# Adversary-aware machine learning models for malware detection systems

Asim Darwaish

▶ **To cite this version:**

Asim Darwaish. Adversary-aware machine learning models for malware detection systems. Artificial Intelligence [cs.AI]. Université Paris Cité, 2022. English. NNT : 2022UNIP7283 . tel-04471124

**HAL Id: tel-04471124**
**https://theses.hal.science/tel-04471124**

Submitted on 21 Feb 2024

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Paris Cité


**École Doctorale Informatique, Télécommunication et Électronique «Edite de Paris : ED130»**
**Laboratoire d'Informatique Paris Descartes (LIPADE)**


# Adversary-aware Machine Learning Models for Malware Detection Systems


Par Asim Darwaish


Thèse de doctorat en Informatique (Intelligence Artificielle et Decision)

Dirigée par Farid Nait Abdesselam

Présentée et soutenue publiquement le 24-11-2022

Devant un jury composé de :

Tayssir Touili, Directrice de recherche (HDR), CNRS, Rapporteur

Hossam Afifi, Professeur, Institut Polytechnique de Paris, Rapporteur

Véronique Vèque, Professeur, Université Paris-Saclay, Examinateur

Ahmed Serhrouchni, Professeur, Telecom Paris, Examinateur

Melek Onen, MCF (HDR), EURECOM, Examinateur

Chafiq Titouna, MCF, Institut Gaspard Monge, Examinateur

Université Paris Cité

# Adversary-aware Machine Learning Models for Malware Detection Systems

LIPADE
Laboratoire d'Informatique PAris DEscartes

**Asim Darwaish**

**Reviewers:** Tayssir Touili, Research Director, CNRS
Hossam Afifi, Professor, Institut Polytechnique de Paris

**Members:** Véronique Vèque, Professor, Université Paris-Saclay
Ahmed Serhrouchni, Professor, Telecom Paris
Melek Onen, Assistant Professor, EURECOM
Chafiq Titouna, Assitant Professor, Institut Gaspard Monge

**Advisor:** Farid Naït-Abdesselam, Professor, U. Paris Cité

Université Paris Cité

This dissertation is submitted for the degree of
*Doctor of Philosophy*

LIPADE October 2022

I wholeheartedly dedicate this dissertation work to my adorable son, lovely wife, loving parents and parents-in-law, siblings, relatives, and many friends. A special feeling of gratitude to my wonderful wife, Sumera, and cute son Aahil, who encouraged and supported me throughout the process. A special thanks to my parents; their motivational words and push for determination ring in my ears. I also appreciate my siblings for keeping me in their prayers for achieving this noteworthy milestone. ...

# Acknowledgements

I want to thank the following people, without whom I would not have been able to complete this research. Thanks to the Higher Education of Pakistan (HEC) for granting me a prestigious scholarship and funding to support my Ph.D. studies. Further, I would sincerely thank my supervisor, Dr. Farid Nait-Abdesselam, for his incredible mentorship, guidance, and thoughtful comments on my research and dissertation. I wish to thank my committee members, who were more generous and benevolent with their expertise, agility, and precious time. Thank you, Dr. Hossam Affifi and Dr. Tayssir Touili, for your detailed comments and for evaluating my thesis report. A special thanks to Dr. Ahmed Serhrouchni, Dr. Veronique Veque, Dr. Melek Onen, and Dr. Chafiq Titouna for agreeing to serve on my committee. I cannot forget to thank my family, parents, and friends for all their support during my devoted academic years. Special thanks to my father, M. Darwaish, and mother, Naseem Akhtar, for their countless prayers and wishes. I would also appreciate my parents-in-law Abdul Sattar and Zahida for their keen interest and special blessings. A special shout-out to my wife, Sumera; without her, I would have stopped my research and Ph.D. studies, you have been amazing, and now I'll clean all dishes with you as promised. For my kid, Aahil, I am sorry for being even grumpier than usual while writing my thesis. Last but not least, a few names from my siblings, colleagues, and friends; I would like to pay my sincere gratitude for their continuous support, including Mujeeb-ur-Rehman, Ehsan Shani, Sheeba Darwaish, Muneeba Darwaish, Musab Darwaish, Abdul Sabbor (Honey), Saira Sattar, Ashfa Sattar, Shahzeb, Farrukh Mehboob, Abdul Rafay, and Quamber Ali.

# Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Asim Darwaish
October 2022

# Abstract

The exhilarating proliferation of smartphones and their indispensability to human life is inevitable. The exponential growth is also triggering widespread malware and stumbling the prosperous mobile ecosystem. Among all handheld devices, Android is the most targeted hive for malware authors due to its popularity, open-source availability, and intrinsic infirmity to access internal resources. Machine learning-based approaches have been successfully deployed to combat the evolving and polymorphic malware campaigns. As the classifier becomes popular and widely adopted, the incentive to evade the classifier also increases. Researchers and adversaries are in a never-ending race to strengthen and evade the android malware detection system. To combat malware campaigns and counter adversarial attacks, we propose a robust image-based android malware detection system that has proven its robustness against various adversarial attacks. The proposed platform first constructs the android malware detection system by intelligently transforming the Android Application Packaging (APK) file into a lightweight RGB image and training a convolutional neural network (CNN) for malware detection and family classification. Our novel transformation method generates evident patterns for benign and malware APKs in color images, making the classification easier. The detection system yielded an excellent accuracy of 99.37% with a False Negative Rate (FNR) of 0.8% and a False Positive Rate (FPR) of 0.39% for legacy and new malware variants. In the second phase, we evaluate the robustness of our secured image-based android malware detection system. To validate its hardness and effectiveness against evasion, we have crafted three novel adversarial attack models. Our thorough evaluation reveals that the state-of-the-art learning-based malware detection systems are easy to evade, with more than a 50% evasion rate. However, our proposed system builds a secure mechanism against adversarial perturbations using its intrinsic continuous space obtained after the intelligent transformation of Dex and Manifest files which makes the detection system strenuous to bypass.

**Keywords:** Malware Detection; Android; Adversarial Attacks; Deep Learning; CNN; Evasion Attacks; Robustness; Machine Learning; FGSM.

# Table of contents

# List of figures

# List of tables

# Chapter 1

# Introduction

## 1.1 Background

The smartphone is the biggest revolution of the 21st century and is considered a game-changer in everyday life, including social interaction, finance management, entertainment, education, and many more. Over 3.6 billion smartphone users worldwide, and total mobile app downloads climb to 38 billion (collectively Google Play and iOS App store) as per [135]. Smartphones are indispensable to a human's daily life, containing sensitive data, for instance, credit card information, healthcare data, personal details, location proximity, and many others. Such information needs utmost security from illegitimate access. Android dominating the maximum market share, a popular choice for end-users due to its versatility, also attracts developers to build legitimate apps due to its open-source availability and development ecosystem. On the other hand, it also serves as a hive for adversaries to infiltrate and disseminate malware for illegitimate means. According to AV-Test anti-virus website report [14] on average, half a million Android malware applications are crafted each month to invade unsuspecting users. The expansion of such malware poses severe security threats, for instance, stealing credentials, leaking sensitive information, financial transactions, calling premium numbers, sending SMS messages, and using private data for other means without individual knowledge and permission. Therefore, the anti-malware industry and researchers need a durable defense line against malware threats to build an effective and efficient Android malware detection system.

Since the last decade, machine learning-based systems have exhibited tremendous versatility in combating various polymorphic malware. These approaches like [168, 170, 82]

recently developed to automate Android malware detection by leveraging different feature representations such as Application Programming Interface (API), permissions, App components, dynamic behaviors, system call graph, etc. These ML-driven approaches offer unsurpassed solutions to resist prevailing malware in mobile applications. However, machine learning is prone to adversarial attacks [59, 144]. An adversary tries to mislead the classifier by carefully crafting adversarial attacks on the detection system by changing the feature importance or data distribution. As a result, the learning system becomes ineffective in a production-ready environment and yields fewer True-positives, for example, a malware sample classified as benign by deteriorating the ML-model prediction.

In computer security, defenders and attackers always engage in a never-ending arms race. They continuously assess the system's strengths, opportunities, and vulnerabilities to make it safe or exploit as per their interest to beat each other methodologies. The defender and attacker competition lead to versatile and secure solutions versus sophisticated and polymorphic routes to exploit vulnerabilities. For instance, to make the traditional signature-based malware detection techniques ineffective by re-packaging and code obfuscation. Attackers create polymorphic and dynamic malware by defeating attempts to analyze the inner mechanism. For the last few years, the study of adversarial machine learning has risen. Researchers had successfully evaded various malware detection systems in various domains like PDF malware detection [92, 155], Windows [84, 87], and Cloud-based anti-virus engines [27]. However, investigating adversarial perturbations for android malware detection systems and their resilience under adversarial invasion is scarce.

## 1.2 Research Motivation

Smartphones have emerged as a vital piece of daily life tasks for accessing valuable electronic services such as online shopping, mobile banking, food ordering, governance, eHealth services, and many more. According to [159], the total number of worldwide users of smartphones reached 6.378 billion, meaning that 80.69% of the world population carries a smartphone. These smartphones carry sensitive data that need paramount security attention for mobile computing. The Android smartphone is a popular choice due to its multifaceted offering and captures 85% of the global market [8]. Being an open-source operating system (OS), it attracts application developers to build legitimate mobile applications. On the flip-side, Android OS is a favorite swarm for malware authors due to its popularity, open-source availability, and intrinsic infirmity to access internal resources, as depicted in Figure 1.1.

Anti-virus software provided by anti-malware companies establishes the first line of defense to secure computing and handheld devices against malware threats. The threat of widespread malware has been obstructed by a few classical approaches such as signature detection, static analysis, and dynamic analysis. Signature and heuristic-based detection techniques were commonly used to combat the thriving mobile ecosystem against malware attacks. The signature-based detection needs manual intervention and works for known attacks. It always fails on new variants due to non-availability in the signature database. Static and dynamic analysis approaches are often used to analyze the behavior of malware. The static analysis methods were ineffective in code obfuscation, packaging, and sophisticated malware. The dynamic analysis requires significant resources to execute in the sandbox environment and may not be effective against dynamic evasions, and need more execution time to analyze the behavior of malware.

The detection and isolation of continuously evolving malware is a consistent challenge for the anti-malware industry and research community. Cybersecurity experts are always searching for a durable defense against malware apps for mobile and IoT devices. With the exponential growth of mobile apps, it is notably challenging to manually examine the malicious behavior of each application using signature or heuristic-based approaches. The worldwide adoption and advances of AI (Artificial Intelligence) are the paradigm shift in the technology landscape and enforced its deployment in every domain like Computer vision, healthcare, e-commerce, retail, autonomous vehicles, cybersecurity, and many more. Cybersecurity space is an arms race among attackers and defenders. They continuously and respectively apply different strategies to make the target system vulnerable and secure against new attacks. Defenders have used machine learning and deep learning as effective weapons in security-sensitive tasks. Such tasks include intrusion detection, anti-virus software, phishing attacks, malware detection, and spam filtering.

Machine learning has been proposed and widely adopted by security-sensitive domains to overcome the challenges mentioned above and the limitations of legacy solutions for automated malware detection. Like other domains, machine learning and deep learning-based malware detection systems are prevailing and count as favorite weapons by defenders. Learning-based approaches have proven more potent due to their automation, generalization, optimization, and personalization capabilities. Over the last decade, these learning techniques have offered unprecedented versatility to combat malware attacks on static and dynamic analysis routes. Despite rapid development and superior performance, machine learning is vulnerable to adversarial attacks [144, 59]. Numerous studies in the literature showed

that the learning classifiers are easy to evade by crafting meticulous adversarial examples either in the training or testing phase [45, 71, 59]. An adversary can deceive the target model by crafting subtle perturbations known as Adversarial Examples (AE's). Therefore, deploying learning-based detection and classification systems in security-sensitive domains needs thorough contemplation. Despite the popularity of learning-based detection systems and their wide adaptability, they are the weaker link in security and cannot be trusted blindly.

The study of adversarial learning has been a hotspot research area and mainly focuses on images due to the easy visualization of perturbations [59, 103, 32]. However, research evaluating the impact of adversarial perturbation on image-based learning classifiers for android malware detection systems is still relatively scarce. Few researchers have analyzed the adversarial evasion in PDF malware detection [92, 155], Windows executable [84, 87], and android malware detectors but limited their research to binary space [31, 166, 62]. An adversary constantly explores vulnerabilities of the learning classifier by adding non-perceptible perturbations to compromise the integrity or availability of the system. Generating adversarial examples is easy in images by altering pixel values, but the malware domain has specific constraints due to its binary feature representation. Alteration of a boolean feature can violate the malware functionality constraint or disrupt the original functionality of the application.

## 1.3 Contributions

This research work comprises two phases. In the first phase, we present a novel image-based Android malware detection system using CNN that requires zero code analysis and no fine-grained feature engineering for assigning malware signatures and family classification. We aim to build a lightweight, robust, and scalable malware detection system by applying an intelligent transformation of APK files to RGB images. The system classifies the signature of the resultant image as either malware or benign and further identifies the family of predicted malware. We examine the adversarial attack taxonomy on machine learning-based Android malware detection systems in the second phase. We evaluate the robustness of our image-based android malware detection system called RoboDroid under novel adversarial attacks. The main contribution of the research work is enumerated as follows:

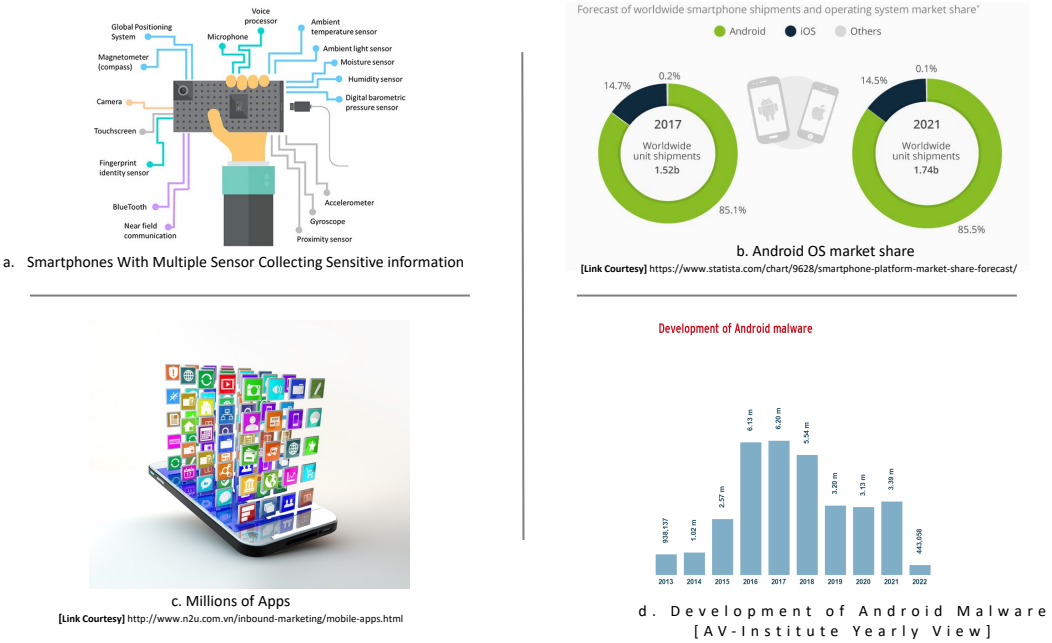



Fig. 1.1 Android OS capture the biggest market share and actively collecting user data through sensor and various app. According to AV-Test report [14]: On Average half a million android malware reported every month

### 1.3.1 An Intelligent Transformation of Android APKs to Images

Our proposed system performs a static analysis of the Android APK file using a reverse engineering tool *Androguard*. The system applies an intelligent transformation of dex and the manifest files to map their behaviors onto an RGB image instead of blind transformation or a simple bytecode conversion into gray-scale images. We devise a predefined exhaustive dictionary [41] based on a literature review and various experiments that enumerate malicious behaviors from the APK file. The dictionary comprises suspected properties, including API calls, permissions, App components, filter intents, etc., and their corresponding pixel values. We continuously update the dictionary on the invention of new malware variants and their respective properties. The system uses this dictionary as a lookup table for mapping the extracted values from the manifest and the dex file onto the respective RGB channels of the image. We trained a CNN on resultant images for malware detection and family classification. The proposed system is lightweight, scalable, and generalized to detect legacy and new malware from the AndroZoo dataset [6].

### 1.3.2 Crafting Novel Adversarial Attack Model

An adversary is always eager to intrude the security fence by compromising the integrity or availability of the system. The goal of an adversary is to force the learning classifier to predict malware samples as benign samples, also known as integrity attacks. However, the prediction of benign samples into malware is known as denial of service attacks (DoS) and counts as availability attacks. Adversarial perturbation is a hotspot area in the computer vision domain to deceive learning classifiers but has different challenges when dealing with the malware (binary) domain. In Computer vision, the goal is to add minute perturbations but non-perceptible to the human eye and strong enough to deceive the trained classifier. However, adding adversarial perturbations to a binary domain is complex because altering Boolean features from 0-1 and vice versa will likely break the malware functionality constraint. The basic rule of adversarial perturbation is not to violate the semantics of malware functionality constraints nor disrupt the original app's execution. Preserving the malware functionality constraints, we crafted two novel adversarial attacks for an Android malware detection system. a) Adding benign-only properties to a malware sample can disrupt the system's integrity. b) Adding a portion of a benign image to a malware sample using Deep Convolutional Generative Adversarial Network (DCGANs) [43]. c) Adding a portion of a faked benign image generated through GAN and altering newly added benign properties using modified FGSM (Fast Gradient Sign Method). The second and third attacks directly apply to image-based android malware detection systems. However, using our predefined dictionary, it can be launched for binary space by converting the pixel values to corresponding benign properties.

### 1.3.3 Robustness evaluation under newly crafted Adversarial Attacks

The robustness of the proposed system dawns on the fact that adding perturbation to the transformed RGB image of a malware sample has a very high probability of violating the malware functionality constraint and may demolish the app functionality. Therefore, the image-based representation builds a naturally inherent defense mechanism against adversarial attacks and does not require adversarial training, which is computationally expensive. To illustrate the robustness of the image-based malware detection system, we implemented state-of-the-art android malware detection techniques, such as DroidDetector [168], Deep-ClassifyDroid [170] and visualization-based android malware detector [171] along with our proposed approach using the same AndroZoo [6] dataset repository. The RoboDroid and the

other three implemented approaches, DroidDetector, visualization-based detection system, and DeepClassifyDroid, are tested for adversarial robustness and evasion rate under the novel threat model. The experimental results and detailed evaluation revealed that the RoboDroid is more secure than other learning-based android malware detection approaches against adversarial attacks. The results demonstrated that the evasion rate for DeepDroid [168], DeepClassifyDroid [170], and visualization-based approach [171] is more than 50% by adding 12-13% adversarial perturbations. While the evasion rate of RoboDroid (our image-based android malware detection system) reaches 13% with maximum perturbation. However, in a few scenarios, it built a durable defense line and no evasion rate against 3-4% perturbation compared to other approaches.

## 1.4 Organization of this Thesis

The rest of the thesis is broadly organized into two parts and comprises seven chapters.

- **Part 1: Background and State of the art Malware Detection Systems**
- **Malware Forensics: Background, Recent Advances, and Future Challenges:** Chapter 2 sheds light on the history of malware, basic definitions, and guidelines to avoid malware, various types, symptoms of malware, and malware for handheld devices. The chapter also discusses the legacy solutions, recent advances, and current malware detection challenges.
- **Android Malware Detection Systems and Adversarial Attacks:** Chapter 3 presents preliminary concepts pertinent to Android APK structure and adversarial attacks in the context of malware. This chapter also presents the state-of-the-art learning-based android malware detection systems and their limitations. Lastly, it discusses the adoption of adversarial attacks in the malware domain.
- **Part 2: Proposed a Secure Android Malware Detection system under Adversarial Attacks**
- **RGB-Based Android Malware Detection System using Deep Learning:** Chapter 4 presents our proposed Android malware detection system that transforms APK into images and then uses Convolutional Neural Network (CNN) to detect malware and classify its family. The generation of images dataset for malware detection, experimental setup, and empirical results show its viability and detection accuracy.

- **Novel Adversarial Attacks for Learning-Based Android Malware Detection Systems:** Chapter 5 presents three methodologies for crafting novel adversarial attacks for learning-based Android malware detection systems.
- **Evaluation for Adversarial Robustness of Learning-Based Detection Systems:** Chapter 6 presents the detailed experiments for the robustness evaluation of our proposed Image-Based Adversary-Aware Android Malware Detection System in comparison to state-of-the-art learning-based detection systems. The chapter also discusses various defensive strategies and their effectiveness in the presence of our novel attacks.
- **Conclusion:** Chapter 7 concludes the main contributions of this thesis, and also discusses the future directions to extend this research work.

## 1.5 List of Publications

- Malware Forensics: Legacy Solutions, Recent Advances, and Future Challenges. A book chapter published in Advances in Computing, Informatics, Networking and Cybersecurity, (Springer 2022) [105].
- RGB-based android malware detection and classification using convolutional neural network. A paper published in the IEEE Global Communications Conference (GLOBECOM 2020) [40]
- Robustness of image-based android malware detection under adversarial attacks. A paper published in the IEEE International Conference on Communications (ICC-2021) [43]
- An intelligent malware detection and classification system using apps-to-images transformations and convolutional neural networks. A paper published in the 16th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob 2020) [104]
- An Adversary-aware Android Malware Detector Under FGSM-Based Attacks. Submitted to IEEE Transactions on Dependable and Secure Computing (IEEE TDSC 2022)
- Robodroid: A Robust Image-based Android Malware Detection System Against Adversarial Attacks. Submitted to Journal Computers & Security (Elsevier 2022)

# Chapter 2

# Malware Forensics: Background, Recent Advances, and Future Challenges

## 2.1 Introduction

Medical campaigns and awareness programs are launched annually to proactively combat seasonal flu and other viruses. The recent outbreak of the COVID-19 pandemic killed millions of lives, and special measures have been taken to prevent its spread. On the contrary, there is no particular predictable season for PCs, handheld devices, smartphones, servers, and IoT devices to become infected with a virus. There is a continuous threat to computing machines and a never-ending flu season.

Malware applications are continuing to grow across all computing and mobile platforms. Since the last decade, half a million malware applications have emerged as a real threat daily and hamper prosperous mobile and computer ecosystems. With the exponential growth of the smartphone market, detecting and preventing malware for handheld devices require paramount attention. Enrichment and diversification of mobile applications, tools, sensors, and services with underlying sensitive information always allure malware writers to exploit vulnerabilities. Android is an open-source and leader in the smartphone market. It is more vulnerable and has spawned Android malware applications. *Malware* is the payload created by an attacker to compromise system integrity, availability, and confidentiality. Mostly, the interest is to steal confidential information and financial data, crippling critical infrastructure and servers. Malware leads to severe financial losses for companies, institutions, and

individuals. Malware writers keep on coining novice methods and sophisticated routes for creating malware for mobile, computers, and servers.

Malware infections come at us like a torrent of arrows on the battlefield, each with its poison and method of attack with stealth, deceitful and subtle. Therefore, it is imperative to devise a system that can detect and prevent legacy and new malware with high accuracy under different settings. Malware detection and isolation studies are preventive immunization against infection and wounds. In this chapter, we present a brief background and history of malware across different platforms, existing solutions, and malware analysis techniques for detection and prevention.

## 2.2 Background and History of Malware

Given the daily generation of massive malware variants for different platforms, it is impossible to enumerate the complete history of malware here. However, the malware trends that are convenient to manage in recent decades are circumvented below.

"Automatically self-reproducing" consider the groundbreaking of viruses that began in the 1970s and occurred in pre-personal computers. The history of malware and viruses instantiated by a program named Elk Cloner spread by a floppy disk. The virus impacted all the connected disks on Apple II systems in 1982 and was known as the first outbreak in computer security and occurred in the pre-windows PC era. The widespread use of viruses, worms, and malware started in the early 1990s with the emergence of the windows PC. The first malware spread started with the flexible macros of window's application, providing a gateway for malware authors to inject malicious code using MS Word micros. The macro-based injected code infected the documents and templates despite the portable executable (PE) program. The famous IM (instant messaging) warm outbreak stumbled the big giants' Yahoo and MSN messenger by spreading the self-replicating malicious code leveraging the network loopholes. Similarly, from 2005 to 2009, a famous Adware malware proliferated on various devices. It displays the advertisements by opening new windows and pop-ups and deliberately disables users to turn them off. Initially, Adware expanded its spread by piggybacking well-reputed companies, but the legal action against these adware companies soon forced them to shut down the campaigns.

With the boom of social networks, scammers targeted the famous MySpace in 2007 for delivering scoundrel advertisements, offering fake security tools and antiviruses. They

deceive the users via social engineering cheats. Later, Facebook and Twitter became the main targets as MySpace's popularity vanished. The common methods for social networks were links to phishing pages and advertising Facebook applications with malicious extensions. A new variant of malware named ransomware was introduced using the name CryptoLocker and targeted Windows-based PCs from September 2013 to May 2014. The CryptoLocker attack has collected 27 million dollars from victims. Similarly, a different ransomware variant remained effective until 2015 and gathered millions of dollars. In 2017, ransomware became the king of malware, affecting all domains and businesses by encrypting victims' data and asking for payments to release it. Since 2018, cryptocurrency has been hype, which gave birth to a new malware named Cryptojacking, which operates in a stealth mode on a victim's device and uses its resources to mine for cryptocurrency.

### 2.2.1 Definition of Malware

Malware is an umbrella term that means malicious software intends to infect the computing system. In the computer security domain, "the program aims to breach the system's security policy by compromising confidentiality, integrity, or availability." Numerous other definitions exist in the literature that loosely defines *malware* as an "Annoying software program" or "Running a code without the user's consent." The primary purpose of a malware author is to collect illegal monetary benefits by injecting malware to damage the system (PC, desktop, smartphones, tablets, and networks) or make it nonresponsive or disable any services/feature.

### 2.2.2 Symptoms of Malware Infection

The symptoms of malware infection are revealed through various abnormal behaviors. Following are a few pointers that help us to alarm against malware infection

- Slowing down the operating system of the computer, tablet, smartphone, or any other device. Whether we are browsing the Internet or using a local file system.
- Bombardment of annoying ads on the user's screen is a typical sign of adware and is especially harder to close. Moreover, they allure the users to click and offer various fake prizes, etc.
- Recurrent system crashes, for example, a blue screen on a Windows PC.
- Mysterious loss of storage space is another hint of hidden malware in the user's system.

- Sudden resources of the user's system reaching the peak and the cooling fan starting at full swing indicate malware consuming the resources in the background.
- Frequent and automatic change of home browser without permission; similarly, new plugins are installed and appear on the browser without the user's consent.
- Anti-virus scanners or security tools installed on our system do not respond and cannot be updated.
- Loss of important data from the user's system in a stealth mode by famous ransomware and then demands the user to pay to recover it.
- If everything seems perfect, do not get smug. Few potent malware can stay in stealth mode without raising any red flag and keep collecting our confidential data and sensitive files, like bank cards and passwords, etc., for the wrong dirt.

### 2.2.3 Types of Malware

There are various kinds of malware and millions of their families, but the most common and scoundrel's group of malware are discussed below:

- **Adware** is a kind of malware that pops numerous advertisements on the system via a web browser. Usually, it infiltrates the system by impersonating itself as legitimate software or piggybacking on other software.
- **Spyware** is deceitful, resides in the target system in stealth mode, and reports the activities to the malware's author without the victim's permission and knowledge.
- A **virus** is a kind of malware appended to other software and unintentionally executed by the end-user. It starts replicating itself and modifies other installed programs by infecting its piece of code.
- **Worms** are malware similar to viruses that destroy the system's files and user's data.
- **Trojan** It is also known as a Trojan horse. It is a special type of malware that appears to be necessary for the system to deceive the user. Once it has entered into the system, it also gives birth to other malware like viruses and ransomware, and more importantly, it is injected to steal financial information.

- **Ransomware** is the most dangerous malware. It is considered a cyber criminal's weapon that encrypts and steals our important files/data and then demands a ransom to make them available.
- **Rootkit** operates stealthily on the system and keeps itself hidden from the OS and other software. It provides root permission to malware attackers.
- A **keylogger** is a spy on the keystrokes of the touchpad and keyboard; it stores and sends the gathered information to the attacker to manipulate sensitive information such as usernames, passwords, credit cards, and financial information.
- **Cryptojacking** is a specialized malware that steals the victim's resources to mine cryptocurrencies such as Bitcoin or Monero and many other coins.

## 2.3 Malware for Handheld Devices

Malware authors love penetrating the smartphone market because mobiles are sophisticated handheld computers. Nowadays, smartphones carrying multiple sensors endowed with sensitive information consistently allure malware criminals to make more money. There are numerous ways for malware to intrude on our smartphones and exploit malicious activities, for instance, trojan, adware, spyware, ransomware, and many more. A report [135] presented statistics on smartphones and reported almost 3.5 billion smartphone users in 2020 and predicted 4.4 billion users in 2022. Mobile users are a relatively easy target for malware authors compared to conventional desktop users. Most smartphone users do not protect their phones by installing security applications and do not follow security measures as they do for their computers. Due to this negligence, they are more vulnerable to even primitive malware. Moreover, various cheap smartphones are available in the market with preinstalled malware. These smartphones use the open-source Android operating system. Android and iOS are the two big giants in the smartphone market, capturing the 80% and 15% market, respectively. Due to the most significant market capture and open-source platform, Android is the hive for malware authors. The subsequent subsection also spotlights Android and iPhone separately.

### 2.3.1 Malware Symptoms for Android Devices

- A sudden annoying pop-up advertisement tempts the end-users to click and takes them to sketchy websites. This indicates that adware is hiding in our android devices and alluring us to click on the advertisement.
- Surprisingly increases in data usage. Malware consumes users' data plans by displaying numerous ads and sending information to the outworld without the user's permission and knowledge.
- The malware sends SMS and dial in premium numbers in stealth mode resulting in unknown and false charges on the bill.
- Sudden drainage of battery power. The malware uses resources and sucks up the battery's energy compared to normal usage. Trojan performs stealth processing that wipes out the user's battery's power and can potentially damage both the battery and the phone.
- Obnoxious calls and SMS received by contacts from the user's phone number.
- The stealth installation of unknown apps on a user's phone by piggybacking on other famous apps.
- Automatically turning on WiFi, internet, and personal hotspot on your phone.

### 2.3.2 Malware Symptoms for IPhone

Apple iPhone is secure, protected, and studies revealed that they are not easy to invade by malware authors, but it does not mean that malware does not exist on iPhone. Malware can target the iPhone in two rare scenarios. The first way is the state-level adversary authorized by the government to target the obscure security loophole in iOS. Apple has done a great job of making the iOS and app store secure and has a rigorous scanning mechanism on the device and App store. The second route is the jailbreak of the phone by the user and removes all restrictions imposed by Apple. Using the iPhone and iOS does not mean the user is safe from malware. If the user taps a link received in a message from an unknown source, it can lead to the website where login details and other personal information is requested by alluring the end user.

### 2.3.3 General Guidance for Malware Prevention: Dos and Don'ts

Always stay vigilant and avoid domain names ending with odd letters. As discussed in previous sections, pay close attention to all indications and symptoms to prevent malware infection. Do not click on ads and pop-ups; stay away from unknown email attachments. Always use the up-to-date OS, browsers, software, and plugins. Smartphone users only download and install applications from a trusted source, such as the Google Play Store for Android and App Store for iOS, and block third-party sources for app download and installation. The business and corporate sectors should implement strict security policies and their compliance to prevent malware. Finally, always install updated and reliable antivirus or malware detection and prevention systems.

## 2.4 Malware Detection and Prevention: Legacy Solutions

The techniques used to detect malware can be categorized into two main categories: Static and Dynamic Analysis. Fig. 2.1 presents a brief taxonomy of malware detection approaches.

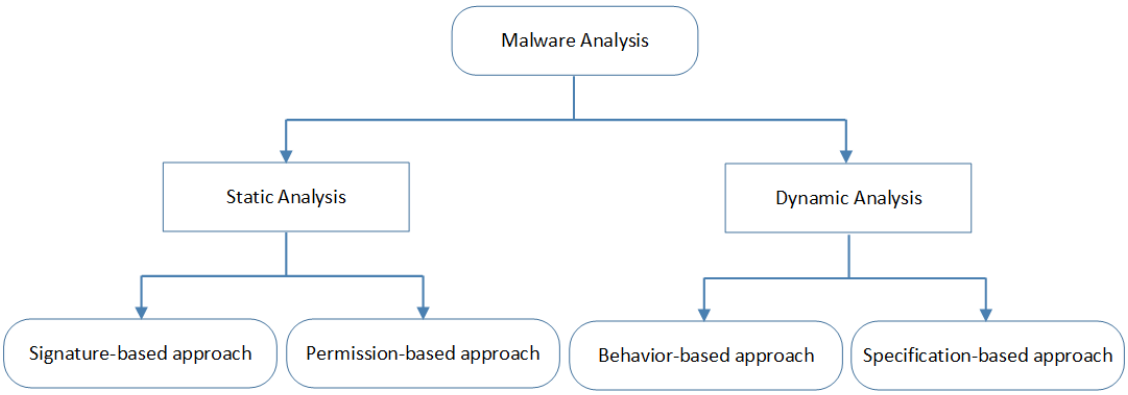



Fig. 2.1 Taxonomy of Malware Detection Approaches.

### 2.4.1 Static Analysis Techniques

In static analysis-based approaches, malicious software is detected without its execution or installation on the target system. The properties and characteristics of the internal structure are used to detect malicious codes. Therefore, the program's executable should unpack using a disassembler or debugger before analyzing any code using reverse engineering methods.

There are various tools to automate the job like IDA Pro [69], OllyDbg [110], XXD [164], Hexdump [65]. IDA Pro and OllyDbg are disassembler programs that create maps of software execution to illustrate binary instructions as an assembly program. The disassembly method helps software experts check the structure of programs and detect malicious instructions. XXD is a simple Linux command that produces hexadecimal or binary dumps of files in several different formats. It can reverse engineer by converting its hex dump format back into the original data. Hexdump is a tool that shows the contents of binary files in hexadecimal, decimal, octal, or ASCII. Hence, XXD and Hexdump are very useful for inspecting the details of the functionality of any program.

Static analysis extracts feature statically, meaning the binary or executable files are checked for malicious behavior before execution. Numerous studies showed that various features could be used to analyze malicious activities, and the following are the most commonly used in the literature.

- API calls their sequence and occurrence patterns. Various studies in the literature used API calls as a feature to detect malware [80, 63, 115, 161, 160].

- Control Flow Graph (CFG). A control flow graph is created after performing the disassembling operation on a binary file. The graph nodes represent instruction blocks, and the edges indicate the control flow of instructions. Eskandari *et al.* [49] introduced the CFG representation method with the combination of called APIs to detect malware. Xu *et al.* [163] presented the combination of CFG and DFG (Data Flow Graph) to detect malware using deep learning techniques.

- Analysis of Application components for Android malware detection [53, 13, 91].

- Opcode (Operation code) sequences; Opcode represents the machine language instructions, classes, and functions along with operands [125]. McLaughlin *et al.* [99] performed a static analysis of raw opcode sequences to detect malware using CNN.

- Dynamic-link library (DLL) imports are used to detect the behavior of malware. Narouei *et al.* [106] presented a heuristic-based approach to detect malware on the route of static analysis using tree representation of DLL.

- N-gram; n-grams are the unique sub-strings of length N from raw bytes. Most of the n-gram feature extraction methods used NLP-centric algorithms to analyze malicious activities. Zhang *et al.* [55] proposed a malware detection method based on n-gram similarity attributes.

- Model checking: Model checking is the technique to detect malware by analyzing and validating the bytecode after the source code compilation [137, 17, 24]

The above static techniques can be categorized into signature-based and permission-based approaches.

#### 2.4.1.1 Signature-based Approach

The signature-based approaches are widely used to detect viruses and malware. A signature is a known pattern or simply a sequence of bytes that can be leveraged to identify specific malware. This approach uses pattern matching schemes to scan for signatures in a database where malware signatures are stored. Updating this database is highly recommended to deal with new types of malware. Therefore, malware detection begins by creating the appropriate signatures for each software that should be analyzed and comparing them with signatures already stored in databases. If a match is flagged, the software is considered malware where any further actions are blocked.

Sung *et al.* [143] proposed a robust signature-based malware detection method to detect mutated and obfuscated malware. The proposed technique is called SAVE for Analysis for Vicious Executable. The authors assumed that all malware variants use the same core signature that presents the set of features of the code. Therefore, detecting one of the variants leads to identifying the rest by examining the detected signature. The SAVE algorithm decompresses the Microsoft Windows Portable Executable (PE) binary code to extract the Windows API calling sequence. Letters are represented by a 32-bit integer id number, where the 16-bit most significant represents the Win32 module, and the other 16-bit represents a specific API in this module. SAVE compares the extracted API calling sequence with a set of signatures already stored in a database using a similarity metric module. This module uses Euclidean distance and well-known functions like Cosine, extended Jaccard, and the Pearson correlation measures. The author's experimentation includes malware detection work such as MyDoom, Bika, Beagle, and Blaster worm. Their comparison with eight antivirus scanners showed that SAVE outperformed the detectors compared.

Christodorescu *et al.* [36] presented a tool called SAFE (Static Analyzer For Executables) to detect malicious patterns in x86 executables. The SAFE tool is resilient to common obfuscation transformations. Later, authors [35] proposed a program to evaluate the resilience of malware detectors against obfuscation transformations usually employed by hackers to mask the malware. In practice, malware authors search for new methods to evade signature-

based malware detection. They developed metamorphic-based techniques that morph the code after each infection, i.e., the code structure of a malware changes, but the malicious functionality remains [145]. Several works focused on detecting metamorphic malware [158, 139, 18], where the authors used opcode-graph [122] or statistical analysis [147]. Shanmugam *et al.* [130] proposed a similarity-based technique in which a simple substitution cipher cryptanalysis is used for measuring the distance between executable. The signature-based technique is also used for android malware detection [172, 56]. The authors in [172] proposed an approach called DroidAnalytics, that is, a signature-based analytical system to automatically collect, manage, analyze, and extract android malware. Gao *et al.* [56] proposed an approach based on the function call graphs extracted from android App packages. The authors claimed that their approach could correctly identify malware and its variants.

Song *et al.* [137] developed a tool called PoMMaDe for Push down Model-checking Malware Detector. The latter detection system models the binary program as a pushdown system (PDS) to track the program's stack. The authors claim that PoMMaDe can detect 600 real malware where 200 of which are new variants. The same authors in [136] proposed an approach where they model the Android apps as a PDS. This technique detects private data leaking working at the Smali code level. Battista *et al.* [17] used model checking to detect Android malware families (the DroidKungFu and the Opfake families). The proposed approach analyzes and verifies the java bytecode generated after a source code compilation phase. The author affirms that the preliminary results seem significant, and the detection of malicious payloads is achievable with a very high accuracy rate within time. A similar approach has been proposed by Canfora *et al.* [24]. The author developed a model checking-based tool called LEILA (formaL tool for idEntifying mobIle maLicious behAviour), where they employed an algebras process named CCS (Calculus of Communicating Systems). The experimental results revealed that LEILA could detect malicious behavior efficiently with an accurate rate of over 0.97%.

The main disadvantage of the specification-based approach in malware detection is the complete and accurate specification of normal behaviors. Even for moderately complex apps, designing a complete specification of all its correct behaviors cannot be achieved. In addition, if we can specify all behaviors of an application with natural language, we cannot ensure the translation correctness into the language machine. Antivirus and malware vendors mostly use signature-based detection approaches for their flexibility, simplicity, and efficiency against various categories of malware. A drawback of these approaches is that: a) it requires an

updated repository of malware signatures; b) the detection always happens after the malware harms systems; c) it fails to identify new malware that utilizes polymorphic methods.

#### 2.4.1.2 Permission-based Approach

Permission-based approaches have attracted researchers since developing intelligent devices using Android, iOS, etc. In such systems, threats have increased by developing a new kind of malware based on permissions. The operating systems deployed on smart devices (Android) operate with permission to authorize new applications to access the stored data and the device itself. Therefore, the system forces applications to declare the permissions required for desired functionality. If the operating system does not grant the required permissions, the application cannot access the particular service; otherwise, it can use it without limit. Permission-based approaches are considered a faster detector to identify malware and applications with suspicious permission requests.

Wu *et al.* [160] proposed a static technique called DroidMat to detect Android malware using a set of information such as API calls, Intent messages passing, deployment of components, and permissions. They extracted the information from the manifest file and monitored App components to track the API calls. Then, the K-means algorithm is applied to enhance the malware modeling capability. Finally, DroidMat uses the k-Nearest Neighbors algorithm (KNN) to classify applications as malware or benign. Like the previous work, Peiravian and Zhu [114] presented a permission-based approach using a set of classifiers. The advantage of this technique is that it does not require the dynamical tracing of the API calls used by the system. In [29], the authors proposed almost the same previous work using a feature set containing permissions and API calls for Android malware. They conducted a series of experiments using various machine learning algorithms such as Multi-layer Perceptron (MLP), Support Vector Machine (SVM), Naive Bayes, and Radial Basis Function (RBF). Liang and Du [94] developed a security scheme called Droid Detective, in which they combined permissions in sets. Droid Detective obtains permission combinations generally required by malware applications, generating a set of rules to detect malware. The results showed that the proposal has a detection rate of over 96%. Chaugule *et al.* [30] developed a tool called SBIDF for specification-based Intrusion Detection Framework. The proposed approach uses keypads and touchscreen interrupts to detect unauthorized malicious behaviors. The authors used TLCK (Temporal Logic of Causal Knowledge ), an application that describes normal behavior patterns. The results showed that SBIDF could detect malware that tries to use sensitive services without the authorization of the users.

Recently, Nivaashini *et al.* [153] proposed an approach in which permission detection is performed automatically. Then, apply a classification model to classify applications. The authors claim that achieving 97.88% is the detection rate. Kapoor *et al.* [77] classify applications on malware into benign using six machine learning algorithms. The logistic Regression Algorithm provides a high accuracy rate equal to 99.34%. In [124], the authors employed a non-parametric technique called Kernel Density Estimation to estimate the probability distribution of data. Then, the authors categorized permission into signature, dangerous, and normal to simplify the analysis.

Despite the performance of the permission-based approach, it still presents several disadvantages. This approach generates a high false-positive rate since many benign applications require permissions to function correctly, which are considered dangerous by the malware detector. In addition, malware can perform without requesting any permissions.

### 2.4.2 Dynamic Analysis Techniques

In dynamic analysis-based approaches, malware detection occurs after the application installation. This type of technique analyzes the behavior of apps during their execution. Dynamic analysis continuously checks apps' API calling, instruction traces, memory writes, etc. The advantage of dynamic-based approaches is the ability to detect new, unknown, and obfuscated polymorphic malware. The main categories of these approaches are behavior-based approaches and specification-based approaches.

#### 2.4.2.1 Behavior-based Approach

The behavior-based malware detection approach examines the execution of the application to ascertain if it works as malicious or benign software. When an application launches in the system, the behavior-based technique analyzes its activities like the system calls used by the application. Generally, in such approaches, the execution is done within a sandboxed scenario, and a set of run-time tests are verified. The main advantage is that allowing comprehension of all malware behaviors.

Kolbitsch *et al.* [83] proposed an approach based on graphs for detecting malware. A behavior model is developed after analyzing a malware application in a controlled environment called Anubis. This behavior model is represented as a graph where the edges are the call's transition while the nodes are the system calls. Then, the obtained model is

compared to known graphs to detect malware programs. The authors claim that the proposed approach presents effective detection with an acceptable performance overhead. Burguera *et al.* [22] proposed an efficient tool that can realize a deep analysis to detect Android malware dynamically. The proposed approach is based on observing the behavior of applications to identify malware. Thus, the tool employs the concept of crowd-sourcing to collect traces from real users and then perform an analysis on a remote server. The latter is a collector of information from its clients (Crowdroid). All data are analyzed, and a behavior database is generated. This database is used to detect malware by applying the k-means algorithm. The authors achieved a detection rate of 100% when using self-written malicious apps and 85% accuracy when using the Monkey Jump to the app.

Mohaisen *et al.* [102] presented a behavior-based malware system called AMAL. This proposed system comprises two sub-systems named AutoMal and MaLabel. AutoMal collects low granularity behavioral artifacts that indicate memory, registry, and file system utilization. Besides that, the MaLabel uses artifacts to build and train classifiers to detect malware programs. The evaluation result showed that MaLabel realizes a 99.5% of precision and 99.6% recall. Similar to the approach presented in [83], Polenakis and Nikolopoulos [108] proposed a graph-based technique where they developed a model which exploits the relationships that exist between system calls. The proposed solution uses the system-calls dependency graphs that are already generated. The authors employed three metrics; Δ-similarity, SaMe-similarity, and NP-similarity metrics for the process of detection and classification.

Saracino *et al.* [127] presented a system called MADAM, a host-based malware detection system designed for Andriod equipment that can perform analysis on four levels; package, application, user, and kernel. MADAM can classify malware based on its behavioral classes. MADAM detects and blocks at least 96% of malware. Choi *et al.* [34] proposed a new technique where they used the FPGrowth algorithm and Markov Logic Networks algorithm. The FPGrowth algorithm is used to find associations that exist between patterns to create malware behavior patterns. And then, the Markov algorithm is applied as an inference step to measure the correlation between patterns.

Behavior-based detection is considered an efficient and effective method to deal with new malware variants. Nevertheless, these approaches consume more execution time to determine behaviors, extract features, and detect abnormal behaviors. In addition, some malware programs are very complex, which makes building behavior patterns a big challenge.

#### 2.4.2.2 Specification-based Detection

The promulgation of law inspires specification-based approaches, typically anomaly-based detection systems. Specification-based approaches rely on legal and valid rules and behaviors to distinguish between malign and benign applications. The specification-based system dynamically runs the application and analyzes the authorized behaviors and rules with a pre-determined valid set of rules and behaviors. Apps non-compliant with a valid set of pre-determined rules are classified as malicious; otherwise, benign.

Generally, specification-based approaches involve two phases, training and testing. In the training phase, an authorized set of rules and behaviors collectively called specifications are designed for a particular app and device. The testing phase analyzes whether the behaviors exhibited by a given app during the execution comply with specified rules or not. Sheen *et al.*[133] proposed a collaborative Android malware detection system using APIs and permissions as a feature set and fusion of multiple classifiers for final prediction. They have achieved remarkable accuracy, but computation and resource consumption are incredibly high. Enck *et al.*[48]proposed a system called TaintDroid that tracks the flow of private and sensitive information by treating third-party apps as untrusted. They need real-time monitoring and users' private data to generate specifications for malware detection. They have labeled sensitive data, established rules, and applied flags for data distribution within the application, program variables, and inter-process/program communication. They found that many inter-process and program connections/communication violate the privacy rules for data flow, but their detection time was too high. Rehman *et al.*[119] proposed a Markov Logic Network for malware detection. The learning module of his approach contains specification rules and treats training data as input, and the inferencing module draws consequences according to the pre-determined knowledge base of querying data. The main challenge with specification-based approaches is the articulation and conformity of pre-determined rules due to the wide range.

#### 2.4.2.3 Challenges with Static and Dynamic Analysis Techniques

Static analysis is a famous track for researchers and anti-malware specialists to examine malware behavior. The static analysis is helpful to pinpoint the specific vulnerability in the code and has a quick turnaround for bug resolution. However, static analysis methods fail to build effective defenses against code obfuscation and polymorphic malware creation techniques [134]. Signature-based static analysis techniques are unable to detect zero-day

and unknown malware. Static analysis methods are unable to capture vulnerability introduced at runtime. Malware authors use various obfuscation techniques to disrupt the static analysis by hiding, encrypting, and packaging malicious activities. Han *et al.* [63] presented an explainable malware detection method with the fusion of statically and dynamically extracted APIs. They have stated that the syntax of dynamic and static API may differ, but semantically they are similar. Moreover, various encryption techniques invaded malware detection systems, making it difficult to extract such features without decryption statically. Packing malicious content by adding multiple layers of compression and encryption is often used by malware attackers to hide malicious payloads. Anezi *et al.* [2] presented an approach for packed malware and commented that 80% of new malware are packed, and 50% of them are the simple repacking of legacy malware.

The dynamic analysis examines the malware during the execution of the program and monitors system calls, traces instructions, registry changes, read/writes operations, file systems, memory, network usability, and more. The dynamic analysis can be run in real, sandbox, or virtual environments. However, it requires more resources to perform the execution, is costly compared to static analysis, and generates more false-negative and false-positive results. Moreover, it takes more resolution and computation time to traceback the vulnerability to the exact location in the code. Dynamic analysis is ineffective for malicious content resides inside the non-execution part of code; it only works for the execution pieces. Various emulation techniques to detect sandbox environments discussed in [98] stated that malicious activities could successfully evade dynamic analysis. It has been noted that static and dynamic analyses are mostly viewed in opposition. However, numerous strengths and weaknesses are associated with both approaches.

## 2.5 Recent Advances and Challenges

The detection and isolation of malware are challenging due to variant nature, diversity, code obfuscation, and other sophisticated routes of malware creation. Establishing a robust defense line against malware apps for mobile and IoT devices is imperative. With the paramount growth of mobile apps, it is notably challenging to examine each application manually for malicious behavior. Traditional machine learning-based approaches use handcrafted features and need extensive source code analysis for signature detection.

Advances in AI (Artificial Intelligence) are the paradigm shift in the technology landscape and advancing enormously in various domains like computer vision, Natural language

processing, self-driving vehicles, data analytics, security diagnostic, e-health diagnosis and prognosis, and many more. For example, machine learning-based data-driven analytics has revolutionized healthcare, finance, and other business models. Detection, monitoring, and classification in the security domain extracting valuable information from text and images from enormous social media data and other means are possible due to these emerging technologies. The taxonomy of recent advances for malware detection is divided into machine learning-based and deep learning-based techniques and circumvent computer vision (CV) image-based approaches and Natural language processing (NLP) related text-based techniques.

### 2.5.1 Machine Learning-based Malware Detection

To protect computers, servers, and handheld devices, anti-virus software provided by anti-malware companies set up the first line of defense. Traditionally, signature and heuristic-based malware detection methods are often used, i.e., the short sequence of unique byte-code assigned to each known malware by a domain expert.

Over the last decade, the research community has incurred tremendous efforts for malware detection using machine learning and data mining techniques. The machine learning approaches for malware detection are further grouped into two main types *Supervised/Classification* techniques and *unsupervised/Clustering* techniques. These techniques are widely adopted for the signature detection of previously unseen malware and/or malware family. All the machine learning techniques heavily depend on feature extraction using a file's static, dynamic, or hybrid analysis. The first step of all ML-based approaches is to perform feature engineering to capture the characteristics of a malware sample file, for instance, API calls, permissions, intent, services, protected string, opcode sequences, etc. The second step is input representation, training the classifier for signature detection, and assigning the family label using the classification or clustering technique. The trained classifier is then deployed locally or on a cloud for the detection of an unknown sample. The basic accuracy metrics of a malware classifier are usually determined through True Positive, False Positive, True Negative, and the False Negative Rate. However, for clustering-based techniques, Macro and Micro F-1 measure is used to evaluate the system. In the subsequent subsections, we have further enlightened the details of these metrics.

#### 2.5.1.1 Supervised Learning for Malware Detection

*Supervised learning* is the mechanism in which samples are provided with correct labels to a statistical model in the preliminary phase called training. The model can correctly classify the unseen sample based on the distribution learned from training data. An example is a listing of one thousand android APKs (Android Application Packaging Kit) with their malware and benign signature. The supervised classification task is completed in two sequential steps; classifier construction and model deployment/usage. For the abovementioned example, how a supervised machine learning model will work?

In the first step, one thousand android APKs with correct malware and benign labels are available to the system. The system processes each APK file to extract the underlying characteristics, converts them into a feature vector, and assigns the provided label from the training set. The converted feature vector and class label of each APK file from the training set is used to train the classifier, for example, K-Nearest neighbor (KNN), Support Vector Machine (SVM), Artificial Neural Network (ANN), Decision Tree, Bayesian Belief Network, etc. In the second step of model usage, the constructed model is deployed as a part of an application, and an unknown APK file, either malware or benign, is exposed to the model. The system performs the feature extraction of a new sample (similar to the training set) and gives a verdict of malware or benign based on learning from the training set. Researchers have presented various supervised learning techniques for malware detection; a few commonly used ones are discussed below.

- **Decision Tree (DT)** : A decision tree is a conventional classifier used for categorical data. The Decision trees are built using if-then-else rule-based approach, presenting the application features or variables in a tree structure. A heuristic-based greedy mechanism is used to construct the tree, which involves the information gain and entropy score. The leafThe leaf node represents the class labels, while the root/decision node represents the features/variables, has two or more branches, and is calculated using information gain and entropy. The dataset is split into smaller subsets, and entropy measures the homogeneity/heterogeneity of the data using the following equations 2.1. If the number of samples in a dataset is equally distributed, then entropy is 1. If all the samples are homogeneous, then entropy is zero. For example, a dataset having ten samples and all benign, then entropy becomes zero; if five are benign and five are malware, then entropy is one.

$$Entropy = -\sum p(x) * \log_2 p(x) \tag{2.1}$$

where p(x) represents the number of examples in a given class. The decision trees are computationally less expensive and easy to understand as a complete path from the root node to the leaf node makes one rule and shows the decision. On the other hand, DTs are more prone to over-fitting, and noisy data leads to the wrong classification. The over-fitting issue can be solved by pruning the decision tree. Numerous studies are available in the literature to solve the malware detection issue but lack robustness and need heavy feature engineering.

- **Random Forest (RF)** Random forest is the advanced and improved version of the decision tree. More specifically, it is the collection of decision trees. It yields better results than the decision tree by choosing an independent dataset for each tree in the forest using random or bootstrap data sampling. These sampling techniques ensure that decision trees in the forest have diversity and are dissimilar. The same approach is used for constructing each tree, as discussed above for DT. The algorithm makes voting on each decision tree's result in the forest and performs the final classification. A few popular studies like [4, 175] used the random forest for android malware detection and yielded state-of-the-art results.

- **K-Nearest Neighbour (KNN)** KNN is the oldest and most simplistic algorithm used for classification and regression by measuring the similarity/distance metric between samples or variables. An unknown query instance classifies into the same class as the nearest neighbor belongs (winning class of its neighbor samples). K represents the size of the neighbors involved in the decision process for classification. Usually, K is an odd number, and typical values lie between 3 to 8. Different distance metrics are used to measure the similarity of variables of an unknown sample to a training set sample, for example, Manhattan distance, Hamming distance, and Minkowski distance. However, the most popular one is Euclidean distance. KNN is not an optimal choice for malware detection because, practically, malware datasets are highly imbalanced with many benign samples and fewer malware samples. Therefore, most neighbors belong to the dominant class, which may resultresult in a high False Positive and False Negative Rate.

- **Naive Bayes Classifier (NBC)** Naive Bayes is a supervised learning technique applicable to binary and multi-class classification problems using Bayes Theorem as given in equation 2.2. NBC evaluates the probability of each attribute independently and calculates the class probability and conditional probability to make the prediction. An advanced version of Naive Bayes is the Bayesian Belief Network, which shows the

graphical structure representing the dependencies of variables but not conditionally independent. Bayesian Belief Network comprises two components. The first is the directed acyclic graphs depicting all random variables as nodes, and the edges between nodes represent the probabilistic probabilities between these variables. The second is the conditional probability table for variables. Several studies are available in the literature using NBC and Bayesian Belief Network for static and dynamic malware detection using opcode sequences, API calls, and permission analysis.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.2}$$

- **Support Vector Machine (SVM)** Support Vector Machine is a machine learning-based classification method mostly adopted for binary classification problems, and the objective of this classifier is to search for a hyperplane that maximizes the distance between two given classes. The term "support Vector" refers to data points exist closer to the hyperplane, and the distance between the support vector and hyperplane is referred to as the margin. The objective function of SVM is to maximize these margins. SVM with multiple hyperplanes can be used for multi-class classification. SVM has been widely used in malware detection; the famous Drebin paper [13] uses SVM for Android malware detection after applying feature engineering to extract relevant features that contribute to benign and malware.

- **Artificial Neural Network (ANN)** Artificial Neural network, as the name depicts, is inspired by the human brain and mimics the functionality of the human biological nervous system. In layman's terms, Neural Networks are a collection of input and output neurons, each associated by weight except the final decision-making neuron. During the training phase, weights are randomly assigned initially, and then the network adjusts using backpropagation to predict the actual class correctly. There are various variants of Neural Networks for Malware detection using images, binary, and text data.

- **Deep Neural Network (DNN)** also known as Deep Learning (DL), is a focal concern for academia and industry for different applications, for instance, computer vision, Natural Language Processing (NLP), speech recognition, and cognitive learning, etc. DNN with multi-layer architecture allows superiority in feature learning compared to ANN and overcomes challenges like layer-wise pre-training, etc. DNN involves more than one hidden layer in contrast to simple ANN and is conducive to learning multiple abstractions, advanced feature extraction, and higher-level concepts. However,

DNNs are compute-intensive with GPU requirements and time-consuming to learn the underlying model. There are various variants of DNN which are extensively used for malware detection, such as Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), Deep Belief Networks (DBN), and Deep Auto-encoders.

#### 2.5.1.2 Unsupervised Learning for Malware Detection

In machine learning, all the classification algorithms require labeled inputs for predicting the correct outcome. However, semi-supervised learning does not require a fully labeled training dataset. The semi-supervised techniques are handy in scenarios where the scarcity of labeled data or partially labeled inputs are provided. The research community is actively working on automatic malware detection and categorization using unsupervised learning where no labels are available. Clustering is a well-known unsupervised learning technique for segregating the data into groups called clusters using different predefined distance measures like Euclidean distance, Manhattan distance, etc. The clustering aims to maximize the inter-cluster distance while minimizing the intra-cluster distance. K-mean and Hierarchical clustering are famous techniques used for unsupervised learning. Hierarchical clustering is suitable for malware datasets that exhibit irregular cluster structures, while K-means clustering is more effective in globular representation.

In malware detection, different malware families represent different clusters. A file sample with the same traits and features can be assigned to a respective cluster using the clustering technique by measuring the distance. Similarly, clustering can be used in malware detection in both static and dynamic analysis. In dynamic analysis, the grouping is based on profile behavior, runtime execution, and API calling structures. In static analysis, Malware can be identified and grouped using the same permissions, features, and similarity analysis. Researchers have presented various unsupervised approaches for automated malware detection using structural information (function call graph analysis). After the pre-processing, the fine-grained feature extraction step is performed using call graphs that determine the file sample belongs to which malware cluster by applying a discriminate distance metric to marginalize the inter-cluster distance. The choice of clustering technique in malware detection depends on the dataset, extracted features, platform OS, Windows or Android, and their underlying feature distribution. The malware expansion is not limited to a single platform but hits all platforms with the same force. The research community must set up an apt combat mechanism across all platforms.

#### 2.5.1.3 Cross Platform Malware Detection

Malware is a kind of software developed to harm the target system deliberately. Due to recent internet advancements, the exponential growth of malware is targetting every platform, including Linux, macOS, Windows, Android, etc. The concept of malware has existed since the inception of computing in 1949, and numerous malware attacks have posed severe threats to legitimate users. Evolving expansion of malware is not limited to one platform. According to the report of the AV-Test Institute [14], each month, half a million android malware is created. Similarly, the number is near one million for Windows, and 350,000 malware are created daily for different platforms. With the emergence and popularity of IoT devices, malware attackers have broader target markets to penetrate. Malware is generally not only categorized into various families but also platform-specific and may have various variants of one family. Malware can infiltrate into the system via multiple routes (a) vulnerable services over the network allow automatic entry (b) Download from the internet, and studies depicted that about 80% of the malware came from famous websites (c) malware authors allure the individuals to execute the malicious code on their machines, for instance, asking them to install a codec to watch a movie or a tv program online or clicking the adds. It has been noticed that the malware author spy on the target system and collects sensitive information, affects the system's performance, and creates overload processes.

It is a universal truth that non of the platform is 100% entirely bulletproof against malware and spyware. There are different pros and cons associated with each platform regardless of how well engineered an operating system is. Due to the never-ending race of defenders and attackers, there is always room for malware injection. Studies have revealed that the Windows platform is more prone to malware attacks. The vulnerability is not due to any inherent flaw of the operating system but due to its high market share and sheer size of users. Hackers and malware authors create numerous attacks against the window compared to Linux or macOS due to low effort and high return ratio. Various studies and research community targets mostly Windows (PE) for malware detection and present unsurpassed approaches. Linux is less prone to malware due to its low market share of 3%. Similarly, macOS and iOS also have less market share than Windows and Android. Therefore the attack pool is marginally less to compromise and infiltrate. Moreover, macOS has apple system integration protection (SIP) which does not allow access and modify the core files and processes. On the other hand, Android is open source, capturing the 85% smartphone market, and is a hive for malware authors. In addition to the Google Play store, various third-party app stores allow the end-user to install different applications. This thesis focuses on Android OS in the

context of malware detection; that is why the subsequent chapters are narrowing the funnel around Android malware detection.

# Chapter 3

# Android Malware Detection

The smartphone is the biggest revolution of the 21st century and is proven a game-changer in everyday life, including social interaction, finance management, entertainment, education, and many more. Smartphones with more than 4 billion apps, including iOS and Android, transform human interaction. Android occupying the largest market share of 85% and an open-source operating system, is the most vulnerable to hackers and malware authors. According to the AV-Test antivirus institute, half a million android malware is crafted each month to invade unsuspecting users. The expansion of malware posed higher security threats, for instance, stealing credentials, leaking sensitive information, financial transactions, calling premium numbers, sending SMS messages, etc., and using their private data for other means without individual knowledge and permission. Therefore, detecting android malware is of utmost concern for both the antivirus industry and researchers to protect legitimate users from these threats. To understand the preliminaries and basic structure of the android app hierarchy, let us deep dive into the Android platform architecture and structure of an Android application called APK (Android application packaging Kit). Contrary to desktop-based PE files, an android application is packaged and wrapped in APK zip format for distribution and installation of an android app on a smartphone. Java is the language choice for all android applications and is compiled into Dalvik bytecode wrapped in a .dex file. The main structure of APK includes a manifest file, a dex file, assets, and resources.

## 3.1 Android Platform Architecture

Android is developed as an open-source, Linux-based software stack created by Google and promoted by Open Handset Alliance (OHA) under Android Open Source Project (AOSP). It consists of developers, 34 carriers partners, chip-makers, and Original Equipment Manufacturers (OEMs). Android applications are written Java, and code is transformed into Dalvik bytecode using runtime Java Virtual Machine (JVM) called Dalvik Virtual Machine (DVM, however, native code and shared libraries are often written in C/C++. Figure 3.1 from the Android developer platform [9] depicts the architecture and major components of the Android architecture. The android platform is based on the Linux kernel due to enormous benefits like a robust and flexible driver model, process and memory management, efficient networking support, and many more core services to accommodate many devices. Typically, Android supports two main Instruction Set Architectures, i.e., ARM and x86.

- ARM: It is specific for smartphones and tablets kind of devices, ARM supports 32-bit, and ARM 64 supports the 64-bit operating system. ARM follows the RISC (Reduced Instruction Set Computer) architecture common to modern smartphone and tablet devices to enable powerful computing with optimized battery consumption. Nowadays, most Android devices use this ARM 64 architecture.

- X86: is common for mobile internet devices powered by AMD and Intel. X86 processor uses CISC (Complex Instruction Set Computer) and is specifically designed for speed. On the flip side, it consumes more battery power which is the biggest drawback of its adoption on mobile devices.

The key components of the Android architecture platform depicted in Figure 3.1 are enumerated as follows:

- **Linux Kernel:** The foundation of the Android platform is based on the Linux kernel tailored for mobile devices to manage the core modules of the Android platform, such as memory and process management, security, networking, and many others. Linux offers proven features in terms of security, process management, and flexibility to device manufacturers to write hardware drivers. The Android Runtime (ART) in Android OS depends on Linux to facilitate threading, multi-tasking, and low-level memory management.

- **Hardware Abstraction Layer (HAL):** In computer programming, HAL provides an interface to interact between the operating system and hardware device at an abstract

or general level. HAL can be pinged either from the Linux kernel or the device driver in Andriod OS, similar to other operating systems. In Android, HAL exposes hardware devices to a high-level Java API framework through standard interfaces, which enables the Android operating system to be agnostic to low-level driver implementation. HAL offers flexibility, cohesion, and modularity to implement desired functionality without affecting the higher-level system. The implementation of HAL is packaged in multiple library modules. Each module has a specific interface implemented for a particular type of hardware component and loaded at an appropriate time.

- **Android Runtime (ART):** Android Runtime ART and its predecessor Dalvik were created purely for Android projects to provide a layer that can support and ensure portability across various devices. A few examples of core libraries in ART are independent of Java implementation, such as Apache Harmony and Bouncy castle for cryptography. ART, as a runtime, can run virtual machines on low-memory devices to execute Dalvik executable format and dex bytecode. Android version 5.0 onwards, each android application runs in its process and has a dedicated instance of ART. Before 5.0, Dalvik was used to handle runtime; therefore, an application that runs smoothly on ART should run on Dalvik, but vice versa may not be true.

- **Native Libraries:** Native libraries are the essential module of the Android architecture platform and enable the implementation of various services and app components such as HAL and ART. These services are developed in native code and need libraries written in C and C++. Android framework APIs expose the low-level functionalities and the intensive computational power of native libraries to be used in apps. The Android NDK (Native Development Kit) provides C and C++ code tools in android apps and native platform libraries to access and manage physical components such as touch input, camera, Bluetooth, and other sensors.

- **API Framework:** Java API framework is available to application developers on top of native libraries and Android Runtime. This Android API framework provides higher-level services to application developers in the form of Java classes. The framework contains various java classes and interfaces that act as building blocks to develop android applications, such as developers using Android APIs like telephony, resources, locations, UI (user interface), and many more to build desired apps. The key services provided by the Android API framework are listed below:

  - **Content Provider -** Enable the app to share data with other applications, publish data, and access data from other apps.
  - **Notification Manager -** Provide capabilities to show custom notifications and alerts to users in the status bar.
  - **Activity Manager -** It manages the life cycle of the app, responsible for dispatching message events, optimizes memory management, and controls the navigation/activity stack.
  - **Resource Manager -** Allows access to non-code resources embedded in the app. It enables to import, creation, and manage UI resources in an app, for instance, interface layout, strings, graphics, colors, etc.
  - **View System -** Contains an exhaustive set of views used to create the application's UI.

- **System Apps:** The application layer is the user interactive layer in the android architecture. This layer contains pre-installed default applications for end-users like a camera, contacts, messages, gallery, etc. The third-party applications installed by the user from the google play store also reside here. The users cannot delete the pre-installed application because the uninstallation of default apps may disrupt the android functionality. Every installed application on Android OS run an independent memory in Linux Kernel with a Linux user ID. However, it is possible to configure Linux user ID shared among multiple apps.

## 3.2 Android Application Structure & Representation

APK file is considered an Android binary. APK stands for Android Application Packaging Kit or sometimes Android Package Kit. It encompasses all the ammunition that needs to install correctly on any Android device. An APK file is usually a zip folder to install and distribute Android applications. The structure of an APK is presented in Figure 3.2 with a brief description of each directory. Our focus in this thesis oscillates around manifest.xml and classes.dex file and discussed in detail in subsequent sub-sections.

The zip APK folder may contain many nested folders and files, but the conventional structure of an APK is listed as follow:

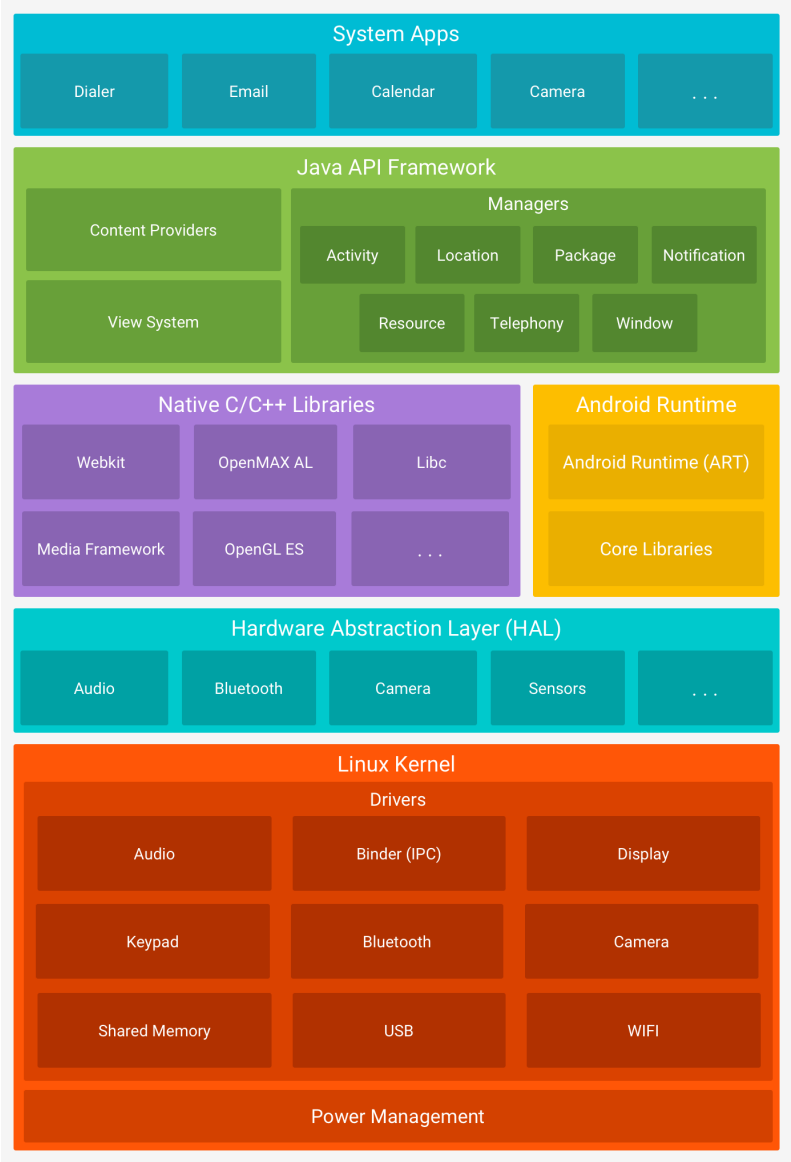



Fig. 3.1 Image credit to Android Developers Platform [9]: Architecture of Android Platform

- META-INF/: The folder contains the manifest file having verification information about the Java jar file. The folder verifies every file in APK on signing the application. Any minor alteration to the APK requires resigning the APK; otherwise, the operating system will dissolve the installation.

- lib/: This folder contains native libraries and platform-dependent compiled code (similar to DLL in windows), which compiles the code in .so format from NDK (Android Native Kit). As discussed in section 3.1 Android is cross-platform and supports multiple processor architectures like ARM, ARM64, and X86. This folder

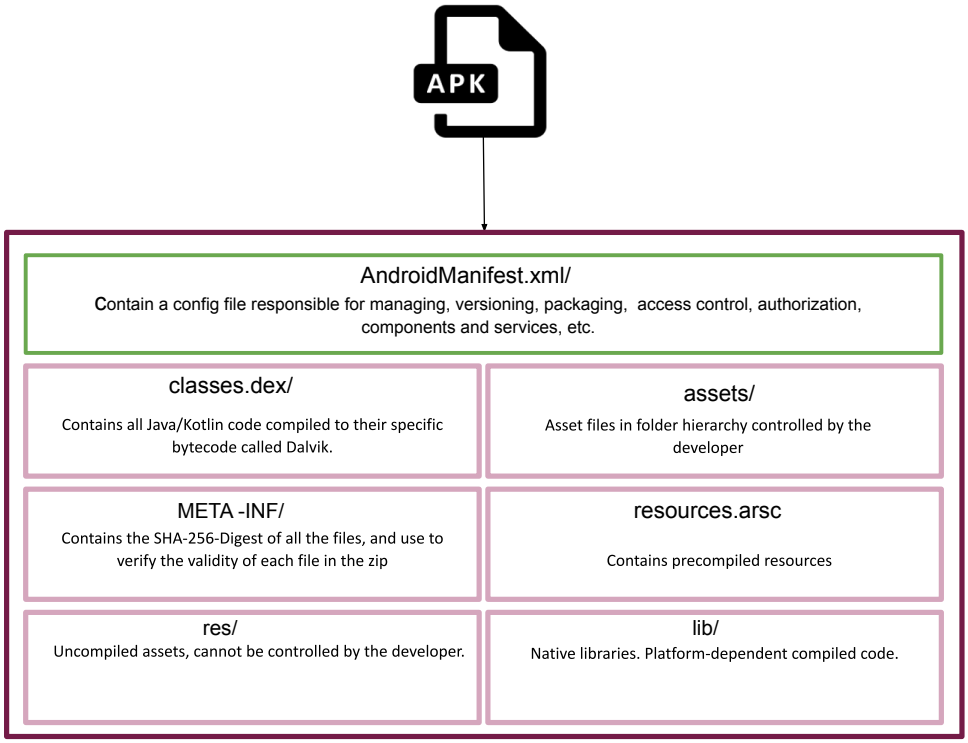



Fig. 3.2 Structure of Android APK File.

comprises subfolders for each supported processor, like 'armeabi-v7a, arm64-v8a, x86, and x86_64. The absence of a subfolder indicated that support is unavailable for the missing processor architecture.

- res/: This folder contains uncompiled resources (similar to assets) with a predefined hierarchy and cannot be changed by developers. These resources are alternate options for multi-lingual support, various screen orientations, and different OS versions.
- assets/: The graphical resources usually displayed in applications like icons, visuals, videos, document templates, etc., are stored in this folder. These resources can be modified and controlled by a developer.
- resources.arsc/: This folder contains precompiled resources and information that act as a bridge between code(classes.dex) and resources (res). An example of precompiled resources is binary XML, a particular format used in Android OS to save XML, and resources contain strings, colors, styles, etc.
- AndroidManifest.xml: contain a configuration file responsible for managing the security and versioning control, including access, authorization, services, etc.

- classes.dex: similar to exe file, a wrapper containing the java code compiled into a Dalvik bytecode.

The main components of the Android APK structure pertinent to this research work are further explained in the following subsections.

### 3.2.1 Manifest.XML File

Android OS uses the component-based framework for mobile application development, deployment, installation, and maintenance. Manifest.xml file act as a configuration file for the android application that manages the access control, authorization, versioning control, permissions, and app components. App components have four main modules, such as *Activities, Providers, Receivers, and Services*. The detailed purpose of these app components declared in the manifest file are listed below:

- **Activities** enable Graphical User Interface (GUI) interaction while services are responsible for background communication and processing of inter and intra-app components.
- **Providers** act as a database management system for the app. It is a subclass of *ContentProvider* that grants access to structured data. The provider uses its UI to ingest and work with this data.
- **Receivers** are background processes liable to respond to messages broadcast by the system or by other applications.
- **Service** are declared as one of the components of an application in the manifest file responsible for background processing. It usually enables long-running operations, for instance, continuously fetching the data without interrupting the user's interaction and activity.
- **Filter Intents** Filter intents governs the action of each component. It specifies the intent type to which content provider, broadcast receiver, activity, and service can listen and respond.
- **Permissions** are the core part of the Android Manifest.xml file manages access control. This part contains custom permissions and various protection levels. It contains permissions needed to access the protected modules of the operating system or other apps. The permissions used to access content from this app are also declared here. Following are a few commonly used permissions levels/types in the manifest file.

- Normal: permissions are considered zero risk or very minimal to the user's privacy. The system automatically grants such permissions at installation time when an app needs to access resources or data outside its sandbox. A few examples of Normal permissions are CHANGE_NETWORK_STATE, SET_CLOCK, SET_ALARM, etc.
- Signature: Permissions are also granted at installation time to only those applications available in Android image or the asking app signed with the same certificate as the app that declared permissions. Some common examples of signature permissions are READ_VOICEMAIL, WRITE_VOICEMAIL, BIND_TV_INPUT, etc.
- Dangerous: permissions that involve accessing users' data and private information or potentially impacting users' privacy are considered dangerous. The app must prompt and ask user consensus to use private data/resources, for example, access to a contact list, locations, etc. Examples of dangerous permissions are SEND_SMS, READ_CONTACTS, READ_CALENDAR, etc.

The android app declares all its components in the android manfiest.xml file to manage the structural information. Before initiating any component, the system reads the manifest file and checks the app components, permissions, and filter intents. Furthermore, the actions of each component are governed by filtered intents that specify the type of intents an activity, provider, services, or receivers can respond to this intent. For instance, an activity can initiate a phone call via filtered intent, or the receiver can monitor SMS. The manifest file governs the application functionality and acts as a road map of the android application. The manifest also enumerates all the permissions requested by the app to perform specific functionalities (e.g., wifi access, internet access, SMS send permission, etc.). Hence, permissions, intents, and app components handle the interaction between app to app and with OS. A sample snippet of the Android Manifest file representing app components can be seen in Figure 3.3. On the route of static, dynamic, or hybrid analysis, learning-based android malware detection techniques extract these ingredients from the manifest and then train a learning algorithm for malware classification [160, 47, 114, 29, 77, 168, 82]. Different studies follow their respective methodologies and use all app components and permission or selective app components and permissions for malware detection. Fewer studies only relying on permissions with different accuracy metrics and reasoning [94, 29, 153, 77, 47, 124, 93].

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="payspace.ssidit.pp.ua.payspacemagazine">

    <uses-permission android:title="android.permission.INTERNET" />
    <uses-permission android:title="android.permission.ACCESS_NETWORK_STATE" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@android:style/Theme.Holo.Light">
        <activity
            android:label="@string/app_name"
            android:title=".MainActivity">
            <intent-filter>
                <action android:title="android.intent.action.MAIN" />

                <category android:title="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:label="@string/title_activity_settings"
            android:title=".SettingsActivity"></activity>
    </application>

</manifest>
```

Fig. 3.3 Snippet of Android Manifest.XML File

### 3.2.2 Classes.Dex File

Dex is an executable file in Android containing the Dalvik bytecode compiled from Java source code and interpreted by a Dalvik Virtual Machine. Android applications are always written in java and compiled into Dalvik bytecode. Dex file is the file format that contains the compiled code, including all the user-implemented classes, functions, and objects. Dex file also contains different API calls used by the android app to access the different operating system activities, creating new instances and resources. The codebase of an android application resides in the Dex file that is ultimately executed by ART (Android Runtime). Each APK contains single or multiple classes.dex file comprised methods and classes called in the app. Dex file resembles java class file and compiled to native code on the device at installation time. In older versions of Android, it was run under DVM (Dalvik Virtual Machine), and in newer versions with ART. In short, API calls, new instances, and different opcode sequences for a dex file can be used to determine the behavior of an android app. It is imperative to mention that the dex file is unreadable without any reverse engineering tool. Numerous studies used reverse engineering tools to decompile the dex file into a smali file, then parse it to extract features like opcodes sequences, system calls graph, API calls, etc., for malware detection. Smali is the intermediate between java and Dalvik bytecode. The details of opcodes and smali file format are enumerated below.

- **Opcodes:** The opcode is the terminology abbreviated for operation codes used in computer science. It is a part of machine language or assembly instructions that determines the operation to be executed through the opcode table. Apart from opcode itself, it has one or more specifiers to execute the instruction set called operands and usually data on which operation performs.

- **Smali File:** Smali is a file format with an extension of .smali used in Android Assembly Language Format, developed by Google. The developer code written in Jave compiled in .Dex file can be inspected by decompiling the .Dex file into Smali format using Smali/Baksmali, acting as assembler/disassembler. The format is used for low-level inspection of Android app content. The content in Smali is similar to Java source code but not the exact source or reconstruction of original Java code, but human-readable. Using smali code, a developer can recompile and repack APK, commonly known as reverse engineering. It is imperative to mention that decompiling a .Dex file into Java is more readable than Smali but most likely irreversible. In the case of smali, it is harder to read but flexible to make changes, modifications, and repackage an Android App.

The complete APK building process that includes all the ingredients of the APK structure as discussed above is depicted in Figure 3.4.

### 3.2.3 A Short Review on Reverse Engineering Tools for APK Analysis

As mentioned above, security specialists and researchers cannot directly read the .dex file. Therefore, for detailed assessment, they need a reverse engineering tool to convert the .dex file into a smali file format (the intermediate between java and Dalvik bytecode). Following are some of the favorite reverse engineering tools used by Android security experts and engineers to analyze the APK file.

#### 3.2.3.1 Tools for On-device Analysis

There are a couple of reverse engineering tools offering on-device conversion for lightweight assessment and analysis. However, they are not very useful and successful due to limited resources. A few well-known on-device APK analysis tools are enumerated below:

- ***apktool:*** APKtool is the most commonly used tool [11] for decoding the content of APK into a project-shape directory structure, containing almost near to the original

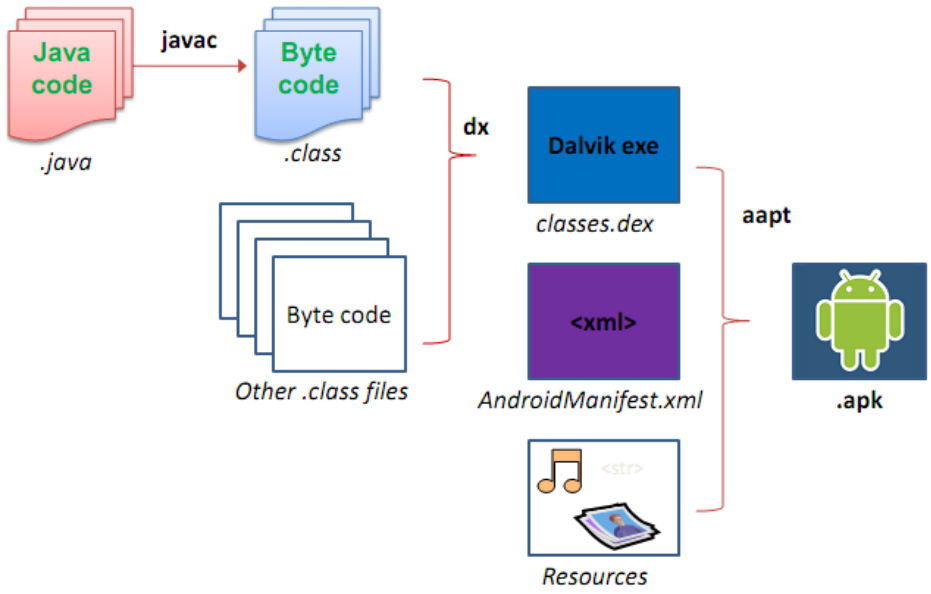



Fig. 3.4 Android APK Build Process [Image credit [10]]

java code called smali format. The syntax of the smali file format is easier to read, analyze, manipulate, modify, enable a few repetitive tasks, and is easy to repackage into APK/JAR. This tool intends not to use for illegitimate uses like piracy but to modify features, enable support for custom platforms, and thoroughly analyze the application security.

- ***dex2jar***: dex2jar is a reverse engineering tool that provides a lightweight API to decompile the dex file into a readable java source. Alongside parsing, dex2jar is handy to optimize the dex file and also capable of assembling back the jar/smali file into the dex file.

- ***Dare***: Octeau *et al.* [109] shown that the Dare is a more appropriate choice than dex2jar for retargeting Dalvik bytecode by leveraging its strongly typed inference algorithm and intermediate representation. Dare potentially offers various techniques to analyze classes.dex file was initially developed for Java applications. Another

feature of Dare is to verify the Java bytecode by providing flexibility to rewrite the unverifiable input bytecode.

- **Dedexer**: Dedexer is another on-device reverse engineering tool used to decompile .dex files into an assembly format similar to Jasmine syntax. It generates the project directory containing a separate file for each class. However, it cannot repackage and reassemble like other tools discussed above.

- ***JEB:*** JEB [73] is a popular reverse engineering platform available in professional and community versions for precise and robust decompilation of Android apps and Dalvik bytecode. It offers debugging, analysis, and decompiling services to support multiple processor architectures and cross-platforms like Windows, Linux, and Mac. It is easy to use and navigate through an interactive GUI for analyzing the inner content of Android APK. It provides functionalities like refactoring, obfuscated code organization, and analysis output modification. It offers Java and Python plugins to automate reverse engineering tasks and enable customization. It can also be accessed through JEB API for automation.

#### 3.2.3.2 Androguard

Androguard [7] is an open-source reverse engineering tool used to analyze the contents of an Android APK in static analysis mode. Androguard generated control flow graphs of classes and methods and provided a Python API to access disassembled resources like fundamental building blocks in static structure, generated control flow graphs, and various other instructions. Androguard returns a similarity index for two similar and related apps to detect repackaging using NCD (Normalized Compression Distance) algorithm. Androguard mainly returns five methods to show the similarity between two apps if the second app is the repackaged one. To reveal comparison, it depicts *NEW, SKIPPED, IDENTICAL, and SIMILAR* method. On the flip side, it also returns the differentiating aspects between the two methods by analyzing the basic building blocks. In this research work, we primarily use Androguard to analyze the contents of APK. The details of its usability and integration are available online [7].

The brief working summary of Androguard used in this research work is given as under: When using a python script, we pass a path of an Android APK it returns three objects $a, d, dx$ according to the documentation of Androguard [120]. $a$ points to an APK object that contains the information about the APK file, mainly dealing with the AndroidManifest.xml

file and resources, etc., such as package name, version details, permissions, app components, and activities. *d* returns an array of DalvikVMFormat that corresponds to the dex file and helps to excavate information about methods, classes, and strings to analyze the dex file. *dx* is the analysis object used to handle multiple dex files in an APK. It is an optimal choice due to its ability to link information from multiple dex files with the help of particular classes.

#### 3.2.3.3 Andromaly

Shabtai *et al.* [129] presented a lightweight anomaly detection system using machine learning on the track of dynamic analysis. The system operates in hybrid mode, i.e., on-device and cloud, and collects real-time metrics such as Network traffic data, battery usage, active processes, CPU consumptions, etc. The proposed system comprises four modules: Feature Extractor, Processor, Main Sever, and GUI. The *feature extraction* module contains a feature manager that collects various metrics by periodically sending a trigger request to Android Kernel and the application framework. It also performs some raw and initial preprocessing on collected features. *Processor* is responsible for analysis and detection. It analyzes the input feature vector from Main Service and performs threat analysis using TWU (Threat Weighting UNIT) based on ensemble learning models. The processor can be built on top of rule-based learning, knowledge-based classifier, or machine learning-based anomaly detection methods. *Main Service* manipulates the feature extraction module, anomaly/malware detection, and alert management part. It mainly acts as a bridge, gathers newly extracted features, and sends them over to the processor for final recommendation. It has various components such as loggers responsible for logging the information pertinent to debugging, calibration, etc., and configuration files responsible for application configuration and management, for instance, defining sampling intervals, specifying a threshold for alerts, and active processors. The operation manager is responsible for switching modes resulting in the activation/deactivation of specific processors. Alert manager filtering the false alarms predicted through a processor. *GUI* enables users to configure different settings, alerts, and parameters.

#### 3.2.3.4 Andrubis

Andrubis [154] is a web-based platform developed on top of existing malware analysis tools such as APKtool, Androguard, Droidbox [90], and TaintDroid [48]. Users can submit an APK through a web interface to Andrubis, which returns the verdict of malicious or benign by performing static and dynamic analysis. Andrubis operates on two-level the java code

executed by Dalvik VM and native code at the system level and offers various stimulation techniques and post-processing capabilities.

#### 3.2.3.5 Bouncer

Google offers a dynamic analysis platform based on a virtual machine to protect its Google Play Store. Bouncer checks for malicious applications and content from third-party developers before uploading an app to the google play store. Researchers have shown that bouncer protects the play store from the malicious app, but still, malware authors can significantly evade bouncer, and malicious apps can enter into the user's smartphone.

#### 3.2.3.6 TaintDroid

William *et al.* [48] presented a system called TaintDroid by extending the Android analysis platform to track the disclosure and leakage of privacy and sensitive information within third-party applications. The system provides an automated way of labeling and tracing private data leaving an app and the destination and logging this tracking information. TaintDroid tracks the privacy-relevant information at four levels, including the variable level, method level, message level, and file level.

### 3.2.4 Feature Representation

The research community has used different approaches to represent the android application. Most studies use a reverse engineering tool to extract the features from the manifest and dex files and represent the android app as a binary vector space. After the preprocessing and feature representation, different machine learning and deep learning-based algorithms are trained to detect legacy and new malware. The feature extraction and representation for android application is illustrated in Table 3.1

A more generic way to express the feature vector 1,..., M, including all permission, API calls, system calls, and intents, is to represent in a binary vector $X\varepsilon\{0,1\}^M$, where $X_i$ represents whether the application has particular feature i, i.e., $X_i$ =1, or not,i.e., $X_i = 0$. Due to different functionalities and purposes, numerous applications are available in the market, and pertinent to these functionalities, $M$ can be massive. In contrast, one application might

contain a few features, resulting in a very sparse feature vector. An Android app can be represented as a binary vector as depicted in equation 3.1 in the illustration to Table 3.1.

$$x_i = \begin{pmatrix} 1 \\ 0 \\ \dots \\ 0 \\ 1 \\ \dots \\ 1 \\ 1 \\ \dots \\ 1 \\ 0 \\ \dots \end{pmatrix} \begin{cases} \text{ACCESS\_NETWORK\_STATE} & \\ \text{CALL\_PHONE} & \text{Permissions} \\ \dots\dots\dots & \\ \text{Splash\_View} & \\ \text{lib.activity.SmsPatternForm} & \text{Activities} \\ \dots\dots\dots & \\ \text{intent.action.GTALK\_CONNECTED} & \\ \text{BROADCAST\_CHANNEL} & \text{Filtered Intents} \\ \dots\dots\dots & \\ \text{Landroid/telephony/TelephonyManager:getDeviceId} & \text{API Calls} \\ \text{Landroid/app/AcitivityManager:killBackgroudProcess} & \\ \dots\dots\dots & \end{cases} \tag{3.1}$$

| File Type | Feature | Sample |
|---|---|---|
| Manifest | Permissions | android.permission.ACCESS_NETWORK_STATE<br>android.permission.CALL_PHONE |
| | Activities | Activity_Splash_View<br>com.haunsoft.moneyapp_lib.activity.SmsPatternForm |
| | Services | com.android.providers.update.OperateService<br>com.android.providers.sms.SMSService |
| | Filtered Intents | android.intent.action.GTALK_CONNECTED<br>android.intent.category.BROADCAST_CHANNEL |
| Dex | API Call | Landroid/telephony/TelephonyManager:getDeviceId<br>Landroid/app/AcitivityManager:killBackgroudProcess |
| | Opcode | invoke-super(parameter),method_to_call |

Table 3.1 Illustration of Feature Representation of an Android App

After the binary vector representation of an android application, different machine learning techniques are applied to train the malware detection system. Besides binary representation, numerous studies are presented representing the app in a two-dimension array, pixel values (images) and word sequences, etc. Then on top of these representations, learning-based classifiers are trained to detect the malware and classify the corresponding

malware's family. Our focus is on image-based APK representation, and the next chapter elaborates on these details.

## 3.3 Static & Dynamic Analysis of Android Malware Detection

The basic details of dynamic and static analysis in general malware detection are already discussed in section 2.4. This section has narrowed down the analysis funnel around Android malware detection. Static analysis-based approaches perform code reviews and code-smells in the source code by disassembling the APK file without execution. The closest analogy to static analysis is the in-depth code review by an expert but in an automated way. The static analysis-based approaches badly failed against code obfuscation techniques like string encryption, repackaging, changing classes and method names, altering control flow, reordering opcodes, inserting junk code, etc. Dynamic analysis is an apt choice to analyze malware and overcome code obfuscation challenges in Android applications. The dynamic analysis continuously checks for malicious content at execution time by running in a protected environment. The app under consideration is emulated with all the required resources and inputs to examine malicious activities. Therefore, the effectiveness of dynamic analysis is heavily dependent on the quality and coverage of inputs. Moreover, code compilation and build time errors can not be determined in such kinds of analysis, and these analyses are slower than static analyses.

### 3.3.1 Static Approaches for Android Malware Detection

Numerous approaches have used on-device static analysis tools to examine malicious content. Feng *et al.* [52] used a static analysis approach to extract semantic and syntactic features using a signature-based malware detection system. However, signature-driven techniques are ineffective against new, unknown and polymorphic malware. Faruki *et al.* [51] presented a prototype named "AndroSimilar" to make the signature-based detection more effective and robust. The author claimed that the presented system is robust against code obfuscation and repackaging. It can defend against unknown and new malware with more tolerance for malware evasion. The system is a syntactic footprint that searches for statistically similar regions between known and unknown malware variants. Various researchers have proposed component-based analysis to analyze android APK for security assessment as an

alternative to signature detection techniques. The proposed methodologies excavate critical information and content from the manifest file, bytecode, and app components, such as Activities, Services, Receivers, filter intents, permissions, etc. They analyze the behavior of extracted content and its interaction with bytecode [33, 54, 97, 94].

Researchers have also proposed various approaches to statically analyze the APK content using permissions as a feature from the manifest file [128, 16, 153, 77]. Android OS has introduced a permissions module in the manifest file to protect users' privacy and sensitive data from malware authors. However, permissions are not solely enough to decide between malicious and benign apps. Sanz *et al.* [126] used permission as a feature vector and applied traditional machine learning algorithms like the random forest and Naive Bayes to distinguish between benign and malicious Android apps. Huang *et al.* [67] also used requested permissions and trained a classifier to predict malicious apps. Wang *et al.* [153] used multilevel permissions to detect malware. They have trained the machine learning model on nearly 10 thousand samples with 5K benign and 5K malware samples. Arora *et al.* [12] presented an approach called permpair(abbreviated for permission pairs) that searches for a dangerous combination of permission pairs. The proposed system constructs and compares a graph of malware and benign samples for malware detection using specific patterns of extracted permissions. Sahin*et al.* [123] proposed an attribute-based feature selection method to choose significant permission and used a machine learning classifier for the android malware detection system. The system used a linear regression model to eliminate unnecessary and less important features.

The dex file decompiled using a reverse engineering tool to read and perform security assessment contains methods, classes, and instructions. The Dalvik bytecode contains semantically rich information regarding the behavior of the app. Researchers have performed detailed analysis by converting this semantically rich information into control flow, and data flow graphs and then identifying potentially dangerous functionalities like data leakage, violation of privacy, and misuse of phone resources [60, 81, 174]. Wenbo Yeng *et al.* [167] proposed an AppSpear system for unpacking the encrypted malware using bytecode decryption and reassembling of dex file. The author showed that the malware author heavily relies on anti-malware analysis techniques to inject packed malware. The proposed system performs automated unpacking and rescues protected bytecode without the packer's knowledge. Various other studies have used control flow graphs and data flow graph methods to track the constant arguments in API calls, for example, sending premium messages to hard-coded

phone numbers. API calls from bytecode have been used in such studies to monitor privacy leakage, repackaged contents, and detect malicious activities [173, 81, 1, 76, 80, 115].

### 3.3.2 Pros and Cons of Static Analysis

Static analysis poses a minimal cost because it catches the bug at the early stages of development and is easy to fix. The static analysis can pinpoint the location of vulnerability, bug, or other issues without execution and stop the malware from entering a system. It is a proactive, fast, cost-efficient, and thorough approach that offers full code coverage. Unlike dynamic analysis, it does not rely on input data, and the security expert does not need to create a diverse and full-blown input dataset. The static analysis fails against code obfuscation techniques, metamorphic, and polymorphic malware. The static analysis is not robust to capture memory leaks because it needs execution. Code obfuscation is the process of modifying the executable file in a way that makes it complex, unreadable to humans, and non-usable to a hacker but has zero impact on the output and functionality of the application. Polymorphic and metamorphic malware techniques make the malware/virus challenging to catch and detect. Polymorphic malware uses a variable encryption key to appear the copy of malware/viruses completely different. Metamorphic techniques rewrite the code in an automated way instead of using the variable encryption key for the said purpose.

### 3.3.3 Dynamic Approaches for Android Malware Detection

As mentioned above, the static approaches are cost-efficient and quick to analyze the program but fail against code obfuscation techniques. Dynamic analysis is the right choice to overcome such scenarios of code obfuscation, polymorphic, and metamorphic malware. Researchers have proposed profile-based anomaly detection approaches for identifying malicious content [129, 121]. They have shown that malware authors launch DoS (Denial of Service) attacks and overly consumed hardware resources. Shabtai*et al.* [129] and [121] collected different parameters at various levels of Android OS like CPU, memory utilization, network traffic, consumption of battery power, etc. in malware and benign apps. On top of this collected data, they have trained a machine learning model for anomaly detection. Few other approaches employed dynamic analysis to monitor privacy, and sensitive data leakage like calling or sending messages to premium numbers without user consent [48, 22, 153]

Shijo*et al.* [134] presented an approach that combines static and dynamic features and represents an Android APK as an integrated feature vector to classify unknown executable files. On the track of dynamic analysis, the proposed system extracts the sequence of API calls and trains SVM and Random forest algorithms for malware detection. Wong*et al.* [157] presented an approach named IntelliDroid that constructs the set of inputs conducive to dynamic analysis tools like TaintDroid, and the injected input must not hamper system functionality. The system estimates specific APIs in an APK as malicious behaviors and triggers relevant inputs to monitor targeted API calls. They show that IntelliDroid is cheap and fast and can avoid the running cost of approximately 95% of an application during dynamic analysis while still detecting all malicious behaviors.

Yan*et al.* [165] highlighted that the emulator (VM) used to monitor application behavior itself is suspectable of malicious attacks. They have introduced the VM introspection technique to examine the behavior of APK externally. Lorenzo*et al.* [44] presented a dynamic analysis called VizMal to visually analyze the execution traces of an Android application to detect malicious behaviors. The system localizes the malicious content during the application execution at each second by displaying visual results. The execution trace of an Android APK is represented as a row of colored boxes. However, each box shows the one-second execution time, while the shape and the color of the box indicate the malicious activity at each second. They have trained a neural network called LSTM (Long Short-Term Memory), a variant of RNN (Recurrent Neural Network), to label every single interval of execution.

### 3.3.4 Pros and Cons of Dynamic Analysis

Complete access to source code is not required in the case of dynamic analysis. However, it relies on code instrumentation, i.e., only addition of specific code fragments are added to application execution to monitor errors. Dynamic analysis is good at handling code obfuscation, catching memory leaks, and producing fewer false positives. As complete code coverage is not 100% guaranteed, therefore, completeness of security assessment and zero errors cannot be guaranteed. Dynamic analysis performs pathetically on logic errors; for example, always True condition is not a bug in dynamic analysis. Dynamic analysis cannot precisely point out the vulnerability or error in the program. It is time-consuming, expensive, requires more resources than static analysis, and is heavily dependent on input data. The severe complication of dynamic analysis is missing some malicious execution

paths due to input dependencies. Moreover, anti-emulation techniques and detecting sandbox environments can evade dynamic analysis.

## 3.4 Learning-based Android Malware Detection & Limitations

With the proliferation of Android devices, many malware apps have emerged as a real threat to the end-users. Numerous machine learning approaches have been designed to combat, detect and isolate such applications. Since the last decade, ML-based frameworks have achieved unsurpassed versatility to defend various malware by leveraging different feature representations, including Application Programming Interface (API), permissions, App components, dynamic behaviors, system calls graph, etc. These approaches use various ML algorithms like Random Forest, K-nearest neighbors (KNN), Support Vector Machine (SVM), and different variants of Deep Neural Networks on top of extracted features for malware classification [114, 29, 77, 95, 170].

Detecting malware is challenging due to the polymorphic nature, obfuscation, and numerous sophisticated routes of creating malware. Moreover, with the proliferation of mobile apps, it is remarkably challenging to analyze each application manually for malicious activities. However, traditional machine learning-based approaches heavily depend on fine-grained feature extraction and extensive source code analysis for malware detection. Most machine learning-based approaches analyze malicious behavior of android APK files, either using static or dynamic application behavior. The research community is a bit inclined toward the static behavioral analysis of malicious apps using machine learning, which involves small run-time overhead, is scalable, considered more efficient, and features are extractable without the execution of a program. However, the common issues of these approaches are a high false-positive rate, heavy feature engineering required, and usual failure on dynamic code load and obfuscation. In contrast, approaches rely on dynamic behavior analysis and extract features based on execution/emulation by incorporating run-time code loading, obfuscation, API, and system calls but need proper inputs to determine execution paths.

Malware is continuously emerging, and new variants are being introduced daily. Traditional approaches like signature-based, handcrafted feature selection have limitations in coping with this challenge and struggle to detect new malware and corresponding families. Rule-based and machine learning techniques have been in practice for a decade to detect

malware in static and dynamic environments. However, these approaches are not robust enough to detect new malware families and struggle with the automatic extraction of fine-grained features. Moreover, recent studies revealed that Machine Learning is vulnerable to adversarial attacks, and the trained models can be compromised in a production-ready environment. This imposes a new challenge, and malware authors launch different adversarial attacks on machine learning-based approaches to compromise their integrity and availability. An adversary aims to mislead the trained classifier to produce the least True positives by classifying malware samples as Benign.

### 3.4.1 Machine Learning Based Android Malware Detection

Over the last decade, machine learning approaches have gained unprecedented adoption for malware detection. The successful adoption and deployment of machine learning classifiers to the malware domain became true for three reasons. a) An abundant ground truth(labeled data) available to the anti-malware industry and research community. b) The substantial increase in compute power by modern technologies and the availability of GPUs/CPUs/memory at a very cheap cost. c) The expansion and extensive research in the field of machine learning. Figure 3.5 presents the workflow of machine learning models that involves the iterative process of tuning the model and parameters according to the objective function. Instead of dealing with direct APK and raw malware data, the traditional malware approaches first decompile the APK file using tools mentioned in section 3.2.3 and perform preprocessing of data to extract meaningful features.

Confora*et al.* [23] presented an approach that extracts specific opcodes and the frequency of their occurrence in Dalvik bytecode using static analysis. The proposed technique was trained and tested on the Drebin dataset using six different machine learning classification models: Random Forest, Random Tree, LadTree, NBTree, RepTree, and J48. The system achieved an overall accuracy of 95% in distinguishing between malware and benign applications. Fuyong *et al.* [55] presented an approach that calculates the similarity between n-gram attributes, selects features using information gain, and uses NLP techniques for malware detection. They have compared their results with Naive Bayes, SVM, Decision Tree c4.5, and Bayesian Networks. A few Android malware detection approaches used ensemble algorithms to determine the best combination for malware detection. These approaches used different datasets to extract permission and intents on the route of static analysis. They applied numerous machine learning techniques like Random Forest, Decision

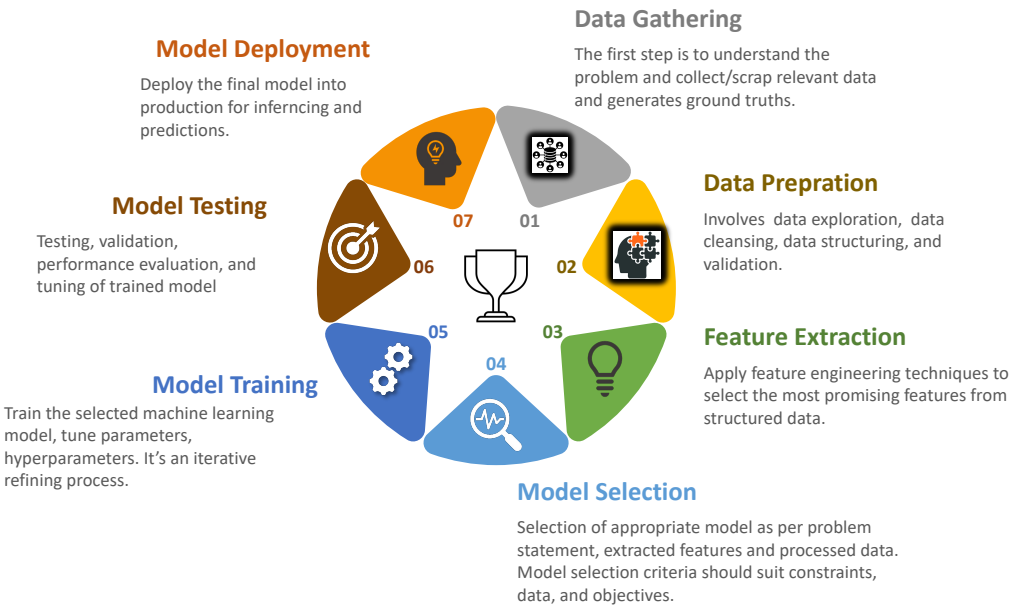



Fig. 3.5 Machine Learning Workflow Process from Data Gathering to Deployment

Trees, Multi-layer perceptrons, and Logistic Regression to classify malicious and Benign applications [38, 70, 101].

Alam *et al.* [5] presented a scheme called DroidNative that analyzes both bytecode and native code to classify malicious apps. The proposed approach used specific control flow patterns to eradicate the effects of code obfuscations. The authors used Drebin and Contagio datasets to conduct their experiments and achieved an accuracy of 93.5% using machine learning models. Kouliaridis*et al.* [85] developed an open-source tool named Mal-warehouse to collect data and monitor the utilization of resources such as CPU, battery consumption, network traffic, and memory utilization using an open-source tool called "MIET". The data collection tool is also equipped with detection models like SVM, Random Forest, and AdaBoost for malware classification and achieved an AUC score of 85.4%.

Wang*et al.* [152] used Decision Tree c4.5 to analyze the network traffic flow and detected malware with high accuracy. The author has conducted experiments on Drebin set and compared his results with state-of-the-art Android malware detection approaches. Four different variants of K-nearest neighbors (KKN) based on Hamming distance were implemented by Taheri *at al.* [146] for android malware detection. The author used filter intents, permissions, and API calls from AndroZoo [6], Drebin [13], Contagio [37], and MalGenome [57] datasets

and applied Random Forest Regressor algorithm to select the top 300 features. The evaluation criteria involved various machine learning algorithms like SVM, Decision Tree, Random Forest, and MLP in examining the malware detection accuracy.

Kouliaridis et al. [86] presented a heterogeneous ensemble android malware detection system created on numerous base models either on static or dynamic analysis. The author has used permissions, filter intents, API calls, network traffic, and inter-process communication attributes as a feature set from famous datasets like VirusShare [150], Drebin [13], and AndroZoo [6]. The author has evaluated his approach using various classifiers such as KNN, AdaBoost, SVM, Random Forest, and Logistic Regression and achieved high accuracy on the most challenging AndroZoo dataset. Another ensemble-based malware detection approach is presented by Potha *et al.* [118]. They have demonstrated that larger external instances with homogeneous size outperformed ensemble models with heterogenous and small-size external instances. The author has achieved an accuracy of 99.4% on AndroZoo, 99.3% on VirusShare, and 99.7% on the Drebin dataset.

### 3.4.2 Deep Learning Based Android Malware Detection Systems

Deep learning is the subfield of machine learning focusing more on various levels of representations analogous to feature ranking and importance. The higher level of features are inferred from the lower level, and at times lower level features help determine the higher-level features. Before deep learning, techniques are applied to draw a separation of feature levels from malware samples to perform classification, which lacks in mirroring the overarching behaviors of malware. Moreover, classification formulated on varying and diverse attributes leads to doubt on dimensions, computational resources, and training time. Deep learning-based malware classification offers scalable models to handle massive data without deep feature analysis and without expanding resources like memory, etc. Deep learning classifies the malware by observing the general pattern and distinguishing various malware attacks and their variants. The studies revealed that deep learning-based solutions offer profound classification and robust malware detectors with improved accuracy and generalization due to relying on multiple levels of feature orchestration and learning compared to traditional machine learning methods [168, 78, 82, 20].

As discussed in section 2.5.1.1, researchers primarily use DNN, RNN, CNN, DBN, RBM, and deep auto-encoders in android malware detection systems. The selection of an apt classification algorithm for detection is made considering its impact on detection accuracy

and performance. Few studies in the literature for android malware detection systems trained the deep learning method DBN (Deep belief Network) by leveraging the combo of dangerous API calls and risky permissions as a feature set to recognize the malware from benign samples automatically. Few approaches use continuous space and convert the feature set into an image and train a CNN for malicious behavior; for instance, one of the studies converts the requested permissions into an image file and trains a CNN classifier for malware detection [66]. Few studies followed the NLP route, designed a model that deals with system call sequences as texts, considered the malware detection function as theme extraction, and trained a Recurrent Neural Network (RNN) to identify the malicious sample [106]. Another NLP base approach treated the dex file as text input, automatically learned the features from raw data, and then applied multiple CNN layers for classification.

### 3.4.3 Open Issues with Deep Learning Based Malware Detection

This section elaborates on a few well-known open issues relating to deep learning in android malware detection

- The first and the most severe issue pertinent to all learning-based approaches is the misleading of the classifier. All the classifiers can be deceived.
- Malware analysts are always interested in the decision. They need to understand the reasoning behind the decision made, but deep learning lacks visibility and transparency to explain the interpretation of the detection decision formulated by its method.
- Deep learning-based android malware detection systems are not fully autonomous and need retraining if a totally out-of-distribution malware sample shows up. It needs continual retraining and parameter tuning.
- Human experts are required for feature engineering, and continual model updates are mandatory. There is no guarantee that deep learning-based trained models are equally effective and efficient against the new samples unrelated to previously trained distribution.
- Deep learning analyzes the complex correlations of input and output feature sets with no inherent explanation of causality.

- Deep learning model hosted on the cloud for malware detection sends data to a remote server for analysis; this mechanism needs data privacy protection which is not yet addressed fully.

## 3.5 Introduction to Adversarial Attacks

Over the last decade, machine learning and deep learning-based solutions provided unparalleled flexibility and versatility in addressing android malware detection. However, these approaches are self-vulnerable to adversarial attacks. In computer security, defenders and attackers always engage in never-ending arms. They always examine the system and assess strengths, opportunities, and vulnerabilities to save or exploit the system as per their interest and always try to beat each other methodologies. Competition leads to versatile and secure solutions versus sophisticated and polymorphic routes to exploit vulnerabilities. For instance, to make the traditional signature-based malware detection techniques ineffective by re-packaging and code obfuscation. Attackers create polymorphic and dynamic malware by defeating attempts to analyze the inner mechanism. Researchers have successfully evaded various malware detection systems in various domains like PDF malware detection, Windows-based malware detection, Cloud-based anti-virus engines, and android malware detection systems.

An adversary can generate adversarial samples to launch attacks on the target model by capitalizing on the knowledge and parameter settings. Various methods have been proposed in the literature to attack the deep learning model by carefully crafted adversarial samples to make the deep learning model fool. Adversarial examples are closer to real distribution with a slight perturbation imperceptible to a human eye and mislead the model to predict the targeted or incorrect class. The malware authors are attracted by this vulnerability of machine learning and misleading the learning-based android malware detection systems to produce minimum True positive. The metric used to measure the security of machine learning and deep learning models depends on the goals and capabilities of an adversary. The adversarial goals are mainly related to compromising the output integrity of learning models and can be achieved by Confidence Reduction and Misclassification.

## 3.6 Background of Adversarial Attacks

Learning-based techniques use different application representation methods, usually a binary vector, and extract the relevant features to train the classification model. The main feature of such approaches is the generalization capability and detection of unseen malicious samples with acceptable accuracy. Machine learning approaches like [168, 170, 78, 82] are recently developed to automate android malware detection by using different feature representations, for instance, Application Programming Interface (API), permissions, filter intents, app components, and dynamic behaviors. These approaches played a significant role in combating various malware. However, machine learning is prone to security risk as first discovered by [15] and neural networks are vulnerable to adversarial attacks investigated by Szeged and Ian Goodfellow [144, 59] in the computer vision domain.

Machine learning algorithms are designed with the underlying assumption that training and test data belong to the same probability distribution of past occurrences. The system learns in the training phase and predicts the label on the test/real environment (not seen in the training phase). A skilled adversary (i.e., attacker) can carefully craft an attack to violate this assumption and deceive the malware classifier. Researchers have confirmed this intuition. An adversary evasion is categorized into "Causative" and "Exploratory" attacks. In causative attacks, the skilled adversary changes the dataset distribution by injecting poisoning samples into training data or changing the labels. However, in exploratory attacks, he explores weaknesses in the trained classifier. These attacks produce adversarial examples that subsequently mislead the learning classifier by compromising the detection system's integrity and/or availability. The study of adversarial machine learning is currently the hotspot area and has largely impacted computer vision and the Natural Language Processing domain. Few studies have successfully evaded Windows-based malware detection systems [84, 87], Cloud-based anti-virus engines [27], and PDF malware detection systems [92, 155]. However, the application of adversarial machine learning to android malware detection has been scarce. Adversaries and researchers are always in a never-ending cyber-warfare race; sooner or later, adversaries will exploit the learning-based android malware detection systems.

## 3.7 Adversarial Perturbation

Adversarial attacks on learning-based detection systems raised serious threats for researchers and practitioners of various domains. The goal of an adversary is to add minute perturbation

to the original input sample that cannot be perceptible to the human eye and leads to misclassification of the target model with high confidence score as proposed by szegedy *et al.* [144]. Lets mathematically represent a Machine Learning model M which can correctly classify a data instance X as $Y_{true}$ and can be expressed as:

$$M(X) = Y_{true}$$

An adversary can carefully construct an adversarial example $X^{/}$ by adding a minute perturbation into the input data instance $X + \rho$ and deceive the trained classifier M to misclassify the input $X^{/}$, which is similar to $X$. The same classifier, M was classifying it correctly before the added perturbation. Moreover, the perturbation is bounded by a value $\rho$ to limit the number of features to manipulate. Mathematically it can be expressed as

$$M(X^{/}) \neq Y_{true}$$

Most of the research to generate adversarial examples is conducted on a natural image dataset. The main idea that leads to adversarial examples was first coined in natural images and came from the fact that no computer system can precisely and correctly classify all the objects in a given image. There is always a subset of space that leads to incorrect recognition while a human can easily recognize them. The false recognition of the subset space may be due to physical perturbation that includes shape deformation, improper illumination, distortion, occlusion, misorientation, etc. These perturbations highly influence the extracted/selected feature in the classification process and lead to false predictions. However, adversarial malware examples must preserve the intrinsic structure of the original application, and added perturbations must not change the malware functionality. The binary feature representation of an android APK makes it complex and challenging to generate more powerful adversarial examples.

## 3.8 Adversarial Threat Model & Attack Axes

A threat model defines an adversary with his ready payload, i.e., threat Vector and Surface, Knowledge, and capabilities. The popular threat models usually oscillate around the white box and black box attacks in machine learning domains and involve three integral parts: Threat Vector and Surface, Knowledge, and capabilities. Threat Vector and surface represent the interaction of an adversary to a model under analysis. In threat vector and surface,

the adversary can provide legit inputs to the model and record the corresponding outputs. The threat surface is the collection of all such inputs and outputs of the targeted model. However, the adversary's access to these surfaces is controlled by the level of knowledge and capabilities. Generally, the assessment criteria to evaluate the robustness of learning-based algorithms against adversarial attacks involves three axes. a) Adversary's Knowledge - the knowledge and information available to an adversary about the targeted model and categorized as complete, partial, and no information. b) Adversary's Goals - also known as attack specificity and corresponds to attacker intention of security violation, for instance, integrity, availability, or privacy. c) Adversary's capabilities - refer to his potential to manipulate the data and types of attacks that can be mounted. The taxonomy of adversarial attack concerning axes mentioned above is given in Figure 3.6. The details about the three attack axes are given in the following sections.

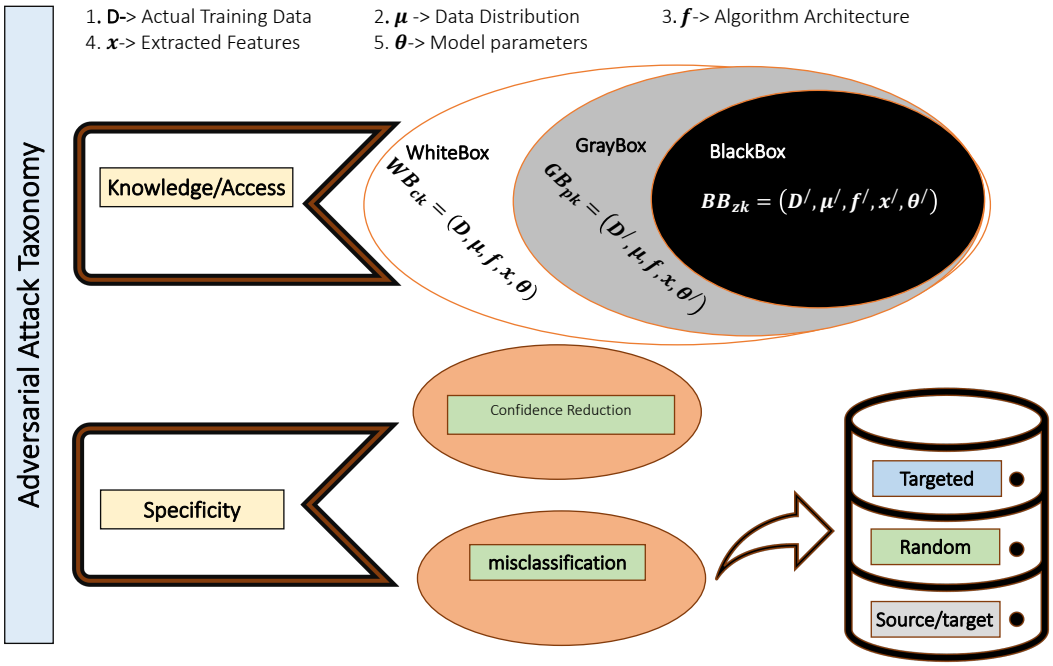



Fig. 3.6 Adversarial Attack surface Taxonomy

### 3.8.1 Adversary's Knowledge

The level of knowledge about the targeted system varies according to the adversary's capabilities. An adversary is presumed as more powerful if he has more information about the

target model. The adversary uses the available knowledge to construct attacks on a learning classifier. These attacks are grouped into the following three categories.

#### 3.8.1.1 Whitebox Attacks- Complete knowledge (CK)

In white-box attacks, the adversaries have complete knowledge about the target model $f$, including algorithm architecture, training data $D$, data distribution $\mu$, extracted features $X$, and model parameters $\theta$. Complete knowledge of the model can be mathematically expressed in equation 3.2.

$$WB_{CK} = (D, \mu, f, X, \theta) \tag{3.2}$$

The adversaries identify the most vulnerable feature space of the target model $f$ by utilizing available information and then alter an input using adversarial sample generating methods. Accessing the model's internal weight/parameters for white-box attacks corresponds to a strong adversarial capability.

#### 3.8.1.2 Graybox Attacks - Partial Knowledge (PK)

Different settings can be considered in a partial knowledge scenario, but generally, an adversary does not have the complete knowledge like Whitebox settings. Typically, an adversary has access to the extracted features, data distribution, and the nature of the algorithm, for instance, the architecture of the algorithm or whether it is a linear or nonlinear model. However, an adversary neither has access to the exact trained data nor trained parameters of the classifier. Moreover, by capitalizing the data distribution information, an adversary can generate substitute data $D^s$ and then train a substitute model on surrogate data, enabling the adversary to estimate the parameters $\theta^s$. Mathematically the equation can be written as:

$$GB_{PK} = (D^s, \mu, f, X, \theta^s) \tag{3.3}$$

#### 3.8.1.3 Blackbox Attacks - Zero Knowledge (ZK)

An adversary does not know about the target model $f$ in black-box attacks. She instead analyzes the model's vulnerability by using information about the past input/output pairs, e.g., an adversary attacks a model by inputting a series of adversarial samples and observing corresponding outputs. Recently researchers have shown that Machine/Deep learning models

are vulnerable without substantial information about the algorithm, training data, feature space, or learned parameters. If an adversary can query the BlackBox learning algorithm and gets the output as a confidence score or class label, she can threaten the underlying classifier. However, the attacking adversary must have the information on the task/objective of the targeted classifier; for example, the classifier is designed for text classification, image classification, object detection or malware detection, etc. This minimum information allows an adversary to choose a specific transformation to generate and evaluate changes in the input feature space. Otherwise, the adversary is blind to the purpose of the classifier, and crafted perturbation will not have any effect. Mathematically Blackbox with zero knowledge can be expressed as:

$$BB_{ZK} = (D^s, \mu^s, f^s, X^s, \theta^s) \tag{3.4}$$

### 3.8.2 Adversary's Goals

Attacker goals are also known as specificity attacks used to violate the security fence of the learning-based classifier. Firstly, an adversary aims to compromise the integrity (malware samples classified as benign), availability (benign applications are detected as malware, also known as DoS), and privacy (leaking confidential information about its users) of the detection system. The goal of an adversary can be divided into targeted and indiscriminate attacks. For example, suppose an adversary wants to pass the specific inputs through the classifier or wants to manipulate the specific class labels. In that case, these are targeted attacks, while the indiscriminate attack is random and does not care about specific input or class labels. An adversary can compromise the integrity of the learning classifier by three methods, i.e., Confidence Reduction and Misclassification. The misclassification is further achieved through random, targeted, and source/target misclassification, as enumerated below:

- **Confidence Reduction** Mostly, the output of the machine learning model is the predicted class label and associated confidence score. The adversary's goal is to reduce the confidence score for the target model. For instance, a classifier predicts a network packet as malicious with a low confidence score.

- **Random Misclassification** The adversary intends to change the output of a target model to a random class different from the actual class. For example, the class of a network packet changed to any other class except the malicious.

- **Targeted Misclassification** An Adversary attempts to change the output of a class into the specific target class. For example, Input fed to a malware classifier is classified as benign.
- **Source/Target Misclassification** The goal of an adversary is to change the output of a specific class into a specific target class. For example, the Benign class is always classified as Malicious.

### 3.8.3 Adversary's capabilities

The adversarial capabilities are directly attributable to the adversary's abilities and the intended type of attack to be mounted. To generate adversarial examples in the malware domain is constrained by the level of access/knowledge available to an adversary. For example, an adversary can easily manipulate features and perform specific transformations at compile-time if she has access to source code. However, it is challenging in the case of no source code access. The adversarial capabilities are grouped as training phase capabilities and testing phase capabilities. In the training phase, the adversary tries to corrupt the target model by altering the training dataset with partial or full access to training data. Data injection, modification, and logic corruption achieve the training phase capabilities. Attacks crafted for the training phase are also known as poisoning attacks to change the training set distribution. For instance, modifying the training data, deleting or inserting new data, manipulating data features or labels, etc. However, the testing phase capabilities depend on the available knowledge. Adversarial attacks on the testing stage, also known as evasion attacks, are classified into two categories, i.e., Whitebox attacks and Blackbox attacks, as discussed above.

## 3.9 A Short Review on Adversarial Attack and Adaption to Malware Detection

According to [144], these perturbations contradict achieving high generalization performance of a neural network in various domains. These minor perturbations to input images are not visible to human eyes and still confuse the prediction models with a high confidence score. After the conception of adversarial examples, researchers have a different point of view about the existence of adversarial examples. Some [100] say adversarial examples lie on

the boundary of a manifold and due to poor generalization of a classifier in the premises of adversarial examples, or these adversarial examples exist due to high variance. In contrast, other schools of thought have a point of view about their existence due to the high nonlinearity of neural networks [113]. However, [59] came up with the contradiction to the speculation of nonlinearity and demonstrated by adding minute perturbation to a well-regularized model and conducting experiments on a linear model. He has demonstrated that both the regularized and the linear model could not defend against adversarial examples in defense performance improvement. Goodfellow *et al.* [59] articulated that the cause of adversarial examples is the linear outcome in high dimensionality. For a linear classifier in high-dimension space, every input feature of a linear classifier is normalized. Therefore minute perturbation to each input in one dimension cannot deviate the classifier to misclassification. While tiny perturbation in all dimensions of an input results in the misclassification of a classifier. Figure3.7 supports the Goodfellow argument that the linear models are vulnerable to adversarial examples by adding minute perturbation to each dimension in a particular direction of the input.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Orignal Example | 1 | 2 | 3 | -1 | -2 | 2 | 1 | -3 | 2 | 1 |
| Weights | -1 | -1 | 1 | -1 | -1 | -1 | 1 | 1 | -1 | 1 |
| Adversarial Example | 0.5 | 1.5 | 3.5 | -1.5 | -1.5 | 1.5 | 1.5 | -2.5 | 1.5 | 1.5 |

Prior to perturbation:
-1-2+3+1+2-2+1-3-2+1 = -2
Class "A" probability is calculated as :
1/(1+exp(-(-2))) = 0.1192

Post perturbation:
-0.5-1.5+3.5+1.5+1.5-1.5+1.5-2.5-1.5+1.5 = 3
Now Class "A" probability :
1/(1+exp(-(3))) = 0.952

Class "A" probability increased from 12% to 95% after Perturbation

Fig. 3.7 Depicting the increase in Class "A" probability from 12% to 95% prior and post perturbation to each dimension in specific direction.

Several studies are presented on evasion and poisoning attacks for learning-based systems, making different assumptions on the amount of knowledge available to adversaries about the targeting system in the training, testing, and deployment phases. For instance, Srndic *et al.* [92] studied in windows executable where an adversary had a high amount of information available about the internal structure of the system. Similarly, sharif *et al.* [131] studied attack scenarios on facial recognition systems where an adversary had access to feature space and recognition algorithm parameters. Xu *et al.* [162] studied the evasion attack on the Linux system in restricting information access and launching Blackbox attacks on the targeted classifier that yields a confidence score in continuous values against the query sample. However, these attacks are prevented by simply hiding the confidence score. Moreover, the applicability of such attacks to android malware detection is still a question due to the binary

output of benign and malware decisions. Few other studies have introduced adversarial malware examples on Windows PE (portable executable) by altering and adding the bytes in the unused space of a binary file without affecting the original functionality [84, 87, 88]. Such attacks focus on the unused bytes of the application and do not require complete source code to evade the classifier but still require Whitebox settings to attack the target model.

Few other studies have introduced problem-driven approaches to launch adversarial attacks in Blackbox settings. Song *et al.* [138] introduced an adversarial attack by randomly generating the sequence of macro-actions and applying recursively until the targeted classifier evaded. Examples of macro-actions include adding a byte at the end of a binary, adding a byte in unused binary space, adding a new section, removing the debugging information, etc. Furthermore, the macro-actions can be divided into micro-actions which can give more insights into the reason for evasion by a particular action. For instance, appending byte action can be broken down by appending only one byte at a time. The method yielded good results against windows PE in static and dynamic analysis. Nevertheless, the transformation of this method to Android APK is questionable. Pierazzi *et al.* [116] introduced an adversarial attack on Android OS using Drebin dataset [13] in Blackbox settings. The author has injected the benign code blocks to change the feature extracted by Drebin and misleads the classifier.

Learning-based android malware detection systems need to be secured against adversarial attacks. We proposed an image-based learning model [42] for android malware detection that can yield best-in-class accuracy for malware detection and family classification and establish a solid defense against adversarial attacks. The next chapter will circumvent the details of this image-based learning model for android malware detection.

# Chapter 4

# RGB-Based Android Malware Detection using Deep Learning

## 4.1 Overview

With the proliferation of handheld devices and numerous routes for malware creation, detecting new sophisticated malware becomes a real challenge. Contrary to conventional machine learning approaches, which require feature engineering and source code analysis, we propose using a new RGB-based imaging technique for android malware detection and classification. Our system is built on a static analysis of the android application packaging (APK) file to combat malware threats. First, we perform a novel transformation of the APK file into a lightweight RGB image using a predefined dictionary and intelligent mapping. Second, we train a convolutional neural network on the obtained images for signature detection and malware family classification. The experimental results on the AndroZoo [6] dataset show that our system can classify both legacy and new malware applications with high accuracy of 99.37%, a False Negative Rate (FNR) of 0.8%, and a False Positive Rate (FPR) of 0.39%.

## 4.2 Image-based Android Malware Detection Techniques

Natraj *et al.* [107] are pioneers in coining visualization-based malware detection as a part of Kaggle competitions. The author converted the binary executable to a grayscale image in Windows PE and applied the K-nearest neighbor to measure the similarity. Ganesh *et al.*

presented a visualization-based Android malware detection system that performs permission mining only and converts the extracted information into an image to train a Convolutional Neural Network (CNN). The proposed system predicts the malicious app with 93% accuracy by learning the patterns instead of signatures. However, the author tested it only using permissions, and evaluation was conducted on imbalanced data. Similarly, Shiqi *et al.* [64] converted an APK to a grayscale image and used Deep Belief Network (DBN) for malware detection. The author has conducted experiments on the Drebin dataset by leveraging image texture and API calls. Huang *et al.* [66] presented an approach that converts APK into fixed-sized images and trains a CNN for automated feature extraction and malware detection with 93% accuracy. Ding *et al.* [46] proposed an approach to convert API calls from dex files to two-dimensional images. The author has trained a CNN on top of extracted APIs and compared his results with traditional machine learning approaches.

Numerous approaches have been proposed in the literature to convert binary files into grayscale images and then apply ML or DL-based classifiers for malware detection, for instance, [79, 39, 75]. These approaches usually rely on the small source code from the dex file ignoring permissions and other multiple contributing factors from manifest.xml and dex file or vice versa (only using API calls and ignoring app components). They perform a blind transformation of the binary code into a grayscale image and apply pattern recognition algorithms. They achieve a certain level of detection accuracy but are not robust enough against new malware and mostly fail against encryption and code obfuscation. There are few image-based approaches developed for Android malware detection, and the field is comparatively new in the context of images. Moreover, these approaches struggle in family classification due to blind transformation, resulting in multiple false alarms and yet to be evaluated under adversarial attacks. We have designed an intelligent system that smartly maps an APK file into an RGB image. It is practically viable, lightweight, scalable, and robust enough to detect legacy and new malware and their family identification.

## 4.3 Proposed Methodology

Most malware detection studies adopt static (without executing the malware) or hybrid (by executing the malware) analysis. Following the static analysis track, we propose a novel malware detection system that performs the intelligent mapping of an APK file into an RGB image without source code analysis and feature engineering. The proposed system learns the malicious patterns from manifest and dex files by leveraging the app components, API calls,

and opcodes. The transformed resultant RGB images of APK files are preprocessed using different interpolation techniques and then used for training a convolution neural network for malware detection and family classification. The proposed framework aims to build a lightweight, robust, and scalable solution that acts as a combat force against legacy and new android malware. The system-level architecture of our RGB-based Android detection system and its workflow is shown in Figure 4.1. Following are the main contributions of the RGB-based android malware detection framework.

- **An intelligent mapping of an APK file to an RGB image:**

  The system uses a reverse engineering tool to analyze the manifest.xml and dex file. It performs an intelligent and strategic mapping of an Apk file to an RGB image rather than a blind transformation of a binary file to a grayscale image. We have devised a meticulous predefined dictionary [41] condensing suspected API calls, permissions, and various malware behaviors learned and collected from numerous studies in the literature. The system extracts all the app components, including permissions, intents, activities, services, and filter intents, from the manifest.xml with protection level normal and maps them on the green channel. We map all the customary API calls and opcodes in the second step on the red channel. In the third step, we map all the suspected behavior on the blue channel using our predefined dictionary, which comprises (a) suspected API calls from the dex file, (b) permissions, intents, activities, services, and receivers from the manifest.xml file, and (c) protected strings without any superfluous information. The resultant RGB image is preprocessed and normalized using nearest-neighbor interpolation as a ready appetite for the CNN classifier.

- **Generalized and lightweight solution** The proposed framework is a generalized solution that detects new and legacy malware with a high confidence score. The system is robust against malware of polymorphic and varying nature without relying on fingerprint features, signature detection methods, and zero code analysis. The system is lightweight and scalable to detect malware on-device solutions and in cloud settings for two reasons (a) the transformation of an APK file (usually in MBs) into a lightweight RGB image of size 3-4KB. (b) The lightweight static analysis tool uses Andaguard, which yields the best and fast results even on large-size APK files compared to other tools like Ninjadroid. Along with the binary classification of malware and benign, the system can identify the family of reported malware; experiment and discussion sections elaborate more on malware and family classification. The experimental results on the

AndroZoo dataset from 2009-to 2020 indicate that the proposed system is practical, lightweight, scalable, and a generalized malware and family classification solution.

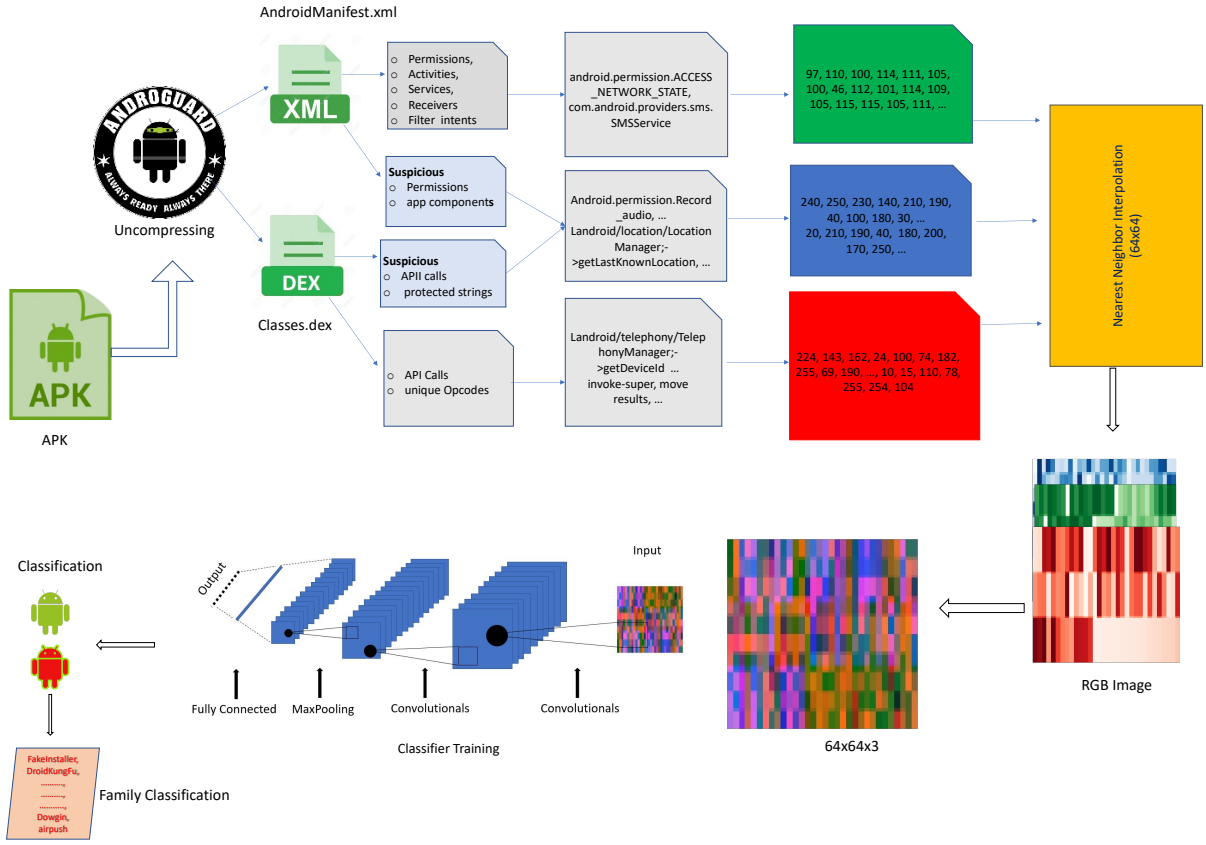



Fig. 4.1 System working flow for android malware signature detection and family classification

### 4.3.1 Tactful Transmutation of APK file into an RGB Image

Firstly, our system statically analyzes the AndroidManifest.xml file using the lightweight reverse engineering tool *Androguard* as depicted in Figure 4.1. The system extracts all the permission and app components from the manifest file, including activities, intents, services, providers, and receivers. The system converts all the extracted app components and permissions into pixel values using each character's ASCII code and applies filtration using our predefined dictionary [41]. We map all the extracted permissions and app components

on the green channel with permission level Normal in a specific order to generate certain patterns. We envision that the pattern will help the learning classifier to detect malware. The system does not map the suspected app components and permissions on the green channel having a matching index in the dictionary. We capture all the suspected behavior on the blue channel discussed later in this section. The conversion of the manifest file onto a green channel of an image can be visualized in Figure 4.2.

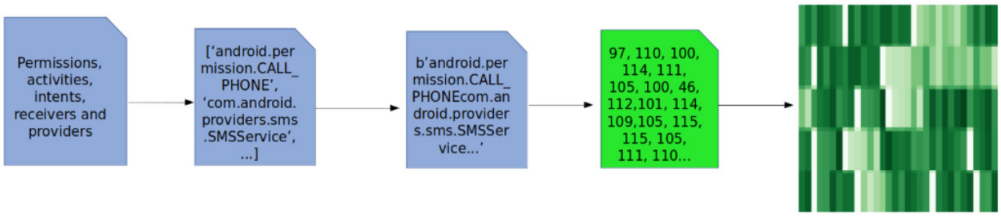



Fig. 4.2 Green Channel: Conversion of Permissions and app components from AndroidManifest.XML

After mapping the manifest file on the green channel, the system analyzes the dex file and extracts all API calls and unique opcode sequences from Dalvik bytecode. The system performs the conversion of all extracted API calls by summing the ASCII value of each character (M) and then taking the mod by 255 to normalize the pixel value in the range [0-255] as given in equation 4.1 where M represents each ASCII character of API call. After the conversion, we perform filtration and only map the standard API calls on the red channel using the predefined dictionary and save suspected API calls for the blue channel.

$$Pixel_{Value} = \sum_{i=1}^{n} (M_i) \bmod 256 \tag{4.1}$$

For example, Ljava/net/HttpURLConnection-connect is an API call and the conversion is performed as character by character ASCII value as follow:
Decimal value = [76, 106, 97, 118, 97, 47, 110, 101, 116, 47, 72, 116, 116, 112, 85, 82, 76, 67, 111, 110, 110, 101, 99, 116, 105, 111, 110, 59, 45, 62, 99, 111, 110, 110, 101, 99, 116]
Sum = 3526 Pixel-value = $3526 \bmod 256 = 198$

We also map unique opcodes (operation code) from the dex file along with API mapping. Dex file may contain millions of opcodes to execute low-level instructions, and mapping all the opcodes on the red channel can make the resultant image noisy. We have only used unique opcode sequences from the dex file and map on the red channel. Every opcode has a predefined value range from [0-255]. For example, invoke-super parameter, method-to-call;

It invokes the virtual method of the immediate parent class. The opcode 'invoke-super' has a Hexa value of '6F' and a decimal value of '111'. The mapping of the red channel can be visualized from Figure 4.3.

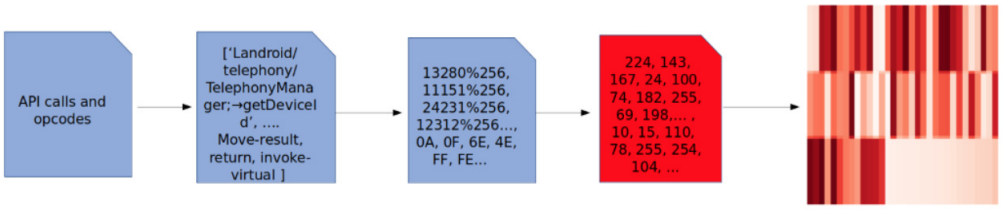



Fig. 4.3 Red Channel: Conversion of API calls and unique opcode sequences from Dex file

Lastly, we gather all the suspected malicious components from the manifest and dex files and map them onto the blue channel, as illustrated in Figure 4.4. The system matches the extracted permissions, app components, API calls, opcodes, and protected strings with an exhaustive predefined dictionary. We have populated this meticulous dictionary based on different malware studies [74, 47, 76, 132]. If any suspicious activity is not listed in the dictionary, it is still mapped on the red and green channels and incorporated into the decision process. We are also updating this dictionary based on analyzing different datasets and new malware. It is imperative to mention that we are only mapping protected strings that encapsulate malicious activities such as API calls, permission, or any app component. Because Androguard returns both protected and rough strings, converting all these strings leads to noisy images and classification cumbersome.

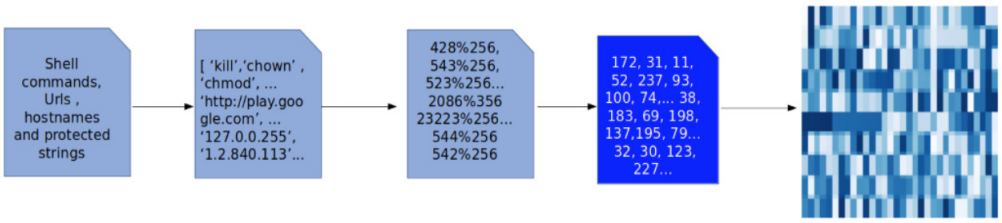



Fig. 4.4 Blue Channel: Conversion of protected strings,suspected permissions, app components and API calls

### 4.3.2 Image Scaling using Nearest Neighbor Interpolation

The size of an android APK depends on an application's functionality, due to which dex and manifest files may have different sizes, which can lead to varying channel sizes in a

transformed image. However, our system needs a consistent channel size as a ready appetite to feed our CNN classifier. Therefore, we trim or elongate the resultant image's channel size into 64x64 (height x width) after transmutation. We use the Nearest Neighbour interpolation (NNI) without losing any information to shrink and elongate the channel size. We have employed different interpolation techniques, and a detailed discussion is given in section 4.4. We aim that the generated pixel value must correspond to an APK's value, not a dummy value. Other Interpolation techniques do not guarantee that the created value is an APK's property because they rely on the surrounding values to estimate new value. However, NNI ensures that the newly created value must belong to an APK's property because it resamples the original image's existing values. For example, we have an input image $I$ of size $mxn$, and the goal is to enlarge it to a size of $p \times q$. First, we need to determine the row and column-wise scaling factor $Sr, Sc$ of the resultant transformed image $O$. The resultant image $O$ pixel value at coordinates $(r,c)$ will be created from the input image $I$ using equation 4.2.

$$x = \lfloor \frac{r}{Sr} \rceil \quad ,and \quad y = \lfloor \frac{c}{Sc} \rceil \tag{4.2}$$

Thus $O_{rc} = I_{xy}$. Where $(x,y)$ are the pixel coordinates in the original input Image $I$. Now that we have all three channels of the same size (64x64), we merge them to create one RGB image, which encapsulates all the information of the APK file and is ready to feed to our convolutional neural network. Few samples of benign and Malware images are depicted in Figure 4.5

The complete tactful transmutation process of an Android APK into an RGB image is explained via pseudo-code in Algorithm 1. Androguard returns three parameters, including $a$ corresponding to the APK object used to analyze the manifest file. $d$ specifies an array of Delvick bytecode objects for analyzing classes and methods in the Dex file. While $dx$ corresponds to the analysis object and contains special classes and linking information, it is handy for analyzing multiple Dex files simultaneously.

## 4.4 Experimental Setup

The section circumvents the CNN architecture, dataset details, experimental configurations, parameter settings, results, and detailed analysis concerning existing approaches and future directions.

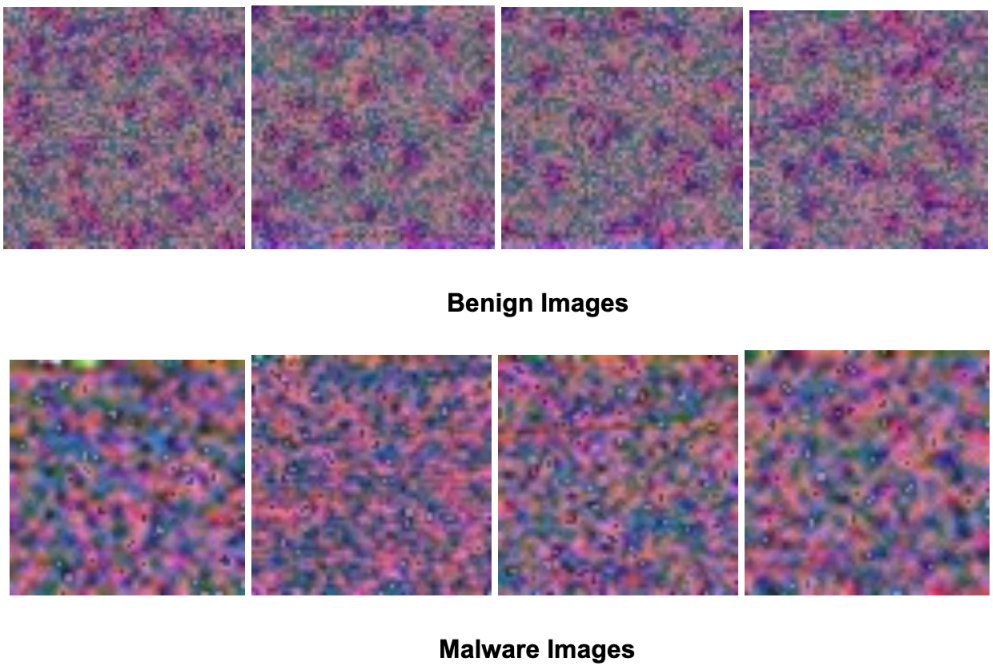



Fig. 4.5 Image Representation of APKs: Malware and Benign Application Samples

### 4.4.1 Convolutional Neural Network for signature detection and family classification

Different Neural Network architectures are helpful in various domains, like Recurrent Neural Nets (RNN) are the first choice for Natural language problems with sequential data. Similarly, Convolutions Neural Networks (CNN) are very common for computer vision tasks, for example, object detection, recognition, object segmentation, tracking, and many more. Due to the inherent properties of images, Convolutional Neural Networks are widely used for images and continuous data and significantly boost performance. As of now, APKs are transformed into images, and applying the CNN seems an intelligent choice for signature detection and family classification. The architecture of CNN for malware detection is depicted in Figure 4.6. As depicted, the input to the CNN classifier is a 64x64 RGB image with three uniform channel sizes. The architecture mainly involves two convolution layers with 32 filters on the first and 64 filters on the second layer. The kernel/filter size at initialization is 3x3 and instantiated using glorat, also known as Xavier uniform. Both the convolution layers use ReLu as an activation function, followed by a max-pooling layer (2,2) and a dropout layer with a rate of 0.25. The architecture uses ‘sigmoid’ activation in the output layer and

**Algorithm 1** Transmutation of an APK to an RGB Image for android malware detection

1: **Predefined_Dictionary** enumerates suspected malware properties collected via several past studies and assigning pixel value to these properties.
2: **Begin**
3: $a, d, dx \Leftarrow$ Androguard(apk)
4: $green\_channel$, $manifest_{susp} \Leftarrow$manifest2Pixel(a)
5: $Interpolation(green\_channel$,size$= (64 \times 64))$
6: Convert Dex2Pixel$(d, dx)$
7: $red_channel, api_calls_{susp} \Leftarrow$api_calls2Pixel$(dx)$
8: $Interpolation(red\_channel$,size$= (64 \times 64))$
9: $opcodes_pixel \Leftarrow$opcodes2Pixel(dx)
10: $suspicious_strings \Leftarrow$strings2Pixel(d)
11: $blue\_channel \Leftarrow manifest_{susp}$+$api\_calls_{susp}$ +$suspicious\_strings$+$opcodes\_pixel$
12: $Interpolation(blue\_channel$,size$= (64 \times 64))$
13: $RGB \Leftarrow merge(red\_channel, green\_channel,$ $blue\_channel)$
14: **End**

binary cross-entropy as a loss function for signature detection, as given in equation 4.3.

$$L(y,\hat{y}) = -\frac{1}{N}\sum_{i=0}^{N}(y * \log(\hat{y_i}) + (1-y) * \log(1-(\hat{y_i})) \tag{4.3}$$

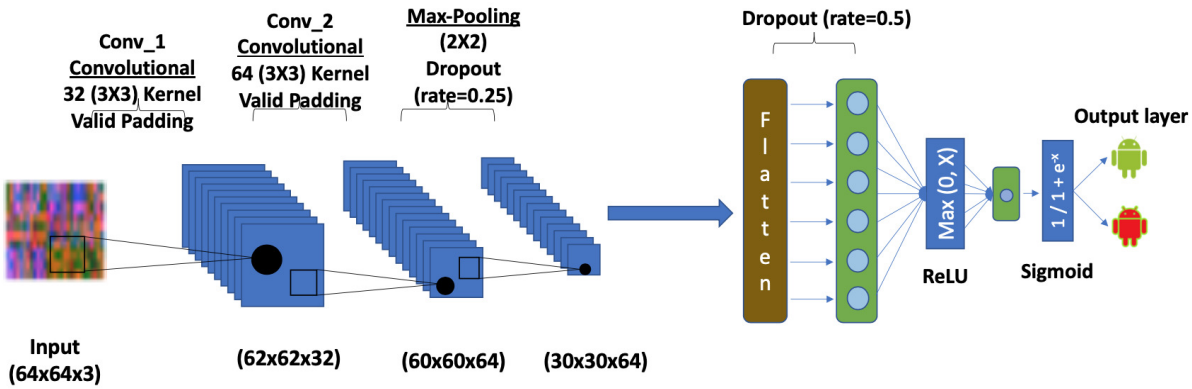



Fig. 4.6 Complete architecture of Deep Malware Detection System

From our experience and literature review, it is well known that sigmoid and hyperbolic tangent (*tanh*) results in gradient vanishing in back-propagation. Therefore, we have used

rectified linear unit (ReLu) as an activation function as given in equation 4.4.

$$\forall_q \varepsilon [1, m_k] f(x) = max\{0, x\} \tag{4.4}$$

Our CNN architecture uses the adaptive learning rate, Adam, starting with 0.001, and decayed to $10^{-5}$ after every epoch. Adam helps to train NN more speedily by enabling the power of Adagrad (Adaptive Gradient Algorithm) and RMSprop (Root Mean Square Propagation). We have used the adaptive learning rate, Adam, with an initial learning rate set to 0.001, which decayed to $10^{-5}$ after each epoch. The rationale behind using Adam is its popularity for training the deep learning model with fast speed, plus it enables the power of Adagrad (Adaptive Gradient Algorithm) and RMSprop (Root Mean Square Propagation). Adagrad works well in sparse gradients, and RMSprop is suitable for mini-batch gradient descent in online settings.

Appetite to our system is the converted APK into RGB image with three channels and a size of 64 x 64. Following the guidance and advice from the literature, we start with a kernel size of 3 x 3 and filter number 32 at the first convolution layer. We initialize the kernel with glorat, also called Xavier uniform [58], which draws samples from a uniform distribution within the limit, where the limit is defined as per equation 4.5.

$$limit = \sqrt{\frac{6}{fan_{in} + fan_{out}}} \tag{4.5}$$

where $fan_{in}$ and $fan_{out}$ shows the number of input and output in weight tensor respectively.

### 4.4.2 Details of Dataset

For experimentation, we use AndroZoo [6] because it is the largest repository that keeps on growing and have APK files available in some other datasets like AMD, Drebin, Google play, and Genome. In AndroZoo, all APKs are labeled benign and malware using numerous Anti Virus (AV) tools. The rating and Antivirus detection (VT) is also marked against each APK depicting how many AVs say the particular APK is malware. Currently, AndroZoo contains 10 million android applications. Among this pool, 7.3 million apps are segregated as benign as their VT detection is zero. However, for the remaining 2.7 million, they are labeled as malware by one or more than one Antivirus. We only choose malware apps where VT detection is $\geq 15$ (at least 15 Anti-virus software marked them malware) to remove the false

positive and ensure high confidence that the app belongs to a malicious class. AndroZoo allows us to access the data via SHA256 and the creation date. The details of the dataset are given in Table 4.1.

Table 4.1 Yearly view of Malware Samples in AndroZoo Dataset

| Year | #APKs | VT ≥15 | Year | #APKs | VT ≥15 |
|---|---|---|---|---|---|
| 2020 | 9978 | 3 | 2014 | 1762714 | 87400 |
| 2019 | 181825 | 1562 | 2013 | 772918 | 47925 |
| 2018 | 428725 | 5153 | 2012 | 566549 | 139169 |
| 2017 | 383406 | 5754 | 2011 | 219627 | 36900 |
| 2016 | 1425425 | 22781 | 2010 | 61122 | 1235 |
| 2015 | 866759 | 49280 | 2009 | 12075 | 61 |

## 4.5 Results & Discussion

Using the provided SHA256 file by AndroZoo, we have split the data into yearly chunks per VT detection of malware apps. The segregation is for our convenience to download, store, analyze data, and train our CNN classifier. In the first experiment, we selected malware apps with VT detection $\geq 35$ and collected a random sample of 4K malware and 4k benign apps from the year chunk [2009-2011]. Similarly, in the second experiment, we kept the same configuration for 12K malware and 12K benign samples from 2012 to 2014. As discussed earlier, a higher VT detection number in the dataset guarantees that the particular app is malware. Unfortunately, we cannot keep the VT number constant because of the non-availability of higher VT detection for all available apps in the AndroZoo dataset. For the third experiment, we collected 30K malware samples with VT detection $\geq 15$ and 30K benign samples from 2015 to 2016. In the last experimental setup, we tested our approach on 11K malware apps with VT detection $\geq 15$ and 11K benign apps from the year range [2017-2020]. We have trained our CNN-based classifier as discussed in section 4.3 with the split ratio of 65% and 35% for train and test, respectively. Our novel approach for smartly mapping the APK into RGB image yielded excellent results for malware detection, as discussed in Table 4.2.

Table 4.2 shows false-negative rate (FNR), false-positive rate (FPR), F1 score (F1), recall (Rec), precision (Prec), Area under the curve (AOC), and accuracy metrics for all the experiments and datasets in respect of signature detection. The mathematical expressions for precision and recall are given in equation 4.6 and equation 4.7.

Table 4.2 RGB-based Malware Signature Detection Using CNN and ResNet

| # M/B | Year | FNR% | | FPR% | | F1 | | AOC | | Acc% | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | CNN | ResNet | CNN | ResNet | CNN | ResNet | CNN | ResNet | CNN | ResNet |
| 4K each | 2009-2011 | 1.7 | 0.62 | 1.5 | 0.25 | 0.98 | 0.99 | 0.997 | 1.0 | 98.21 | 99.41 |
| 12K each | 2012-2014 | 1.2 | 1.39 | 1.9 | 1.48 | 0.98 | 0.984 | 0.997 | 0.99 | 98.39 | 98.18 |
| 30K each | 2015-2016 | 4.9 | 2.7 | 3.5 | 2.3 | 0.95 | 0.96 | 0.977 | 0.977 | 95.73 | 97.42 |
| 11K each | 2017-2020 | 1.72 | **0.85** | 0.72 | **0.39** | 0.98 | **0.99** | 0.999 | **1.0** | 98.77 | **99.37** |

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} \tag{4.6}$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} \tag{4.7}$$

The precision-recall curve shows the trade-off between precision and recall of different thresholds. A high area under the curve demonstrates high recall and precision, where high recall shows a low false-negative rate (the actual app is malware, and the model classifies it as benign). This case is of our highest interest, and our goal is to minimize the false negative. High precision means the false positive rate is low (the actual app is benign, and the classifier predicts it as malware).

### 4.5.1 Results for Malware Family Classification

After signature detection, why malware family classification is essential? Malware family classification is vital for security experts and anti-virus vendors to determine the identification of malware families to assess the threat level and establish an apt defense mechanism. They must know about malware families and behaviors to educate and alert end users about how a particular malware can affect their mobile devices. The AndroZoo dataset contains 3305 different malware families until 2016 inclusive. However, malware family labels are not available for 2017 onward android applications. The details of the top ten families of the AndroZoo dataset are mentioned in Table 4.3

We ran a series of experiments with different malware samples and VT detection for Android malware family classification. Using the SHA256 file, we have downloaded the top 10 malware families for simplicity, and the number of samples for each family is more than 5000. To the best of our knowledge, we are the first to use vast family-wise samples for malware family classification. Following the same methodology discussed in section 4.3, we

Table 4.3 Top 10 Malware Families in AndroZoo Dataset

| Rank | Family Name | Count # | Rank | Family Name | Count # |
|---|---|---|---|---|---|
| 1 | dowgin | 262057 | 6 | artemis | 38041 |
| 2 | kuguo | 107114 | 7 | droidkungfu | 37336 |
| 3 | airpush | 100471 | 8 | leadbolt | 31491 |
| 4 | revmob | 74419 | 9 | adwo | 28733 |
| 5 | youmi | 51762 | 10 | jiagu | 27526 |

have trained a convolutional neural network (CNN) for malware samples having family labels available in AndroZoo dataset. The results for malware family classification are summarized in Table 4.4.

Table 4.4 Results of CNN for Malware Family Classification

| No of Families | No of Samples per Family | FPR % | F1 score | Accuracy % |
|---|---|---|---|---|
| 6 | 600 per family | 9.7 | 0.91 | 91.21 |
| 10 | 175 per family | 5.8 | 0.96 | 96 |
| 5 | 4100 per family | 10.8 | 0.92 | 92.3 |
| 6 | 2300 per family | 11.2 | 0.92 | 92.4 |
| 10 | 4100 per family | 15.84 | 0.88 | 88.91 |

### 4.5.2 Discussion & Analysis

The proposed approach is a practical, feasible solution completely independent of feature engineering, fingerprint examination, and deep source code analysis. The system works for all legacy and new malware and applies to all android datasets. Our proposed system is well generalized, lightweight, and equally viable to cloud and on-device settings. Generally, an android APK's size varies between 3MB to 80MB, and the average Android app size is 15MB [21]. The size of an android APK also depends on the application's functionality; for instance, the average game-specific app is 68MB, and navigation-related is 45MB [21]. The rationale behind the fast and lightweight solution is the APK-to-image conversion in less than a second for an average-sized Android application. The conversion time is dependent on the APK size. However, the resultant image size of android APK always varies between 2-3KB. Uploading the 2-3KB image on the cloud for malware detection is cheaper and faster than a 70MB APK. Table 4.5 presents some exciting results obtained using the proposed approach compared to the existing state-of-the-art techniques for malware detection.

Table 4.5 Comparison of Malware Detection, (F.Eng abbreviated for feature engineering, and F.C stands for family classification).

| Approaches | F.Eng | F.C | Dataset | Acc% | FN Rate% |
|---|---|---|---|---|---|
| [76] | ✓ | × | AMD+Drebin | 97 | – |
| [168] | ✓ | × | Google.P+Genome | 96.7 | – |
| [66] | × | × | Google Play | 93 | 9.0 |
| [141] | ✓ | × | Genome+Drebin | 97.2 | – |
| [72] | ✓ | × | AndroZoo | 97.65 | – |
| [61] | ✓ | × | Drebin | 98 | 7.0 |
| our | × | ✓ | AndroZoo | **99.37** | **0.85** |

Several other approaches exist in the literature for android malware detection. Results for a few of these approaches are mentioned in Table 4.5 in contrast to our RGB-based android malware detection results depicted in Table 4.2. The comparison indicates that our approach yielded excellent results for legacy and new malware and catered to the most extensive dataset without any feature engineering and source code analysis. Moreover, our approach also performs the family classification of detected malware.

Table 4.6 Malware Family Classification Comparison

| Approaches | No of Samples/Family | No of Families | F1 score | Dataset |
|---|---|---|---|---|
| [28] | 59-833 | 15 | 0.82-0.95 | Drebin |
| [171] | 43-925 | 14 | 0.93 | Drebin |
| [142] | 43-925 | 20 | 0.90 | Drebin |
| [149] | 2408 (Total) | 20 | 0.71 | AMD |
| [50] | 643 (Total) | 43 | 0.93 | FalDroid-II |

[168] used less than 2k malware and 20K benign apps, [66] used malware samples only for the year 2017(January to May), [141] used less than 4K malware and 4K benign apps, and![61] used only 5.5K malware apps for their experiments. The approaches mentioned above are not performing malware family classification. Therefore, RGB-based malware family classification results presented in Table 4.4 are compared with a few other trending approaches mentioned in Table 4.6. It is easy to observe that these approaches used a limited number of malware families and the ratio of per family samples was also nominal. Our family classification results exhibit that, as the number of samples per family is less than

200, we have achieved an accuracy of 96%. The accuracy metric decreases as the number of samples increase per family and include more malware families. We have tested our results on the top 10 families from the AndroZoo dataset with more than 4K samples for each family and still achieved remarkable results.

We have applied various CNN flavors like ResNet (Residual Network), VGG, etc., according to their suitability for image recognition tasks. We have not observed a marginal difference in results by applying different CNN variants, as depicted in Table 4.2. However, it is imperative to highlight that different interpolation technique impacts the evaluation metrics. We have employed numerous interpolation techniques to normalize the resultant image, including interlinear interpolation, inter-cubic interpolation, and inter-area interpolation. Inter linear interpolation yielded promising results but cannot guarantee the generated pixel value belongs to an APK property. Therefore, we have implemented the problem suited to Nearest neighbor interpolation, as discussed in section 4.3.

## 4.6 Summary

The focus of this research work is to build a deep learning-based generalized, practical, and lightweight solution for android malware detection. The proposed system examined the AndroZoo dataset without any feature engineering and source code analysis and performed well on the legacy and new malware in the year range [2009-2020]. The system performs the novel, tactful and lightweight transmutation of an android APK file into an RGB image and fixes the resultant image using the NN interpolation. A CNN-based classifier is trained on top of these image-based Android APKs and yields exciting results. In the next chapter, we expose our approach to adversarial settings and test the robustness of RGB-based android malware detection under adversarial attacks.

# Chapter 5

# Novel Adversarial Attacks for Learning-Based Android Malware Detection Systems

The exhilarating pace of smartphone innovation and vast proliferation in everyday life also impose serious security threats. The open-source and largest android market is also the hive for malware authors. Since the last decade, machine learning (ML) has gained much attraction and successfully deployed, offering unsurpassed versatility for automated malware detection. Unfortunately, as ML-based approaches become widely adopted and deployed, adversaries are also in a never-ending race to evade these classifiers for bypassing android malware detection systems. To combat adversarial attacks and secure machine learning-based android malware classifiers, we present a novel image-based android malware classifier that has proven its robustness under various adversarial settings. In this work, we have crafted two novel attacks and reveal that the state-of-the-art Android malware detectors are vulnerable and easily evaded with a more than 50% evasion rate. However, the proposed approach establishes a durable defense line against these adversarial attacks and is too arduous to bypass.

## 5.1 Machine Learning a Weaker Link in the Security

Since the last decade, machine learning-based systems have exhibited tremendous versatility in combating various polymorphic malware. These approaches like [168, 170, 82] are

developed recently to automate android malware detection by leveraging different feature representations, for instance, Application Programming Interface (API), permissions, App components, dynamic behaviors, system call's graph, etc. These ML-driven approaches offer unsurpassed solutions to resist prevailing malware in mobile applications. However, machine learning is prone to adversarial attacks [59, 144]. As discussed in Chapter 3 An adversary tries to mislead the classifier by carefully crafting the adversarial attacks on the detection system by changing the feature importance or data distribution. As a result, the learning system becomes ineffective in a production-ready environment and yields fewer True-positives, for example, a malware sample classified as benign by deteriorating the ML-model prediction. The study of adversarial settings around machine learning has been rising for the last few years. However, the exploration of adversarial perturbation to evade machine learning-based android malware detection are scarce. In this chapter, we evaluate the robustness of our image-based android malware detection system under various adversarial settings.

## 5.2 An Adversary-Aware and Robust Classifier for Android Malware Detection

Learning-based android malware detection systems prevent the smartphone user from illegitimate access. However, malware authors always try to intrude on the security fence, either through integrity attacks,i.e., forcing the malicious app to classify as benign, or (b) availability attacks,i.e., launching the denial of service attack by misclassifying benign apps as malware. Adversarial perturbations that are relatively easy in computer vision by changing a few pixel values and remain non-perceptible to the human eye are harder to translate in the malware domain with the constraint of preserving the original malware functionality. The robustness of our proposed system lies in the fact that adding perturbation to the transformed RGB image of a malware sample has a very high probability of violating the malware functionality constraint. To illustrate the robustness of the image-based malware detection system, we implemented state-of-the-art android malware detection techniques, including DroidDetector [168] and DeepClassifyDroid [170] in parallel to our approach. We have crafted two novel adversarial attacks that preserve the original functionality of malware after the alteration and evaluated our image-based malware detection system and these two approaches [168], and [170] for adversarial robustness.

## 5.3 Adversarial Attack Model

"If you know the enemy and know yourself, you need not fear the result of a hundred battles." (Sun Tzu, The Art of War, 500 BC). Since the inception of Android OS, it has been enjoying a monopoly in the smartphone marketplace. The widespread of Android OS is also drawn the attention of malware authors and threatens the android ecosystem. The existing state-of-the-art android malware detection systems use machine learning techniques and offer unsurpassed versatility for automatic detection with high accuracy. Despite yielding superior performance and high accuracy, these models are vulnerable to adversarial attacks. The existing threat models, which are designed for Windows and other desktop platforms [84, 87, 162] are not directly transferable to Android OS. Because the success of the attacking threat model highly depends on the context of the targeted model. We develop novel adversary attacking models for the Android-specific platform that are transferable to different feature representations.

### 5.3.1 Proposed Threat Model

Crafting a pro-active threat model for android malware detection is challenging due to various adversarial settings based on capabilities, level of access, and knowledge available to an adversary, as discussed in chapter 3. Moreover, preserving the malware functionality constraint for an adversary is mandatory. In adversarial attacks, an adversary attempts to manipulate the features until successful. An adversary evades the system's integrity (malware classify as benign in our case) or makes the system completely unavailable (DOS). Therefore, adversarial attacks belong to an optimization problem. The perturbation constraint, as discussed in section 3.7, crafting an adversarial example (AE) $x'$ from the original malware sample $x$ should not affect the semantic and intrusive functionality of the malware and the execution of the android application. However, the computer vision constraint is that the perturbation should be imperceptible.

Following the same guideline, we have crafted two novel attacks for malicious Android applications that can mislead the detection system and classify malware as benign. This work focuses on adversarial perturbations pertinent to integrity attacks in Whitebox settings. Our threat model has two configurations, (a) adding benign-only properties to malware APK without any modification and preserving original malware functionality. (b) append a portion of a benign image to a malware image using Generative Adversarial Networks (GANs). The

complete system diagram of our image-based malware detector, as discussed in section 4.3 and its impact under adversarial attacks, is depicted in Figure 5.1.

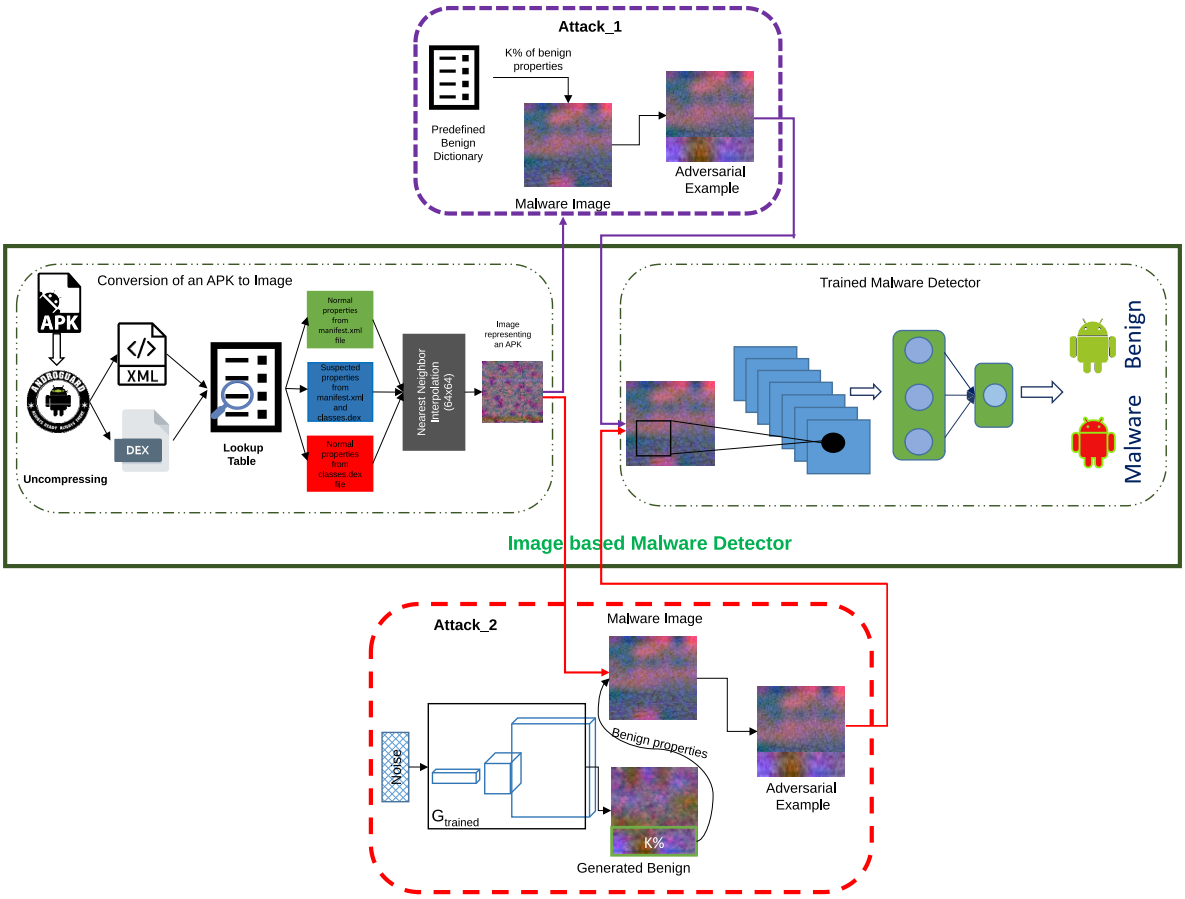



Fig. 5.1 Complete system diagram of image-based malware detection system under adversarial attacks

### 5.3.2 Attack 1: Appending Predefined Benign Properties

Contrary to other approaches like appending random bytes to a binary file or pixel values at the end of converted images lead to dummy values. We add benign properties to a malware sample using our predefined dictionary. Attack 1 guarantees that adding a benign property to a malware sample preserves the original malware functionality constraint. Moreover, it prevents modification/deletion of any existing feature from the malware sample. As malware APK is represented by an input image $x$ with $n$ numbers of features from manifest and dex

file, The trained classifier $M$ can be evaded by adding benign perturbation $\rho$ with percentage $k$ of a malware sample size $s$. Equation 5.1 controls the addition of perturbation $\rho$ to a malware sample.

$$x' = x + \rho \quad | \quad \rho = \frac{k * s}{100} \tag{5.1}$$

As discussed in section 4.3.1, our classifier is trained on resultant RGB images. The resultant image corresponds to the mapping of dex and manifest file information. The blue channel contains suspicious information from both files, the red contains Normal APIs, and the green channel contains Normal information from the manifest file. Attack 1 append benign properties with an equal distribution of perturbation $\rho$ on all channels using equation 5.2, i.e., $\rho/3$ % of predefined benign permissions and app components on the green channel, $\rho/3$ % of API method calls values at the red channel and $\rho/3$ % of benign permissions, app components, and API method calls on the blue channel.

$$Perturb_{channel} = \frac{\rho}{3} \ \forall \ channels \Rightarrow R, G, B \tag{5.2}$$

Malware samples before and after perturbation can be visualized in Figure 5.2.

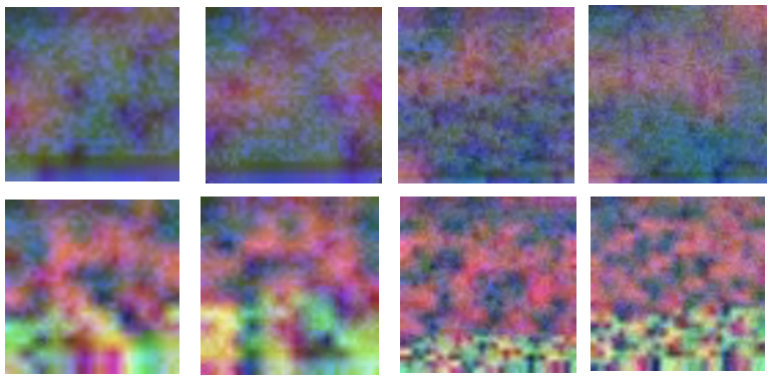


Fig. 5.2 Original Malware samples at top. Bottom row depicts perturbed malware images predicted as benign

### 5.3.3 Attack 2: Adding Proportionate of Benign Images Using Deep Convolutional GANs (DCGANs)

In the second attack, we use GANs to add a portion of the benign image to a malware sample to evade the learning-based classifier. Attack 2 also complies with the original malware

functionality constraint and guarantees that the appended image part belongs to actual benign properties. One can back-propagate using the lookup table (Pixels to benign properties mapping and vice versa). This attack involves a two-step process (a) To learn the benign distribution using only benign samples from the dataset. (b) Selection of benign pixels from the generated images (corresponds to benign properties) and append to a malware image to make it an adversarial example. We train DCGANs with benign APKs only and generate fake benign images. The generated benign samples were tested using CNN classifier as a sanity check and yielded 99.85% accuracy. In the second step, we select benign pixel values from these generated benign images and append them to a malware sample using equation 5.1 and 5.2. The pictorial view of our second attack is given in Figure 5.3.

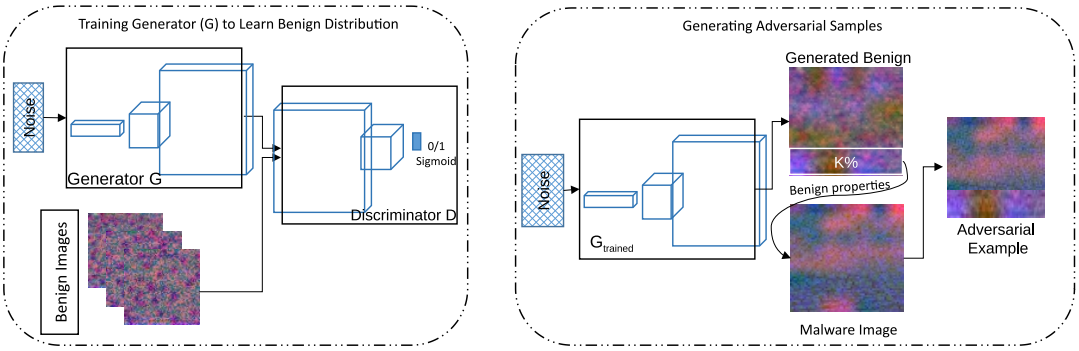



Fig. 5.3 Adversarial Examples Generation using GANs

## 5.4 Experiments & Robustness Evaluation

This section elaborates on the experimental setup, robustness, and evaluation of our adversarially robust image-based android malware detection system, adversarial attack generation, implementation of DroidDetector [168], and DeepClassifyDroid [170] and their robustness under our attack model.

For our experimentation and evaluation, we use the same AndroZoo [6] dataset repository comprising more than 10 million Android APKs (7.3M benign and 2.7M malware) and continuously growing and updating every week. This repository merges various other datasets and contains APKs available in Drebin, Genome, and Google Play. Further details about the AndroZoo dataset are explained in our previous work [104]. Our image-based

android malware detection system achieved an accuracy of 99.37% and FNR 0.8% on the legacy and new malware samples from the AndroZoo dataset as discussed in chapter 4 and [40]. For adversarial robustness of our image-based android malware detection system, we have also implemented DroidDetector [168] and DeepClassifyDetector [170] using the same AndroZoo dataset in Whitebox settings.

### 5.4.1 Implementation of DroidDetector and its robustness Evaluation Under Attack 1

DroidDetector [168] extracts 120 permissions by parsing the AndroidManifest.xml file and 59 sensitive API method calls from classes.dex file in the static phase. While in the dynamic phase, it monitors 13 actions only. Overall it extracts 192 features (179 static) for each APK and represents a binary vector where '1' indicates that the feature $x_i$ is called by the corresponding APK and '0' means not called. DroidDetector used Google-play and Genome Dataset and trained the Deep Belief Network (DBN) using benign and malware samples (880 APKs each). It achieved an accuracy of 96.76 % on both static and dynamic features. However, the accuracy was 89.03 % on 179 static features. Since we are interested in static features, we implemented DroidDetector with 179 static features comprising 120 permissions and 59 sensitive API method calls. We have trained the DBN using the AndroZoo dataset with the same parameter settings and architecture. We have achieved a bit higher accuracy of 94.5% using 880 samples (each benign and malware) and 95.75 % using 4K samples (each benign and malware).

In the next step, we have launched the adversarial attack on DroidDetector to check its robustness under adversarial examples. By exposing DroidDetector to Attack 1 as per section 5.3.2 we ensure that the malware functionality will remain untouched because we can only add predefined benign properties $B$ to the malware sample. To add a benign feature, $x_i$, we can set $x_i = 1$ if it is not already called by the android application (representing as a binary vector of 179 features), we cannot add extra malware functionality nor remove any existing feature as per equation 5.3

$$x_i = \begin{cases} 1 & \text{if } x_i \in B \text{ and } x_i = 0 \\ & \text{otherwise same} \end{cases} \tag{5.3}$$

It is imperative to highlight that we add benign properties with a protection level of "Normal," which means lower risk permissions. Attack 1 successfully evaded DroidDetector on both training and test samples as depicted in Table 5.1 which determines that Attack 1 is strong enough to mislead the DroidDetector(DroidD) on known and unknown samples.

Table 5.1 Robustness of our approach vs DroidDetector [168] under adversarial attack:1

| **Perturbation%** | **Accuracy** % | | **Evasion Rate** % | |
|---|---|---|---|---|
| | **DroidDetector [168]** | **Our** | **DroidDetector [168]** | **Our** |
| 0 | 95.75 | 99.37 | 0 | 0 |
| 10 | 78 | 89.2 | 17.75 | 10.5 |
| 11 | 72 | 88.4 | 23.75 | 10.97 |
| 12 | 59.3 | 88.4 | 36.45 | 10.97 |
| 13 | 45.7 | 86.7 | 50.05 | 12.67 |

### 5.4.2 Implementation of DeepClassifyDroid and its robustness Evaluation Under Attack 1

For evaluation, we have implemented another CNN-based Android malware detection system called DeepClassifyDroid [170]. The system was trained on the Drebin dataset containing Genome dataset samples also. They used four feature sets: permissions, filter intents, API calls, and constant strings. They represent the android APK as a binary matrix of shape $n \times p$ of size (256x256) where 1 indicates the corresponding feature $x_i$ is called while 0 means not called by the application. We have implemented the DeepClassifyDroid with similar architecture and parameter settings. We have collected 65536 (256*256) features based on their calling frequency in 50K APKs from the AndroZoo dataset. We select the top 60500 API calls (at least called by 12 APKs), 500 permissions (at least called by 8 APKs), 3936 constant strings (at least called by 7 APKs), and 600 intents (at least called by 5 APKs). The CNN is trained on 11K samples (5500 each benign and malware) from the AndroZoo dataset and achieved the same level of accuracy 97.6% (DeepClassifyDroid 97.4%).

As a next step, we evaded the DeepClassifyDroid by Attack 1. We have created the predefined dictionary of benign properties from the pool of 50K APKs from the AndroZoo dataset. A feature can be called Bening Property $B$ if it is called at least $x$ times by a benign APK but never called by a malware APK. Following the Attack 1 constraint of preserving

malware functionality, we cannot modify nor delete any property. We add a benign property to a malware sample using the equation 5.3. However, the maximum perturbation that can be added to a malware sample is bound by the equation 5.1. Table 5.2 presents that our image-based android malware detection system is adversarially more robust as compared to DeepClassifyDroid (abbreviated as DCD).

Table 5.2 Robustness of our approach vs DeepClassifyDroid [170] under attack:1

| **Perturbation%** | **Accuracy** % | | **Evasion Rate** % | |
|---|---|---|---|---|
| | **DeepClassifyDroid [170]** | **Our** | **DeepClassifyDroid [170]** | **Our** |
| 0 | 97.60 | 99.37 | 0 | 0 |
| 1 | 78.72 | 95.87 | 18.88 | 3.5 |
| 3 | 72.60 | 97.51 | 25.00 | 1.86 |
| 5 | 70.16 | 94.57 | 27.44 | 4.8 |

### 5.4.3 Attack 2 Effectiveness

We have evaded our image-based malware detection system, DroiDetector, and DeepClassifyDroid using the second attack model discussed in section 5.3.3. Attack 2 is crafted using DCGAN, directly applicable to the image domain. However, for binary vector space like DroidDetector, we have adjusted it using a lookup table (Pixel to corresponding benign properties). DCGAN is an extension of GAN to use convolutional and convolutional-transpose layers instead of fully connected layers. The generator of DCGANs uses a convolutional transpose layer, and the discriminator uses a simple convolutional layer.

Unlike the conventional pooling layer in CNN to downsample the input image, the Generator in GANs needs an upsampling layer. We have used the Conv2DTranspose layer provided by Keras to upsample the input. We trained the Generator $G$ using benign images from the real data $P_r$ to learn benign distribution. Both Generator and Discriminator model contains four Conv2D layers with 'LeakyReLU' (alpha=0.2) as an activation function and Batch Normalization. As a result, the $G_{trained}$ generated benign samples $P_g$ with 99.98 % accuracy validated by our image-based malware classifier. We use the 'min - max' loss function derived from cross-entropy to quantify the distance between real $P_r$ and generated $P_g$ distributions.

The properties generated by GANs are pixel values on RGB channels and can be easily translated to binary space vector using the lookup table [41] to evade DroidDetector and DeepClassifyDroid. For instance, GAN's generated pixel value 200 corresponds to the android.permission.camera. Attack 2 adds benign properties to a malware sample using equation 5.3 and ensures it preserves the original malware functionality constraint. The robustness of our approach versus DroidDetector (DroidD), and DeepClassifyDroid (DCD) under Attack 2 can be analyzed Table 5.3 and Table 5.4.

Table 5.3 Robustness of our approach vs DroidDetector [168] under Attack:2

| **Perturbation%** | **Accuracy** % | | **Evasion Rate** % | |
|---|---|---|---|---|
| | **DroidDetector [168]** | **Our** | **DroidDetector [168]** | **Our** |
| 10 | 85.95 | 91.37 | 9.8 | 8.0 |
| 11 | 80.05 | 86.27 | 15.7 | 13.01 |
| 12 | 80.05 | 83.97 | 15.7 | 15.4 |
| 13 | 82.08 | 85.7 | 19.6 | 13.67 |

Table 5.4 Robustness of our approach vs DeepClassifyDroid (DCD) [170] under attack:2

| **Perturbation%** | **Accuracy** % | | **Evasion Rate** % | |
|---|---|---|---|---|
| | **DCD [170]** | **Our** | **DCD [170]** | **Our** |
| 1 | 74.6 | 97.57 | 23.0 | 1.8 |
| 3 | 68.26 | 95.37 | 29.34 | 4.0 |
| 5 | 63.39 | 92.87 | 34.21 | 6.5 |

## 5.5 Summary

The smartphone is an essential device to human life endowed with various sensitive data and personal information that needs paramount security from malware attackers. Learning-based malware detection systems have achieved unmatched recognition to protect smartphone users from illegitimate access. However, learning-based systems are vulnerable to adversarial attacks. Malware authors can add non-perceptible perturbation to real input and mislead the target classifier. In this work, we have shown the robustness of our image-based android malware detection system, which is resilient to adversarial attacks compared to other learning-based approaches. We have implemented two more approaches in parallel to our image-based malware detection system for the evaluation. The results illustrated the resilience of all

three android malware detectors under two novel adversarial attacks. Our system proved more robust against adversarial perturbations and only evaded by 13% while the evasion rate was more than 50% for DroidDetector and DeepClassifyDroid. In the future, we will craft more diverse adversarial attacks in Blackbox to evaluate its inherent defense mechanism. Moreover, we will measure the impact of adversarial perturbation at a different location of the converted image and provide a defense strategy to minimize the 13% evasion rate.

# Chapter 6

# Robustness Evaluation and Modified FGSM Attack on Malware Detection Systems

## 6.1 Overview

In addition to the monopoly of Android OS, the post-COVID world has compounded the dependencies on the smartphone for numerous routine tasks. The widespread proliferation of android OS and increased dependencies allure the malware designers to threaten the prosperous mobile ecosystem. Learning-based classifiers are in action to battle these sophisticated and polymorphic methods of malware creation. Despite yielding superior performance for distinguishing malicious and benign content, learning-based techniques are vulnerable to adversarial attacks, as discussed in Chapter 3 and 5. Malware authors are targeting the adversarial weakness to evade learning-based classifiers consonantly. Robust Android malware detection systems against adversarially modified examples are the utmost requirement for the Anti-malware industry and cybersecurity community. This chapter addresses the effects of a novel adversarial attack on a visualization-based android malware detection system using a modified Fast Gradient Sign Method (FGSM). Moreover, our research work is a stepping stone for establishing the benchmarks for adversary-aware android malware detection systems. Previously, various studies have proposed attacks on different models using different datasets and claim high evasion rates. Our experimental results reveal that our image-based android malware detection system is robust against adversarial attacks.

Furthermore, our detection system (without any additional defensive strategy) evaded with a minimum misclassification rate in contrast to the state-of-the-art deep learning-based approaches that employ defensive methodologies.

## 6.2 Smartphone; A Digital personal Assistant and Adversarial Android World

Smartphones have emerged as a vital piece of daily life tasks for accessing valuable electronic services such as online shopping, mobile banking, food ordering, governance, eHealth services, and many more. According to [159], the total worldwide users of smartphones reached 6.378 billion, meaning that 80.69% of the world population carries a smartphone. The smartphone is a digital assistant to organize work, routine tasks, studies, and private life. These smartphones are loaded with sensitive data that need paramount security for mobile computing. The Android smartphone is a popular choice due to its multifaceted offering and captures 85% of the global market [8]. Being an open-source operating system (OS), it attracts application developers to build legitimate mobile applications. On the flip side, Android OS is a favorite swarm for malware authors due to its popularity, open-source availability, and intrinsic infirmity to access internal resources.

Machine learning-based approaches have been offered unprecedented flexibility to combat malicious activities on static and dynamic analysis routes. However, learning-based classifiers easily evade with adversarial examples, as discovered by [144]. Therefore, deploying a learning-based detection system in security-sensitive fields must be contemplated. An adversary can deceive the detection model by crafting minute, subtle perturbations known as Adversarial Examples (AE's). The study of adversarial learning has been a hotspot research area and mainly focuses on images due to the easy visualization of perturbations [59, 103, 32]. However, research evaluating the impact of adversarial perturbation on image-based learning classifiers for android malware detection systems is still quite scarce. Few researchers have analyzed the adversarial evasion in PDF malware detection [92, 155], Windows executable [84, 87], and android malware detectors but limited their research to binary space [31, 166, 62].

## 6.3 Visualization-based Android Malware Detection Systems under Adversarial Attacks

Adversarial malware detection is a vast field and needs extensive review. This section presents a concise review that accords to visualization-based android malware detection systems and the behavior of learning-based techniques under various adversarial settings.

Security experts usually analyze the malware through static, dynamic, or hybrid analysis methods. Machine learning-based techniques have notably automated the static/dynamic malware analysis by representing the Android APK (Application Packaging Kit) as a binary feature vector [170, 82, 12, 168, 93]. However, a few researchers have proposed visualization-based techniques to counter malware. Such approaches transform the bytecode into a gray-scale image and train a learning-based classifier for malware detection [79, 66, 39]. The main concerns with such image-based approaches are their blind transformation, poor detection accuracy, and excessive false alarms against sophisticated malware.

Researchers and adversaries are always in a never-ending race to efficiently detect malware and evade detection systems. Very few adversarial approaches have successfully evaded android malware detection in binary feature space with limited perturbation options of addition and deletion [31, 166, 62]. Several studies present evasion and poisoning attacks based on the knowledge available to an adversary at different stages of ML classifier construction. Srndic *et al.* [92] presented adversarial attacks in Whitebox settings for windows executable with excessive insights available to an adversary. Sharif *et al.* [131] attacked the facial recognition system having access to feature space and trained parameters of the target model. Xu *et al.* [162] analyzed the behavior of evasion attacks in Blackbox settings on Linux OS with limited information. In this scenario, the classifier assigns a confidence score as a continuous value against a query sample. Such attacks can be prevented by simply hiding the confidence score. A few studies poisoned the Windows PE (portable executable) with adversarial examples by appending/altering the bytes in the non-reachable area of the binary file without disrupting the original functionality [84, 87, 88].

A few studies used problem-driven approaches in Blackbox settings to generate adversarial attacks. Song *et al.* [138] introduced the random generation of macro actions recursively to evade the target classifier. Following are the examples of macro-actions (a) Appending random bytes at the end of binary, (b) adding bytes in unused space of the binary, (c) adding a new section, and (d) removal of debugging information. Furthermore, the micro-actions subset of macro-actions provides more insights into evasion for a specific action. For exam-

ple, appending byte action can be implemented by appending one byte at a time. The said method yielded effective results in dynamic and static analysis of Windows PE, but their adoption of Android APK is questionable. Pierazzi *et al.* [116] proposed an evasion attack in Blackbox settings for Android OS. The author uses Drebin dataset [13] to inject the benign code blocks to disrupt the feature importance and mislead the classifier.

Several studies present defensive strategies to counter adversarial attacks but primarily target to image domain. Carlini *et al.* [25] empirically demonstrated that the efficient defensive technique could not combat the carefully crafted adversarial perturbation in specificity attacks. Adversarial training is effectively used by [89, 144, 148] to improve the generalization capability of the classifier and an adequate strategy against evasion attacks. Conversely, the augmentation of training data with adversarial examples is a computationally expensive process. The author in [148] presented an ensemble-based adversarial training to make the classifier robust against Blackbox settings. Papernot *et al.* [111] invented distillation as a defensive strategy to counter adversarial attacks by training the Artificial Neural Network (ANN) with soft labels. However, [61] and [140] evaluated the impact of distillation in malware detection and showed that distillation is more appropriate in the continuous domain but ineffective in the binary space. Wang *et al.* [151] introduced an approach to randomly disable features in training and inference time to make gradient computation expensive. This approach disables a random set of features at training and inference time. Bhagoji *et al.* [19] introduced a dimensionality reduction approach to enhance the security of the learning classifier by applying linear transformation with principal component analysis (PCA). Similarly, a few other approaches are proposed; for instance, *Zhang et al.* [169] enable adversary-aware feature selections as an optimization function that aims to gain higher generalization and detection accuracy and more resilience against adversarial attacks.

We have built an image-based android malware detection system using an intelligent APK representation and adversary-aware feature mapping [40] also discussed in chapter 4. We proposed two adversarial attacks to evade the image-based android malware detection system and evaluate its robustness against adversarial attacks [43] also explained in chapter 5. Further extending our research, we develop another novel adversarial attack using modified FGSM and plan to analyze its impact on our image-based detection system. Moreover, many researchers claimed different evasion rates using varying datasets, classifiers, feature sets, and threat models. There has been no benchmarking to evaluate the impact of these attacks, the effectiveness of defensive strategies, and cross-classifier comparisons.

## 6.4 Proposed Methodology for Modified FGSM Attack

This section highlights the new FGSM attack model for the visualization-based Android malware detection systems. It also highlights defensive strategies to combat adversarial attacks and benchmark state-of-the-art learning-based approaches.

An adversary always explores vulnerabilities of the learning classifier by adding non-perceptible perturbations to compromise the integrity or availability of the system. Generating adversarial examples is easy in images by altering pixel values, but the malware domain has specific constraints due to its binary feature representation. Alteration of a boolean feature can violate the malware functionality constraint or disrupt the original functionality of the application. This section extended the evaluation of adversarial robustness and proposed a new modified FGSM-based attack on the proposed android malware detection system [40].

Preserving the malware functionality constraint, we have presented the modified FGSM method to generate AEs. The modified FGSM method guarantees that the perturbed value belongs to a benign property instead of a dummy or random value, unlike the computer vision (CV) field and existing evasion attacks. Moreover, FGSM maximizes the cost function by jumping to the next benign value in the direction of the most significant loss. We have evaluated our image-based android malware detection system [40] under modified FGSM attack compared to visualization-based deep learning classifier for Android malware detection [171]. Furthermore, this study pioneers benchmarking of Adversary-aware android malware detection systems. In addition to FSGM, we also evaluated the impact of our previously proposed adversarial attacks [43] by employing defensive strategies to state-of-the-art learning-based Android malware detection systems [168, 170]. Our image-based adversary-aware android malware detection system yielded a minimum evasion rate without incorporating any defensive strategy compared to other learning-based techniques with defensive policies.

### 6.4.1 Background of FGSM Attack

Originally, Szegedy [144] diagnosed the weak immunity of Artificial Neural Networks (ANN) against adversarial attacks in image recognition and detection. Numerous researchers have examined the behavior of ANN under adversarial attacks in Android malware detection. However, recent studies have limited the binary representation of Android APK. In [43], we have evaluated the robustness of our image-based android detection system under novel

adversarial attacks. A quick pictorial recap of our image-based android malware detection system under novel adversarial attacks presented in chapter 4 and chapter 5 is shown in Figure 6.1 to establish the context with FGSM. Extending our research to adversarial android

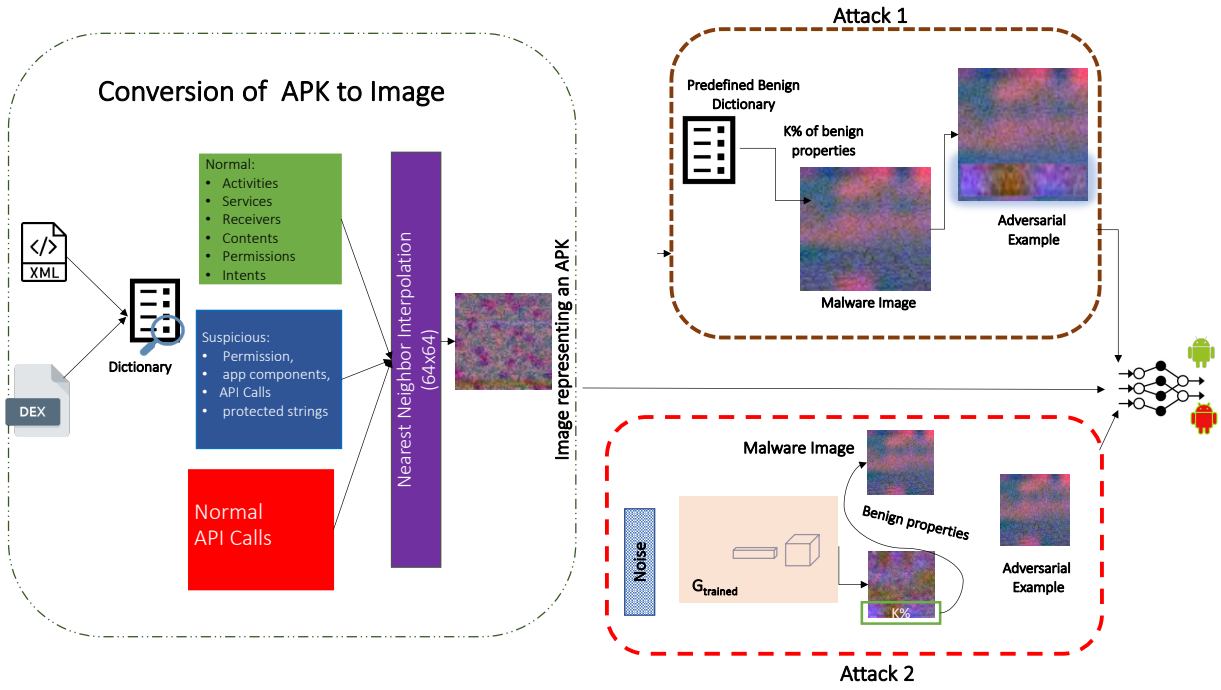



Fig. 6.1 Recap: Image-based Android Malware detection system under Adversarial Attacks

malware detection, a modified FGSM method to evade different android malware detection systems is proposed here. The adversarial examples in the computer vision domain are designed to make perturbations to original input imperceptible to the human eye and strong enough to deceive the trained classifier. Various techniques have been proposed to craft adversarial attacks in the computer vision field like JSMA (Jacobian-based saliency map attack) [156], C&W attacks [26], Deepfool [103], FGSM [68]. For malware, the original malware functionality should remain intact as opposed to the visual perception of the human eye in computer vision.

In general, FGSM [59] operates on the linearity of ANN and crafts an adversarial example. Let us now consider a malware Android APK sample $x$ added with a tiny perturbation $\rho$, deceiving the model and yielding a false output label for malware sample $x$ given in the equation.

$$x + \rho = x^{/} \tag{6.1}$$

Let us consider a dot product between adversarial example $x'$ and weight vector $w_1, .....w_n$ as given in eq 6.2.

$$w^n \cdot x' = w^n \cdot x + w^n \cdot \rho \tag{6.2}$$

Where $x'$ is the adversarial sample and $w^n\rho$ indicates the activation part of ANN. The minute added perturbation $\rho$ triggers the activation function by $w^n \cdot \rho$. This can be further increased by the direction of $\rho = sign(w)$. Since the perturbation $\rho$ increases linearly with the dimension $n$ of $w$ and impacts the activation function to change and validates "a minute change to an original input sample $x$ in high dimensions [which] can bring significant change in output." Adversarial malware examples computed by FGSM in the malware domain are given in equation 6.3.

$$x' = x + \varepsilon * sign(\nabla_x J(\theta, x, y)) \tag{6.3}$$

Where $x$ is the original input sample, $x'$ is the adversarial example, $\varepsilon$ signifies the amount of noise to be added to a classifier, $y$ is the original class label for input $x$, $J(\theta, x, y)$ is the loss function, and $\theta$ indicates the model parameter.

### 6.4.2 Modified FGSM Attack for Image-based Android Malware Detection Systems

The problem with the abovementioned techniques like JSMA, C&W, Deepfool, and standard FGSM does not guarantee that the modified (perturbed) values belong to a benign property. These techniques are missing the context of APK property (either benign or malware) and focus on generating a combination of dummy values (neither benign nor malware) to evade the detection system. Another fact about the aforesaid techniques is the normalization constraints, i.e., $L_2\,Norm$ or $L_\infty\,Norm$ that distinguish between the original image and adversarial image in real-life objects. For instance, let us consider an image of a cat have one of the pixel values 220 (white color). It can only perturb with a lower perturbation value from 220 to 221 (white) to look the same as per perturbation constraint in the CV domain. On the flip side, if the image is perturbed from a pixel value of 220 to 10 (black), it will be obvious to the naked human eye. Unlike the CV domain, it does not matter in the malware sample if we replace a pixel value of 220 with another benign property representing a pixel value of 10. The visual perturbation constraint does not hold for the malware domain. However, the perturbation should guarantee that the original malware functionality remains unchanged.

The modified FGSM attack comprises a two-step process. The first step involves the generation of fake benign APKs using Deep Convolutional Generative Adversarial Networks (DcGANs). The system learns the benign distribution from the AndroZoo dataset [6], and the generated APKs are validated using our image-based android malware detection system [40]. A small portion of newly GANs generated APK (few pixel values) is appended with an original malware sample to craft an adversarial example by preserving the malware functionality constraint. The details of the abovementioned first step are explained in Algorithm 2.

**Algorithm 2** Crafting adversarial malware sample using DcGANs

**Step I**
**Require:** $x^b$ input sample representing an image of benign APK, $D(x_i^b, ....x_n^b)$ the dataset of benign APKs in AndroZoo
**Require:** $m$ is the batch size, $n = len(D(x_n^b))/m$ number of batches, and $q$ number of epochs.
**Ensure:** $G_{trained}$ Generator to produce fake APK images containing benign properties.
$\mathbb{P}_r$ : real benign distribution
$P(z)$ : Prior noise sample distribution
$G$ : Generator Model
$D$ : Discrimantor Model
**for** $l = 0, ....., q$ **do**

    **for** $t = 0, ....., n$ **do**
        Sample $\{x^i\}_{i=1}^m \sim \mathbb{P}_r$ ▷ A mini batch from real benign distribution
        Sample $\{z^i\}_{i=1}^m \sim P(z)$ ▷ a mini batch from noise
        $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m [\log D(x^i) + \log(1 - D(G(z^i)))]$ ▷ Update the discriminator
    **end for**
    Sample $\{z^i\}_{i=1}^m \sim P(z)$ ▷ A mini batch from noise
    $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m [\log(1 - D(G(z^i)))]$
**end for**
**Step II**
**Require:** $k$ perturbation percentage, $B_{prop}$ benign properties from generated APK image, $X_g$ fake benign image, $X$ input image representing malware APK, $\rho$ amount of perturbation.
**Ensure:** $X'$ Adversarial malware sample
$X_g \leftarrow G_{trained}(z)$ ▷ Generate a fake benign image from Step-I
$\rho \leftarrow X_g \times \frac{k}{100}$ ▷ K is the % of benign properties selected from a benign image
$X' \leftarrow X + \rho$ ▷ Where $\rho$ is bound by the size of original malware sample $X$,i.e., $\rho \leftarrow \frac{Size(X) \cdot k}{100}$
**End**

In the second step, we can further strengthen our threat model using FGSM. The system could not further modify the appended benign properties in Attack-1 and Attack-2 presented

in chapter 5. To maximize the loss for the perturbed malware samples, we can make our threat model more effective by modifying the appended pixel values in Attack1 and Attack2 with modified FGSM. The proposed system does not nudge the pixel value with $\pm\,\varepsilon$ but jumps to the next benign property generated by GANs in the gradient direction. In this way, the system guarantees that the value added to malware samples is purely a benign property and is not a dummy value like [3, 96, 117]. Standard FGSM adds or subtracts pixel values according to the gradient sign from a given image. The standard FGSM does not apply to our approach because it may lead us to a dummy value. We cannot add/subtract a value to a pixel value in a given image (a pixel value in our approach represents an android property). Despite adding/subtracting random or dummy values, the system chooses the next benign property generated by GANs or offered in our exhaustive dictionary [41] according to the gradient sign. The overarching systematic diagram for modified FGSM-based attack on image-based Android malware detection system is given in Figure 6.2.

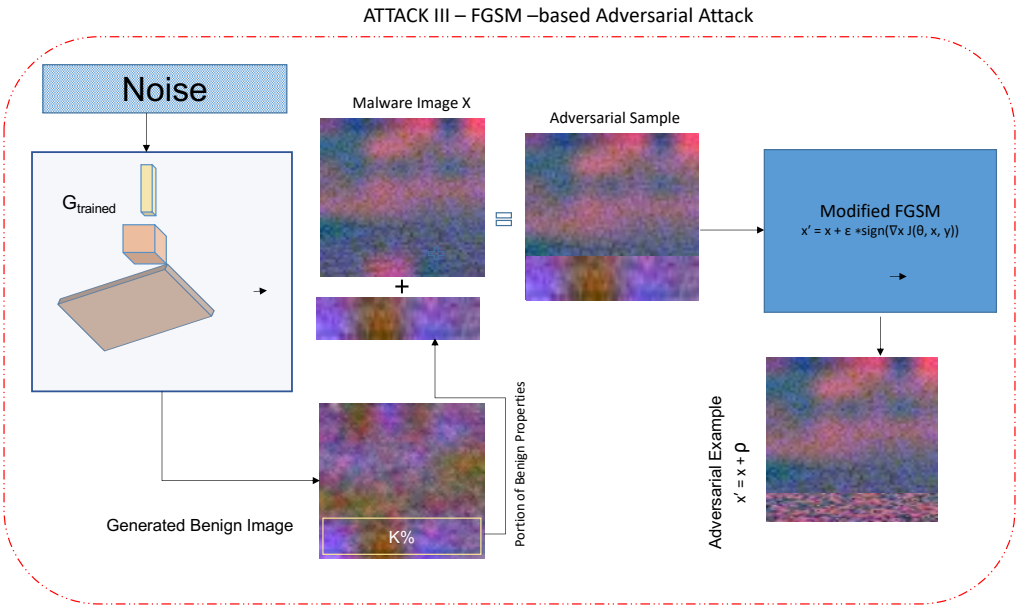



Fig. 6.2 Block Diagram of FGSM-based Adversarial Attack for Image-based Android Malware Detection System

Let us consider the example mentioned in Figure 6.3 to understand the modified FGSM. The first array contains the benign properties generated by DcGANs using Algorithm 2 given as:

Benign Properties generated by DcGANs:

$[110, 213, 99, 75, 145, 245, 254, 183, 190, 100, 22, 60, 112, 165, 215]$

Sorted ordered:

$[22, 60, 75, 99, 100, 110, 112, 145, 165, 183, 190, 213, 215, 245, 254]$

Chapter 4 presents the complete transmutation process of Android APK into an RGB image, also explained in [40]. The resultant RGB image can be visualized as pixel values as depicted in Figure 6.3. Each pixel value represents an Android application property from App components, permissions, or API calls. The pixel values in the original image representing an APK can be altered using the gradient sign by jumping to the next benign property from the sorted array as depicted in Figure 6.3. For instance, in this example first pixel value 200 represents a benign property and, as per the direction of the gradient, needs to move in the negative direction. Therefore, the pixel value 200 should be perturbed with a benign property from the sorted array with the pixel value 190. Algorithm 3 describes the pseudo-code of the above-mentioned modified FGSM attack for an image-based android malware detection system.

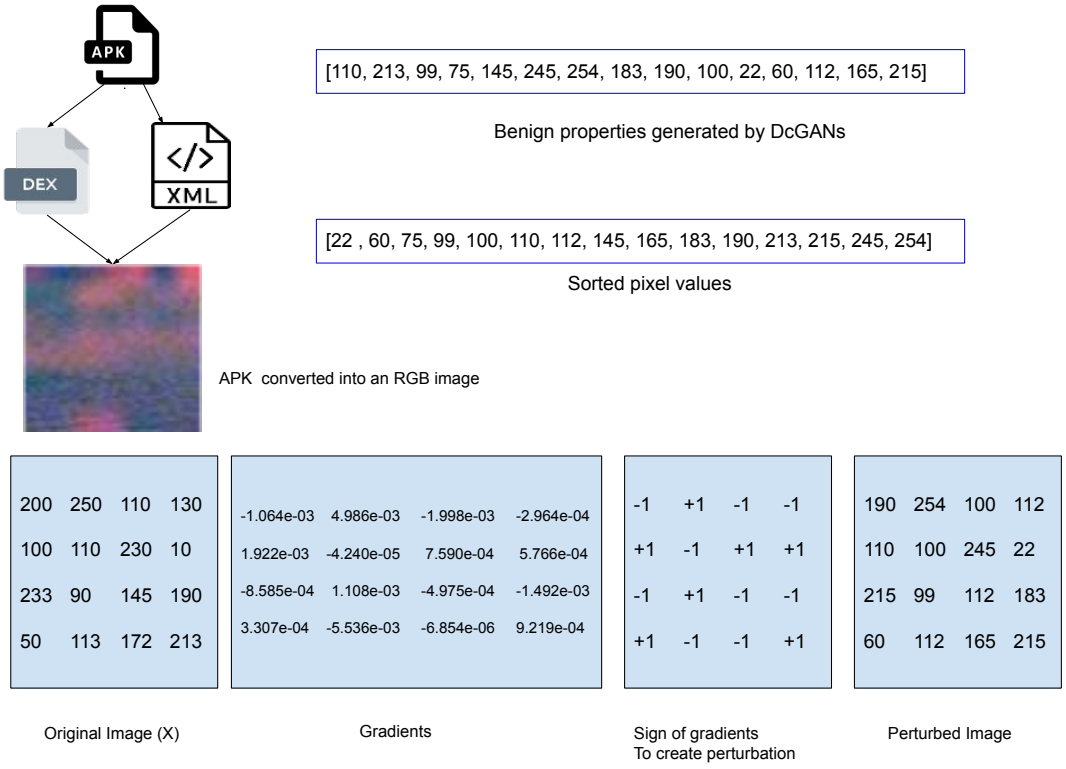



Fig. 6.3 A demonstration of perturbing a malware sample using modified FGSM

**Algorithm 3** FGSM-based adversarial attack for image-based Android malware detection system

**Require:** $BP$ benign properties, $X$ input image representing malware APK, $\rho$ perturbation, $k$ amount of perturbation to be added (number of pixels).
**Ensure:** $X'$ Adversarial malware sample

**Begin**
Dictionary $D \leftarrow BP_{DcGan} + BP_{attack-1}$ ▷ Exhaustive dictionary of Benign properties from Attack-1 and Attack-2 (generated by DcGANs)
$\nabla_x J(X, Y_{true})$ ▷ Take a gradient of loss w.r.to $X$
$Sign(\nabla_x J(X, Y_{true}))$ ▷ Take sign of gradient
$X_{ij}^{-ve} \leftarrow X[Sign(\nabla_x J(X, Y_{true}))][-k:]$ ▷ Store indexes with $-ve$ gradient as per size of $k$.
$X_{ij}^{+ve} \leftarrow X[Sign(\nabla_x J(X, Y_{true}))][-k:]$ ▷ Store indexes with $+ve$ gradient as per size of $k$.
$\rho \leftarrow [BP_{DcGan} < X_{ij}^{-ve}]$ ▷ Assign next benign property in $-ve$ direction
$\rho \leftarrow [BP_{DcGan} > X_{ij}^{+ve}]$ ▷ Assign next benign property in $+ve$ direction
$X' = X + \rho$
**return** $X'$
**End**

### 6.4.3 Benchmark Learning-based Android Malware Detection Systems Under Adversarial attacks

The biggest problem in adversarial learning for android malware detection is establishing a benchmark, baseline, and ground truth. There is neither a consistent dataset nor a learning-based classifier available to evaluate the attack's impact and robustness of a classifier against adversarial perturbations. Different studies claim a high evasion rate against popular classifiers, but no mechanism exists to inter-evaluate them on the same dataset and similar settings. Therefore, we have used the same AndroZoo dataset and parameter settings for all top classifiers to evaluate their robustness against adversarial perturbations. Moreover, we have also generated a massive dataset for adversarial APKs using our threat model [43] and online available [41] for the research community. The generated data can be converted from images to APK and vice versa using lookup table [41]. We have also implemented distillation as a defense to these state-of-the-art classifiers and evaluated attack effectiveness and robustness. This is available for the research community to benchmark their classifiers against novel adversarial attacks, defensive methodologies, and inter-evaluation of attacks, classifiers, and defenses [41]. In the future, we plan to extend this benchmarking setup with the help of the research community to keep adding more attacks in Blackbox and Whitebox settings, learning-based SOTA classifiers for android malware detection, new defensive strategies, and inter-evaluation using this benchmark.

## 6.5 Experiments for Robustness Evaluation

This section first provides the details of the dataset, experimental setup, and robustness evaluation of our image-based android malware detection system under modified FGSM attack. It also circumvents the analysis of other learning-based approaches under adversarial attacks. This section also presents a benchmark to evaluate our image-based android malware detection system, which sets itself apart from state-of-the-art learning classifiers with and without defensive strategies.

### 6.5.1 Dataset Details

We are using the AndroZoo [6] for all experiments, evaluation, and benchmarking against other learning classifiers for android malware detection. The AndroZoo data set is a growing dataset that updates daily and contains 18 million Android APKs. The dataset is a super set for all other Android APKs used in various other datasets in the literature, such as Drebin [13], Genome [57], and Google Play. The details of AndroZoo already mentioned in chapter 4 and 5.

### 6.5.2 Attack-3 Evaluation: Robustness of Image-based Android Malware Detector under Modified FGSM Attack

The proposed system uses DcGANs to generate fake benign images. DcGAN is a variant of GANs that uses a convolutional transpose layer in the generator part and a convolutional layer in the discriminator part instead of a fully connected layer. The system trained the Generator $G_{trained}$ using benign APKs (images) from the AndroZoo data of real distribution $P_r$. The system yielded benign samples $P_g$ as an output using $G_{trained}$. The generated benign images are cross-validated through our image-based malware detection system [40] with an accuracy of 99.98 %. The system chooses a $k$ proportion of the fake benign image and appends it to the original malware sample $x$. The system modifies only the appended portion using modified FGSM as given in Algorithm 3. $k$ indicates the total amount of perturbation $\rho$ to be added to the original malware sample $x$, $s$ is the application size and bounded by equation 6.4. We have used different sizes of k to evade our image-based android malware detection system, DCD (DeepClassifyDroid) [170] and other image-based malware classifier [171]. If the size of $k$ is 3% to perturb the original malware sample $x$, that means 1% perturbation $\rho$ is allowed

on each channel. Every channel has 4096 $(64 \times 64)$ pixel values which means a maximum of 14-pixel values can be altered on each channel to generate an adversarial malware sample $x'$. Table 6.1 enumerates the different perturbation percentages we have used to evade various classifiers.

$$x' = x + \rho \quad | \quad \rho = \frac{k * s}{100} \tag{6.4}$$

Table 6.1 Perturbation percentages and corresponding pixel values altered for Attack-3

| **Perturbation $\rho$%** | **Per channel $\rho$%** | **# Pixels per channel** | **# pixel per image** |
|---|---|---|---|
| 0.5 | 0.17 | 7 | 21 |
| 1 | 0.34 | 14 | 42 |
| 2 | 0.67 | 27 | 81 |
| 3 | 1 | 41 | 123 |
| 6 | 2 | 82 | 246 |
| 9 | 3 | 123 | 369 |
| 12 | 4 | 164 | 492 |

To evaluate its evasion rate, we have exposed our adversarially robust image-based android malware detection system to the modified FGSM called Attack-3. The detailed experimental setup, parameter settings, and other configurations of our image-based Android malware detection system are discussed in chapter 4 and presented in [40]. The detection system yielded 99.37% accuracy and 0.8% False Negative Rate (FNR) on the AndroZoo dataset. To draw a fair comparison concerning robustness under Attack-3, we have also implemented other visualization-based approach [171] alongside side DeepClassifyDroid (DCD) [170].

We achieved the same accuracy and F-1 score by implementing the visualization-based approach [171] on the AndroZoo dataset. The system visually explores the disassembly (dex) file to identify the malware. The implemented approach first converts the dex file into smali code and extracts 255 opcodes ranging from 0x00 to 0xFF. Extracted opcodes are converted into pixel values and maps on the Red channel. The system performs the coloring of risky APIs and maps sensitive APIs on the Green channel and risky APIs to the Blue channel. The system merged all three channels to generate a featured image and used a machine learning model to identify family fingerprints. For evaluation, the implementation of DeepClassifyDroid (DCD) [170] using the AndroZoo dataset is similar as presented in chapter 5 and published in [43] for the comparisons of Attack-1 and 2. DCD uses CNN

as a classifier for android malware detection that represents an Android APK as a 2D binary vector (single channel image). By converting it into a 1D binary vector, 1 indicates the corresponding feature $x_i$ is called by APK, while 0 means missing in this application. Table 6.2 shows the result for evasion rate under modified FGSM attack. The results indicate that the modified FGSM attack is strong enough to deceive image-based android malware detection systems. Our approach's evasion rate is meager compared to other image-based detection systems. Moreover, as the size of $k$ for perturbation $\rho$ increases, the attack yields a high misclassification rate against [171] as compared to our detection system [43].

Table 6.2 Robustness of our approach vs [171] and DCD [170] under modified FGSM (Attack:3)

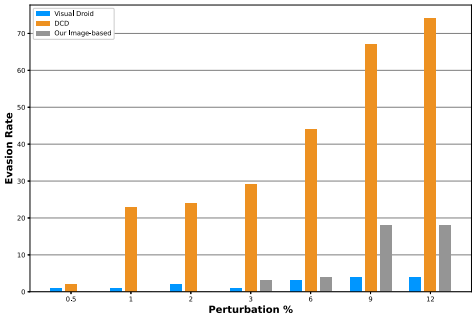
| **Perturbation%** | **# Iterations** | **Evasion Rate %** | | |
|---|---|---|---|---|
| | | **[171]** | **DCD [170]** | **Our [43]** |
| 0.5 | 1 | 1 | 2 | - |
| | 5 | 1 | 5 | - |
| | 10 | 2 | 9 | - |
| 1 | 1 | 1 | 23 | - |
| | 5 | 1 | 26 | 1 |
| | 10 | 3 | 29 | 1 |
| 2 | 1 | 2 | 24 | - |
| | 5 | 3 | 25 | 4 |
| | 10 | 6 | 25 | 4 |
| 3 | 2 | 1 | 29 | 3 |
| | 5 | 5 | 33 | 6 |
| | 10 | 9 | 39 | 11 |
| 6 | 1 | 3 | 44 | 4 |
| | 5 | 19 | 53 | 7 |
| | 10 | 49 | 67 | 13 |
| 9 | 1 | 4 | 67 | 18 |
| | 5 | 36 | 69 | 18 |
| | 10 | 72 | 83 | 19 |
| 12 | 1 | 4 | 74 | 18 |
| | 5 | 57 | 85 | 18 |
| | 10 | 83 | 89 | 21 |

### 6.5.3 Results and Discussion

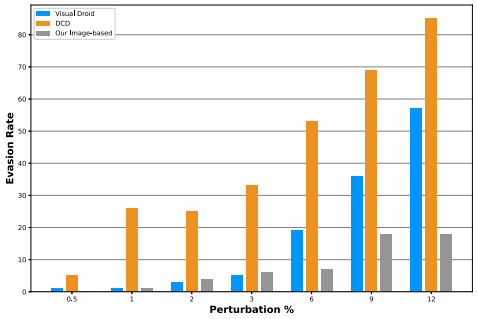
Most of the FSGM-based studies presented in the literature add random bytes. The randomly appended bytes are then modified using the FGSM method over the number of iterations to maximize the loss. These studies do not guarantee that the modified portion of the resultant adversarial sample $x'$ belongs to an application property or Benign properties. Therefore, it can be valid in theory but cannot create a practical attack. Our modified FGSM method adds only the Benign-application property using GANs and jumps to the next benign property using the gradient direction in subsequent iterations instead of nudging the pixel value as discussed in section 6.4.2. The results illustrated in Fig 6.4 indicate that the modified FGSM method applies to visualization-based Android malware detection systems and can evade image-based techniques strongly. We have compared the results with two other visualization-based approaches, DeepClassifyDroid (DCD) [170] and VisualDroid [171]. The VisualDroid [171] is purely an image-based technique that maps the API packages, opcodes, and risky APIs on color images and then uses CNN for malware and family classification. DCD [170] used four feature sets: permissions, filter intents, API calls, and constant strings. They represent an APK as a binary matrix of shape $n \times p$ of size $256 \times 256$, equivalent to a single image channel of size $256 \times 256$. Figure 6.4.a shows that the Attack-3 (Modified FGSM) has maximum evasion for DCD, then our image-based detection system, and then VisualDroid under the first iteration. However, Figure 6.4.b and c indicate that the evasion rate of our image-based detection system is only 18-20% in contrast to DCD and VisualDroid using five and ten iterations of FGSM with the maximum allowed perturbation of 12%.

## 6.6 Effectiveness of Proposed Attacks in the presence of Defensive Strategy

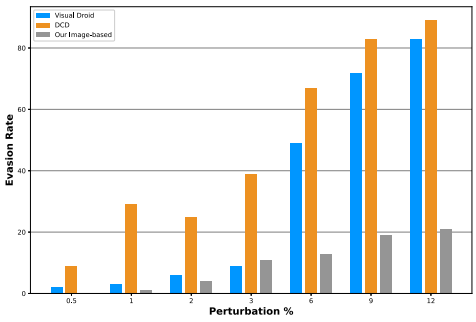
Distillation as a defensive strategy against adversarial examples was coined by Papernot *et al.* [112]. Initially, it was intended to run the trained model on cheaper hardware. In distillation, a smaller ANN trains on the output of another trained ANN with high temperature $T$ and predicts the class probabilities of the trained network instead of hard labels. ANN typically uses softmax as an activation function in the output layer which converts the logit $z_i$ (numeric value) into class probability $P_i$ by comparing it with other logits as given in eq 6.5.




(a) Attack3 evasion rate with only one iteration of modified FGSM




(b) Attack3 evasion rate with only five iterations of modified FGSM




(c) Attack3 evasion rate with only 10 iterations of modified FGSM

Fig. 6.4 Attack-3 effectiveness and Robustness evaluation of our image-based detection system versus DCD [170] and VisualDroid [171]

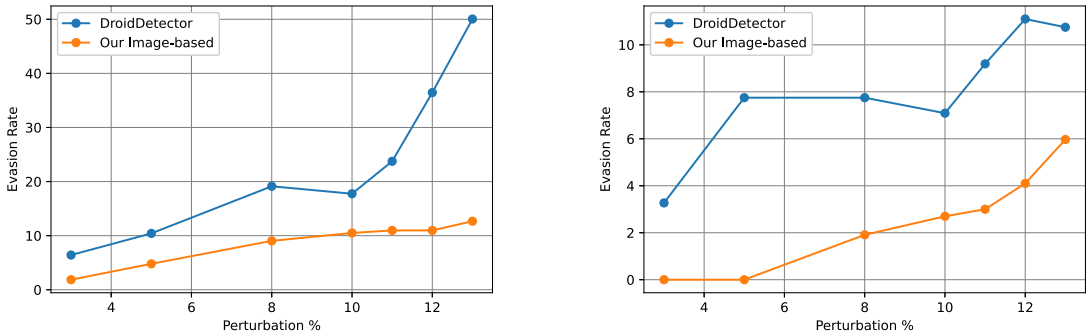
$$P_i = \frac{e^{z_i/T}}{\sum_{j=0}^{n-1}(e^{z_j/T})} \tag{6.5}$$

Where $T$ is the temperature usually set to 1 for inference, the distilled model used a higher value of $T$ during the training, and once it has been trained, it uses $T = 1$. A higher value of $T$ generates ambiguous probability distribution, while the smaller value of $T$ yields discrete probability distribution. In our case, we have trained a CNN classifier $F$ with the temperature $T = 40$ in the last layer of the sigmoid activation function using the original labels from the AndroZoo dataset. As a next step, we have trained a distilled classifier $F'$ on soft labels predicted by classifier $F$ by keeping the architecture similar to $F$ except $T = 1$ in the last layer of the sigmoid.

The effectiveness of an attack is determined by its ability to craft adversarial examples that deceive the learning classifier. We have presented two novel adversarial attacks in chapter 5 also published in [43] and evaluated their evasion rate concerning other learning-based classifiers [170, 168]. In this research work, we have implemented distillation [112] as a defensive strategy to various learning-based classifiers and benchmark the impact of Attack1, Attack 2, and 3 on these classifiers in the presence of defensive strategy. We have also implemented distillation as a defense to our image-based android malware detection system and compared the outcomes with [171], DroidDetecor [168], and DCD [170].

### 6.6.1 Effectiveness of Attack-1 and Attack-2 under Distillation

Figure 6.5 shows the evasion rate of Attack-1 for DroidDetector and our image-based android malware detection system before and after employing the distillation as a defensive strategy. The results illustrated in Figure 6.5 indicate that distillation adds a defense layer against Attack-1 for DroidDetector [168] and our image-based detection system published in [40]. However, the impact of distillation as a defense is not substantial. It does not make the classifier foolproof, but still acceptable and reduces the evasion rate from 36% to 10% against 12% perturbation for DroidDetector, as depicted in Figure 6.5. Moreover, it is easy to visualize that our image-based detection without any defense strategy invaded with almost the same evasion rate as DroidDetector evaded after employing the distillation against Attack-1.

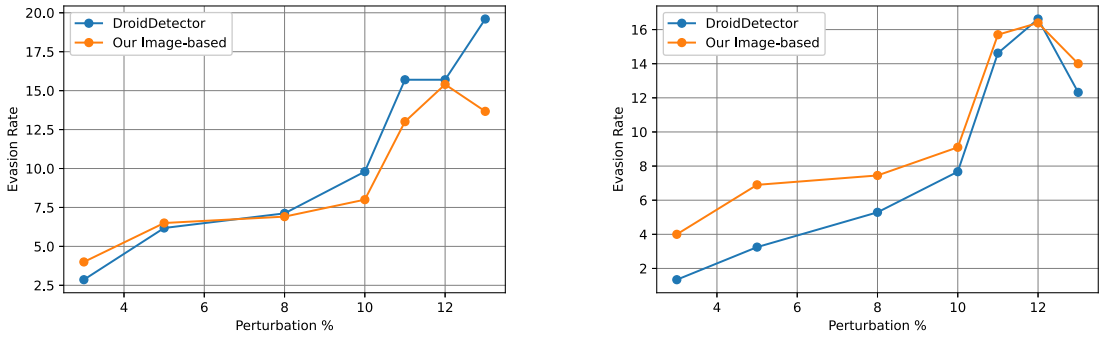



(a) Attack-1 evasion rate without Distillation

(b) Attack1 evasion rate under Distillation

Fig. 6.5 Attack-1 effectiveness under distillation for our image-based detection system versus DroidDetector [168]

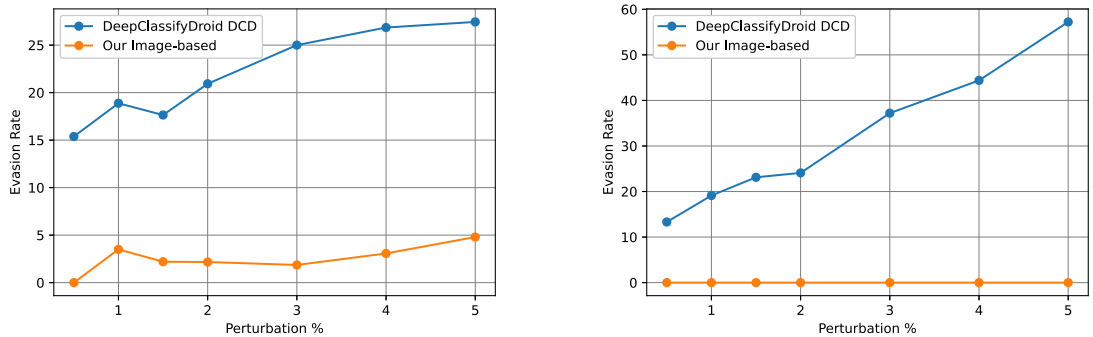
Attack-2 discussed in section 5.3.3 generates fake benign images and corresponding benign properties. Attack-2 uses these generated fake benign properties and appends them to malware samples to create adversarial examples. Figure 6.6a shows the evasion rate of DroidDetector [168] versus our image-based detection system [43] without incorporating any defensive strategy. It is easy to visualize that Attack-2 is more targeted toward visualization-based approaches but still effective against non-images-based techniques like DroidDetector and Drebin, which used binary vector representation using our lookup table [41]. However, it does not evade non-image-based approaches with higher evasion rates than DCD and VisualDroid. We incorporated distillation as a defense in our image-based technique and DroidDetector to evaluate the effectiveness of Attack-2 in the presence of a defensive strategy. Figure 6.6b shows that the distillation could not establish any defense line against Attack-2. Our non-distilled image-based detection system yielded the same evasion rate as the distilled DroidDetecor. However, distillation as defense shows minimal improvement for DroidDetector, as shown in Figure 6.6. The inherent mechanism of APK to image transformation using intelligent mapping of App components and suspected APIs makes the system hard to invade. However, carefully crafted adversarial attacks in Whitebox settings can evade the learning classifier with slightly higher perturbations.




(a) Attack-2 evasion rate without Distillation (b) Attack2 evasion rate under Distillation

Fig. 6.6 The effectiveness of Attack-2 under distillation for our image-based detection system versus DroidDetector [168]

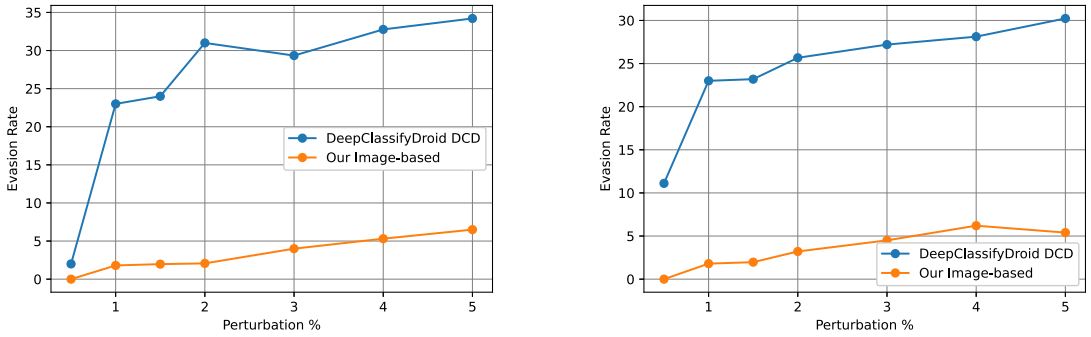
Further extending our evaluation benchmark, we also implemented DeepClassifyDroid (DCD) [170] as discussed in section 5.4.2. The approach is similar to images in terms of APK representation that uses a binary matrix of shape $n \times p$ and represents an APK as a single channel of size $256 \times 256$. Figure 6.7 illustrates the effectiveness of Attack-1 on DCD [170] and our image-based detection system [43] in the presence and absence of distillation as a defense. It is obvious from Figure 6.7a that DCD has an evasion rate of 28%, while our image-based approach evaded with only 5% against the maximum allowed perturbation of 5%. As we observed, the distillation can defend Attack-1 to some extent. Similarly, Figure 6.7b shows that distillation added some defense against Attack-1 in the case of our image-based detection system. However, in the case of DCD, it could not establish any defense against Attack-1.




(a) Evasion rate by Attack-1 without Distillation (b) Evasion rate by Attack-1 under Distillation

Fig. 6.7 Attack-1 effectiveness under distillation for our image-based detection system versus DeepClassifyDroid (DCD) [170]

Figure 6.8 presents the effectiveness of Attack-2 with and without distillation as a defense for DCD and our image-based detection systems. Attack-2 targets image-based approaches, and it is easy to infer from Figure 6.8b that distillation as a defense failed against Attack-2 for both DCD and our approach. The evasion rate of Attack-2 stays almost the same for both DCD and our image-based detection system in Figure 6.8a and Figure6.8b. Moreover, it is easy to visualize that distillation has a negligible effect on image-based android malware detection systems. Our image-based detection system [40] without any defensive strategy is still better than DCD [170] after employing distillation.




(a) Evasion rate by Attack-2 without Distillation (b) Evasion rate by Attack-2 under Distillation

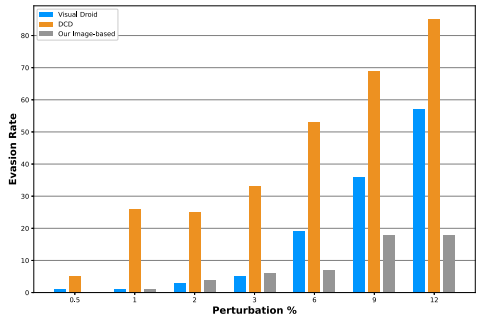
Fig. 6.8 The effectiveness of Attack-2 under distillation for our image-based detection system versus DeepClassifyDroid (DCD) [170]

### 6.6.2 Effectiveness of Attack-3 under Distillation

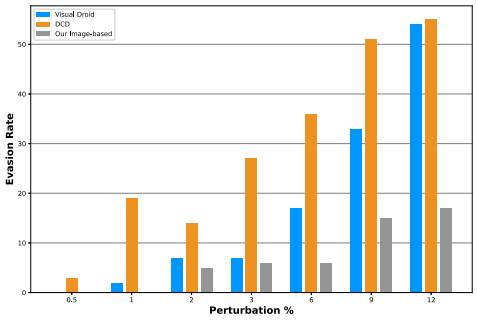
The results of evasion for modified FGSM attack (Attack-3) are given in Table 6.2. We implemented distillation as a defense for our image-based detection system, VisualDroid [171] and DCD [170] to evaluate the effectiveness of Attack-3 in the presence of a defensive strategy. Figure 6.9 shows that distillation is not a suitable defense against Attack-3 using five iterations. Although it has added a minor improvement in the evasion rate for DCD [170] and visualization-based detector [171], both classifiers still got evaded with more than 50% evasion rate. However, the evasion rate of our non-distilled image-based detection system depicted in Figure 6.9a is still significantly better than both approaches after incorporating the distillation as a defense given in Figure 6.9b. Furthermore, we examined the behavior of knowledge distillation against Attack-3 using ten iterations as depicted in Figure 6.10. The evasion rate of Attack-3 climbed above 60% for VisualDroid and DCD in the presence of distillation, as shown in Figure 6.10b.

Table 6.3 Evasion of Attack-3 (Modified FGSM) under distillation for Visual Droid [171], DCD [170] and our [43]

| **Perturb $\rho$ %** | **# Iterations** | **Evasion Rate After Distillation %** | | |
|---|---|---|---|---|
| | | **Visual Droid [171]** | **DCD [170]** | **Our** |
| 0.5 | 5 | 0 | 3 | - |
| | 10 | 1 | 7 | - |
| 1 | 5 | 2 | 19 | - |
| | 10 | 3 | 28 | - |
| 2 | 5 | 7 | 14 | 5 |
| | 10 | 11 | 24 | 6 |
| 3 | 5 | 7 | 27 | 6 |
| | 10 | 14 | 36 | 12 |
| 6 | 5 | 17 | 36 | 5 |
| | 10 | 46 | 47 | 16 |
| 9 | 5 | 33 | 51 | 15 |
| | 10 | 69 | 61 | 17 |
| 12 | 5 | 54 | 55 | 17 |
| | 10 | 78 | 64 | 22 |

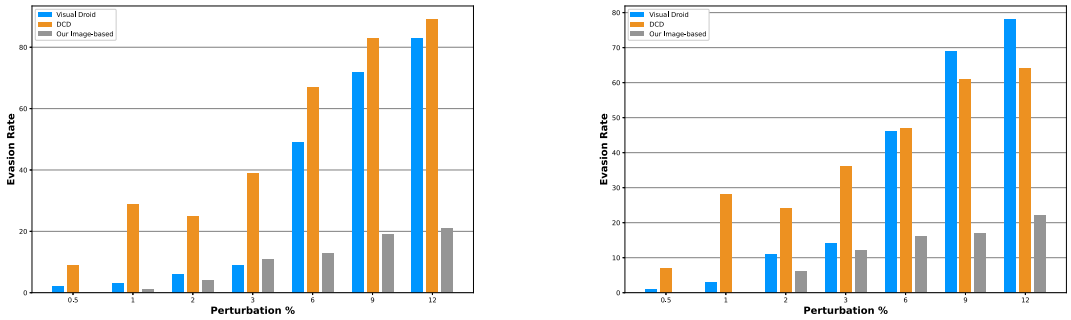



(a) Evasion rate by Attack-3 without Distillation using five iterations




(b) Evasion rate by Attack-3 under Distillation using five iterations

Fig. 6.9 The effectiveness of Attack-3 under distillation for our image-based Android malware detection system versus DeepClassifyDroid (DCD) [170],and VisualDroid [171] with maximum five iterations




(a) Evasion rate by Attack-3 without Distillation using 10 iterations

(b) Evasion rate by Attack-3 under Distillation using 10 iterations

Fig. 6.10 The effectiveness of Attack-3 under distillation defense for our image-based Android malware detection system versus DeepClassifyDroid (DCD) [170],and VisualDroid [171] with maximum of 10 iterations

## 6.7 Summary & Conclusion

The smartphone is acting as a modern personal assistant to human life, and its usability is growing exponentially. Android OS holding the maximum market share is on the radar for malware authors. Learning-based malware detection systems have offered exceptional flexibility to detect malicious content, but they are vulnerable to adversarial attacks. A robust Android malware detection systems against adversarially modified examples are the utmost requirement for the Anti-malware industry and cybersecurity community. We have proposed a novel modified FGSM attack that generates adversarial examples for visualization-based Android malware detection systems. The offered system uses DcGAN to generate fake benign properties and appends them to malware samples using modified FGSM. The results showed that Attack-3 is strong enough to deceive various learning-based classifiers. Moreover, we have implemented distillation as a defense and incorporated it into state-of-the-art learning-based malware detection systems compared to our image-based android malware detection system. The distillation was unable to establish any defense line against Attack-2 and Attack-3. However, our non-distilled image-based android malware detection system surpasses other distilled learning-based approaches under these attacks. In the future, we will extend our benchmarking to incorporate more attacks and defenses using the same dataset and parameters to determine the best learning-based classifier under adversarial settings.

# Chapter 7

# Conclusion

## 7.1 General Conclusion

Since the last decade, the smartphone has been the only device in history that has completely changed human life regardless of age on many fronts, be it communication, entertainment, marketing, education, finance, healthcare, business, sharing content, and many more. One of the main questions is, "how is life possible without a smartphone?". The dependency of a common person from morning alarm to calendar schedule, phone calls, messages, scrolling through social media for news and other activities, answering Snapchat, listening to music, watching videos, playing games, ordering food, booking transport, and online shopping - everything is embedded into only one device with exceptional portability and ubiquitous internet connectivity. The smartphone endows millions of applications to assist users in daily routine tasks and capture personal data and sensitive information. The proliferation of smartphones and billions of applications is the sweet spot for application developers and malware authors.

The main contribution of this thesis work oscillates around Android smartphones. We have mentioned that the open-source availability of Android OS attracts developers to build legitimate mobile applications and currently captures the biggest market share in the smartphone industry. At the same time, it is the popular hive for malware authors due to its popularity, open-source availability, and intrinsic infirmity to access internal resources. The first three chapters establish the basic concepts and general guidance for layman users and researchers pertinent to the Android world. We have discussed legacy and modern approaches for Android malware detection on the track of static, dynamic, or hybrid analysis. Since the

last decade, it has been noticed that researchers and the anti-malware industry have widely adopted machine learning and deep learning-based approaches to combat Android malware. Adopting machine learning in mobile security protects users' data, privacy information, and safe end-users from illegitimate actions launched by malware attackers.

Learning-based malware detection techniques have gained unpaired distinction to shield smartphone users from illegitimate access. However, recent advances in cybersecurity and adversarial machine learning have shown that machine learning is the weaker link in security and weakens its battle against adversarial examples. The vulnerability of learning-based techniques against adversarial examples poses a severe security threat to their adoption in security-sensitive domains and can compromise the system's overall security. The rationale behind ineffective adversarial handling is the original design of learning techniques that do not cater to adaptive and intelligent attacks from creative adversaries. A proficient adversary (i.e., attacker) can carefully frame an attack to dupe the malware classifier. Researchers have demonstrated this intuition that the skilled adversary changes the dataset distribution by injecting poisoning samples into training data, changing the labels known as "Causative attacks, " or exploring the weakness in a trained classifier called "Exploratory attacks." These attacks orchestrate adversarial examples that subsequently delude the learning classifier by compromising the detection system's integrity and/or availability. The study of adversarial machine learning is currently the hotspot area and impacted computer vision and the Natural Language Processing domain. However, the application of adversarial machine learning to android malware detection has been scarce. To this end, the work of this thesis entails the solution of two following objectives.

- **Objective 1: Solved the challenge of crafting a novel adversarial threat model and generating adversarial examples for android applications.**
- **Objective 2: Solved the challenge of developing an adversary proof classifier through intelligent application representation, feature selection, and coining defensive strategies against adversarial attacks**

### 7.1.1 Summary of Adversarial Attacks

Researchers have been focusing on natural image datasets to generate adversarial examples. The central idea that leads to adversarial examples was first coined in natural images and came from the fact that no computer system can precisely and correctly classify all the objects in a given image. There is always a subset of space that leads to incorrect recognition while

a human can easily recognize them. The false recognition of the subset space may be due to physical perturbation that includes shape deformation, improper illumination, distortion, occlusion, misorientation, etc. These perturbations highly influence the extracted/selected feature in the classification process and lead to false predictions. However, adversarial malware examples must preserve the intrinsic structure of the original application, and added perturbations must not change the malware functionality. The binary feature representation of an android APK makes it complex and challenging to generate more powerful adversarial examples.

In this thesis work, we have presented several studies on evasion and poisoning attacks for learning-based systems, making different assumptions on the amount of knowledge available to adversaries about the targeting system in the training, testing, and deployment phases. To solve the challenge of generating novel adversarial attacks for the android platform, we have implemented and published a threat model in a Whitebox setting. During the research of this thesis, we have proposed three different methodologies to generate adversarial attacks to deceive learning-based Android malware detection systems. All three attacks comply with the perturbation constraint that crafting an adversarial example (AE) $x'$ from the original malware sample $x$ should not affect the semantic and intrusive functionality of the malware and the execution of the android application. We have evaluated the effectiveness of three novel attacks on various learning-based Android malware detection systems. The high-level summary of these three attacks is enumerated below:

- **Attack 1: Appending predefined benign properties only:** It has been observed that adding random bytes to binary files or appending random pixels at the end of converted files generated dummy values. Contrary to this approach, we purely add benign-only properties to a malware sample using our predefined dictionary [42]. Attack 1 guarantees the malware and application functionality constraint and prevents the modification and deletion of existing features. The effectiveness of Attack 1 is evaluated on two other state-of-the-art learning-based android malware detection systems in parallel to our image-based classifier. We have implemented two approaches, DroidDetector [168] and DeepClassifyDroid [170] using AndrZoo dataset and evaluated their misclassification rate under Attack 1. We have tested the different amounts of perturbations; the addition of 13% perturbation evaded the DroidDetector with a 50% evasion rate in the contest to our image-based approach that evaded with 12%. In the case of DeepClassifyDroid, Attack 1 achieved a 27.8% evasion rate by adding 5% perturbation, while our image-based detection system evaded by 4.8%.

- **Attack 2: Adding proportionate of benign images using Deep Convolutional GANs (DCGANs):** Our second method for crafting adversarial examples is highly effective for image-based learning malware detection techniques, and it also works for other representation methods like binary vector, etc., using the given lookup table. In the second attack, we employed GANs to add a fragment of the benign image to a malware sample to evade the learning-based classifier. Attack 2 also capitulates with the original malware functionality constraint and assures that the appended image part belongs to actual benign properties. Attack 2 entails a two-step process (a) learning the benign distribution using only benign samples from the AndroZoo dataset. (b) Select benign-only pixels from the generated image (corresponds to benign properties) and append them to a malware image to craft adversarial examples. In the second step, we select benign pixel values from these generated benign images and append them to malware samples. We have evaluated the effectiveness of Attack 2 on our image-based detection, DroidDetector, and DeepClasifyDroid. Attack 2 evaded the DroidDetector with 19.6%and our image-based detector with 13% using perturbation threshold at 13%. Attack 2 became more influential on DeepClassifyDroid due to its binary matrix representation closer to image representation. Attack 2 evaded the DeeplClassifyDroid with a 34% evasion rate using 5% perturbation, while the misclassification rate of our image-based detection system was 6.5% under 5% perturbation.

- **Attack 3: Modified FGSM method to generate adversarial examples** Preserving the malware functionality constraint and original application functionality, we have introduced a newly modified FGSM method for crafting adversarial examples. In Attack 1 and 2, the appended benign properties cannot be further modified to evolve and strengthen the attack model. The modified FGSM attack involved two steps; firstly, it generates the fake benign images using DCGANs and appends them to malware samples. Secondly, it maximizes the loss function of perturbed malware samples. Attack 3 does not nudge the pixel value with $\pm\,\varepsilon$ but jumps to the next benign property generated by GANs in the direction of the gradient and assures that the pixel value added to a malware sample is purely a benign property instead of dummy value. We have evaluated the effectiveness of Attack 3 on our image-based detection system, DeepclassifyDroid, and Visualization-based Android Malware detection system (VisualDroid) [171]. Our empirical results indicated that Deepclassify evaded with 89%, VisualDroid evaded with 83%, and our proposed system evaded with a 21% evasion rate.

### 7.1.2 Summary of Image-based Android Malware Detection System

We developed a learning-based classifier incorporating adversary-aware feature selection and secure application representation. The proposed image-based Android malware detection system performs an intelligent mapping of an APK file into an RGB image. We proposed a novel representation of android APK, making it immune and harder against evasion attacks and does not require adversarial training. The proposed framework built a lightweight, robust, and scalable solution that is a combat force against legacy and new android malware and is harder to intrude. The system extracted information from the dex and manifest files and applied filtration based on an exhaustive dictionary, intelligently mapping them on red, green, and blue channels to draw patterns. The resultant RGB image is preprocessed and normalized using nearest-neighbor interpolation and then fed to a CNN. The experimental evaluation revealed that the image-based android malware detection system achieved an accuracy of 99.37% and FNR 0.8% on the legacy and new malware samples from the AndroZoo dataset. Moreover, our system recalls the malware family with high accuracy and low False Positive score compared to state-of-the-art malware family classification methods.

It is imperative to present an effective defense strategy targeting android malware detection systems without comprising detection accuracy and, at the same time, encapsulating security measures against adversarial attacks. For the adversarial robustness of our image-based android malware detection system, we compared our approach against state-of-the-art android malware detection systems DroidDetector [168], DeepClassifyDroid [170], and Visualization-based Android detection system [171]. The results illustrated the resilience of all three other learning-based android malware detectors under novel adversarial attacks. Our system proved more robust against adversarial perturbations and only evaded by 13% while the evasion rate was more than 50% for DroidDetector, DeepClassifyDroid, and VisualDroid.

The main challenge in adversarial attacks is the inconsistency of threat models. The existing studies do not present clear limitations and assumptions of proposed threat models regarding source code availability and attack feasibility concerning computational cost and time constraints. We have established the baseline, ground truth, and inter/intra-evaluation criteria for cross-platform/classifiers, as there is no consistent dataset and learning-based or commercial classifiers. Existing studies claimed a high evasion rate in a different setting and on top classifiers, but inter-evaluation criteria are missing. This study pioneers establish the benchmarking of Adversary-aware android malware detection systems. We have also evaluated the impact of our threat model by employing defensive strategies to state-of-the-art learning-based Android malware detection systems [168, 170, 171]. Our image-based

adversary-aware android malware detection system yielded a minimum evasion rate without incorporating any defensive strategy compared to other learning-based techniques with defensive policies.

## 7.2 Future Work

Numerous studies proposed various defensive strategies to make a learning-based classifier resilient against evasion attacks. However, they do not guarantee a durable defense line against new and evolving attacks. Each defensive strategy has different pros and cons and mostly targets the computer vision domain. Their applicability to the Android malware domain needs thorough research and analysis. Moreover, employing one defensive strategy may not improve the performance of the learning classifier significantly but can make it vulnerable to other adversarial attacks. Following are the future directions and possible extensions of this thesis work, as it is a relatively new field and getting higher research attention.

- **Establish defenses against practical adversarial malware attacks:** Most of the defense studies revealed that adversarial training is one of the domain-agnostic strategies to combat practical attacks. Adversarial training performs well against the attacks used in training but does not guarantee to combat new or unknown attacks. As encryption does in basic security, none of the defensive methodologies built any mathematically guaranteed resilience. Therefore other techniques like smoothing and randomization need to be studied in the context of android malware detection. The transformation and defense performance of these strategies in the malware domain varies. It is imperative to develop a defense strategy or ensemble of defensive strategies tailored to the malware domain that can guarantee a mathematical hardness against adversarial attacks or at least make it computationally infeasible to create adversarial perturbation for an attacker.

- **Extending benchmarks for Android-centric adversarial malware attacks:** We have established the initial benchmarks for adversarial android malware detection in this thesis work. This benchmark needs further investigations from different perspectives, like varying the adversary's resources, introducing a cost factor incurred on each iteration of an adversary attack, or setting a limit on the number of queries allowed to hit the target model. Moreover, this study is limited to exploring adversarial attacks in

WhiteBox settings. The investigation of adversarial attacks in BlackBox and GreyBox settings is deemed required as a next step.

- **Study of Reinforcement learning for crafting novel adversarial threat model in a dynamic analysis environment:** Static analysis-based approaches often fail to classify the malware under different packing and encrypted techniques. The study of reinforcement learning is required to craft adversarial attacks on android malware detection and their behavior in a dynamic analysis. One can investigate Reinforcement Learning(RL) to generate intentional perturbations that can be visualized as a min-max game where a reinforcement learning agent determines a sequence of actions to maximize the return based on the reward function.

- **Accuracy versus security balancing for an adversary-aware android malware detection system:** Detection accuracy and robustness of the classifier is a known challenge. Feature selection and fewer features degrade the system's performance, and a natural trade-off between accuracy and security is common. Currently, no special design decision is involved in dealing with the learning classifier for the malware domain (binary data) as it is available for image classification and text processing. In images, feature extraction is part of the CNN network, and it happens in embedding layers for text data. In future work, security considerations must be incorporated to construct the design of the learning classifier. A comparison should be drawn for malware detection systems with varying features to set up operation points and cope with various performance and security criteria.

# References


[1] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems*, pages 86–103. Springer, 2013.

[2] Mafaz Mohsin Khalil Al-Anezi. Generic packing detection using several complexity analysis for accurate malware detection. *Int. J. Adv. Comput. Sci*, 5(1), 2014.

[3] 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.

[4] 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.

[5] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. Droidnative: Automating and optimizing detection of android native code malware variants. *computers & security*, 65:230–246, 2017.

[6] 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.

[7] Androguard. Open-source reverse engineering tool for static analysis, Last visited May 2022. URL https://code.google.com/archive/p/androguard/.

[8] Android Market. Android smartphone market share report and analysis, Last visited November 2021. URL https://www.idc.com/promo/smartphone-market-share.

[9] Android Platform. Platform architecture for android os, Last visited March 2022. URL https://developer.android.com/guide/platform.

[10] APK building. Android apk build process, Last visited March 2022. URL https://blog.codecarrot.net/how-your-android-code-compiles-to-deliver-apk-package-file/.

[11] APK-tool. A tool for reverse engineering android apk files, Last visited May 2022. URL https://ibotpeaches.github.io/Apktool/.

[12] Anshul Arora, Sateesh K Peddoju, and Mauro Conti. Permpair: Android malware detection using permission pairs. *IEEE Transactions on Information Forensics and Security*, 15:1968–1982, 2019.

[13] 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.

[14] AV-TEST. Av-test,2020 malware statistics, Last visited October 2020. URL https://www.av-test.org/en/statistics/malware/.

[15] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D Joseph, and J Doug Tygar. Can machine learning be secure? In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 16–25, 2006.

[16] David Barrera, H Güneş Kayacik, Paul C Van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 73–84, 2010.

[17] Pasquale Battista, Francesco Mercaldo, Vittoria Nardone, Antonella Santone, and Corrado Aaron Visaggio. Identification of android malware families with model checking. In *ICISSP*, pages 542–547, 2016.

[18] Donabelle Baysa, Richard M Low, and Mark Stamp. Structural entropy and metamorphic malware. *Journal of computer virology and hacking techniques*, 9(4):179–192, 2013.

[19] Arjun Nitin Bhagoji, Daniel Cullina, and Prateek Mittal. Dimensionality reduction as a defense against evasion attacks on machine learning classifiers. *arXiv preprint arXiv:1704.02654*, 2, 2017.

[20] Jarrett Booz, Josh McGiff, William G Hatcher, Wei Yu, James Nguyen, and Chao Lu. Tuning deep learning performance for android malware detection. In *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 140–145. IEEE, 2018.

[21] Brendon Boshell. Average app size,2017 blog by brendon boshell, Last visited April 2020. URL https://sweetpricing.com/blog/2017/02/average-app-file-size/.

[22] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26, 2011.

[23] Gerardo Canfora, Francesco Mercaldo, and Corrado Aaron Visaggio. Mobile malware detection using op-code frequency histograms. In *2015 12th International Joint Conference on e-Business and Telecommunications (ICETE)*, volume 4, pages 27–38. IEEE, 2015.

[24] Gerardo Canfora, Fabio Martinelli, Francesco Mercaldo, Vittoria Nardone, Antonella Santone, and Corrado Aaron Visaggio. Leila: formal tool for identifying mobile malicious behaviour. *IEEE Transactions on Software Engineering*, 45(12):1230–1252, 2018.

[25] Nicholas Carlini and David Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 3–14. ACM, 2017.

[26] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pages 39–57. IEEE, 2017.

[27] Raphael Labaca Castro, Corinna Schmitt, and Gabi Dreo. Aimed: Evolving malware with genetic programming to evade detection. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 240–247. IEEE, 2019.

[28] Tanmoy Chakraborty, Fabio Pierazzi, and VS Subrahmanian. Ec2: Ensemble clustering and classification for predicting android malware families. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[29] Patrick PK Chan and Wen-Kai Song. Static detection of android malware by using permissions and api calls. In *2014 International Conference on Machine Learning and Cybernetics*, volume 1, pages 82–87. IEEE, 2014.

[30] Ashwin Chaugule, Zhi Xu, and Sencun Zhu. A specification based intrusion detection framework for mobile phones. In *International Conference on Applied Cryptography and Network Security*, pages 19–37. Springer, 2011.

[31] 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.

[32] Xuesong Chen, Xiyu Yan, Feng Zheng, Yong Jiang, Shu-Tao Xia, Yong Zhao, and Rongrong Ji. One-shot adversarial attacks on visual tracking with dual attention. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10176–10185, 2020.

[33] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252, 2011.

[34] Chang Choi, Christian Esposito, Mungyu Lee, and Junho Choi. Metamorphic malicious code behavior detection using probabilistic inference methods. *Cognitive Systems Research*, 56:142–150, 2019.

[35] Mihai Christodorescu and Somesh Jha. Testing malware detectors. *ACM SIGSOFT Software Engineering Notes*, 29(4):34–44, 2004.

[36] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. Technical report, WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES, 2006.

[37] Contagio mobile. Android malware samples from contagio mobile, Last visited March 2022. URL http://contagiominidump.blogspot.com.

[38] Lilian D Coronado-De-Alba, Abraham Rodríguez-Mota, and Ponciano J Escamilla-Ambrosio. Feature selection and ensemble of classifiers for android malware detection. In *2016 8th IEEE Latin-American Conference on Communications (LATINCOM)*, pages 1–6. IEEE, 2016.

[39] Fauzi Mohd Darus, Salleh Noor Azurati Ahmad, and Aswami Fadillah Mohd Ariffin. Android malware detection using machine learning on image patterns. In *2018 Cyber Resilience Conference (CRC)*, pages 1–2. IEEE, 2018.

[40] A. Darwaish and F. Naït-Abdesselam. Rgb-based android malware detection and classification using convolutional neural network. In *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, pages 1–6. IEEE, 2020.

[41] Asim Darwaish. Source code and dictionary, rgb-based-andorid-malware-detection [online] available: https://github.com/asimswati553/., Last visited October 2020. URL Available:https://github.com/asimswati553/.

[42] Asim Darwaish and Farid Naït-Abdesselam. Rgb-based android malware detection and classification using convolutional neural network. In *GLOBECOM 2020-2020 IEEE Global Communications Conference*, pages 1–6. IEEE, 2020.

[43] 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.

[44] 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.

[45] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *28th USENIX security symposium (USENIX security 19)*, pages 321–338, 2019.

[46] Yu-Xin Ding, Wei-Guo Zhao, Zhi-Pan Wang, and Long-Fei Wang. Automaticlly learning featurs of android apps using cnn. In *2018 International Conference on Machine Learning and Cybernetics (ICMLC)*, volume 1, pages 331–336. IEEE, 2018.

[47] Nguyen Viet Duc, Pham Thanh Giang, and Pham Minh Vi. Permission analysis for android malware detection. In *The Proceedings of the 7th VAST-AIST Workshop "Research Collaboration: Review and perspective"*, 2015.

[48] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[49] Mojtaba Eskandari and Sattar Hashemi. A graph mining approach for detecting unknown malwares. *Journal of Visual Languages & Computing*, 23(3):154–162, 2012.

[50] 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.

[51] 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.

[52] 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 SIGSOFT international symposium on foundations of software engineering*, pages 576–587, 2014.

[53] Hossein Fereidooni, Mauro Conti, Danfeng Yao, and Alessandro Sperduti. Anastasia: Android malware detection using static analysis of applications. In *2016 8th IFIP international conference on new technologies, mobility and security (NTMS)*, pages 1–5. IEEE, 2016.

[54] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/avik/projects/scandroidascaa*, 2(3), 2009.

[55] Zhang Fuyong and Zhao Tiezhu. Malware detection and classification based on n-grams attribute similarity. In *2017 IEEE international conference on computational science and engineering (CSE) and IEEE international conference on embedded and ubiquitous computing (EUC)*, volume 1, pages 793–796. IEEE, 2017.

[56] Tianchong Gao, Wei Peng, Devkishen Sisodia, Tanay Kumar Saha, Feng Li, and Mohammad Al Hasan. Android malware detection via graphlet sampling. *IEEE Transactions on Mobile Computing*, 18(12):2754–2767, 2018.

[57] Genome. Android malware genome project, Last visited March 2022. URL http://www.malgenomeproject.org.

[58] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[59] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[60] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294, 2012.

[61] 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.

[62] 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.

[63] Weijie Han, Jingfeng Xue, Yong Wang, Lu Huang, Zixiao Kong, and Limin Mao. Maldae: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics. *computers & security*, 83:208–233, 2019.

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

[65] hexdump. display file contents in hexadecimal, Last visited March 2021. URL https://man7.org/linux/man-pages/man1/hexdump.1.html.

[66] TonTon Hsien-De Huang and Hung-Yu Kao. R2-d2: color-inspired convolutional neural network (cnn)-based android malware detections. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 2633–2642. IEEE, 2018.

[67] Chun-Ying Huang, Yi-Ting Tsai, and Chung-Han Hsu. Performance evaluation on permission-based detection for android malware. In *Advances in intelligent systems and applications-volume 2*, pages 111–120. Springer, 2013.

[68] Sandy Huang, Nicolas Papernot, Ian Goodfellow, Yan Duan, and Pieter Abbeel. Adversarial attacks on neural network policies. *arXiv preprint arXiv:1702.02284*, 2017.

[69] IDA. Binary code analysis, Last visited March 2021. URL https://www.hex-rays.com/.

[70] 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.

[71] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 19–35. IEEE, 2018.

[72] Sagar Jaiswal. *Feature Engineering & Analysis Towards Temporally Robust Detection of Android Malware*. PhD thesis, Indian Institute of Technology Kanpur, 2019.

[73] JEB. Jeb is a modular reverse engineering platform used as android debugger and android decompiler, Last visited May 2022. URL https://www.pnfsoftware.com.

[74] Xu Jiang, Baolei Mao, Jun Guan, and Xingli Huang. Android malware detection using fine-grained features. *Scientific Programming*, 2020, 2020.

[75] Jaemin Jung, Jongmoo Choi, Seong-je Cho, Sangchul Han, Minkyu Park, and Youngsup Hwang. Android malware detection using convolutional neural networks and data section images. In *Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems*, pages 149–153, 2018.

[76] Jaemin Jung, Hyunjin Kim, Dongjin Shin, Myeonggeon Lee, Hyunjae Lee, Seong-je Cho, and Kyoungwon Suh. Android malware detection based on useful api calls and machine learning. In *2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*, pages 175–178. IEEE, 2018.

[77] Aditya Kapoor, Himanshu Kushwaha, and Ekta Gandotra. Permission based android malicious application detection using machine learning. In *2019 International Conference on Signal Processing and Communication (ICSC)*, pages 103–108. IEEE, 2019.

[78] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb. Maldozer: Automatic framework for android malware detection using deep learning. *Digital Investigation*, 24:S48–S59, 2018.

[79] Riaz Ullah Khan, Xiaosong Zhang, and Rajesh Kumar. Analysis of resnet and googlenet models for malware detection. *Journal of Computer Virology and Hacking Techniques*, 15(1):29–37, 2019.

[80] Hyunjoo Kim, Jonghyun Kim, Youngsoo Kim, Ikkyun Kim, Kuinam J Kim, and Hyuncheol Kim. Improvement of malware detection and classification using api call sequence alignment and visualization. *Cluster Computing*, 22(1):921–929, 2019.

[81] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD Center. Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 12 (110):1, 2012.

[82] TaeGuen Kim, BooJoong Kang, Mina Rho, Sakir Sezer, and Eul Gyu Im. A multimodal deep learning method for android malware detection using various features. *IEEE Transactions on Information Forensics and Security*, 14(3):773–788, 2018.

[83] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *USENIX security symposium*, volume 4, pages 351–366, 2009.

[84] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European Signal Processing Conference (EUSIPCO)*, pages 533–537. IEEE, 2018.

[85] Vasileios Kouliaridis, Konstantia Barmpatsalou, Georgios Kambourakis, and Guojun Wang. Mal-warehouse: A data collection-as-a-service of mobile malware behavioral patterns. In *2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pages 1503–1508. IEEE, 2018.

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

[87] 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.

[88] Yunus Kucuk and Guanhua Yan. Deceiving portable executable malware classifiers into targeted misclassification with practical adversarial examples. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, pages 341–352, 2020.

[89] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236*, 2016.

[90] Patrik Lantz. An android application sandbox for dynamic analysis, Last visited May 2022. URL https://www.honeynet.org/projects/active/droidbox/.

[91] Arash Habibi Lashkari, Andi Fitriah A Kadir, Laya Taheri, and Ali A Ghorbani. Toward developing a systematic approach to generate benchmark android malware datasets and classification. In *2018 International Carnahan Conference on Security Technology (ICCST)*, pages 1–7. IEEE, 2018.

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

[93] 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.

[94] Shuang Liang and Xiaojiang Du. Permission-combination-based scheme for android mobile malware detection. In *2014 IEEE international conference on communications (ICC)*, pages 2301–2306. IEEE, 2014.

[95] Liu Liu, Bao-sheng Wang, Bo Yu, and Qiu-xi Zhong. Automatic malware classification and new malware detection using machine learning. *Frontiers of Information Technology & Electronic Engineering*, 18(9):1336–1347, 2017.

[96] Xinbo Liu, Yapin Lin, He Li, and Jiliang Zhang. Adversarial examples: Attacks on machine learning-based malware visualization detection methods. *arXiv preprint arXiv:1808.01546*, 10(3326285.3329073), 2018.

[97] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240, 2012.

[98] Dominik Maier, Tilo Müller, and Mykola Protsenko. Divide-and-conquer: Why android malware cannot be stopped. In *2014 Ninth International Conference on Availability, Reliability and Security*, pages 30–39. IEEE, 2014.

[99] Niall McLaughlin, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupé, et al. Deep android malware detection. In *Proceedings of the seventh ACM on conference on data and application security and privacy*, pages 301–308, 2017.

[100] Dongyu Meng and Hao Chen. Magnet: a two-pronged defense against adversarial examples. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 135–147. ACM, 2017.

[101] Nikola Milosevic, Ali Dehghantanha, and Kim-Kwang Raymond Choo. Machine learning aided android malware classification. *Computers & Electrical Engineering*, 61:266–274, 2017.

[102] Aziz Mohaisen, Omar Alrawi, and Manar Mohaisen. Amal: high-fidelity, behavior-based automated malware analysis and classification. *computers & security*, 52: 251–266, 2015.

[103] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2574–2582, 2016.

[104] Farid Naït-Abdesselam, Asim Darwaish, and Chafiq Titouna. An intelligent malware detection and classification system using apps-to-images transformations and convolutional neural networks. In *2020 16th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)(50308)*, pages 1–6. IEEE, 2020.

[105] Farid Naït-Abdesselam, Asim Darwaish, and Chafiq Titouna. Malware forensics: Legacy solutions, recent advances, and future challenges. In *Advances in Computing, Informatics, Networking and Cybersecurity*, pages 685–710. Springer, 2022.

[106] Masoud Narouei, Mansour Ahmadi, Giorgio Giacinto, Hassan Takabi, and Ashkan Sami. Dllminer: structural mining for malware detection. *Security and Communication Networks*, 8(18):3311–3322, 2015.

[107] Lakshmanan Nataraj, Sreejith Karthikeyan, Gregoire Jacob, and Bangalore S Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th international symposium on visualization for cyber security*, pages 1–7, 2011.

[108] Stavros D Nikolopoulos and Iosif Polenakis. A graph-based model for malware detection and classification using system-call groups. *Journal of Computer Virology and Hacking Techniques*, 13(1):29–46, 2017.

[109] Damien Octeau, Daniel Luchaup, Somesh Jha, and Patrick McDaniel. Composite constant propagation and its application to android program analysis. *IEEE Transactions on Software Engineering*, 42(11):999–1014, 2016.

[110] OllyDbg debugger. 32-bit assembler level analysing debugger for windows, Last visited March 2021. URL http://www.ollydbg.de/.

[111] Nicolas Papernot and Patrick McDaniel. Extending defensive distillation. *arXiv preprint arXiv:1705.05264*, 2017.

[112] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE symposium on security and privacy (SP)*, pages 582–597. IEEE, 2016.

[113] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519. ACM, 2017.

[114] Naser Peiravian and Xingquan Zhu. Machine learning for android malware detection using permission and api calls. In *2013 IEEE 25th international conference on tools with artificial intelligence*, pages 300–305. IEEE, 2013.

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

[116] 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.

[117] Robert Podschwadt and Hassan Takabi. On effectiveness of adversarial examples and defenses for malware classification. In *International Conference on Security and Privacy in Communication Systems*, pages 380–393. Springer, 2019.

[118] Nektaria Potha, V Kouliaridis, and G Kambourakis. An extrinsic random-based ensemble approach for android malware detection. *Connection Science*, 33(4):1077–1093, 2021.

[119] Mahmuda Rahman. Droidmln: a markov logic network approach to detect android malware. In *2013 12th International conference on machine learning and applications*, volume 2, pages 166–169. IEEE, 2013.

[120] Read-Androguard. Introduction and usability of androguard, Last visited May 2022. URL https://androguard.readthedocs.io/en/latest/intro/gettingstarted.html.

[121] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April*, 2013.

[122] Neha Runwal, Richard M Low, and Mark Stamp. Opcode graph similarity and metamorphic detection. *Journal in computer virology*, 8(1):37–52, 2012.

[123] Durmuş Özkan Şahin, Oğuz Emre Kural, Sedat Akleylek, and Erdal Kılıç. A novel permission-based android malware detection system using feature selection based on linear regression. *Neural Computing and Applications*, pages 1–16, 2021.

[124] Muhammad Suleman Saleem, Jelena Mišić, and Vojislav B Mišić. Examining permission patterns in android apps using kernel density estimation. In *2020 International Conference on Computing, Networking and Communications (ICNC)*, pages 719–724. IEEE, 2020.

[125] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231:64–82, 2013.

[126] B Sanz, I Santos, C Laorden, et al. Permission usage to detect malware in android. In *Int. Joint Conf. CISIS*, pages 289–298, 2013.

[127] Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, 15(1):83–97, 2016.

[128] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Android permissions: a perspective combining risks and benefits. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pages 13–22, 2012.

[129] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. "andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.

[130] Gayathri Shanmugam, Richard M Low, and Mark Stamp. Simple substitution distance and metamorphic detection. *Journal of Computer Virology and Hacking Techniques*, 9 (3):159–170, 2013.

[131] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proceedings of the 2016 acm sigsac conference on computer and communications security*, pages 1528–1540, 2016.

[132] Akanksha Sharma and Subrat Kumar Dash. Mining api calls and permissions for android malware detection. In *International Conference on Cryptology and Network Security*, pages 191–205. Springer, 2014.

[133] Shina Sheen, R Anitha, and V Natarajan. Android based malware detection using a multifeature collaborative decision fusion approach. *Neurocomputing*, 151:905–912, 2015.

[134] PV Shijo and AJPCS Salim. Integrated static and dynamic analysis for malware detection. *Procedia Computer Science*, 46:804–811, 2015.

[135] Smartphone users. Smartphone users worldwide 2016-2021, Last visited March 2021. URL https://www.statista.com/statistics/.

[136] Fu Song and Tayssir Touili. Model-checking for android malware detection. In *Asian Symposium on Programming Languages and Systems*, pages 216–235. Springer, 2014.

[137] Fu Song and Tayssir Touili. Pushdown model checking for malware detection. *International Journal on Software Tools for Technology Transfer*, 16(2):147–173, 2014.

[138] Wei Song, Xuezixiang Li, Sadia Afroz, Deepali Garg, Dmitry Kuznetsov, and Heng Yin. Automatic generation of adversarial examples for interpreting malware classifiers. *arXiv preprint arXiv:2003.03100*, 2020.

[139] Ivan Sorokin. Comparing files using structural entropy. *Journal in computer virology*, 7(4):259–265, 2011.

[140] Jack W Stokes, De Wang, Mady Marinescu, Marc Marino, and Brian Bussone. Attack and defense of dynamic analysis-based, adversarial neural malware detection models. In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*, pages 1–8. IEEE, 2018.

[141] Xin Su, Dafang Zhang, Wenjia Li, and Kai Zhao. A deep learning approach to android malware feature learning and detection. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 244–251. IEEE, 2016.

[142] Yuxia Sun, Yanjia Chen, Yuchang Pan, and Lingyu Wu. Android malware family classification based on deep learning of code images. *IAENG International Journal of Computer Science*, 46(4), 2019.

[143] Andrew H Sung, Jianyun Xu, Patrick Chavez, and Srinivas Mukkamala. Static analyzer of vicious executables (save). In *20th Annual Computer Security Applications Conference*, pages 326–334. IEEE, 2004.

[144] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[145] Peter Szor and Peter Ferrie. Hunting for metamorphic. In *Virus bulletin conference*. Prague, 2001.

[146] Rahim Taheri, Meysam Ghahramani, Reza Javidan, Mohammad Shojafar, Zahra Pooranian, and Mauro Conti. Similarity-based android malware detection using hamming distance of static binary features. *Future Generation Computer Systems*, 105:230–247, 2020.

[147] Annie H Toderici and Mark Stamp. Chi-squared distance and metamorphic virus detection. *Journal of Computer Virology and Hacking Techniques*, 9(1):1–14, 2013.

[148] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*, 2017.

[149] Sercan Türker and Ahmet Burak Can. Andmfc: Android malware family classification framework. In *2019 IEEE 30th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC Workshops)*, pages 1–6. IEEE, 2019.

[150] VirusShare project. Malware samples at virusshare website, Last visited March 2022. URL https://virusshare.com.

[151] Qinglong Wang, Wenbo Guo, Kaixuan Zhang, Xinyu Xing, C Lee Giles, and Xue Liu. Random feature nullification for adversary resistant deep architecture. *arXiv preprint arXiv:1610.01239*, 2016.

[152] Shanshan Wang, Zhenxiang Chen, Qiben Yan, Bo Yang, Lizhi Peng, and Zhongtian Jia. A mobile malware detection method using behavior features in network traffic. *Journal of Network and Computer Applications*, 133:15–25, 2019.

[153] Zhen Wang, Kai Li, Yan Hu, Akira Fukuda, and Weiqiang Kong. Multilevel permission extraction in android applications for malware detection. In *2019 International Conference on Computer, Information and Telecommunication Systems (CITS)*, pages 1–5. IEEE, 2019.

[154] Lukas Weichselbaum, Matthias Neugschwandtner, Martina Lindorfer, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Tech. Rep. TR-ISECLAB-0414-001*, pages 1–10, 2014.

[155] Xu Weilin, Qi Yanjun, and D Evans. Automatically evading classifiers. *Proc ofthe 23rd Annual Network and DistributedSystem SecuritySymp (NDSS2016). Reston, VA, USA: TheInternetSociety*, 2016.

[156] Rey Wiyatno and Anqi Xu. Maximal jacobian-based saliency map attack. *arXiv preprint arXiv:1808.07945*, 2018.

[157] Michelle Y Wong and David Lie. Intellidroid: a targeted input generator for the dynamic analysis of android malware. In *NDSS*, volume 16, pages 21–24, 2016.

[158] Wing Wong and Mark Stamp. Hunting for metamorphic engines. *Journal in Computer Virology*, 2(3):211–229, 2006.

[159] Worldwide users. Smartphone users worldwide, Last visited November 2021. URL https://www.bankmycell.com/blog/how-many-phones-are-in-the-world.

[160] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *2012 Seventh Asia Joint Conference on Information Security*, pages 62–69. IEEE, 2012.

[161] 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.

[162] Weilin Xu, Yanjun Qi, and David Evans. Automatically evading classifiers. In *Proceedings of the 2016 network and distributed systems symposium*, volume 10, 2016.

[163] Zhiwu Xu, Kerong Ren, Shengchao Qin, and Florin Craciun. Cdgdroid: Android malware detection based on deep learning using cfg and dfg. In *International Conference on Formal Engineering Methods*, pages 177–193. Springer, 2018.

[164] XXD-hexdump. xxd creates a hex dump of an input file, Last visited March 2021. URL https://linux.die.net/man/1/xxd.

[165] Lok Kwong Yan and Heng Yin. {DroidScope}: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis. In *21st USENIX security symposium (USENIX security 12)*, pages 569–584, 2012.

[166] Wei Yang, Deguang Kong, Tao Xie, and Carl A Gunter. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 288–302, 2017.

[167] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *International Symposium on Recent Advances in Intrusion Detection*, pages 359–381. Springer, 2015.

[168] Zhenlong Yuan, Yongqiang Lu, and Yibo Xue. Droiddetector: android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, 21(1):114–123, 2016.

[169] Fei Zhang, Patrick PK Chan, Battista Biggio, Daniel S Yeung, and Fabio Roli. Adversarial feature selection against evasion attacks. *IEEE transactions on cybernetics*, 46: 766–777, 2015.

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

[171] Yong-liang Zhao and Quan Qian. Android malware identification through visual exploration of disassembly files. *International Journal of Network Security*, 20(6): 1061–1073, 2018.

[172] 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.

[173] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 317–326, 2012.

[174] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In *NDSS*, volume 25, pages 50–52, 2012.

[175] Hui-Juan Zhu, Tong-Hai Jiang, Bo Ma, Zhu-Hong You, Wei-Lei Shi, and Li Cheng. Hemd: a highly efficient random forest-based malware detection framework for android. *Neural Computing and Applications*, 30(11):3353–3361, 2018.

Université Paris Cité

# Université Paris Cité

**École Doctorale Informatique, Télécommunication et Électronique «Edite de Paris : ED130»**
**Laboratoire d'Informatique Paris Descartes (LIPADE)**

## Résumé

# Adversary-aware Machine Learning Models for Malware Detection Systems

Par Asim Darwaish

Thèse de doctorat en Informatique (Intelligence Artificielle et Decision)

Dirigée par Farid Nait Abdesselam

Présentée et soutenue publiquement le 24-11-2022

Devant un jury composé de :

Tayssir Touili, Directrice de recherche (HDR), CNRS, Rapporteur

Hossam Afifi, Professeur, Institut Polytechnique de Paris, Rapporteur

Véronique Vèque, Professeur, Université Paris-Saclay, Examinateur

Ahmed Serhrouchni, Professeur, Telecom Paris, Examinateur

Melek Onen, MCF (HDR), EURECOM, Examinateur

Chafiq Titouna, MCF, Institut Gaspard Monge, Examinateur

Université Paris Cité

# Modèles d'apprentissage robustes aux attaques pour les systèmes de détection de logiciels malveillants

LIPADE
Laboratoire d'Informatique PAris DEscartes

**Asim Darwaish**

**Rapporteurs:** Tayssir Touili, Directrice de Recherche (HDR), CNRS
Hossam Afifi, Professeur, Institut Polytechnique de Paris

**Examinateurs:** Véronique Vèque, Professeur, Université Paris-Saclay
Ahmed Serhrouchni, Professeur, Telecom Paris
Melek Onen, Maîtresse de Conférences (HDR), EURECOM
Chafiq Titouna, Maître de Conférences , Institut Gaspard Monge

**Dirigée par :** Farid Naït-Abdesselam, Professeur, Université Paris Cité

Université Paris Cité

Cette thèse est présentée pour l'obtention du diplôme de
*Doctorat en Informatique*

LIPADE October 2022

# Résumé

La popularisation des smartphones et leur caractère indispensable les rendent aujourd'hui indéniables. Leur croissance exponentielle est également à l'origine de l'apparition de nombreux logiciels malveillants et fait trembler le prospère écosystème mobile. Parmi tous les systèmes d'exploitation des smartphones, Android est le plus ciblé par les auteurs de logiciels malveillants en raison de sa popularité, de sa disponibilité en tant que logiciel libre, et de son capacité intrinsèque à accéder aux ressources internes. Les approches basées sur l'apprentissage automatique ont été déployées avec succès pour combattre les logiciels malveillants polymorphes et évolutifs. Au fur et à mesure que le classificateur devient populaire et largement adopté, l'intérêt d'échapper au classificateur augmente également. Les chercheurs et les adversaires se livrent à une course sans fin pour renforcer le système de détection des logiciels malveillants androïd et y échapper. Afin de lutter contre ces logiciels malveillants et de contrer les attaques adverses, nous proposons dans cette thèse un système de détection de logiciels malveillants android basé sur le codage d'images, un système qui a prouvé sa robustesse contre diverses attaques adverses. La plateforme proposée construit d'abord le système de détection des logiciels malveillants android en transformant intelligemment le fichier Android Application Packaging (APK) en une image RGB légère et en entraînant un réseau neuronal convolutif (CNN) pour la détection des logiciels malveillants et la classification des familles. Notre nouvelle méthode de transformation génère des modèles pour les APK bénins et malveillants plus faciles à classifier en images de couleur. Le système de détection ainsi conçu donne une excellente précision de 99,37% avec un Taux de Faux Négatifs (FNR) de 0,8% et un Taux de Faux Positifs (FPR) de 0,39% pour les anciennes et les nouvelles variantes de logiciels malveillants. Dans la deuxième phase, nous avons évalué la robustesse de notre système de détection de logiciels malveillants android basé sur l'image. Pour valider son efficacité contre les attaques adverses, nous avons créé trois nouveaux modèles d'attaques. Notre évaluation révèle que les systèmes de détection de logiciels malveillants basés sur l'apprentissage les plus récents sont faciles à contourner,

avec un taux d'évasion de plus de 50%. Cependant, le système que nous avons proposé construit un mécanisme robuste contre les perturbations adverses en utilisant son espace continu intrinsèque obtenu après la transformation intelligente des fichiers Dex et Manifest, ce qui rend le système de détection difficile à contourner.

**Mots-clés** : Détection de Logiciels Malveillants, Android, Attaques Adverses, Apprentissage Profond, Robustesse, Apprentissage Automatique.

# Contexte

Les smartphones sont devenus un élément essentiel des tâches de la vie quotidienne pour accéder à des services électroniques précieux tels que les achats en ligne, les services bancaires mobiles, la commande de nourriture, la gouvernance, les services de santé en ligne et bien d'autres. Selon [27], le nombre total d'utilisateurs mondiaux de smartphones a atteint 6,378 milliards, soit 80% de la population mondiale. Le smartphone est un assistant numérique pour organiser le travail, les tâches routinières, les études et la vie privée. Ces smartphones regorgent de données sensibles qui nécessitent une sécurité primordiale pour l'informatique mobile. Le smartphone Android est un choix populaire en raison de son offre multiforme et s'empare de 85 % du marché mondial [3]. En tant que système d'exploitation (OS) open source, il attire les développeurs d'applications pour créer des applications mobiles légitimes. D'un autre côté, le système d'exploitation Android est un essaim préféré des auteurs de logiciels malveillants en raison de sa popularité, de sa disponibilité en open source et de son infirmité intrinsèque à accéder aux ressources internes.

Depuis la dernière décennie, les systèmes basés sur l'apprentissage automatique (ML) ont fait preuve d'une grande polyvalence dans la lutte contre divers logiciels malveillants polymorphes. Ces approches comme [29, 30, 15] ont été récemment développées pour automatiser la détection des logiciels malveillants Android en exploitant différentes représentations de fonctionnalités telles que l'interface de programmation d'applications (API), les autorisations, les composants d'application, les comportements dynamiques, le graphique des appels système, etc. Ces ML approches axées sur le cloud offrent des solutions inégalées pour résister aux logiciels malveillants dominants dans les applications mobiles. Cependant, les exemples adverses peuvent facilement échapper aux classificateurs basés sur l'apprentissage automatique, comme décrit dans [11, 24]. Par conséquent, le déploiement d'un système de détection basé sur l'apprentissage dans des domaines sensibles à la sécurité doit être envisagé. Un adversaire peut tromper le modèle de détection en créant des perturbations

subtiles connues sous le nom d'exemples adverses (AE). Un adversaire essaie d'induire en erreur le classificateur en concevant avec soin des attaques adverses sur le système de détection en modifiant l'importance des caractéristiques ou la distribution des données. En conséquence, le système d'apprentissage devient inefficace dans un environnement prêt pour la production et produit moins de Vrais Positifs (True Positive), par exemple, un échantillon de logiciel malveillant classé comme bénin en détériorant la prédiction du modèle ML.

L'étude de l'apprentissage adverse a été un domaine de recherche point chaud et se concentre principalement sur les images en raison de la visualisation facile des perturbations [11, 20, 6]. Cependant, les recherches évaluant l'impact des perturbations adverses sur les classificateurs d'apprentissage basés sur l'image pour les systèmes de détection de logiciels malveillants Android sont encore relativement rares. Peu de chercheurs ont analysé l'évasion adverse dans la détection de logiciels malveillants PDF (Portable Document Format) [18, 25], Windows executable [16, 17] et les détecteurs de logiciels malveillants Android, mais ont limité leurs recherches à l'espace binaire [5, 28, 12].

# Motivations

Les auteurs de logiciels malveillants utilisent des itinéraires sophistiqués et différentes techniques d'obfuscation de code pour créer des logiciels malveillants. La croissance exponentielle des applications pour smartphones aux natures variées et diverses facilite le travail de l'auteur de logiciels malveillants, tandis que la détection et l'isolement de ces logiciels malveillants deviennent difficiles et fastidieux. Il est presque impossible d'examiner le comportement malveillant de chaque application. Par conséquent, une ligne de défense solide contre ces applications malveillantes est jugée nécessaire.

Les logiciels antivirus fournis par les sociétés anti-malware établissent la première ligne de défense pour sécuriser les ordinateurs et les appareils portables contre les menaces de logiciel malveillant. La menace des logiciels malveillants répandus a été entravée par quelques approches classiques, telles que la détection de signature, l'analyse statique et l'analyse dynamique. Les techniques de détection basées sur les signatures et les heuristiques étaient couramment utilisées pour combattre l'écosystème mobile florissant contre les attaques de logiciels malveillants. La détection basée sur la signature nécessite une intervention manuelle et ne fonctionne que pour les attaques connues. Il échoue toujours sur les nouvelles variantes en raison de la non-disponibilité dans la base de données des signatures. Les approches d'analyse statique et dynamique sont souvent utilisées pour anal-

yser le comportement des logiciels malveillants. Les méthodes d'analyse statique étaient inefficaces pour l'obscurcissement du code, l'empaquetage et les logiciels malveillants sophistiqués. L'analyse dynamique nécessite des ressources importantes pour s'exécuter dans l'environnement de test (sandbox) et peut ne pas être efficace contre les évasions dynamiques, et nécessite plus de temps d'exécution pour analyser le comportement des logiciels malveillants. Contrairement à l'analyse manuelle pure, les approches basées sur l'apprentissage automatique ont montré un succès important dans la détection des applications malveillantes (basées sur les signatures, basées sur l'heuristique, etc.), mais dépendent toujours fortement de l'ingénierie des fonctionnalités et de l'analyse approfondie du code source pour attribuer des signatures malveillantes. La puissance de l'apprentissage en profondeur a révolutionné divers domaines, notamment la médecine, l'ingénierie, les sciences sociales, les processus et la gestion des entreprises, la santé, la finance et la cybersécurité. L'une des applications prestigieuses de l'apprentissage automatique (ML) dans la sécurité informatique & de l'information est la détection de logiciels malveillants.

L'apprentissage automatique a été proposé et largement adopté par les domaines sensibles à la sécurité pour surmonter les défis mentionnés ci-dessus et les limites des solutions héritées pour la détection automatisée des logiciels malveillants. Comme dans d'autres domaines, l'apprentissage automatique et les systèmes de détection de logiciels malveillants basés sur l'apprentissage en profondeur prédominent et comptent parmi les armes préférées des défenseurs. Les approches basées sur l'apprentissage se sont avérées plus puissantes en raison de leurs capacités d'automatisation, de généralisation, d'optimisation et de personnalisation. Au cours de la dernière décennie, ces techniques d'apprentissage ont offert une polyvalence sans précédent pour lutter contre les attaques de logiciels malveillants sur les voies d'analyse statiques et dynamiques. Malgré un développement rapide et des performances supérieures, l'apprentissage automatique est vulnérable aux attaques adverses [24, 11]. De nombreuses études dans la littérature ont montré qu'il est facile d'éviter les classificateurs d'apprentissage en créant des exemples adverses méticuleux, que ce soit dans la phase de formation ou de test [10, 14, 11]. Un adversaire peut tromper le modèle cible en créant des perturbations subtiles connues sous le nom d'exemples adverses (AE). Par conséquent, le déploiement de systèmes de détection et de classification basés sur l'apprentissage dans des domaines sensibles à la sécurité nécessite une réflexion approfondie. Malgré la popularité des systèmes de détection basés sur l'apprentissage et leur grande adaptabilité, ils sont le maillon faible de la sécurité et on ne peut pas leur faire confiance aveuglément.

# Contributions

A partir de la problématique évoquée dans la section précédente, nous proposons plusieurs contributions originales dans cette thèse. Ce travail de recherche comprend deux phases. Dans la première phase, nous présentons un nouveau système de détection de logiciels malveillants Android basé sur des images utilisant CNN qui ne nécessite aucune analyse de code et aucune ingénierie de fonctionnalités fines pour la détection de logiciels malveillants et la classification des familles. Notre objectif est de créer un système de détection de logiciels malveillants léger, robuste et évolutif en appliquant une transformation intelligente des fichiers APK en images RVB. Nous envisageons que la transformation systématique et intelligente d'un APK en une image RVB créera des modèles spécifiques pour les images bénignes et malveillantes qui simplifieront le processus de classification pour tout classificateur d'apprentissage. Le système classe la signature de l'image résultante comme étant soit malveillante soit bénigne et identifie en outre la famille de logiciels malveillants prédits. Nous examinons la taxonomie des attaques adverses sur les systèmes de détection de logiciels malveillants Android basés sur l'apprentissage automatique dans la deuxième phase. Nous évaluons la robustesse de notre système de détection de logiciels malveillants Android basé sur l'image appelé RoboDroid face à de nouvelles attaques adverses. La principale contribution des travaux de recherche est énumérée comme suit:

## Une transformation systématique des APK Android en images

Notre système de détection de logiciels malveillants Android basé sur l'image transforme systématiquement l'APK Android en une image RVB. Contrairement à une transformation aveugle du bytecode en une image en niveaux de gris, le système mappe la fonctionnalité des fichiers dex et manifeste sur les canaux rouge, vert et bleu d'une image à l'aide de notre dictionnaire épuisé [8]. Le dictionnaire contient des appels d'API suspects, des autorisations, des composants d'application et d'autres comportements malveillants. Le dictionnaire est conçu en s'inspirant de nombreuses études présentées dans la revue de la littérature et de nos vastes expériences sur l'énorme ensemble de données d'AndroZoo [2]. Le système proposé utilise un outil d'ingénierie inverse léger *Androguard* pour extraire les composants d'application et les appels d'API des fichiers manifeste et dex. Le système convertit les autorisations extraites et les composants de l'application, y compris les intentions de filtrage, les activités, les récepteurs et les services, en valeurs de pixel à l'aide du code ASCII de chaque caractère [7]. Si l'autorisation extraite ou le composant d'application a un niveau

de protection "Normal", il est mappé sur le canal vert ; s'il correspond au dictionnaire, le système le mappe sur le canal bleu. De même, le système proposé extrait les appels API du fichier dex et mappe tous les appels API habituels sur le canal rouge et les appels API suspects sur le canal bleu en consultant le dictionnaire.

Nous envisageons que la transformation systématique et intelligente d'un APK en une image RVB créera des modèles spécifiques pour les images bénignes et malveillantes qui simplifieront le processus de classification pour tout classificateur d'apprentissage. Une autre considération derrière la nouvelle transformation basée sur l'image de l'APK est de le rendre plus difficile à manipuler par les attaquants malveillants. Il est relativement facile pour les attaquants malveillants d'injecter du contenu malveillant dans des représentations binaires et autres de l'APK Android, tandis que la transformation basée sur les pixels ajoute une sorte de cryptage naturel. Même s'ils parviennent à injecter du contenu malveillant, notre transformation innovante par canal le filtre à l'aide du dictionnaire exhaustif [9]. Après la transformation de l'APK en image RVB, nous avons des canaux de tailles variables en raison de la taille d'un APK, du nombre d'autorisations et de composants d'application utilisés, et des API appelées par un APK particulier. Cependant, notre système a besoin d'une image avec toutes les tailles de canal cohérentes pour alimenter le réseau neuronal convolutif (CNN). Par conséquent, nous utilisons l'interpolation du plus proche voisin (NNI) pour rendre la taille du canal cohérente sans perdre les informations [21]. De plus, notre méthode NNI garantit que les valeurs de pixel nouvellement générées doivent appartenir à APK plutôt que d'être interpolées avec des valeurs factices. Le système forme un CNN sur les images résultantes pour la classification des logiciels malveillants et bénins.

## Création d'un nouveau modèle d'attaque adverse

Pour approfondir nos recherches, nous avons conçu deux nouvelles attaques adverses afin d'évaluer la robustesse de notre système de détection de logiciels malveillants Android basé sur l'image. La perturbation adverse est un point chaud dans le domaine de la vision par ordinateur pour tromper les classificateurs d'apprentissage, mais présente des défis différents lorsqu'il s'agit du domaine des logiciels malveillants (binaire). En vision par ordinateur, le but est d'ajouter des perturbations infimes mais non perceptibles à l'œil humain et suffisamment fortes pour tromper le classificateur formé. Cependant, l'ajout de perturbations adverses à un domaine binaire est complexe car la modification des fonctionnalités booléennes de 0 à 1 et vice versa brisera probablement la contrainte de fonctionnalité des logiciels malveillants. La

règle de base de la perturbation adverse est de ne pas violer la sémantique des contraintes de fonctionnalité des logiciels malveillants ni de perturber l'exécution originale de l'application. En préservant les contraintes de fonctionnalité des logiciels malveillants, nous avons conçu deux nouvelles attaques adverses pour un système de détection de logiciels malveillants Android. a) Ajouter des propriétés bénignes uniquement à un échantillon de logiciel malveillant pour perturber l'intégrité du système, appelée Attaque 1. b) Ajouter une partie d'une image bénigne à un échantillon de logiciel malveillant à l'aide de Deep Convolutional Generative Adversarial Networks (DCGANs) [9], appelé Attaque 2. Nous prévoyons que toute légère perturbation ajoutée par un adversaire aux images APK transformées a une forte probabilité de perturber les logiciels malveillants ou les fonctionnalités de l'application. Quelques études ont évalué la robustesse adverse des systèmes de détection de logiciels malveillants Android, et la plupart d'entre elles ajoutent du bruit aléatoire ou des valeurs fictives pour concevoir des attaques adverses. L'attaque 1 ajoute des propriétés bénignes uniquement à un échantillon de malware et préserve la contrainte fonctionnelle du malware. Nous extrayons les propriétés bénignes uniquement du dictionnaire et les ajoutons à l'image APK du logiciel malveillant, c'est-à-dire comme si nous ajoutions des fonctionnalités bénignes dans un APK.

En préservant la contrainte de fonctionnalité des logiciels malveillants, nous avons introduit l'attaque 2, qui ajoute la partie d'une image bénigne à un échantillon de logiciels malveillants à l'aide de Deep Convolutional Generative Adversarial Networks (DCGAN). Attack2 implique un processus en deux étapes, a) il apprend la distribution bénigne à l'aide d'images bénignes (APK convertis) à partir de l'ensemble de données AndroZoo et génère des images bénignes. b) Sélectionnez les valeurs de pixel bénignes correspondant aux propriétés bénignes des images bénignes générées via DCGAN et ajoutées à l'APK du logiciel malveillant. Les images bénignes générées via les DCGAN sont testées via notre système de détection de logiciels malveillants Android à base d'images en tant que contrôle d'intégrité et donnent une précision de 99,85% Pour évaluer la robustesse de notre système de détection basé sur l'image face à ces deux attaques, nous avons implémenté deux autres approches basées sur l'apprentissage pour la détection des logiciels malveillants Android [29, 30]. Les résultats expérimentaux ont montré que le système de détection de logiciels malveillants Android basé sur l'image est plus résistant à ces attaques en raison de sa représentation d'application de pixels intelligents et construit une défense naturelle inhérente contre les attaques adverses par rapport aux approches susmentionnées.

## Attaque basée sur le gradient pour les systèmes de détection de logiciels malveillants Android basés sur des images.

Diverses techniques ont été proposées pour concevoir des attaques adverses dans le domaine de la vision par ordinateur, telles que JSMA (attaque par carte de saillance basée sur le jacobien) [26], attaques C&W [4], Deepfool [20], FGSM (méthode de signe de gradient rapide) [13]. Le problème avec les techniques mentionnées ci-dessus, comme JSMA, C&W, Deepfool et FGSM standard, ne garantit pas que les valeurs modifiées (perturbées) appartiennent à une propriété bénigne. Ces techniques manquent le contexte de la propriété APK (bénigne ou malveillante) et se concentrent sur la génération d'une combinaison de valeurs fictives (ni bénignes ni malveillantes) pour échapper au système de détection. Un autre fait concernant les techniques susmentionnées est les contraintes de normalisation, c'est-à-dire $L_2$ *Norm* ou $L_\infty$ *Norm* qui font la distinction entre l'image originale et l'image adverse dans des objets réels. Par exemple, considérons une image d'un chat ayant l'une des valeurs de pixel 220 (couleur blanche). Il ne peut perturber qu'avec une valeur de perturbation inférieure de 220 à 221 (blanc) pour avoir le même aspect que la contrainte de perturbation dans le domaine CV. D'un autre côté, si l'image est perturbée d'une valeur de pixel de 220 à 10 (noir), cela sera évident à l'œil nu. Contrairement au domaine CV, peu importe dans l'échantillon de logiciel malveillant si nous remplaçons une valeur de pixel de 220 par une autre propriété bénigne représentant une valeur de pixel de 10. La contrainte de perturbation visuelle ne s'applique pas au domaine du malware. Cependant, la perturbation devrait garantir que la fonctionnalité d'origine du logiciel malveillant reste inchangée.

Dans la suite des modèles d'attaque, nous présentons une autre nouvelle attaque adverse spécifique aux systèmes de détection de logiciels malveillants Android basés sur l'image. L'attaque FGSM (Fast Gradient Sign Method) modifiée (appelée Attaque 3 dans cette thèse) est conçue explicitement pour les systèmes de détection de logiciels malveillants Android basés sur l'image, mais elle renforce également notre modèle de menace existant, y compris l'attaque 1 et l'attaque 2. Attack3 comprend un processus en deux étapes inspiré de nos précédents travaux [9]. La première étape consiste à générer de faux APK bénins à l'aide de DCGAN. Le système apprend la distribution bénigne à partir de l'ensemble de données AndroZoo [2], et les APK générés sont validés à l'aide de notre système de détection des logiciels malveillants Android basé sur l'image [7]. Une petite partie de l'APK nouvellement généré par les GAN (quelques valeurs de pixel) est ajoutée avec un échantillon de logiciel malveillant original pour créer un exemple adverse en préservant la contrainte de fonctionnalité du logiciel malveillant. Dans la deuxième étape, nous pouvons encore

renforcer notre modèle de menace en utilisant le FGSM modifié. Le système n'a pas pu modifier davantage les propriétés bénignes ajoutées dans l'attaque 1 et l'attaque 2. Pour maximiser la perte pour les échantillons de logiciels malveillants perturbés, nous pouvons rendre notre modèle de menace plus efficace en modifiant les valeurs de pixel ajoutées dans l'attaque 1 et l'attaque 2 avec le FGSM modifié. Le système proposé ne déplace pas la valeur du pixel avec $\pm\ \varepsilon$ mais saute à la prochaine propriété bénigne générée par les GAN dans la direction du gradient. De cette façon, le système garantit que la valeur ajoutée aux échantillons de logiciels malveillants est purement une propriété bénigne et non une valeur factice comme in [1, 19, 23]. Le FGSM standard ajoute ou soustrait des valeurs de pixel en fonction du signe du gradient d'une image donnée. La norme FGSM ne s'applique pas à notre approche car elle peut nous conduire à une valeur fictive. Nous ne pouvons pas ajouter/soustraire une valeur à une valeur de pixel dans une image donnée (une valeur de pixel dans notre approche représente une propriété Android). Malgré l'ajout/la soustraction de valeurs aléatoires ou factices, le système choisit la prochaine propriété bénigne générée par les GAN ou proposée dans notre dictionnaire exhaustif [8] en fonction du signe du gradient.

## Évaluation de la robustesse dans le cadre d'attaques adverses nouvellement conçues

La robustesse du système proposé repose sur le fait que l'ajout d'une perturbation à l'image RVB transformée d'un échantillon de logiciel malveillant a une très forte probabilité de violer la contrainte de fonctionnalité du logiciel malveillant et peut démolir la fonctionnalité de l'application. Par conséquent, la représentation basée sur l'image construit un mécanisme de défense naturellement inhérent contre les attaques adverses et ne nécessite pas de formation adverse, ce qui est coûteux en calcul. Pour illustrer la robustesse du système de détection de logiciels malveillants basé sur l'image, nous avons mis en œuvre des techniques de détection de logiciels malveillants Android de pointe, telles que DroidDetector [29], DeepClassifyDroid [30] et des logiciels malveillants Android basés sur la visualisation. detector [31] ainsi que notre approche proposée utilisant le même référentiel d'ensembles de données AndroZoo [2]. Le RoboDroid et les trois autres approches mises en œuvre, DroidDetector, système de détection basé sur la visualisation, et DeepClassifyDroid, sont testés pour la robustesse face à face et le taux d'évasion sous le nouveau modèle de menace. Les résultats expérimentaux et l'évaluation détaillée ont révélé que le RoboDroid est plus sécurisé que les autres approches de détection de logiciels malveillants Android basées sur l'apprentissage contre les attaques adverses. Les résultats ont démontré que le taux d'évasion pour DeepDroid [29],

DeepClassifyDroid [30] et l'approche basée sur la visualisation [31] est supérieur à 50% en ajoutant 12-13% de perturbations adverses . Alors que le taux d'évasion de RoboDroid (notre système de détection de logiciels malveillants Android basé sur l'image) atteint 13% avec une perturbation maximale. Cependant, dans quelques scénarios, il a construit une ligne de défense durable et aucun taux d'évasion contre une perturbation de 3-4% par rapport aux autres approches.

# Liste des publications

- Malware Forensics: Legacy Solutions, Recent Advances, and Future Challenges. Un chapitre de livre publié dans Advances in Computing, Informatics, Networking and Cybersecurity, (Springer 2022) [22].
- RGB-based android malware detection and classification using convolutional neural network. Un article publié dans le cadre de la Conférence mondiale sur les communications de l'IEEE (GLOBECOM 2020) [7]
- Robustness of image-based android malware detection under adversarial attacks. Un article publié dans l'IEEE International Conference on Communications (ICC-2021) [9]
- An intelligent malware detection and classification system using apps-to-images transformations and convolutional neural networks. Un article publié dans le cadre de la 16e Conférence internationale sur l'informatique sans fil et mobile, les réseaux et les communications (WiMob 2020) [21]
- An Adversary-aware Android Malware Detector Under FGSM-Based Attacks. Soumis aux transactions IEEE sur l'informatique fiable et sécurisée (IEEE TDSC 2022)
- Robodroid: A Robust Image-based Android Malware Detection System Against Adversarial Attacks. Soumis à Journal Computers & Security (Elsevier 2022)

# References


[1] 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.

[2] 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.

[3] Android Market. Android smartphone market share report and analysis, Last visited November 2021. URL https://www.idc.com/promo/smartphone-market-share.

[4] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pages 39–57. IEEE, 2017.

[5] 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.

[6] Xuesong Chen, Xiyu Yan, Feng Zheng, Yong Jiang, Shu-Tao Xia, Yong Zhao, and Rongrong Ji. One-shot adversarial attacks on visual tracking with dual attention. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10176–10185, 2020.

[7] A. Darwaish and F. Naït-Abdesselam. Rgb-based android malware detection and classification using convolutional neural network. In *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, pages 1–6. IEEE, 2020.

[8] Asim Darwaish. Source code and dictionary, rgb-based-andorid-malware-detection [online] available: https://github.com/asimswati553/., Last visited October 2020. URL Available:https://github.com/asimswati553/.

[9] 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.

[10] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *28th USENIX security symposium (USENIX security 19)*, pages 321–338, 2019.

[11] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[12] 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.

[13] Sandy Huang, Nicolas Papernot, Ian Goodfellow, Yan Duan, and Pieter Abbeel. Adversarial attacks on neural network policies. *arXiv preprint arXiv:1702.02284*, 2017.

[14] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 19–35. IEEE, 2018.

[15] TaeGuen Kim, BooJoong Kang, Mina Rho, Sakir Sezer, and Eul Gyu Im. A multimodal deep learning method for android malware detection using various features. *IEEE Transactions on Information Forensics and Security*, 14(3):773–788, 2018.

[16] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European Signal Processing Conference (EUSIPCO)*, pages 533–537. IEEE, 2018.

[17] 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.

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

[19] Xinbo Liu, Yapin Lin, He Li, and Jiliang Zhang. Adversarial examples: Attacks on machine learning-based malware visualization detection methods. *arXiv preprint arXiv:1808.01546*, 10(3326285.3329073), 2018.

[20] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2574–2582, 2016.

[21] Farid Naït-Abdesselam, Asim Darwaish, and Chafiq Titouna. An intelligent malware detection and classification system using apps-to-images transformations and convolutional neural networks. In *2020 16th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)(50308)*, pages 1–6. IEEE, 2020.

[22] Farid Naït-Abdesselam, Asim Darwaish, and Chafiq Titouna. Malware forensics: Legacy solutions, recent advances, and future challenges. In *Advances in Computing, Informatics, Networking and Cybersecurity*, pages 685–710. Springer, 2022.

[23] Robert Podschwadt and Hassan Takabi. On effectiveness of adversarial examples and defenses for malware classification. In *International Conference on Security and Privacy in Communication Systems*, pages 380–393. Springer, 2019.

[24] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[25] Xu Weilin, Qi Yanjun, and D Evans. Automatically evading classifiers. *Proc ofthe 23rd Annual Network and DistributedSystem SecuritySymp (NDSS2016). Reston, VA, USA: TheInternetSociety*, 2016.

[26] Rey Wiyatno and Anqi Xu. Maximal jacobian-based saliency map attack. *arXiv preprint arXiv:1808.07945*, 2018.

[27] Worldwide users. Smartphone users worldwide, Last visited November 2021. URL https://www.bankmycell.com/blog/how-many-phones-are-in-the-world.

[28] Wei Yang, Deguang Kong, Tao Xie, and Carl A Gunter. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 288–302, 2017.

[29] Zhenlong Yuan, Yongqiang Lu, and Yibo Xue. Droiddetector: android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, 21(1):114–123, 2016.

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

[31] Yong-liang Zhao and Quan Qian. Android malware identification through visual exploration of disassembly files. *International Journal of Network Security*, 20(6): 1061–1073, 2018.