mesleh, and Albara Awajan. Intelligent mobile malware detection using permission requests and api calls. *Future Generation Computer Systems*, 107:509–521, 2020.

[42] Satheesh Kumar Sasidharan and Ciza Thomas. Prodroid—an android malware detection framework based on profile hidden markov model. *Pervasive and Mobile Computing*, 72: 101336, 2021.

[43] Hui-Juan Zhu, Zhu-Hong You, Ze-Xuan Zhu, Wei-Lei Shi, Xing Chen, and Li Cheng. Droiddet: effective and robust detection of android malware using static analysis along with rotation forest model. *Neurocomputing*, 272:638–646, 2018.

[44] Ahmad Salah, Eman Shalabi, and Walid Khedr. A lightweight android malware classifier using novel feature selection methods. *Symmetry*, 12(5):858, 2020.

[45] Fauzia Idrees, Muttukrishnan Rajarajan, Mauro Conti, Thomas M Chen, and Yogachandran Rahulamathavan. Pindroid: A novel android malware detection system using ensemble learning methods. *Computers & Security*, 68:36–46, 2017.

[46] Fauzia Idrees and Muttukrishnan Rajarajan. Investigating the android intents and permissions for malware detection. In *2014 IEEE 10th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 354–358. IEEE, 2014.

[47] Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, Guillermo Suarez-Tangil, and Steven Furnell. Androdialysis: Analysis of android intent effectiveness in malware detection. *computers & security*, 65:121–134, 2017.

[48] Quentin Jerome, Kevin Allix, Radu State, and Thomas Engel. Using opcode-sequences to detect malicious android applications. In *2014 IEEE international conference on communications (ICC)*, pages 914–919. IEEE, 2014.

[49] Gerardo Canfora, Andrea De Lorenzo, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Effectiveness of opcode ngrams for detection of multi family android malware. In *2015 10th International Conference on Availability, Reliability and Security*, pages 333–340. IEEE, 2015.

[50] Jianguo Jiang, Song Li, Min Yu, Gang Li, Chao Liu, Kai Chen, Hui Liu, and Weiqing Huang. Android malware family classification based on sensitive opcode sequence. In *2019 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–7. IEEE, 2019.

[51] Abdulbasit Darem, Jemal Abawajy, Aaisha Makkar, Asma Alhashmi, and Sultan Alanazi. Visualization and deep-learning-based malware variant detection using opcode-level features. *Future Generation Computer Systems*, 125:314–323, 2021.

[52] Abdurrahman Pektaş and Tankut Acarman. Learning to detect android malware via opcode sequences. *Neurocomputing*, 396:599–608, 2020.

[53] Junwei Tang, Ruixuan Li, Yu Jiang, Xiwu Gu, and Yuhua Li. Android malware obfuscation variants detection method based on multi-granularity opcode features. *Future Generation Computer Systems*, 129:141–151, 2022.

[54] Zahoor-Ur Rehman, Sidra Nasim Khan, Khan Muhammad, Jong Weon Lee, Zhihan Lv, Sung Wook Baik, Peer Azmat Shah, Khalid Awan, and Irfan Mehmood. Machine learning-assisted signature and heuristic-based detection of malwares in android devices. *Computers & Electrical Engineering*, 69:828–841, 2018.

[55] Wei Wang, Zhenzhen Gao, Meichen Zhao, Yidong Li, Jiqiang Liu, and Xiangliang Zhang. Droidensemble: Detecting android malicious applications with ensemble of string and structural static features. *IEEE Access*, 6:31798–31807, 2018.

[56] Justin Del Vecchio, Steven Y Ko, and Lukasz Ziarek. Representing string computations

as graphs for classifying malware. In *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, pages 120–131, 2020.

[57] Caijun Sun, Hua Zhang, Sujuan Qin, Nengqiang He, Jiawei Qin, and Hongwei Pan. Dexx: a double layer unpacking framework for android. *IEEE Access*, 6:61267–61276, 2018.

[58] Parvez Faruki, Vijay Ganmoor, Vijay Laxmi, Manoj Singh Gaur, and Ammar Bharmal. Androsimilar: robust statistical feature signature for android malware detection. In *Proceedings of the 6th International Conference on Security of Information and Networks*, pages 152–159, 2013.

[59] Wenjun Hu, Jing Tao, Xiaobo Ma, Wenyu Zhou, Shuang Zhao, and Ting Han. Migdroid: Detecting app-repackaging android malware via method invocation graph. In *2014 23rd International Conference on Computer Communication and Networks (ICCCN)*, pages 1–7. IEEE, 2014.

[60] Min Zheng, Mingshen Sun, and John CS Lui. Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 163–171. IEEE, 2013.

[61] Xin Wang, Dafang Zhang, Xin Su, and Wenjia Li. Mlifdect: Android malware detection based on parallel machine learning and information fusion. *Security and Communication Networks*, 2017, 2017.

[62] Fadi Mohsen, Halil Bisgin, Zachary Scott, and Kyle Strait. Detecting android malwares by mining statically registered broadcast receivers. In *2017 IEEE 3rd International Conference on Collaboration and Internet Computing (CIC)*, pages 67–76. IEEE, 2017.

[63] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIG-*

SOFT international symposium on foundations of software engineering, pages 576–587, 2014.

[64] Roopak Surendran, Tony Thomas, and Sabu Emmanuel. A tan based hybrid model for android malware detection. *Journal of Information Security and Applications*, 54:102483, 2020.

[65] Jinpei Yan, Yong Qi, and Qifan Rao. Lstm-based hierarchical denoising network for android malware detection. *Security and Communication Networks*, 2018, 2018.

[66] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Zhenzhou Tian, Qinghua Zheng, and Ting Liu. Android malware familial classification and representative sample selection via frequent subgraph analysis. *IEEE Transactions on Information Forensics and Security*, 13(8): 1890–1905, 2018.

[67] Tatiana Frenklach, Dvir Cohen, Asaf Shabtai, and Rami Puzis. Android malware detection via an app similarity graph. *Computers & Security*, 109:102386, 2021.

[68] Yang Yang, Xuehui Du, Zhi Yang, and Xing Liu. Android malware detection based on structural features of the function call graph. *Electronics*, 10(2):186, 2021.

[69] Suparerk Ngamwitroj and Benchaphon Limthanmaphon. Adaptive android malware signature detection. In *Proceedings of the 2018 International Conference on Communication Engineering and Technology*, pages 22–25, 2018.

[70] Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *2010 5th International Conference on Malicious and Unwanted Software*, pages 55–62. IEEE, 2010.

[71] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Pro-*

ceedings of the 28th Annual ACM Symposium on Applied Computing, pages 1808–1815, 2013.

[72] Arvind Mahindru and Paramvir Singh. Dynamic permissions based android malware detection using machine learning techniques. In *Proceedings of the 10th innovations in software engineering conference*, pages 202–210, 2017.

[73] Altyeb Altaher and Omar Mohammed Barukab. Intelligent hybrid approach for android malware detection based on permissions and api calls. *International Journal of Advanced Computer Science and Applications*, 8(6):60–67, 2017.

[74] Jyoti Gajrani, Umang Agarwal, Vijay Laxmi, Bruhadeshwar Bezawada, Manoj Singh Gaur, Meenakshi Tripathi, and Akka Zemmari. Espydroid+: Precise reflection analysis of android apps. *Computers & Security*, 90:101688, 2020.

[75] Yubin Yang, Zongtao Wei, Yong Xu, Haiwu He, and Wei Wang. Droidward: an effective dynamic analysis method for vetting android applications. *Cluster Computing*, 21(1): 265–275, 2018.

[76] Jyoti Malik and Rishabh Kaushal. Credroid: Android malware detection by network traffic analysis. In *Proceedings of the 1st acm workshop on privacy-aware mobile computing*, pages 28–36, 2016.

[77] Shanshan Wang, Qiben Yan, Zhenxiang Chen, Bo Yang, Chuan Zhao, and Mauro Conti. Detecting android malware leveraging text semantics of network flows. *IEEE Transactions on Information Forensics and Security*, 13(5):1096–1109, 2017.

[78] Shanshan Wang, Zhenxiang Chen, Qiben Yan, Ke Ji, Lizhi Peng, Bo Yang, and Mauro Conti. Deep and broad url feature mining for android malware detection. *Information Sciences*, 513:600–613, 2020.

[79] Mehedee Zaman, Tazrian Siddiqui, Mohammad Rakib Amin, and Md Shohrab Hossain.

Malware detection in android by network traffic analysis. In *2015 international conference on networking systems and security (NSysS)*, pages 1–5. IEEE, 2015.

[80] José Gaviria de la Puerta, Iker Pastor-López, Igone Porto, Borja Sanz, and Pablo García Bringas. Detecting malicious android applications based on the network packets generated. *Neurocomputing*, 456:629–636, 2021.

[81] Gerardo Canfora, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Detecting android malware using sequences of system calls. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, pages 13–20, 2015.

[82] Jorge Maestre Vidal, Marco Antonio Sotelo Monge, and Luis Javier García Villalba. A novel pattern recognition system for detecting android malware by analyzing suspicious boot sequences. *Knowledge-Based Systems*, 150:198–217, 2018.

[83] P Vinod, Akka Zemmari, and Mauro Conti. A machine learning based approach to detect malicious android apps using discriminant system calls. *Future Generation Computer Systems*, 94:333–350, 2019.

[84] Marko Dimjašević, Simone Atzeni, Ivo Ugrina, and Zvonimir Rakamaric. Evaluation of android malware detection based on system calls. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, pages 1–8, 2016.

[85] Xi Xiao, Shaofeng Zhang, Francesco Mercaldo, Guangwu Hu, and Arun Kumar Sangaiah. Android malware detection based on system call sequences and lstm. *Multimedia Tools and Applications*, 78(4):3979–3999, 2019.

[86] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid android: versatile protection for smartphones. In *Proceedings of the 26th annual computer security applications conference*, pages 347–356, 2010.

[87] Hongyu Yang and Ruiwen Tang. Power consumption based android malware detection. *Electrical and Computer Engineerin*, 2016.

[88] Zigrid Shehu, Claudio Ciccotelli, Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Towards the usage of invariant-based app behavioral fingerprinting for the detection of obfuscated versions of known malware. In *2016 10th International Conference on Next Generation Mobile Applications, Security and Technologies (NGMAST)*, pages 121–126. IEEE, 2016.

[89] Jelena Milosevic, Miroslaw Malek, and Alberto Ferrante. A friend or a foe? detecting malware using memory and cpu features. In *SECRYPT*, pages 73–84, 2016.

[90] Mohammed S Alam and Son T Vuong. Random forest classification for detecting android malware. In *2013 IEEE international conference on green computing and communications and IEEE Internet of Things and IEEE cyber, physical and social computing*, pages 663–669. IEEE, 2013.

[91] Fehmi Jaafar, Gagandeep Singh, and Pavol Zavarsky. An analysis of android malware behavior. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 505–512. IEEE, 2018.

[92] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*, pages 95–109. IEEE, 2012.

[93] Zimin Lin, Rui Wang, Xiaoqi Jia, Shengzhi Zhang, and Chuankun Wu. Classifying android malware with dynamic behavior dependency graphs. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 378–385. IEEE, 2016.

[94] Abdurrahman Pektaş and Tankut Acarman. Deep learning for effective android malware detection using api call graph embeddings. *Soft Computing*, 24(2):1027–1043, 2020.

[95] Roopak Surendran, Tony Thomas, and Sabu Emmanuel. Gsdroid: Graph signal based compact feature representation for android malware detection. *Expert Systems with Applications*, 159:113581, 2020.

[96] Vasileios Kouliaridis, Georgios Kambourakis, Dimitris Geneiatakis, and Nektaria Potha. Two anatomists are better than one—dual-level android malware detection. *Symmetry*, 12 (7):1128, 2020.

[97] Saba Arshad, Munam A Shah, Abdul Wahid, Amjad Mehmood, Houbing Song, and Hongnian Yu. Samadroid: a novel 3-level hybrid malware detection model for android operating system. *IEEE Access*, 6:4321–4339, 2018.

[98] Ly Hoang Tuan, Nguyen Tan Cam, and Van-Hau Pham. Enhancing the accuracy of static analysis for detecting sensitive data leakage in android by using dynamic analysis. *Cluster Computing*, 22(1):1079–1085, 2019.

[99] Afifa Maryam, Usman Ahmed, Muhammad Aleem, Jerry Chun-Wei Lin, Muhammad Arshad Islam, and Muhammad Azhar Iqbal. chybridroid: A machine learning-based hybrid technique for securing the edge computing. *Security and Communication Networks*, 2020, 2020.

[100] Randal S Olson, Nathan Bartley, Ryan J Urbanowicz, and Jason H Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the genetic and evolutionary computation conference 2016*, pages 485–492, 2016.

[101] Hemant Rathore, Sanjay K Sahay, Piyush Nikam, and Mohit Sewak. Robust android malware detection system against adversarial attacks using q-learning. *Information Systems Frontiers*, pages 1–16, 2020.

[102] Alessio Merlo, Antonio Ruggia, Luigi Sciolla, and Luca Verderame. You shall not repackage! demystifying anti-repackaging on android. *Computers & Security*, page 102181, 2021.

[103] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In *European Symposium on Research in Computer Security*, pages 37–54. Springer, 2012.

[104] Sibei Jiao, Yao Cheng, Lingyun Ying, Purui Su, and Dengguo Feng. A rapid and scalable method for android application repackaging detection. In *International Conference on Information Security Practice and Experience*, pages 349–364. Springer, 2015.

[105] Mingshen Sun, Mengmeng Li, and John CS Lui. Droideagle: Seamless detection of visually similar android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 1–12, 2015.

[106] Shahid Alam and Ibrahim Sogukpinar. Droidclone: Attack of the android malware clones-a step towards stopping them. *Computer Science and Information Systems*, 18(1):35–35, 2020.

[107] Shirish Singh, Kushagra Chaturvedy, and Bharavi Mishra. Multi-view learning for repackaged malware detection. In *The 16th International Conference on Availability, Reliability and Security*, pages 1–9, 2021.

[108] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. Codematch: obfuscation won't conceal your repackaged app. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 638–648, 2017.

[109] Yuta Ishii, Takuya Watanabe, Mitsuaki Akiyama, and Tatsuya Mori. Appraiser: A large scale analysis of android clone apps. *IEICE TRANSACTIONS on Information and Systems*, 100(8):1703–1713, 2017.

[110] Gaofeng He, Lu Zhang, Bingfeng Xu, and Haiting Zhu. Detecting repackaged android malware based on mobile edge computing. In *2018 Sixth International Conference on Advanced Cloud and Big Data (CBD)*, pages 360–365. IEEE, 2018.

[111] Shahid Alam and Ibrahim Sogukpinar. Droidclone: Attack of the android malware clones-a step towards stopping them. *Computer Science and Information Systems*, 18(1):67–91, 2021.

[112] Roopak Surendran. On impact of semantically similar apps in android malware datasets. *arXiv preprint arXiv:2112.02606*, 2021.

[113] Nedim Srndic and Pavel Laskov. Practical evasion of a learning-based classifier: A case study. In *2014 IEEE symposium on security and privacy*, pages 197–211. IEEE, 2014.

[114] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331, 2018.

[115] Lingwei Chen, Shifu Hou, and Yanfang Ye. Securedroid: Enhancing security of machine learning-based detection against adversarial android malware attacks. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 362–372, 2017.

[116] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In *European symposium on research in computer security*, pages 62–79. Springer, 2017.

[117] Alejandro Calleja, Alejandro Martín, Héctor D Menéndez, Juan Tapiador, and David Clark. Picking on the family: Disrupting android malware triage by forcing misclassification. *Expert Systems with Applications*, 95:113–126, 2018.

[118] Joshua Garcia, Mahmoud Hammad, and Sam Malek. Lightweight, obfuscation-resilient detection and family identification of android malware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(3):1–29, 2018.

[119] Guozhu Meng, Yinxing Xue, Chandramohan Mahinthan, Annamalai Narayanan, Yang Liu, Jie Zhang, and Tieming Chen. Mystique: Evolving android malware for auditing anti-malware tools. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*, pages 365–376, 2016.

[120] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. Evaluation of android anti-malware techniques against dalvik

bytecode obfuscation. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 414–421. IEEE, 2014.

[121] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435*, 2016.

[122] Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, and Fabio Roli. Yes, machine learning can be more secure! a case study on android malware detection. *IEEE Transactions on Dependable and Secure Computing*, 16(4):711–724, 2017.

[123] Hemant Rathore, Piyush Nikam, Sanjay K Sahay, and Mohit Sewak. Identification of adversarial android intents using reinforcement learning. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.

[124] Xuetao Zhang, Jinshuang Wang, Meng Sun, and Yao Feng. Andropgan: An opcode gan for android malware obfuscations. In *International Conference on Machine Learning for Cyber Security*, pages 12–25. Springer, 2020.

[125] Heng Li, ShiYao Zhou, Wei Yuan, Jiahuan Li, and Henry Leung. Adversarial-example attacks toward android malware detection system. *IEEE Systems Journal*, 14(1):653–656, 2019.

[126] Salman Jan, Toqeer Ali, Ali Alzahrani, and Shahrulniza Musa. Deep convolutional generative adversarial networks for intent-based dynamic behavior capture. *International Journal of Engineering & Technology*, 7(4.29):101–103, 2018.

[127] Deqiang Li, Qianmu Li, Yanfang Ye, and Shouhuai Xu. Arms race in adversarial malware detection: A survey. *ACM Computing Surveys (CSUR)*, 55(1):1–35, 2021.

[128] Rahim Taheri, Reza Javidan, Mohammad Shojafar, P Vinod, and Mauro Conti. Can

machine learning model with static features be fooled: an adversarial machine learning approach. *Cluster computing*, 23(4):3233–3253, 2020.

[129] Xiangjun Li, Ke Kong, Su Xu, Pengtao Qin, and Daojing He. Feature selection-based android malware adversarial sample generation and detection method. *IET Information Security*, 15(6):401–416, 2021.

[130] Gerardo Canfora, Andrea Di Sorbo, Francesco Mercaldo, and Corrado Aaron Visaggio. Obfuscation techniques against signature-based detection: a case study. In *2015 Mobile systems technologies workshop (MST)*, pages 21–26. IEEE, 2015.

[131] Charles Smutz and Angelos Stavrou. When a tree falls: Using diversity in ensemble classifiers to identify evasion in malware detectors. In *NDSS*, 2016.

[132] Zainab Abaid, Mohamed Ali Kaafar, and Sanjay Jha. Quantifying the impact of adversarial evasion attacks on machine learning based android malware classifiers. In *2017 IEEE 16th international symposium on network computing and applications (NCA)*, pages 1–10. IEEE, 2017.

[133] Mila Dalla Preda and Federico Maggi. Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. *Journal of Computer Virology and Hacking Techniques*, 13(3):209–232, 2017.

[134] Lingwei Chen, Shifu Hou, Yanfang Ye, and Lifei Chen. An adversarial machine learning model against android malware evasion attacks. In *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint Conference on Web and Big Data*, pages 43–55. Springer, 2017.

[135] Lingwei Chen, Shifu Hou, Yanfang Ye, and Shouhuai Xu. Droideye: Fortifying security of learning-based classifier against adversarial android malware attacks. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 782–789. IEEE, 2018.

[136] Hongyi Chen, Jinshu Su, Linbo Qiao, and Qin Xin. Malware collusion attack against svm: Issues and countermeasures. *Applied Sciences*, 8(10):1718, 2018.

[137] Melissa Chua and Vivek Balachandran. Effectiveness of android obfuscation on evading anti-malware. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 143–145, 2018.

[138] Mauro Conti Vinod P., Akka Zemmari. A machine learning based approach to detect malicious android apps using discriminant system calls. *Future Generation Computer Systems*, 2018.

[139] Marco Melis, Davide Maiorca, Battista Biggio, Giorgio Giacinto, and Fabio Roli. Explaining black-box android malware detection. In *2018 26th European Signal Processing Conference (EUSIPCO)*, pages 524–528. IEEE, 2018.

[140] Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. Android hiv: A study of repackaging malware for evading machine-learning detection. *IEEE Transactions on Information Forensics and Security*, 15:987–1001, 2019.

[141] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1332–1349. IEEE, 2020.

[142] Michael Cao, Sahar Badihi, Khaled Ahmed, Peiyu Xiong, and Julia Rubin. On benign features in malware detection. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1234–1238. IEEE, 2020.

[143] Hemant Rathore, Sanjay K Sahay, and Mohit Sewak. Are android malware detection models adversarially robust? poster abstract. In *Proceedings of the 20th International Conference on Information Processing in Sensor Networks (co-located with CPS-IoT Week 2021)*, pages 408–409, 2021.

[144] Hamid Bostani and Veelasha Moonsamy. Evadedroid: A practical evasion attack on machine learning for black-box android malware detection. *arXiv preprint arXiv:2110.03301*, 2021.

[145] Harel Berger, Chen Hajaj, Enrico Mariconti, and Amit Dvir. Crystal ball: From innovative attacks to attack effectiveness classifier. *IEEE Access*, 10:1317–1333, 2021.

[146] Asim Darwaish, Farid Naït-Abdesselam, Chafiq Titouna, and Sumera Sattar. Robustness of image-based android malware detection under adversarial attacks. In *ICC 2021-IEEE International Conference on Communications*, pages 1–6. IEEE, 2021.

[147] G Renjith, P Vinod, and S Aji. Evading machine-learning-based android malware detector for iot devices. *IEEE Systems Journal*, 2022.

[148] Hemant Rathore, Adithya Samavedhi, Sanjay K Sahay, and Mohit Sewak. Are malware detection models adversarial robust against evasion attack? In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–2. IEEE, 2022.

[149] Hemant Rathore, Sujay C Sharma, Sanjay K Sahay, and Mohit Sewak. Are malware detection classifiers adversarially vulnerable to actor-critic based evasion attacks? *EAI Endorsed Transactions on Scalable Information Systems*, pages e74–e74, 2022.

[150] Vasileios Syrris and Dimitris Geneiatakis. On machine learning effectiveness for malware detection in android os using static analysis data. *Journal of Information Security and Applications*, 59:102794, 2021.

[151] Lukas Schott, Jonas Rauber, Matthias Bethge, and Wieland Brendel. Towards the first adversarially robust neural network model on mnist. *arXiv preprint arXiv:1805.09190*, 2018.

[152] Florian Tramer and Dan Boneh. Adversarial training and robustness for multiple perturbations. *Advances in Neural Information Processing Systems*, 32, 2019.

[153] Raj Samani. Mcafee mobile threat report, 2020. URL https://www.mcafee.com/content/dam/consumer/en-us/docs/2020-Mobile-Threat-Report.pdf. Accessed on: 2022-09-25.

[154] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 431–444, 2013.

[155] Yauhen Leanidavich Arnatovich, Lipo Wang, Ngoc Minh Ngo, and Charlie Soh. A comparison of android reverse engineering tools via program behaviors validation based on intermediate languages transformation. *IEEE Access*, 6:12382–12394, 2018.

[156] Jie Zhang, Cong Tian, and Zhenhua Duan. An efficient approach for taint analysis of android applications. *Computers & Security*, 104:102161, 2021.

[157] Donald Eastlake and Paul Jones. Us secure hash algorithm 1 (sha1), 2001.

[158] Ronald Rivest and S Dusse. The md5 message-digest algorithm, 1992.

[159] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation*, 3:91–97, 2006.

[160] Anitta Patience Namanya, Irfan U Awan, Jules Pagna Disso, and Muhammad Younas. Similarity hash based scoring of portable executable files for efficient malware detection in iot. *Future Generation Computer Systems*, 110:824–832, 2020.

[161] Frank Breitinger, Christian Winter, York Yannikos, Tobias Fink, and Michael Seefried. Using approximate matching to reduce the volume of digital data. In *IFIP International Conference on Digital Forensics*, pages 149–163. Springer, 2014.

[162] Hasnat Ali, Komal Batool, Muhammad Yousaf, Muhammad Islam Satti, Salman Naseer, Saleem Zahid, Akber Abid Gardezi, Muhammad Shafiq, and Jin-Ghoo Choi. Security

hardened and privacy preserved android malware detection using fuzzy hash of reverse engineered source code. *Security & Communication Networks*, 2022.

[163] Vassil Roussev. An evaluation of forensic similarity hashes. *digital investigation*, 8: S34–S41, 2011.

[164] Haipeng Cai, Na Meng, Barbara Ryder, and Daphne Yao. Droidcat: Effective android malware detection and categorization via app-level profiling. *IEEE Transactions on Information Forensics and Security*, 14(6):1455–1470, 2018.

[165] Alimardani Hamidreza and Nazeh Mohammed. Permission-based analysis of android applications using categorization and deep learning scheme. In *MATEC Web of Conferences*, volume 255, page 05005. EDP Sciences, 2019.

[166] Anthony Desnos and G Gueguen. Androguard-reverse engineering, malware and goodware analysis of android applications. *URL code. google. com/p/androguard*, 153, 2013.

[167] Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, Guillermo Suarez-Tangil, and Steven Furnell. Androdialysis: Analysis of android intent effectiveness in malware detection. *computers & security*, 65:121–134, 2017.

[168] Kartik Khariwal, Jatin Singh, and Anshul Arora. Ipdroid: Android malware detection using intents and permissions. In *2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*, pages 197–202. IEEE, 2020.

[169] Deqing Zou, Yueming Wu, Siru Yang, Anki Chauhan, Wei Yang, Jiangying Zhong, Shihan Dou, and Hai Jin. Intdroid: Android malware detection based on api intimacy analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3):1–32, 2021.

[170] Shahid Alam, Soltan Abed Alharbi, and Serdar Yildirim. Mining nested flow of dominant apis for detecting android malware. *Computer Networks*, 167(00):107026, 2020.

[171] Xin-She Yang. A new metaheuristic bat-inspired algorithm. In *Nature inspired cooperative strategies for optimization (NICSO 2010)*, pages 65–74. Springer, 2010.

[172] Seyedali Mirjalili, Seyed Mohammad Mirjalili, and Andrew Lewis. Grey wolf optimizer. *Advances in engineering software*, 69:46–61, 2014.

[173] Xin-She Yang. Firefly algorithms for multimodal optimization. In *International symposium on stochastic algorithms*, pages 169–178. Springer, 2009.

[174] Andrea De Lorenzo, Fabio Martinelli, Eric Medvet, Francesco Mercaldo, and Antonella Santone. Visualizing the outcome of dynamic analysis of android malware with vizmal. *Journal of Information Security and Applications*, 50:102423, 2020.

[175] Krishna Sugunan, T Gireesh Kumar, and KA Dhanya. Static and dynamic analysis for android malware detection. In *Advances in Big Data and Cloud Computing*, pages 147–155. Springer, 2018.

[176] Lucky Onwuzurike, Mario Almeida, Enrico Mariconti, Jeremy Blackburn, Gianluca Stringhini, and Emiliano De Cristofaro. A family of droids-android malware detection via behavioral modeling: Static vs dynamic analysis. In *2018 16th Annual Conference on Privacy, Security and Trust (PST)*, pages 1–10. IEEE, 2018.

[177] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 143–153, 2019.

[178] Haoyu Wang, Junjun Si, Hao Li, and Yao Guo. Rmvdroid: towards a reliable android malware dataset with app metadata. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 404–408. IEEE, 2019.

[179] Nikola Milosevic, Ali Dehghantanha, and Kim-Kwang Raymond Choo. Machine learning

aided android malware classification. *Computers & Electrical Engineering*, 61:266–274, 2017.

[180] Md Shohel Rana, Sheikh Shah Mohammad Motiur Rahman, and Andrew H Sung. Evaluation of tree based machine learning classifiers for android malware detection. In *International Conference on Computational Collective Intelligence*, pages 377–385. Springer, 2018.

[181] Hanqing Zhang, Senlin Luo, Yifei Zhang, and Limin Pan. An efficient android malware detection system based on method-level behavioral semantic analysis. *IEEE Access*, 7: 69246–69256, 2019.

[182] Hongpeng Bai, Nannan Xie, Xiaoqiang Di, and Qing Ye. Famd: A fast multifeature android malware detection framework, design, and implementation. *IEEE Access*, 8: 194729–194740, 2020.

[183] Jeffrey Mcdonald, Nathan Herron, William Glisson, and Ryan Benton. Machine learning-based android malware detection using manifest permissions. In *Proceedings of the 54th Hawaii International Conference on System Sciences*, page 6976, 2021.

[184] Abdelouahab Amira, Abdelouahid Derhab, ElMouatez Billah Karbab, Omar Nouali, and Farrukh Aslam Khan. Tridroid: a triage and classification framework for fast detection of mobile threats in android markets. *Journal of Ambient Intelligence and Humanized Computing*, 12(2):1731–1755, 2021.

[185] Alejandro Guerra-Manzanares, Hayretdin Bahsi, and Sven Nõmm. Kronodroid: Time-based hybrid-featured dataset for effective android malware detection and characterization. *Computers & Security*, 110:102399, 2021.

[186] Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on automatic machine learning*, pages 66–74. PMLR, 2016.

[187] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[188] Usman Ahmed, Jerry Chun-Wei Lin, and Gautam Srivastava. Mitigating adversarial evasion attacks of ransomware using ensemble learning. *Computers and Electrical Engineering*, 100:107903, 2022.

[189] Xiaolei Liu, Xiaojiang Du, Xiaosong Zhang, Qingxin Zhu, Hao Wang, and Mohsen Guizani. Adversarial samples on android malware detection systems for iot systems. *Sensors*, 19(4):974, 2019.

[190] Harel Berger, Chen Hajaj, and Amit Dvir. When the guard failed the droid: A case study of android malware. *arXiv preprint arXiv:2003.14123*, 2020.

[191] Yinxing Xue, Guozhu Meng, Yang Liu, Tian Huat Tan, Hongxu Chen, Jun Sun, and Jie Zhang. Auditing anti-malware tools by evolving android malware and dynamic loading technique. *IEEE Transactions on Information Forensics and Security*, 12(7):1529–1544, 2017.

[192] Deqiang Li, Qianmu Li, Yanfang Ye, and Shouhuai Xu. Sok: Arms race in adversarial malware detection. *arXiv preprint arXiv:2005.11671*, 2020.

[193] Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. Deceiving end-to-end deep learning malware detectors using adversarial examples. *arXiv preprint arXiv:1802.04528*, 2018.

[194] Deqiang Li and Qianmu Li. Adversarial deep ensemble: Evasion attacks and defenses for malware detection. *IEEE Transactions on Information Forensics and Security*, 15: 3886–3900, 2020.

[195] Shen Wang, Zhengzhang Chen, Xiao Yu, Ding Li, Jingchao Ni, Lu-An Tang, Jiaping Gui, Zhichun Li, Haifeng Chen, and S Yu Philip. Heterogeneous graph matching networks for unknown malware detection. In *IJCAI*, pages 3762–3770, 2019.

[196] Rizwan Hamid Randhawa, Nauman Aslam, Mohammad Alauthman, Husnain Rafiq, and Frank Comeau. Security hardening of botnet detectors using generative adversarial networks. *IEEE Access*, 9:78276–78292, 2021.

[197] David Lopez-Paz and Maxime Oquab. Revisiting classifier two-sample tests. *arXiv preprint arXiv:1610.06545*, 2016.

[198] Gabriel Resende Machado, Eugênio Silva, and Ronaldo Ribeiro Goldschmidt. Adversarial machine learning in image classification: A survey toward the defender's perspective. *ACM Computing Surveys (CSUR)*, 55(1):1–38, 2021.

Appendix A

Bat Algorithm

The bat algorithm is inspired by the echolocation behaviour of micro bats. Bats have extraordinary capability to find prey, avoid obstacles and dIFferentiate between insects in the dark by using sonar. BA is formulated by mimicking the characteristics of bats based on the following rules:

- Bats use echolocation to estimate distance, find prey, avoid obstacles and distinguish between target by nature.
- Bats fly at a random velocity v_i at position x_i with fixed frequency f_{min}, varying wave lenghts λ and loundness A_o in order to search for potential prey. Bats have the capability to automatically adjust the wavelength (or frequency) of their emitted pulse and can adjust the pulse emission rate $r \in [0, 1]$ depending on the position of target.
- The loudness of Bats can vary from a large value A_o to a constant minimum value A_{min}.

Based on the aforementioned rules, BA algorithm is presented in Algorithm 10.

Algorithm 10: Bat Algorithm

```
Input: The objective function $f(x)$, $x = \{x_1, x_2, x_3 \dots x_d\}$
Output: Optimal position of bat $x_*$
 1: Initialize the bat population at position $x_i$
    $i = \{1, 2, 3 \dots n\}$
 2: Define pulse frequency $f_i$ at $s_i$
 3: initialize pulse rate $r_i$ and loudness $A_i$
 4: while $t < max\_iterations$ do
 5:     generate new solutions by adjusting frequency $f_i$, updating velocity $v_i$ and location $x_i$
 6:     if $rand > r_i$ then
 7:         select a solution among the best solutions
 8:         generate a local solution based on the selected solution
 9:     end if
10:     generate a new solution by flying randomly
11:     if $rand < A_i$ and $f(x_i) < f(x_*)$ then
12:         Accept the new solutions
13:         Increase $r_i$ and $A_i$
14:     end if
15:     Rank the bats and find the current best $x_*$
16: end while
17: Return $x_*$
```

Appendix B

Firefly Algorithm

FA is a meta-heuristic algorithm for optimization problems inspired by the flashing patterns and behavior of fireflies. Firefly algorithm is formulated by using following three rules:

- Fireflies are attracted to other fireflies based on the intensity of their brightness.
- The attractiveness and brightness of firefly decreases as it moves away from other fireflies. Fireflies start to move randomly IF they are unable to find a brighter firefly.
- An objective function is used to determine the brightness of a particular firefly.

Based on the aforementioned rules, the pseudo code of FA is presented in algorithm 11.

Algorithm 11: Firefly Algorithm

```
Input: The objective function $f(x)$, $x = \{x_1, x_2, x_3 \dots x_d\}$
Output: brightest fireflies
1: Initialize the fireflies population $x_i$, $i = \{1, 2, 3 \dots n\}$
2: Define light the Intensity $I$ based on $f(x)$
3: Define the absorption coefficient $\gamma$
4: while $t < max\_Generation$ do
5:    for $i = 1 : n$ (all n fireflies) do
6:       for $j = 1 : i$ (n fireflies) do
7:          if $I_j > I_i$ then
8:             move firefly i toward firefly j
9:          end if
10:         Vary attractiveness with distance r via $\exp(-\gamma r)$
11:         Evaluate new solutions and update light intensity
12:      end for
13:   end for
14:   Rank fireflies and find the current best
15: end while
16: Return brightest fireflies
```

Appendix C

Grey wolf optimizer

GWO is a meta-heuristic algorithm inspired by the social hierarchy and hunting strategy of grey wolves. Grey wolves live in a pack of 5 to 12 and are divided in to 4 dIFferent classes (alpha, beta, delta and omega) based on individual responsibilities. Alpha wolf is the head of the pack (regardless of gender) and is responsible to organize, make decisions and lead the pack. Beta wolf is second to the superior in the pack. It assists alpha wolf in decision making and has the authority to take over the command in case of injury or senility of alpha wolf. Delta wolves are the scouts and have the responsibility for security and hunting activities for the pack. Finally, Omega wolves are the elders or the frail wolves. Mostly, they have the responsibility to take care of the off springs. Grey wolves are known for their extraordinary technique for hunting by employing following 3 steps:

- Track, tail and approach towards the prey.
- Encircle, harass and move towards the prey until it becomes to a stationary STATE.
- Simultaneously attack the prey .

Equation C.1 shows the mathematical representation of encircling the prey characteristics of grey wolves.

$$X(t+1) = X_p(t) - A.\|C.X_p(t) - X(t)\| \tag{C.1}$$

where X represents the position of the wolf, the current iteration is presented by t and X_p is the current location of the prey. The controlled coefficients A and C are calculated with the help of equation C.2 and equation C.3

$$A = 2a.r_1 - a \tag{C.2}$$

$$C = 2a.r_2 \tag{C.3}$$

where r_1 and r_2 are randomly generated during iterations from a range of $[0, 1]$ respectively. The controlled vector a linearly decreases from 2 to zero during the iterations as shown in equation C.4.

$$a(t) = 2 - 2.(t/T) \tag{C.4}$$

where T represents the maximum number of iterations. The other wolves in the pack update their position based on the position of alpha (α), beta (β) and delta (δ) wolves as shown below:

$$D_\alpha = \|C_1 \times X_\alpha - X\| \tag{C.5}$$

$$D_\beta = \|C_2 \times X_\beta - X\| \tag{C.6}$$

$$D_\delta = \|C_3 \times X_\delta - X\| \tag{C.7}$$

$$X_1 = X_\alpha - A_1.D_\alpha \tag{C.8}$$

$$X_2 = X_\beta - A_2.D_\beta \tag{C.9}$$

$$X_3 = X_\delta - A_3.D_\delta \tag{C.10}$$

$$X(t+1) = \frac{X_1 + X_2 + X_3}{3} \tag{C.11}$$

where the distance of current wolf from α, β and δ is represented by Equation C.5, Equation C.6 and Equation C.7 respectively. Consequently, Equation C.8, Equation C.9 and Equation C.10 present the position of remaining grey wolves based on the current location of α, β and δ. Finally, Equation C.11 is used to calculate the updated the position of the wolf. Based on the aforementioned explanation, the pseudo-code of GWO is presented in Algorithm 12.

Algorithm 12: Grey wolf optimizer Algorithm

```
Input: The objective function $f(x)$, $x = \{x_1, x_2, x_3 \dots x_d\}$
control coefficient $a$
number of iterations
Output: Optimal position of $X_\alpha$
 1: Initialize the grey wolves population $x_i$, $i = \{1, 2, 3 \dots n\}$
 2: IdentIFy $X_\alpha$, $X_\beta$ and $X_\delta$ based on objective function
 3: while $t < max\_Iterations$ do
 4:    for each grey wolf in pack do
 5:       compute $A$ and $C$by Eq. C.2 and C.3
 6:       Update the position of current wolf using $X_\alpha$, $X_\beta$ and $X_\delta$ by Eq.C.11
 7:    end for
 8:    Update $a$, $A$ and $C$
 9:    Calculate the fitness of each wolf
10:    Update $X_\alpha$, $X_\beta$ and $X_\delta$
11: end while
12: Return Best solution $X_\alpha$
```