Design & Development of Malware Detection Technique for Android Based Smart Devices

A Thesis

Submitted in partial fulfilment of the requirements

for the award of the degree of

Doctor of Philosophy (Ph. D.)

by

Rahul Gupta

(Roll No. 2K17/PhDIT/07)

Under the supervision of

Dr. Kapil Sharma
Professor
Department of Information Technology,
Delhi Technological University,
Delhi –110 042

Dr. Ramesh Kumar Garg
Professor
Deenbandhu Chhotu Ram University of Science and Technology, Murthal, Sonipat, Haryana

Delhi Technological University
Shahbad Daulatpur, Main Bawana Road
Delhi-110042

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Shahbad Daulatpur, Main Bawana Road, Delhi-42

CERTIFICATE BY THE SUPERVISOR(s)

Certified that **Rahul Gupta (**2K17/PhDIT/07) has carried out their research work presented in this thesis entitled **"Design & Development of Malware Detection Technique for Android Based Smart Devices"** for the award of **Doctor of Philosophy** from Department of Information Technology, Delhi Technological University, Delhi, under our supervision. The thesis embodies results of original work, and studies are carried out by the student himself and the contents of the thesis do not form the basis for the award of any other degree to the candidate or to anybody else from this or any other University/Institution.

Dr. Kapil Sharma
Professor
Department of Information Technology,
Delhi Technological University,
Delhi –110 042

Dr. Ramesh Kumar Garg
Professor
Deenbandhu Chhotu Ram University of Science and Technology, Murthal, Sonipat, Haryana

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Shahbad Daulatpur, Main Bawana Road, Delhi-42

CANDIDATE'S DECLARATION

I **Rahul Gupta** hereby certify that the work which is being presented in the thesis entitled **Design and Development of Malware Detection Technique for Android Based Smart Devices,** in partial fulfilment of the requirements for the award of the Degree of Doctor of Philosophy, submitted in the Department of Information Technology , Delhi Technological University is an authentic record of my own work carried out during the period from August 2017 to June 2024 under the supervision of **Dr. Kapil Sharma**, Professor of Information Technology Department, Delhi Technological University, Delhi, India and **Dr. R.K. Garg**, Professor, Deenbandhu Chhotu Ram University of Science and Technology, Murthal, Sonipat, Haryana.

The matter presented in the thesis has not been submitted by me for the award of any other degree of this or any other Institute.

Candidate's
Signature

ACKNOWLEDGEMENTS

In servitude to God, I express my deepest gratitude for His almighty grace, which has guided and supported me throughout the course of this thesis.

First and foremost, I am sincerely grateful to my advisors, **Prof. Kapil Sharma** and **Prof. R.K. Garg**, for their invaluable guidance, support, and encouragement. Their expertise and insights have been instrumental in shaping this research and bringing it to fruition.

I am deeply indebted to my mother, **Late Smt. Rekha Gupta**, for her blessings and teachings and my father, **Sh. Mahesh Gupta**, for his unwavering love and belief in me throughout this PhD journey. None of this would have been possible without their unconditional love and trust.

Many thanks to my wife **Mrs. Anjali Gupta**, and my sister **Ms. Palak Gupta** for their constant support, understanding, and prayers. A special mention goes to my little son, **Himaksh Gupta**, who brings so much joy into my life. I can never thank you all enough; your belief in me has been a profound source of strength and motivation.

I would also like to express my heartfelt gratitude to the **Hon'ble Vice Chancellor** of Delhi Technological University and the **Head of the Department** of Information Technology, Delhi Technological University, for their unwavering support and encouragement. Special thanks go to my friends **Dr. Ansul Arora** for his insightful comments and stimulating discussions. His camaraderie and encouragement have made this journey more enjoyable and fulfilling.

Thank you all for being a part of this journey.

Rahul Gupta

August, 2024

List of Publications

Journals:

1. Rahul Gupta, Kapil Sharma, and R.K. Garg, “Innovative Approach to Android Malware Detection: Prioritizing Critical Features Using Rough Set Theory,” Electronics, 13(3), 482, (2024).
2. Rahul Gupta, Kapil Sharma, and R.K. Garg, “Covalent Bond Based Android Malware Detection Using Permission and System Call Pairs,” Computers, Materials & Continua, 78(3), 4283-4301, (2024).

International Conferences:

1. Rahul Gupta, Kapil Sharma, and R.K. Garg “Android Malware Detection based on Feature-pair Bonding: A Hybrid Detection Model” in the 2023 5th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N), Nodia, India
2. Rahul Gupta, Kapil Sharma, and R.K Garg “A Visual Android Malware Detection Technique based on Process Memory Dump Files” in the Proceedings of Fifth International Conference on Computing, Communications, and Cyber-Security: IC4S'05 Volume 2, Uttarakhand, India.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

API	Application Programming Interface
APK	Android Application Package
BEN	Benign Android Apps
CNN	Convolutional Neural Network
ANN	Artificial Neural Network
EM	Encrypted Android Malware
FNR	False Negative Rate
FPR	False Positive Rate
TPR	True Positive Rate
KNN	K Nearest Neighbors
JNI	Java Native Interface
OS	Operating System
PCs	Personal Computers
RAM	Random Access Memory
TNR	True Negative Rate
URL	Uniform Resource Locator
SVM	Support Vector Machine
RF	Random Forest
iOS	iphone Operating System

ABSTRACT

The widespread integration of smartphones into modern society has revolutionized communication, work, entertainment, and access to information, with Android-based devices dominating the market, accounting for approximately 70% of global smartphone usage. However, this popularity has made Android devices prime targets for malware attacks, posing serious threats due to the sensitive personal and financial data they store. Consequently, there is an urgent need for innovative and effective malware detection techniques.

Our study addresses this challenge by introducing three novel approaches to Android malware detection. First, we applied rough set theory to select and rank static features such as permissions, API calls, system commands, and opcodes, using a Discernibility Matrix to assign importance to each feature and calculate reducts—streamlined subsets that enhance detection accuracy while minimizing complexity. Machine learning algorithms, including Support Vector Machines (SVM), K-Nearest Neighbor (KNN), Random Forest, and Logistic Regression, were employed to achieve an impressive 97% detection accuracy, surpassing many state-of-the-art techniques.

Secondly, we pioneered a hybrid method by establishing covalent bonds between permissions and system calls, combining static and dynamic analysis to uncover malicious behavior. A novel Covalent Bond Strength Score was introduced to assess the combined impact of these pairs, with distinct scores for benign and malicious behaviors. This approach provided a comprehensive framework for malware detection, achieving a detection accuracy of 97.5%, further improving upon existing methods.

Lastly, we developed a visual malware detection technique based on Android process memory dumps. The memory dump files were transformed into grayscale images, from which features such as color histograms, Hu moments, and Haralick textures were extracted. These features were used to train machine learning classifiers to differentiate between benign and malicious applications. Among the classifiers tested, Random Forest delivered the best performance.

In conclusion, our integrated approaches provide robust frameworks for Android malware detection, each contributing significant advancements to the field and demonstrating superior performance compared to existing technique

Chapter One: INTRODUCTION

This chapter introduces the concept of malware, types of malwares, Android malware and security issues in Android. The objectives of the research work are highlighted. Chapter wise thesis coverage is summarized at the end of the chapter.

1.1 Malware

Malware refers to any type of malicious software or program code created with the intent to harm, exploit, or infiltrate a device without the user's consent. Examples of malware include Trojans, rootkits, and backdoors. These harmful programs can carry out various harmful activities, such as stealing sensitive information, encrypting or deleting data, taking control of critical system functions, and monitoring a user's actions without their knowledge or approval. Malware often disrupts normal operations, compromising both the security and privacy of the affected device.

1.1.1 Types of Malwares

The different types of malware function in distinct ways, depending on their purpose and design. The Figure 1.1 Shows various type of malware based on the purpose and design.

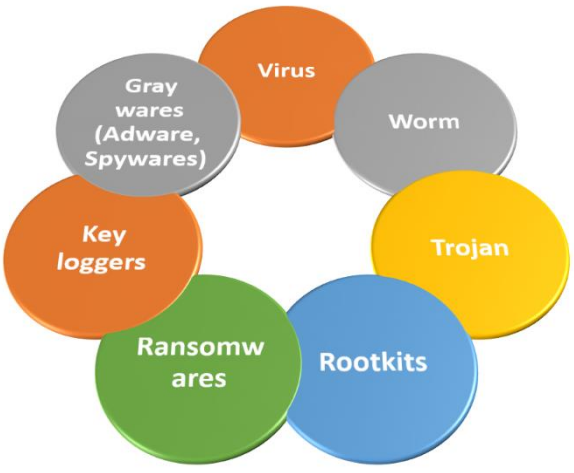

Figure 1.1 Categories of Malwares

1.1.1.1 Viruses

A Virus is a form of malicious software that embeds itself into legitimate files or programs and becomes active when the infected file is run. Its main goal is to replicate and spread across systems, typically through file sharing, email attachments, or exploiting network vulnerabilities. Once triggered, a virus can cause various types of harm, such as damaging or erasing data, disrupting system operations, stealing sensitive information, or opening pathways for further attacks. Certain viruses, like polymorphic variants, can modify their code to avoid detection by security software. The consequences can range from minor performance issues to significant data breaches and security threats.

1.1.1.2 Worm

A worm is a form of malware that replicates itself and spreads through networks without attaching to a specific file or program, setting it apart from traditional viruses. Worms take advantage of weaknesses in operating systems, applications, or network protocols to access systems, often doing so without requiring any action from the user. Once inside, a worm can quickly duplicate itself and send copies to other devices within the same network or across the internet. Unlike viruses, which rely on users to execute infected files, worms operate independently and can spread independently, making them especially dangerous.

Worms can cause various levels of harm. In some instances, they may simply consume network bandwidth or overload system resources, leading to performance issues or crashes. However, more advanced worms are capable of delivering malicious payloads, such as installing backdoors for unauthorized access, deleting files, or distributing other types of malware like ransomware or spyware. Some worms can spread globally, causing widespread chaos, as seen with the ILOVEYOU and WannaCry worms, which infected millions of devices worldwide. Because of their ability to propagate quickly and cause damage without human interaction, worms are considered one of the most dangerous types of malware in networked environments.

1.1.1.3 Trojan

A Trojan, also known as a Trojan horse, is a type of malicious software that masquerades as a legitimate or beneficial application to deceive users into installing

it. Unlike viruses or worms, Trojans do not replicate or spread on their own but rely on social engineering techniques, such as pretending to be a helpful tool, game, or software update, to lure users into running the malicious code.

Once installed, a Trojan can perform a variety of harmful tasks, depending on its design. Some Trojans act as backdoors, granting attackers remote access to the infected system, enabling them to steal data, install more malware, or take control of the device. Others may track keystrokes to collect sensitive information like passwords or financial details, while some can disable security measures, making the system more vulnerable to additional attacks. Banking Trojans specifically aim to steal credit card numbers, banking credentials, and other financial data.

Trojans are typically spread through phishing emails, compromised websites, or deceptive downloads. After being installed, they can remain hidden for long periods, secretly gathering information or maintaining access for attackers. Well-known examples include the Zeus Trojan, which was involved in widespread financial theft, and Emotet, which has been used to spread ransomware or support botnet operations. Because they don’t self-replicate and often appear as legitimate programs, Trojans are particularly challenging to detect, making them one of the most dangerous forms of malware.

1.1.1.4 Rootkits

A rootkit is an advanced form of malware designed to grant unauthorized access to a system while remaining hidden from detection. By providing attackers with administrative or "root" level privileges, rootkits allow them to control the system at a fundamental level without being noticed. These tools are particularly dangerous because they can conceal both themselves and other malicious software from conventional antivirus programs and system monitoring tools, making detection and removal exceptionally challenging.

Rootkits achieve this by altering system files, intercepting system calls, or embedding themselves directly into the operating system's kernel. Once established, they enable a wide range of malicious activities, such as data theft, remote command execution, user activity monitoring, disabling security mechanisms, and facilitating the

deployment of other malware. Due to their stealth capabilities, rootkits allow attackers to maintain long-term control over compromised systems without alerting the user to the breach.

Installation of rootkits can occur through various methods, including phishing attacks, drive-by downloads, or the exploitation of software vulnerabilities. Rootkits are often classified by their operational depth, with kernel-level rootkits being the most severe, as they integrate into the core of the operating system, making them nearly impossible to detect or remove without specialized tools. User-level rootkits, on the other hand, focus on specific applications or services.

A notable example of rootkit misuse was the Sony BMG incident, where a rootkit was covertly installed on users' computers via music CDs to enforce digital rights management (DRM), inadvertently creating significant security vulnerabilities. Given their ability to avoid detection and provide deep system control, rootkits represent a significant threat in the field of cybersecurity, and their removal often requires advanced techniques such as booting into secure environments or using specialized rootkit detection tools.

1.1.1.5 Ransomwares

Ransomware is a type of malware that restricts access to a computer system or its data, usually by encrypting files, and demands a ransom payment, often in cryptocurrencies like Bitcoin, for the decryption key. However, paying the ransom does not guarantee that the data will be recovered, and attackers may withhold the key or even re-target the victim later.

This malware is typically spread via phishing emails, malicious attachments, or compromised websites that exploit security flaws. Once activated, ransomware locks users out by encrypting critical files or the entire system, rendering them unusable. A ransom message is then displayed, providing instructions on how to make the payment, usually with a deadline and the threat of permanent data destruction if the ransom is not paid.

Ransomware attacks can differ in complexity. For instance, crypto-ransomware encrypts files, making them unusable without the decryption key, while

locker ransomware prevents access to the system itself without necessarily encrypting files. More advanced versions, such as double extortion ransomware, not only encrypt files but also threaten to release sensitive information unless the ransom is paid, intensifying pressure on the victim.

Significant ransomware attacks have affected many industries, including healthcare and critical infrastructure. Noteworthy examples include the WannaCry attack in 2017, which compromised hundreds of thousands of systems globally by exploiting a Windows vulnerability, and the Colonial Pipeline attack in 2021, which caused fuel supply disruptions across parts of the U.S.

Preventing ransomware requires a combination of defensive measures, including regular software updates, strong passwords, multi-factor authentication, consistent backups, and user education to avoid phishing scams. Victims are often discouraged from paying the ransom, as it does not ensure data recovery and may encourage future attacks. Instead, reporting incidents to authorities and using backups or decryption tools is the recommended approach for recovery.

1.1.1.6 Keyloggers

Keyloggers are a type of malicious software or hardware designed to covertly record every keystroke on a device, enabling attackers to gather sensitive information such as passwords, credit card details, and personal messages. These tools operate stealthily, making them difficult for users to detect. There are two primary categories: software-based and hardware-based keyloggers. Software keyloggers are usually introduced into a system through phishing attacks or malware infections, capturing keystrokes at different levels, from application-level to deeper kernel-level operations. Hardware keyloggers, on the other hand, are physical devices inserted between the keyboard and computer, or integrated into the keyboard itself, logging keystrokes and either storing the data or sending it wirelessly.

Keyloggers are often deployed via malicious emails, infected downloads, or compromised websites, and can remain hidden while collecting information over long periods. The stolen data is then sent to attackers, who use it for identity theft, financial fraud, or unauthorized system access. This makes keyloggers particularly hazardous

for both individuals and businesses. For individuals, they can lead to the loss of personal data and financial harm. In organizations, keyloggers can result in serious security breaches, exposing confidential information and giving attackers access to critical systems.

Effective prevention of keylogger threats requires a layered defense strategy. Antivirus and anti-malware software are useful in detecting and removing most software keyloggers, though some advanced variants can evade detection. Anti-keylogging tools, two-factor authentication, and keeping systems updated with security patches can also help reduce risks. To protect against hardware keyloggers, maintaining physical security and regularly inspecting computer devices for suspicious attachments is crucial. Keyloggers have been involved in some of the most notorious cyberattacks, such as the Zeus Trojan, which targeted online banking data, illustrating the significant damage they can cause if not addressed.

1.1.1.7 Graywares

Grayware refers to software that falls between legitimate programs and malicious software. While it may not inflict direct harm like viruses or ransomware, it can still degrade system performance, compromise privacy, and negatively affect user experience. Grayware encompasses various unwanted applications, including adware, spyware, and potentially unwanted programs (PUPs), which are often unknowingly installed alongside free software or through deceptive ads and insecure websites. For instance, adware inundates users with intrusive advertisements, while spyware tracks user activity and gathers sensitive data without permission. PUPs may introduce unnecessary toolbars or alter browser settings without consent. Though typically less destructive, grayware can slow down systems, invade user privacy by collecting information for third parties, and increase the risk of more dangerous malware infections. Grayware often spreads unnoticed through bundled software or phishing attacks. Despite being less harmful than other types of malware, grayware can still lead to significant performance issues, security vulnerabilities, and privacy concerns, making its removal and prevention essential. Preventive measures include downloading software from trusted sources, carefully reviewing installation options to avoid bundled software, and using security tools to detect and remove grey ware.

1.2 Malware in Smartphones

Smart devices are electronic gadgets that can connect, share, and interact with their users and other devices. They utilize advanced computing and connectivity technologies to provide enhanced functionality and convenience. Smart devices encompass a wide range of gadgets that enhance our daily lives through advanced technology and connectivity.

There are different types of smart devices available in today's world like smartphones, smart watches, smart home devices, smart lighting, smart security systems, smart appliances, smart TVs and entertainment systems, smart cars, smart health devices etc. The smartphone has established themselves as the dominant smart devices in the modern digital landscape. Their ubiquity, versatility, and continuous innovation set them apart from other smart devices, making them indispensable to our daily routines.

Smartphones have become the personal desktop computers of the modern era, integrating seamlessly into our daily lives and profoundly influencing various facets of contemporary society. These devices offer a wide range of capabilities once reserved for Personal Computers (PCs), including browsing the internet, managing emails, capturing high-quality photos and videos, and using navigation tools. Their versatility extends to online shopping, gaming, social networking, and location-based services [1, 2, 3, 4].

In recent years, the increased connectivity options in smartphones, such as Bluetooth, GPRS, and Wi-Fi, [5, 6] have significantly enhanced the availability of these ubiquitous services [7, 8]. This connectivity, coupled with feature-rich apps, has made smartphones far more powerful than early PCs, escalating their popularity [9, 10]. Smartphones are not just tools for communication; they also support education, personal organization, health monitoring, and control of smart home devices, revolutionizing how we work, entertain ourselves, and access information.

In today's digital age, smartphones are indispensable, functioning as mini-computers that fit into our pockets. They enable us to perform a wide array of tasks

that were once limited to desktops, transforming various aspects of our lives. Here's a deeper look into their impact

- ✓ **Communication**: Beyond calls and texts, smartphones allow instant messaging, video calls, and social media interactions, keeping us connected globally.
- ✓ **Work**: With mobile office apps, email management, and remote conferencing tools, smartphones facilitate productivity and remote work, blurring the lines between office and personal time.
- ✓ **Entertainment**: Streaming services, gaming apps, and multimedia capabilities provide endless entertainment options that are accessible anytime and anywhere.
- ✓ **Information Access**: Real-time news updates, educational content, and digital libraries make smartphones a vital tool for staying informed and continuing education.
- ✓ **Health and Fitness**: Health apps and wearable integrations help monitor physical activity, diet, and overall well-being, promoting a healthier lifestyle.
- ✓ **Convenience**: Online shopping, mobile banking, and digital payments simplify daily transactions, making financial management and shopping more efficient.
- ✓ **Navigation and Travel**: GPS and location-based services guide us through unfamiliar territories, enhancing travel experiences and daily commutes.
- ✓ **Smart Home Integration**: Smartphones control various devices, from lighting and security systems to home entertainment, creating a more connected and automated living environment.

Smartphones have thus become integral to modern life, offering functionalities that significantly enhance our efficiency, connectivity, and overall quality of life. The dominance of smartphones is evident in their market share, which is now 20% higher than desktops[1]. The Figure 1.2 shows the Desktop vs Mobile vs Tablet market share

[1] https://techjury.net/blog/mobilevsdesktop-usage/

worldwide from 2014 to 2024. This shift underscores the growing preference for mobile devices, highlighting how smartphones have surpassed desktops in terms of usage[2]. Their convenience, portability, and multifunctionality make them indispensable in our fast-paced, technology-driven world.

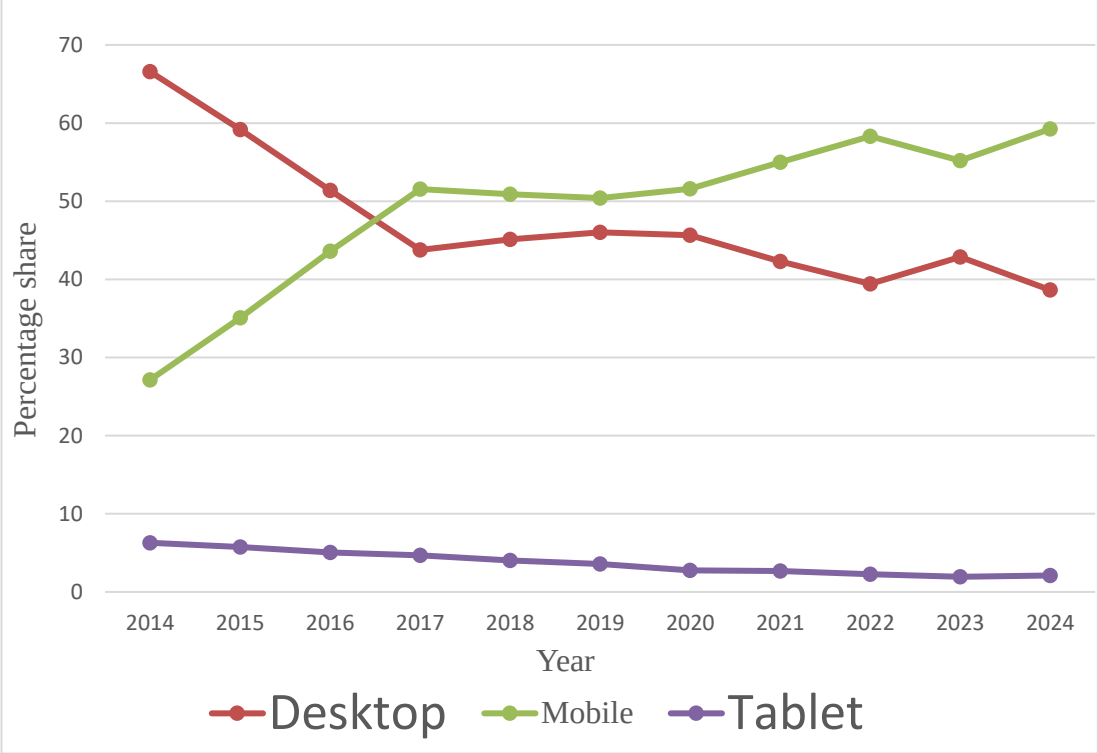

Figure 1.2 Desktop vs Mobile vs Tablet Market Share Worldwide

Among the various types of smartphones available in the market, those running the Android operating system are the most popular. This popularity can be attributed to the fact that Android is an open-source platform adopted by numerous manufacturers. According to a report by Statcounter[3], the Android operating system dominates the global mobile market, holding a 70% share. This substantial market share is a key factor behind the frequent malware attacks targeting the Android platform in recent years. The Figure 1.3 shows the Mobile Operating System Market Share Worldwide 2014 to 2024. The figure depicts that Android has been the leading

[2] https://gs.statcounter.com/platform-market-share/desktop-mobile tablet/worldwide/#yearly-2014-2024
[3] https://gs.statcounter.com/os-market-share/mobile/worldwide/#yearly-2014-2024

mobile OS since 2014, and its market share has consistently increased in the last few years.

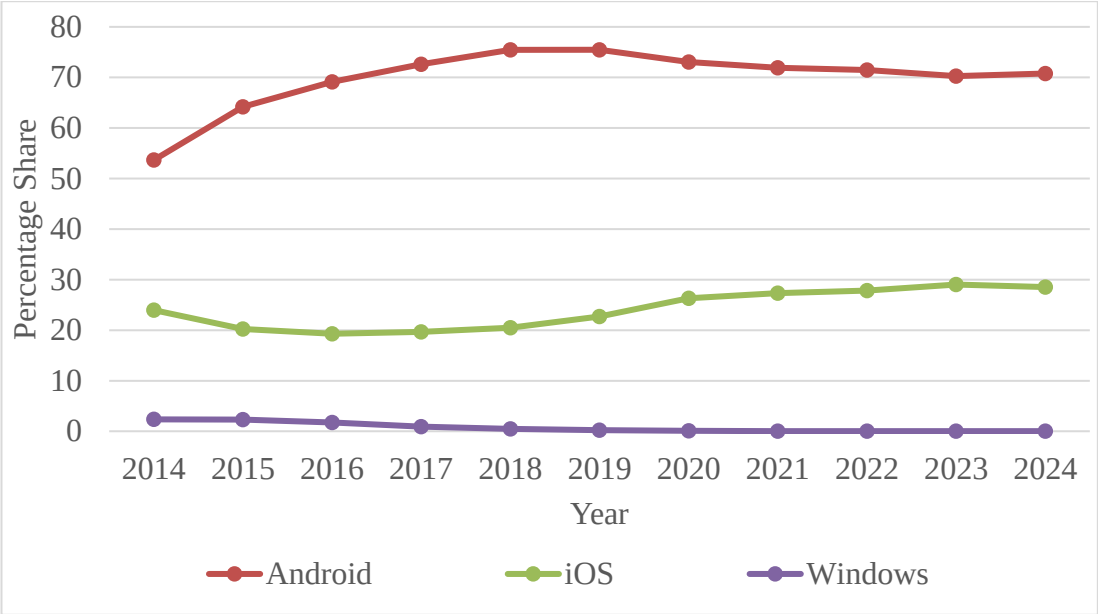

Figure 1.3 Mobile Operating System Market Share Worldwide 2014 - 2024

Android's widespread adoption and open-source nature make it an attractive target for cybercriminals, leading to a higher incidence of security threats than other mobile operating systems. The Android operating system has held a dominant position in the smartphone industry for the past decade. This dominance is partly due to its open-source nature, which encourages widespread adoption by numerous manufacturers. Within the Android API framework, functions that provide access to sensitive system resources are available. Unfortunately, this accessibility has been exploited by cyber attackers who develop and distribute malicious applications via alternative app stores or through social media advertisements. Attackers can also embed harmful components within legitimate Android applications.

These malicious applications enable attackers to perform various harmful activities, including stealing personal information, sending unauthorized SMS messages, and remotely controlling the device. As a result, it is crucial to implement robust security measures to protect smartphones from these threats. Given the high market share of Android devices, the frequency and sophistication of these attacks

underscore the need for vigilant security practices to safeguard user data and maintain device integrity [11, 12, 13].

1.3 Android Mobile Malware

In several ways, mobile devices or smartphones present more significant security risks to users compared to traditional PCs [11] . These devices are equipped with sensors that can inadvertently expose stored information, including images, videos, and even the user's location [12]. Moreover, many users store sensitive data, such as banking details or authentication credentials, on their smartphones, making these devices attractive targets for attackers. The rapid rise in smartphone popularity, along with widespread user adoption, has been accompanied by a corresponding increase in malware attacks.

According to a report[4] , several mobile Trojan subscribers were discovered on Google's official app marketplace in 2022. A blog post by the renowned antivirus firm McAfee revealed that 60 Android apps, with 100 million downloads, were spreading a new strain of malware to unsuspecting users[5] . TechRadar reported a new ransomware named "Daam," which can evade antivirus software[6]. Statista[7] shared data showing that 5.6 million Android malware samples were identified in 2020, with millions more detected annually from 2016 to 2020. The surge in Android malware and riskware activity throughout 2023 marks a concerning shift after a period of relative calm. Reaching levels reminiscent of early 2021 by year-end[8]. Android devices are 50 times more likely to be infected with malware than iOS devices[9]. These findings highlight the urgent need for robust Android malware detection mechanisms

[4] https://securelist.com/mobile-threat-report-2022/108844/
[5] https://www.mcafee.com/blogs/other-blogs/mcafee-labs/goldoson-privacy-invasive-and-clicker-android-adware-found-in-popular-apps-in-south-korea/
[6] https://www.techradar.com/news/this-dangerous-new-malware-also-has-ransomware-capabilities
[7] https://www.statista.com/statistics/680705/global-android-malware-volume/
[8] https://securelist.com/mobile-malware-report-2023/111964/
[9] https://www.getastra.com/blog/security-audit/malware-statistics/

to effectively identify and mitigate malicious applications. The increase in malware attacks on the Android platform can be attributed to several factors [16]:

- ✓ The widespread use of Android worldwide means that many users store sensitive information on their smartphones, making them lucrative targets for identity theft by malware developers.
- ✓ Android's open-source kernel policy allows attackers to gain a deep understanding of potential vulnerabilities within the system's architecture.
- ✓ Third-party app markets provide an easy avenue for distributing malicious applications.
- ✓ The similarity between desktop operating systems and Android makes it easier for attackers to adapt their techniques from desktop environments to the Android platform.

There are several methods through which malicious content can be introduced into smartphones, including SMS/MMS, Bluetooth, app markets, internet downloads, and update attacks [17]. Here's a brief overview of each infection vector:

- ✓ **SMS/MMS:** Messaging services like SMS and MMS have been exploited as attack vectors by malware such as ComWar and Yxe on the Symbian platform and FakeToken on Android.
- ✓ **Bluetooth:** Attackers can use Bluetooth technology to spread malware between devices within communication range. For example, the Symbian-based Cabir and Android-based BlueFrag malware utilized Bluetooth to propagate across devices and steal data.
- ✓ **App Markets:** App markets are a common entry point for malware distribution. Many Android malware samples are introduced through these markets, often using the repackaging technique. This involves disassembling an existing app, embedding malicious code, and repackaging it. Malware families like jSMSHider, DroidDream, and BgServ on Android were spread using this method.
- ✓ **Internet Downloads:** The drive-by-download technique, commonly used on desktop systems, also applies to smartphones. Users are tricked into clicking malicious links, which then download harmful components in the background.

Malware such as GGTracker, JiFake, and Zitmo on Android, PhoneCreeper on Windows, and Ikee on iOS have infected devices through this method.

- ✓ **Update Attacks:** Many applications require periodic updates to stay current. Initially, an app might be harmless, but it can download malicious code during an update, thereby infecting the smartphone. Examples include AnserverBot, Plankton, and BaseBridge, which targeted Android devices using this attack strategy.

Table 1.1 outlines the different threats that smartphone malware poses to users and devices. These threats include system damage, financial loss, and data leakage, among others. In addition to these, mobile devices can be exploited by malware developers for cyberbullying and sending spam messages on Online Social Networks (OSNs) [18, 19, 20].

Table 1.1Threats Posed by Smartphone Malware

Threats		**Malware Example**
System Damage	Disable System Functions like Block the Calling Service	Fakebank (Android), Skulls (Symbian)
	Battery Draining	DrainerBot (Android),Cabir (Symbian)
	Change system configuration such as Wallpaper	ExpensiveWall (Android), Ikee (iOS)
Financial Loss	Send SMS / MMS	FakePlayer, HippoSMS (Android)
	Dialling premium numbers	BaseBridge, BeanBot (Android)
Information Leakage	Privacy Breach	BaseBridge (Android)
	Stealing Banking Information	EventBot (Android), ZeusMitMo (Symbian)
Remote Control	Mobile Botnet	ADRD, AnserverBot (Android)

1.4 Security Issues in Android

Android, being one of the most widely used mobile operating systems, faces various security issues due to its open nature, diverse hardware ecosystems, and popularity. While Google and manufacturers continually update and improve security measures, several vulnerabilities still exist that can be exploited by attackers. Here are some of the key security issues in Android:

1. **Information leakage:** It happens when users unknowingly grant too many permissions to apps, and the Android operating system doesn't enforce strict enough limits on how those permissions are used. This enables apps to access personal data like location, contacts, messages, and even the device's microphone and camera, which can then be misused or shared with third parties without the user's consent. Many apps, particularly those from third-party sources, request more permissions than they need, posing a privacy risk. Older Android versions worsened this issue by lacking detailed permission control, forcing users to either accept all permissions or skip the app altogether. While newer Android versions allow users to selectively manage permissions, many still approve access without understanding what data is being collected or how it's utilized. Additionally, some apps include third-party libraries or SDKs that gather and share user data for advertising or analytics, increasing privacy concerns. This leads to risks such as unauthorized data collection, profiling, and exposure to security threats like phishing or identity theft. To reduce these risks, users should carefully examine app permissions, remove unnecessary access, and use tools that focus on privacy. Android's evolving permissions system, alongside informed user practices, is key to minimizing information leakage and safeguarding user privacy.
2. **Privilege escalation:** Privilege escalation in Android occurs when an attacker gains elevated permissions or control over a device by exploiting system vulnerabilities, particularly within the kernel. The kernel, as the core component of the operating system, manages hardware resources and controls key system operations. When attackers find and exploit weaknesses in the

kernel, they can increase their access from a regular user or app level to that of a root or system administrator, effectively bypassing security controls and taking full control of the device. Attackers often gain initial access through methods like malicious apps or phishing, then leverage kernel vulnerabilities, such as buffer overflows or race conditions, to execute unauthorized code and escalate their privileges. With elevated access, they can alter system settings, steal sensitive data, install malware, or create backdoors for future attacks. This kind of attack is particularly dangerous because root-level access allows an attacker to override most security protections on the device. They can steal personal information, install persistent malware, or even make the device unusable by altering critical system files. Notable exploits, like the "Dirty COW" vulnerability, highlight how attackers have used kernel flaws to gain root access. Preventing privilege escalation requires keeping the system up-to-date with security patches, carefully managing app permissions, and avoiding practices like rooting that disable essential security features. By applying regular updates and leveraging security tools such as SELinux, the risk of privilege escalation can be significantly reduced, though it remains a serious threat if kernel vulnerabilities go unaddressed.

3. **Repacking of Application:** Repackaging of applications on Android is a significant security threat where attackers modify genuine apps by reverse engineering them and injecting malicious code before redistributing them to unsuspecting users. This process starts when attackers obtain the APK (Android Package) file of a legitimate app, decompile it using common tools, and then introduce harmful elements like spyware, malware, or adware into the app’s code. After making these changes, the attacker repackages the app, making it appear as though it is the original, unaltered version. The modified app is then shared through unofficial app stores, third-party websites, or via direct download links, often marketed as free or enhanced versions of popular apps. Because the repackaged app typically retains its core functionality, users may not realize that they have downloaded a compromised version. While the app continues to function as expected, it may secretly collect sensitive information such as passwords, financial details, or location data and transmit

it to the attacker. Additionally, repackaged apps can deliver harmful software like ransomware, install further malware, or convert the device into part of a botnet. This attack is facilitated by Android's open ecosystem, which allows apps to be downloaded from various sources beyond the Google Play Store, where security checks might be less strict or nonexistent. Users are often enticed by unofficial sources offering premium features or unlocked content, making them more susceptible to these threats. To defend against repackaged apps, it's important to download apps only from trusted sources, carefully review app permissions, use mobile security tools, and avoid apps that request excessive or unnecessary permissions.

4. **Denial of Service Attack:** A Denial of Service (DoS) attack on a smartphone occurs when an attacker deliberately overwhelms the device's resources, such as the CPU, memory, or network bandwidth, making it difficult or impossible to use. Malicious apps can be designed to carry out such attacks by overloading system resources or triggering excessive background processes that strain the device. For example, a malicious app may continuously send data requests or initiate tasks that exhaust the device's processor, leading to sluggish performance, system freezes, or crashes. In severe cases, the phone may become unresponsive, preventing users from performing basic functions like making calls, sending messages, or using other apps. DoS attacks may also target the device's network connection by flooding it with excessive traffic, causing significant slowdowns or disconnecting the device from the internet altogether. These attacks can interfere with daily usage, disrupt business activities, and cause the battery to drain faster than usual, forcing users to reboot their devices or uninstall problematic apps to regain control. In some cases, these malicious apps may also be part of a larger network of infected devices, turning the smartphone into a participant in a distributed denial-of-service (DDoS) attack. To prevent DoS attacks, users should avoid suspicious apps, manage app permissions carefully, and keep their devices updated with security patches to address potential vulnerabilities.
5. **Colluding:** Colluding attacks occur when multiple applications installed on a device work together to take advantage of the system's shared user ID (UID)

feature, typically available to apps signed with the same developer certificate. On Android, apps with the same certificate can share a UID, allowing them to exchange data and permissions without the usual separation enforced between different apps. This collaboration enables these apps to pool their permissions, allowing them access to more system resources and sensitive data than they could individually. Each app might request only a few basic permissions, but by sharing their access through a common UID, they collectively gain unauthorized access to critical information. For instance, one app may have permission to access the internet, while another can access a user's contacts or location data. While these permissions might seem benign on their own, the combination of these permissions across multiple colluding apps can result in sharing sensitive data, leading to privacy issues, data theft, or abuse of device resources. Colluding apps can also bypass typical security detection, as each app might appear harmless when viewed separately. Users may not realize that seemingly unrelated apps are working together to exploit system capabilities. To guard against these attacks, users should carefully review app permissions, limit the installation of apps from the same developer unless necessary, and use security tools that monitor for unusual data sharing between apps.

6. **Fragmentation and Delayed Updates:** Fragmentation and delayed updates present major security issues in the Android ecosystem, largely due to the diversity of devices, manufacturers, and customizations. Since Android is open-source, manufacturers can modify the operating system to fit their specific hardware, resulting in different versions and models across devices. When Google releases a new update or security patch, it has to pass through various manufacturers and carriers, who must adapt it to their devices before it reaches users. This process can lead to significant delays or, in some cases, prevent certain devices from receiving updates altogether. This problem is worsened by the fact that many devices continue to operate on outdated Android versions, making them susceptible to known security vulnerabilities. The delay in pushing out security patches and system updates leaves millions of devices at risk for cyberattacks. Additionally, many manufacturers stop supporting older models after a certain period, leaving them without necessary

updates. This fragmentation weakens Android's overall security and makes it challenging for users to access the latest protections and features. As a result, users need to be extra cautious, often relying on third-party security solutions or upgrading to newer devices that receive more timely updates.

7. **Malicious Apps and Google Play Store:** Malicious apps remain a significant security threat for Android users, even on the Google Play Store, which is generally seen as the safest place to download apps. Despite Google's security measures, such as Play Protect, some harmful apps manage to bypass detection and become available for users to download. These malicious apps often disguise themselves as legitimate tools, games, or services, tricking users into installing them. Once installed, they may engage in a variety of harmful activities, such as stealing personal data, tracking user behaviour, displaying unwanted ads, or installing additional malware. These apps sometimes request excessive permissions, such as access to contacts, messages, or location data, which they can misuse for purposes like identity theft or fraud. While Google continually works to find and remove these apps, the sheer number of apps on the platform means that some still slip through, especially when initially harmless apps turn malicious after updates. Users who download apps from unofficial sources face an even higher risk, as those apps are not subject to Google's security screenings. To stay secure, users should carefully manage app permissions, regularly review their installed apps, and avoid downloading software from untrusted sources.
8. **Weak Encryption and Data Security:** Encryption is designed to protect sensitive information, such as personal data, financial transactions, and communications, by converting it into a coded format that only authorized users with the correct decryption key can access. However, when encryption is either weak or poorly implemented, it can be easily compromised by attackers, leaving confidential data vulnerable to unauthorized access, theft, or manipulation. On Android, encryption is intended to protect both data stored on the device and information transmitted over networks. Vulnerabilities arise when apps use outdated encryption methods or fail to apply encryption, leaving user data, particularly at risk when transmitted over unsecured networks like

public Wi-Fi. For example, if login credentials are not encrypted properly, hackers can intercept them during transmission, potentially leading to account breaches. Additionally, some Android devices, especially older ones or those running outdated software versions, may not have encryption enabled by default, increasing the likelihood of data being exposed. Applications that store sensitive data locally without adequate encryption can make that data accessible if the device is lost, stolen, or compromised. Developers may also make critical mistakes in implementing encryption, such as embedding hardcoded encryption keys within the app, which can be extracted through reverse engineering. Weak encryption leaves individual users vulnerable to risks such as identity theft and financial fraud, and it also poses broader security concerns for organizations where employees use Android devices to access corporate networks and sensitive data. To minimize these risks, both Android devices and apps must adhere to robust encryption standards, ensure that software is regularly updated, and apply proper encryption techniques. Users should also take extra steps, such as enabling device encryption, using secure networks like VPNs, and avoiding apps that fail to follow best security practices.

9. **Rooting and jailbreaking** are processes that give users full access to their Android or iOS devices by bypassing manufacturer or operating system restrictions. While these actions provide benefits such as greater customization, the ability to remove unwanted pre-installed apps, and the option to install third-party apps unavailable on official platforms, they also introduce significant security risks. By obtaining root or administrative privileges, users can modify system files and install custom software, but this often disables the device's built-in security measures. Devices that have been rooted or jailbroken are particularly susceptible to malware, as they allow apps from unverified sources, which may not have been subjected to security screenings. Furthermore, these devices are more prone to data breaches, instability, and performance problems, especially when using custom ROMs or unauthorized modifications. Another drawback is that rooting or jailbreaking typically voids the device's warranty, leaving manufacturers unwilling to offer support for any

issues. Additionally, these devices may not receive important software updates or security patches, increasing their vulnerability to cyberattacks. While rooting or jailbreaking unlocks advanced customization and features, the associated security risks and potential loss of device stability and manufacturer support make it a risky decision.

10. **Open Wi-Fi Networks and Man-in-the-Middle Attacks:** Open Wi-Fi networks, commonly found in public places like cafes, airports, and hotels, pose serious security risks due to their lack of encryption. These networks are accessible to anyone within range, often without requiring a password or any form of authentication. While convenient, they provide an easy target for attackers looking to exploit vulnerabilities and intercept data being transmitted between a device and the network. Since these networks lack proper encryption, sensitive information such as login credentials, financial details, emails, and other personal data can be exposed to malicious actors. Users may mistakenly assume their connection is secure, not realizing that they are at greater risk of being targeted by cybercriminals. One of the primary threats on open Wi-Fi is a Man-in-the-Middle (MitM) attack, where an attacker inserts themselves between the user and the Wi-Fi network, intercepting the data that is being exchanged. In this attack, the hacker can monitor all communications, including sensitive information like passwords, credit card numbers, and private messages. Additionally, the attacker can manipulate the data being sent, potentially injecting malicious code that compromises the security of the device. MitM attacks are particularly dangerous because they often go unnoticed, as the attacker can make the connection appear normal to the user. In some cases, attackers may set up rogue Wi-Fi networks that appear to be legitimate public networks, tricking users into connecting. Once connected, the attacker gains full access to the user's internet traffic, allowing them to steal information or distribute malware. To protect against the risks of open Wi-Fi networks and MitM attacks, users should avoid sending sensitive information over unsecured networks, use Virtual Private Networks (VPNs) to encrypt their data, and ensure that they only visit websites secured with HTTPS. It is also wise to disable automatic connections to public Wi-Fi and exercise caution

when connecting to unknown or unsecured networks. Taking these steps can help reduce the chances of falling victim to a MitM attack or other forms of data interception on open Wi-Fi networks.

1.5 Motivations and Research Gaps

The work proposed in thesis aims to design and develop malware detection techniques for Android based smart phones. Smartphones have gained popularity over desktops because of their portability, continuous connectivity, and versatility. They provide users with the convenience of accessing the internet, apps, and communication tools from anywhere, making them an integral part of everyday life. Consequently, smartphones have become the primary device for many people, overtaking desktops in usage and engagement.

Android-based smartphones are more attack-driven in comparison to other mobile operating system-based smartphones. The increase in malware attacks on Android smartphones can be attributed to several factors. Android's widespread adoption makes it an attractive target for cybercriminals, and its open-source nature facilitates the exploitation of vulnerabilities. Fragmentation within the Android ecosystem often results in delayed security updates, leaving many devices vulnerable. Additionally, third-party app markets, which frequently lack robust security measures, and the common practice of repackaging legitimate apps with malicious code, exacerbate the issue. User behaviour, such as granting excessive permissions or downloading apps from unreliable sources, further heightens the risk. These elements combined have contributed to the sharp rise in malware attacks on Android devices.

The primary focus of this section is to describe the motivation behind the research work carried out in this thesis. The motivations are based on the research gaps identified during the literature survey. The following research gaps were identified from the literature survey:

1. On Android devices, one of the built-in defense mechanisms is the permissions system, which controls the access privileges granted to applications [21]. Despite this, the system has proven inadequate in preventing malware attacks. For example, when downloading an app, users must grant all the requested

permissions to proceed with the installation. Most users tend to overlook the permissions list and grant them without much consideration. Even those who do review the permissions may struggle to recognize potential risks. This vulnerability has been exploited by attackers to infiltrate the Android platform in recent years. Consequently, there is a need for more effective Android malware detection mechanisms to protect against these threats.

2. Most research in the field of Android malware analysis, particularly within the static analysis category, has concentrated heavily on the permissions component of the Android manifest file [22, 23, 24, 25, 26]. This focus is due to the critical role that permissions play in determining what a particular app can do on a device, such as accessing the camera, contacts, or location data. Significantly less amount of research considers another component of the manifest file. Also, .dex files which contain all Android classes compiled into dex file format are not largely utilized for malware analysis and detection in comparison to permissions.
3. Some malware samples are advanced enough to bypass static detection through update attacks. Therefore, a dynamic detection model is necessary to identify and counteract these types of threats. Many malware samples can evade static detection because they obfuscate their malicious component or download their malicious component at run time. Static-based solutions may not detect such stealthy obfuscated malware. Regarding dynamic analysis, Memory forensics has been used in Windows/Linux desktop systems to detect malicious activities [27, 28, 29, 30] to detect malicious activities, but to the best of our knowledge, it is still unexplored in the field of Android malware detection.
4. Both dynamic and static analysis methods have their strengths and weaknesses when it comes to malware detection. Static analysis often struggles with code obfuscation techniques, as well as polymorphic and metamorphic malware, whereas dynamic analysis is more effective in these cases by examining the runtime behaviour of a program, which is difficult to obfuscate. However, dynamic analysis is time-consuming, as each malware sample must be executed within a secure environment that differs from a real runtime setting, potentially leading to different behaviours [31, 32]. To overcome the

limitations of both approaches, a combined method that integrates static and dynamic features appears promising for malware classification. Despite this, most research has focused on either static or dynamic analysis individually, with limited exploration of hybrid approaches.

1.6 Research Objectives

The previous section briefly explained the motivations and research gaps in the literature. This section briefly highlights the research objectives as follows:

1. To design and develop a novel malware detection technique using static features either from Android manifest files or .dex files or combination of both.
2. To design and develop a new hybrid malware analysis technique based on static and dynamic features in optimal combination for Android devices.
3. To develop a memory forensics-based technique for classifying Android-based applications for malware detection.

1.7 Contribution of Research Work

Figure 1.4 depicts the broad overview of the research performed at different stages of this thesis. We proposed three Techniques for Android malware detection. We briefly explain the contribution of each of the proposed technique as follows:

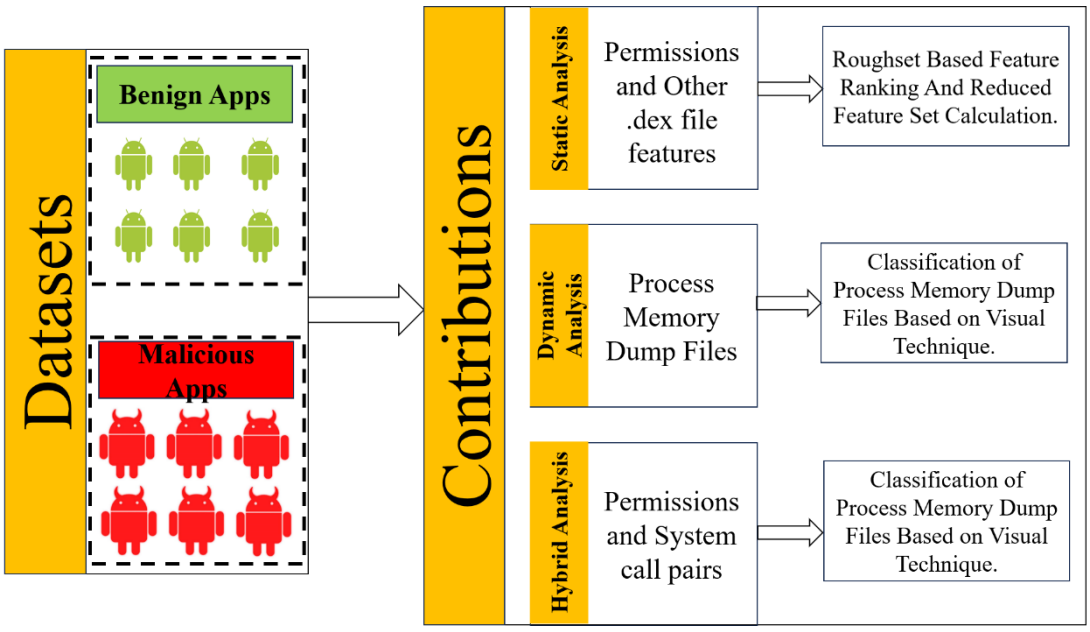

Figure 1.4 Overview of the proposed work of the thesis

1. We have used permissions, API calls, system commands, and opcodes with rough set theory for Android malware detection. To the best of our knowledge, we are the first to apply rough set theory to the static features mentioned above. The rough set theory has several advantages, such as attribute selection and its ability to work with qualitative and quantitative attributes. We used a Discernibility Matrix to rank and further calculate the reduct of the above features. Ranking of features is done to highlight essential features. Reduct, a reduced feature set, is estimated to improve the overall detection rate with the most minor features. We applied several Machine Learning (ML) algorithms such as Support Vector Machines (SVM), K-Nearest Neighbor, Random Forest, and Logistic Regression for malware detection. Our results demonstrate an overall accuracy of 97%, better than many state-of-the-art detection techniques proposed in the literature.
2. We proposed a covalent bond-based Android malware detection model using permissions and system call pair. We use the analogy of covalent bonds between two atoms in chemistry to form covalent bonds between every permission and system call. We also calculate bond strengths between permission and system call pairs to denote the strength of the bond they create between them. The estimated bond strength helps detect an Android application as malicious or benign. Our detection results demonstrate an overall accuracy of 97.5%, better than many state-of-the-art detection techniques proposed in the literature.
3. We developed a visual malware detection technique based on process memory dump files. An Android process memory dump, referred to as a memory dump or core dump, captures the memory snapshot of a running process on an Android device at a specific instance. It encompasses details concerning memory allocation, variables, registers, and other pertinent data structures linked to the process.

1.8 Organization of Thesis

The dissertation explaining research work during the Ph.D. is organized into six chapters. Chapter 1 gives the introduction to the field of Android malware detection

and outlines the motivation behind the research. Chapter 2 is dedicated to the literature review of existing studies and research in the field of Android malware detection. Chapter 3 focuses on using permissions, API calls, system commands, and opcodes with rough set theory for Android malware detection. Chapter 4 is dedicated on building the permission and system call covalent bond pairs to identify and analyze the impact of these pairs for malware detection on Android. Chapter 5 is dedicated to propose a novel Android malware detection mechanism based on visual techniques. The mechanism is based on converting Android process memory dump files into grayscale images. Chapter 6 summarizes the conclusions inferred from this research work and highlights the potential future work in this ar

Chapter Two:
LITERATURE REVIEW

This chapter presents a comprehensive review of techniques for Android malware detection. The Android malware detection techniques can be broadly divided into three types: static analysis, dynamic analysis and hybrid analysis. Static detection is the art of malware detection technique in which the features are extracted from the source code without executing the source code. Dynamic detection is the technique in which the run time behaviour of code is examined while the code is under execution. Hybrid detection is the combination of both as it uses the methodology of both static and dynamic analysis.

2.1 Static detection

This section describes research in the area of static malware detection techniques and centers on three main techniques, namely detection based on manifest files, API calls and Java code. Therefore, the section is divided into three sub sections, including, manifest file-based detection, API calls-based detection and java code-based detection.

2.1.1 Manifest File Based Detection

In this subsection, we covered studies that have been conducted about the features extracted from the manifest files of Android applications in the context of malware detection. According to Grace et al. [33], applications that integrated ad libraries into their main programs posed a threat to Android devices, as the offline work tended to emphasize the insecure aspects of the bundled ad libraries to host app relationships. Others like Enck et al. [34] formalised a simple certification scheme in terms of the security properties of the applications in order to provide an anti-virus against the highly aggressive applications. Considering mobile malware related anxious security concerns [23], Li and colleagues developed SIGPID, a malware detection that reduce over-privileged permission identification of malware applications by applying 3 levels of pruning. Further, Talha et al. [35] created a client

– server application APK-auditor which helps in detection of malicious applications based on maintenance of the Android profile database using permission analysis.

The authors in [36] developed a context category ontology based on permissions to identify the potential risk of information leakage caused by malicious activities. Song et al. [37] created a prototype called ASE, which uses four levels of filtering based on static analysis to classify an application as either benign or malicious. DroidChain [38] is another detection approach that employs static analysis combined with a behaviour chain model to identify four types of malicious behaviours: privacy leakage, SMS financial fraud, malware installation, and privilege escalation. ProDroid [39] is a behaviour-based detection model that leverages biological sequence techniques and a Markov chain model to compare the classes and APIs of decompiled apps with stored malicious behaviour patterns. Moonsamy et al. [40] analyzed both requested and used permissions to extract contrasting permission sets, which were then used to classify applications as either benign or malicious.

Idrees et al. [41] employed intent filters and permissions to classify applications as either benign or malicious. Wang et al. [21] focused on ranking requested permissions by risk, selecting the most risky subset to train machine learning models. The authors in [42] introduced DroidRanger, a tool for detecting malicious applications using permission behaviours and heuristic filtering, which also successfully identified zero-day malware. Qiu et al. [43] uniquely annotated detected malware capabilities, particularly concerning security and privacy issues. In [44], APIs, intents, and permissions were analyzed to establish similarity associations with malware samples, detecting malicious applications using Hamming distance.

Bai et al. [45] tried to develop a fast malware detection system by taking into account numerous features such as permissions and opcode sequences. Drebin [46] is a light-weight method for cell phone malware detection, capable of identifying malicious apps within ten seconds after the download of such an app. Varsha et al. [47] evaluated the feature selection technique of the most valuable features from different sets of static features. Mahindru et al. [48] tested ten different feature selection techniques to choose the most optimal set of features for efficiently detecting malicious applications.

Khariwal et al. [49] proposed a novel method to find the best permissions and intents combined to detect malicious applications. PermPair [50] is another malicious application detection technique that creates permission pairs from each application and further constructed malicious and normal permissions pair graphs used for the detection mechanism. The work in [51] uniquely compared the dynamics between requested permissions and intent filters. In manilyzer [52], stress was given on using different manifest components along with requested permissions. Sanz et al. [53] developed a malware detection model based on used permissions. Li et al. [54] developed a malware detection model using multiple features both from the manifest file as well as from the source file, whereas Sato et al. [55] used multiple features from the manifest file only.

2.1.2 API Calls-Based Detection

Several researchers have utilized static API calls to identify Android malware. The Droidmat model [56] employed a combination of manifest file features and API calls and applied K-means and KNN algorithms for malware detection. Another study [57] examined user-triggered dependencies and sensitive APIs in malicious apps, while Zhang et al. [58] constructed dependency graphs of API calls to categorize malicious apps into Android malware families using similarity metrics. The authors of [59] introduced a model called Apposcopy, which examined control-flow and dataflow properties derived from API calls to detect malware. Wang et al. [60] focused on analyzing string features like permissions and intents, as well as structural features such as API calls and function call graphs, on detecting malicious behaviour in Android apps. Similarly, the work described in [61] involved the analysis of API calls and their call graphs for malware detection.

2.1.3 Java Code-Based Detection

Zhu et al. in [62] developed an image-based malware detection method that extracts important parts of Dalvik code and converts it to RGB images. Fang et al. [63] also used RGB images generated from Dex files but, apart from classifying an application as benign or malicious, did malware familial classification. The work [64] eliminated code confusion and calculated scores for every code word based on their

importance, which deep learning models then used to detect malicious applications. CDGDroid [65] is another technique to detect Android malware based on control flow graphs and data flow graphs that are constructed from the code of the application with the help of program analysis techniques and later on used as features for the CNN model. Xiao et al. [66] developed a method that captures the system call sequence from the code of the application, and the captured system call sequence is used to train LSTM to detect malicious applications. To form the malware detection model, MSNDroid [67] incorporated native-layer code features and combined them with permissions and Java layer components.

2.2 Dynamic Detection

This section describes the techniques available in literature for performing malware detection using dynamic analysis. While static analysis and detection methods are fast, they often struggle against malware that uses encryption, polymorphism, or code transformation. In contrast, dynamic analysis involves running the mobile application within a controlled environment, such as a virtual machine or emulator, allowing researchers to observe its behaviour in real time. This approach was developed to address the shortcomings of static analysis, particularly in detecting malware that downloads harmful code during runtime to avoid static detection. Dynamic analysis operates by executing the app in a secure environment that simulates all necessary resources, enabling the identification of malicious activities. Although several dynamic analysis techniques have been implemented, they are limited by the resource constraints inherent in smartphones. Like static analysis, dynamic analysis employs a variety of features for detection and analysis, such as OS-level features and network traffic data, which we will discuss in detail later.

2.2.1 OS-Based Detection

TaintDroid model, which relies on dynamic taint analysis, evaluated system call sequences and tracked the flow of sensitive information within third-party applications. The researchers found that even many legitimate apps could potentially leak private data stored on mobile devices. Several systems, including those referenced in [68, 69, 70], were developed based on the TaintDroid model to detect privacy leaks

in Android applications. Yang et al. [71] expanded on the TaintDroid framework to identify data leaks and determine whether these leaks were intentional by the user. However, these studies primarily focused on analyzing data leaks rather than detecting malicious applications.

Shabtai et al. [72] investigated dynamic features such as CPU usage, the number of active processes, and Wi-Fi packet transmission to identify malware. The CopperDroid model [73] analyzed system calls from malware samples to determine whether malicious behaviour originated from Java, JNI, or native code execution. Afonso et al. [74] explored a combination of dynamic API calls and system calls for identifying malicious apps.

The CrowDroid model [75] extracted system calls and employed partitioning clustering techniques to differentiate between malicious and benign applications. The DroidTrace model [76], utilizing ptrace, monitored various dynamic features, including system call sequences, file operations, and network connections, for malware detection. Almeida et al. [77] evaluated runtime traces such as system calls, network traffic behaviour, and real-time user inputs like interactions with apps to assess the risk posed by applications. Jang et al. [78] used volatile memory acquisition techniques to detect malicious Android applications.

2.2.2 Network Traffic Based Detection

Few studies in the literature have focused on using network traffic to analyze behaviour and detect Android malware. This section reviews such works. In a study [79], authors utilized an Android emulator to capture the network traffic of both malicious and benign apps. Among 16 network traffic features, 7 were identified as effective in distinguishing between normal and malicious traffic. Chen et al. [80] analyzed network traces of malicious apps after capturing their traffic and found that over 70% of the samples produced malicious traffic within the first five minutes. They noted that features like DNS queries and HTTP requests could be used to detect malware. The authors of [81] grouped malware families based on their HTTP traffic analysis, examining features like the number of GET/POST requests and the amount of data sent in POST messages, using the BIRCH algorithm to cluster similar malware

families. They observed that malware samples from the same families exhibited similar HTTP features and thus were grouped in the same cluster.

Wang et al. [82] clustered Android malware samples by analyzing the similarities in their HTTP traffic flows. Alan et al. [83] demonstrated that popular Android apps could be identified by applying supervised machine learning algorithms to the packet sizes of the first 64 packets they generate, although they did not include malware apps in their analysis. Mauro et al. [84] analyzed encrypted traffic from benign Android apps but did not consider malicious samples in their study, nor did they propose a detection model for malicious Android applications. Wang et al. [85] employed Natural Language Processing techniques on HTTP headers to detect malicious apps. Shabtai et al. [86] applied machine learning algorithms to traffic features to generate normal traffic patterns, which were then used to identify malicious apps. Wang et al. [87] conducted multilevel network traffic analysis, incorporating both HTTP request features and TCP flow-based features, and tested their data using the Decision Tree algorithm. They extended this work in [88] by comparing HTTP-based and TCP-based detection models, developing working prototypes for both to allow users to choose the model best suited to their needs. In another study [85], they extracted text-level features from HTTP flows, used the Chi-Square Test for feature selection, and applied SVM to the selected features for malware detection. They suggested extending their model to analyze and detect encrypted malware traffic but did not implement this design. The authors of [89] analyzed DNS and HTTP traffic from Android smartphones, applying various machine-learning algorithms to detect malware. Other studies, such as [90, 91, 92, 93, 94], have also utilized machine learning algorithms to detect malicious activity on Android networks.

2.3 Hybrid Detection

Only a few studies have integrated static and dynamic detection features into a single model to propose a hybrid detection approach. This section reviews such hybrid models for Android malware detection. Saracino et al. [95] examined various static and dynamic features, including system calls, API calls, user activity logs, and permissions, to identify malware. The authors used statistical inference to correlate these features, thereby detecting app misbehavior. Han et al. [96] extracted 120 static

features, such as APK size, developer information, and API calls, along with 767 dynamic features, including SMS activity, file operations, and cryptographic usage. They applied multiple feature transformations to map the features into a new feature space and used ensemble classifiers to detect malicious samples. Sun et al. [97] developed a hybrid model that generated static and dynamic graphs from manifest file components and system calls, respectively. Xia et al. [98] conducted static API analysis and dynamic bytecode analysis to detect data leaks from apps. The Riskranker model [99] analyzed dynamic features like run-time Dalvik code loading, along with static features such as permissions, to identify malware. The Marvin model [100] utilized machine learning classifiers on hybrid features like app name, class structure, file operations, intent receivers, network behavior, and phone activity. The SAMADroid model [101] extracted system calls and manifest file components and applied various machine learning classifiers for malware detection.

In another study, the authors [102] adapted an open-source framework called CuckooDroid to analyze static manifest file components and dynamic API calls for detecting malicious behavior within apps. Similarly, Patel et al. [2] proposed a hybrid framework by analyzing manifest files and runtime API calls for malware detection. Yuan et al. [103] used deep learning to analyze a combination of static permissions and dynamic behaviors, such as user-app interactions, to identify malicious apps. Liu et al. [104] applied machine learning to a combination of permissions and system calls for malware detection. In a related approach, the authors in [105] developed a machine learning-based hybrid model using permissions and both static and dynamic API calls. Chakraborty et al. [106] applied an ensemble Classification and Clustering approach to manifest file components and dynamic logs, such as SMS logs, generated during app execution. Their model aimed to detect malware and predict the malware family to which a sample belongs.

Chapter Three:
INNOVATIVE APPROACH TO ANDROID MALWARE DETECTION: PRIORITIZING CRITICAL FEATURES USING ROUGH SET THEORY

In this chapter we propose a technique for Android malware detection using rough set theory. In section 3.1, we highlight the motivation behind the work done and briefly explained the overview of the proposed technique. In section 3.2 we explain in detail the methodology of the proposed technique. In section 3.3 the details of results are discussed and presented. The section 3.4 is dedicated to discussions and findings. In section 3.6 limitation of the proposed work is discussed. The section 3.7 summarizes the chapter with future directions.

3.1 Introduction

The smartphone has practically become the personal desktop computer of the modern day and enables us to execute nearly all tasks that one would do on a desktop. It brings into our lives components that affect how we communicate, work, entertain ourselves, and access information. Besides a call and text, smartphones are used to an impressive extent for going on the Internet, social media, email management, photo and video capture, GPS navigation, online shopping, banking, tracking health and fitness, learning, managing personal tasks, and smart home automation. In this context, mobile market use by smartphones out beats desktop usage by 20%.

In the numerous types of smartphones, the most popular Smartphone falls in the category with the Android OS. This trend has been able to be so massive partly because of its open-source nature, as many use the Android OS. The Android leads in the chart, taking 70% market share in the global mobile operating system. This openness has exposed Android to frequent malware attacks in the recent years amid its popularity. In this context, there is a huge requirement to develop efficient Android malware detection mechanisms to fight and eliminate evil applications.

Malware analysis is a technique that understands the functionality and origins of malware. It includes three types [107]: static, dynamic, and hybrid analysis. These approaches can be applied in developing detection models, where applications on

Android may either be malicious or benign. There are three kinds of detection models: static, dynamic, and hybrid. Extracted Features in static detection can be obtained through static analysis that is carried out without installing or running the application. Dynamic detection works on the executing of the application to capture its features at run-time. A hybrid model of detection combines both static and dynamic analysis to extract a more comprehensive set of features from the application.

In static detection methods: There are several common techniques for feature extraction in n static detection methods. The most popular ones among them are manifest file-based detection, API call-based detection, and Java code-based detection. Manifest file-based detection refers to extracting features from the Android application's manifest file. For example, Li et al. [23] were able to achieve 90% accuracy using permissions from the manifest file. This foundation was expanded by the work of Arora et al. [50], who used permission pairs retrieved from the manifest file to obtain 95.44% accuracy. IPDroid [49] combined permissions with intents found in the manifest file and made use of a Random Forest classifier to obtain 94.73% accuracy.

Feature detection through API call entails the detection of APIs invoked by Android applications. Droidmat [56] integrated components of the manifest file and API calls to detect malicious apps with 97.87% precision. Elish et al. [57]constructed a detection model based on sensitive API calls invoked by the users while Zhang et al. [61] developed association rules between API calls with a whopping 96% precision.

The Java code-based detection methods use Dex files that contain Java code in Android applications for extracting features. Zhu et al. [62] converted the vital parts of Dalvik byte code into RGB images and trained the Convolution neural network to devise the malware detection system with an accuracy of 96.9%. Fang et al. [63] also converted Dex files to RGB images to do malware familial classification with a precision of 96%. The work [64] is based on eliminating code confusion and achieves an overall accuracy of 92.67%.

3.1.1 Contributions

In the current work, we have used permissions, API calls, system commands, and opcodes with rough set theory for Android malware detection. To the best of our knowledge, we are the first to apply rough set theory to the static features mentioned above. The rough set theory has several advantages, such as attribute selection and its ability to work with qualitative and quantitative attributes. We used a Discernibility Matrix to rank and further calculate the reduct of the above features. Ranking of features is done to highlight essential features. Reduct, a reduced feature set, is estimated to improve the overall detection rate with the most minor features. We applied several Machine Learning (ML) algorithms such as Support Vector Machines (SVM), K-Nearest Neighbor, Random Forest, and Logistic Regression for malware detection. Our results demonstrate an overall accuracy of 97%, better than many state-of-the-art detection techniques proposed in the literature. The main contributions of this paper are summarized below.

- ✓ Firstly, we performed data pre-processing, in which we eliminated co-related features and features not dependent on the class variable.
- ✓ We calculated the ranking score with the help of the discernibility concept of rough set theory to rank the features according to their importance.
- ✓ We utilized an algorithm for rough set reduct computation to minimize the number of features in each category, employing the ranking score and discernibility principles from rough set theory..
- ✓ We further applied machine learning algorithms to evaluate the detection accuracy with the reduct calculated in the previous step.
- ✓ We compared the results of our proposed model with other state-of-the-art detection techniques, and our results highlight that the proposed model outperforms similar state-of-the-art models.

3.2 Methodology

This section describes the overall approach to classifying Android applications as malware or benign. The process is divided into four phases, as depicted in Figure

3.1. The first phase of the approach is the pre-processing phase, the second phase is feature ranking, the third phase is the Rough Set Reduct Computation Phase, and the fourth phase is the detection phase.

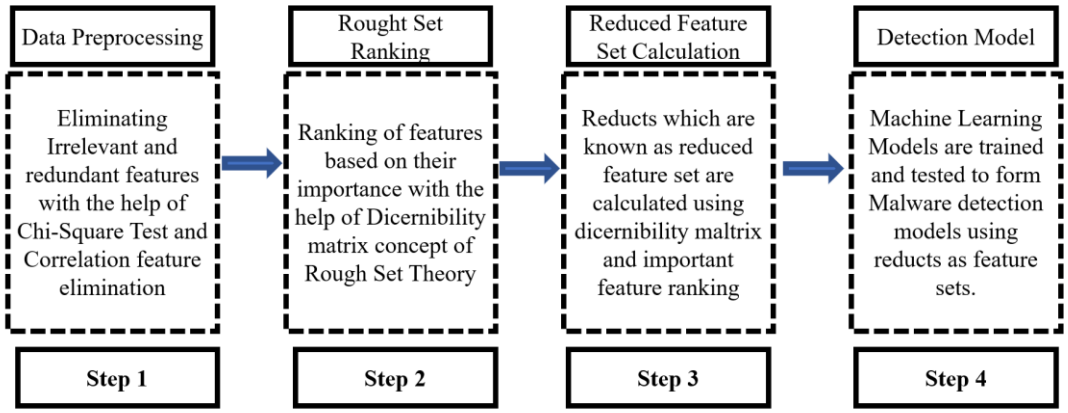

Figure 3.1Proposed Methodology

3.2.1 Data Pre-Processing Phase

These data pre-processing phase is more focused on the primary feature selection phase. The whole process of this phase is depicted in Figure 3.2.

The figure referred to here summarizes the whole process of phase 1. The proposed technique first considers the OmniDroid Dataset [108] and Androzoo Dataset [109]. The OmniDroid dataset is the data set in which various features are extracted from an extensive collection of 22,000 APKs. The dataset consists of an equal number of benign and malicious applications, i.e., 11,000 each. An additional 8000 applications are taken from the Androzoo Data set, spreading from 2015 to 2023, making it more diverse. These 8000 apps consist of an equal number of benign and malicious applications, i.e., 4000 each. The features from these 30,000 apps were extracted with the AndroPytool [110]. The AndroPyTool extracts features from the Android application supplied as input to the tool. Specifically, the AndroPyTool extracts three types of features: pre-static features, static features, and dynamic features. This paper focuses on static features, i.e., permissions, API calls, system commands, and opcodes. The following is the description of each static feature considered in this paper.

1. Permissions: Every Android application requests and requires a particular set of permissions for its functioning. The apps need these permissions to access some data or specific resources. These permissions are listed in the Android Manifest file. The OmniDroid dataset consists of 5501 unique permissions.
2. API Calls: The Application Programming Interface (API) is a set of code snippets that the underlying systems use for communicating. API calls are the calls to such code snippets with some functionality that must be invoked to perform specific tasks. The dataset in consideration consists of 2128 API Calls.
3. System commands: Android applications must access the kernel to perform specific tasks and services. So, the services that need to be accessed by the app are done by calling the OS routines. The calls to such kinds of OS routines are known as system commands. The OmniDroid dataset consists of 103 system commands.
4. Opcodes: The Dalvik Bytecode generated by compiling the Android apps consists of instructions that need to be executed in terms of opcodes. The data set consists of 224 opcodes.

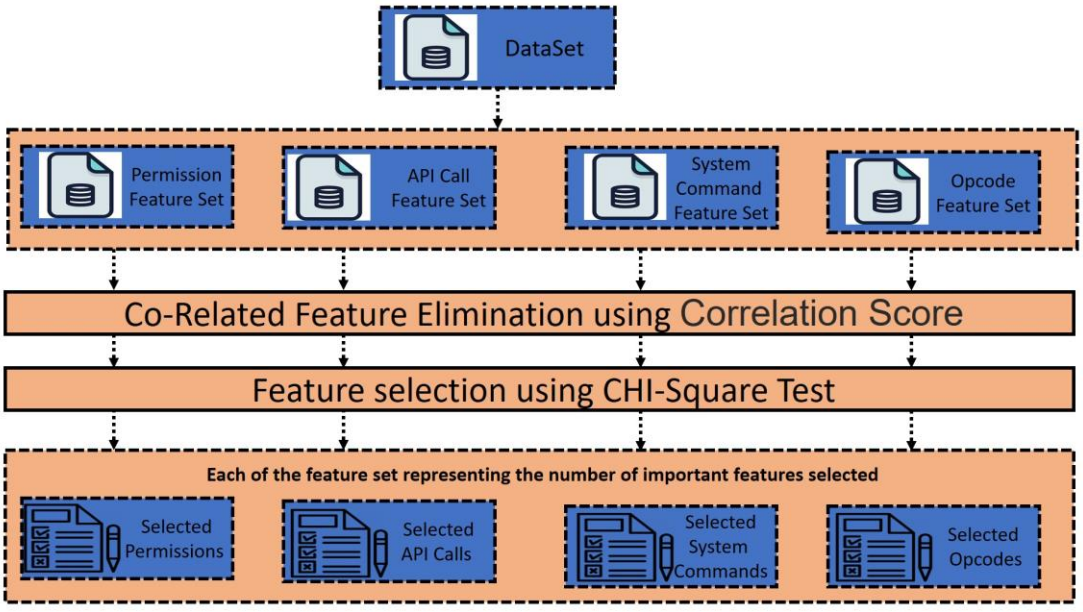

Figure 3.2Data Pre-Processing

First, process is applied individually for each of the features above. For all used features: permissions, API calls, system commands, opcodes, a correlation score is calculated for each set of features, which is used to filter out highly correlated features since attribute features usually show strong correlations and typically have high linear dependence as well as the same effect on the dependent variable. If two features are highly correlated, one can safely be removed. In eliminating one feature, we keep the other, provided one's correlation is 90 percent or higher. From this alone, 4428 out of 5501 permissions, 1589 out of 2128 API calls, 93 out of 103 system calls, and 159 of 224 opcodes remained.

Further, the Chi-Square test is executed to select a subset of features for each feature set. The Chi-square test is a statistical test used to determine whether there is a significant association between two categorical variables. The chi-square test works based on the following equation:

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i} \tag{3.1}$$

The formula for the Chi-square test involves several key terms and calculations, as shown in Equation (3.1), where χ^2 is the Chi-square test statistic, n is the number of categories in the contingency table, O_i is the observed frequency of category i, and E_i is the expected frequency of category i under the null hypothesis. The sum is taken over all categories in the contingency table.

The process begins by assuming the null hypothesis, which it presumes no association exists between the feature variable and the class variable. Testing this is accomplished by using the Chi-square test, as the value in Equation (3.1) is calculated for the computation of the Chi-square test statistic. It then compares the computed value against values in the Chi-square distribution table to determine the corresponding p-value. The null hypothesis is rejected if the p-value is less than 0.05. This signifies that the feature and class variables are correlated. The selected features for the new feature subset in each feature set are those whose pvalue is less than 0.05, signifying dependency on the class variable with a 95% confidence level. Ultimately, this process ends up with a reduced feature set for each category, including permissions, API calls, system commands, and opcodes.

For the permission feature set, we achieved 206 permissions selected as a subset of permission out of 4428 permissions. For the API calls feature set, we achieved 1264 API calls selected as a subset of API calls out of 1589 API calls. For system command feature space, out of 94 system commands obtained in the previous step, we achieved a minimized feature space of 52. Similarly, for the opcodes-based feature set, which consisted of 204 opcodes from the previous step, we achieved 158 opcodes. The entire process is also summarized in the Algorithm 1.

3.2.2 Feature Ranking Phase

In this phase, the ranking of minimal feature sets obtained in the previous step is performed to rank the features of each type according to their importance. Feature ranking is performed through the Discernibility Matrix concept of rough set theory. The whole process flow of this phase is depicted in Figure 3.3.

Rough set theory is a mathematical approach to data analysis and data mining. This mathematical tool is powerful in dealing with improper, imprecise, inconsistent, incomplete information, and knowledge [111, 112]. The rough set theory has several advantages, such as attribute selection and its ability to work with qualitative and quantitative attributes. The critical concepts of the rough set theory used in this paper are explained below.

Algorithm 1: Data Pre-Processing

1. **Input:** A feature set of 22,000 APKs regarding four types of features, i.e., Permissions (f_p), API Calls(f_a) , System commands(f_s) , and Opcodes(f_o).
2. **Output:** For each of the feature sets $f_p, f_a, f_s, and f_o$, a minimal feature space of important features is obtained as $min\_f_p, min\_f_a, min\_f_s, and\ min\_f_o$ respectively.
3. **For each** feature set f_i in feature spaces $f_p, f_a, f_s, and f_o$ **do:**
4. Set $|f_i| = N$
5. **For each** feature x_i in feature set f_i **do:**
6. Set $Truth\_val(x_i) = True$
7. **ENDFor**
8. **For each** feature x_i in feature set $\boldsymbol{f_i}$ **do:**

9. **if** $Truth\_val(x_i) == True$ **then:**
10. **For each** feature x_i in feature set $\boldsymbol{f_i}$ where j: $i + 1 \rightarrow N$ **do:**
11. **if** $\left(correl[x_i, x_j] > 0.9\right)$ **then:**
12. do $Select(x_i)$
13. do $Reject(x_j)$ and set $Truth\_val(x_j) = False$
14. **ENDIf**
15. **ENDFor**
16. **ENDIf**
17. **ENDFor**
18. **ENDFor**
19. **For each** of the feature sets f_i in the new feature $f_p, f_a, f_s, and f_o$ obtained **do:**
20. **For each** feature x_i in feature set $\boldsymbol{f_i}$ **do:**
21. Apply Chi-Square Test (x_i)
22. **if** $p\_value[x_i] < 0.05$
23. do $\boldsymbol{Select(x_i)}$
24. **Else**
25. do $\boldsymbol{Reject(x_i)}$
26. **ENDIf**
27. **ENDFor**
28. **ENDFor**

3.2.2.1 Information System

It is defined as an ordered pair, in which the first element of this ordered pair is called the universe. In our case, the universe is the set of both malicious and benign applications that are considered. We represent the ordered pair as $D = (A, F \cup \{l\})$, where D is the data set under consideration, and A is the non-empty finite set called the universe of Android application, consisting of both the malicious and benign types. F is the non-empty set of features in the data set. In our case, these features are in terms of permissions, API calls, system commands, and opcodes. Here, l is the special

attribute known as the label attribute, which stores the type of label corresponding to each application in set *A*. This label attribute stores whether a particular application in set *A* is malicious or benign.

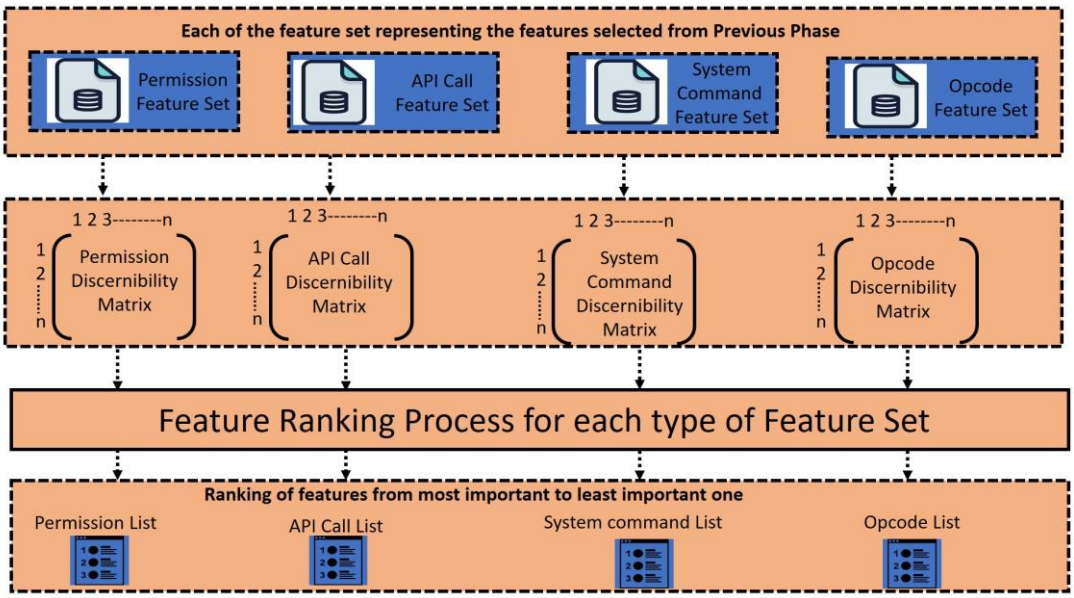

Figure 3.3 Feature Ranking Phase

Table 3.1 Instance of permission information system shows an instance of the permission information system for five applications assumed as A_1, A_2, A_3, A_4, and A_5. Here, feature attributes are the Content Provider Access, Settings App widget Provider, and JPUSH Message. These are permissions, with corresponding values such as 0 or 1 for each application A_i. The value 0 signifies that particular permission is not present in application A_i, whereas the value 1 signifies otherwise. The label column has the value BW, signifying that the particular A_i is Benignware, whereas the label value MW signifies that application A_i is malware, i.e., a malicious one.

Similarly, the information systems for API calls, system commands, and opcodes are shown in Table 3.2,Table 3.3, Table 3.4 respectively. From these information systems, a Discernibility Matrix is formed. The concept of Discernibility is explained in the following section.

Table 3.1 Instance of permission information system

Application	P_1 = CONTENT PROVIDER ACCESS	P_2 = Settings App Widget Provider	P_3 = JPUSH MESSAGE	Label
A_1	0	1	0	BW
A_2	1	1	0	MW
A_3	0	1	1	BW
A_4	0	0	1	BW
A_5	1	1	1	MW

Table 3.2Instance of API calls information system

Application	*AP1 = APICALLandroid.view.SubMenu*	*AP2 = APICALLandroid.net.RouteInfo*	*AP3 = APICALLandroid.app.Activity*	Label
A_1	0	0	0	BW
A_2	1	1	0	MW
A_3	1	1	1	MW
A_4	1	0	1	BW
A_5	1	1	1	MW

Table 3.3 Instance of System Command information system.

Application	*S1 = SYSTEMCOMMAND-svc*	*S2 = SYSTEMCOMMANDstagefright*	*S3 = SYSTEMCOMMANDnandread*	Label
A_1	0	1	0	MW
A_2	1	1	0	MW
A_3	0	1	1	BW
A_4	0	0	1	BW
A_5	1	1	1	MW

Table 3.4 Instance of Opcode information system.

Application	***O1* = *OPCODE-remfloat/ 2addr***	***O2* = *OPCODE-div-int/lit8***	***O3* = *JOPCODEsparse-switch***	**Label**
A_1	1	1	0	BW
A_2	0	1	0	MW
A_3	1	0	1	MW
A_4	0	0	1	BW
A_5	0	1	1	BW

3.2.2.2 **Discernibility Matrix**

This matrix is created from the information system. The Discernibility Matrix is a symmetric $|A|X|A|$ matrix corresponding to each information system. Each entry C_{ij} is defined as $\{f \in F | f(A_i) \nvDash f(A_j)\}$ *if* $l(A_i) \nvDash l(A_j)$, Φ otherwise. Table 3.5–Table 3.8 show the instances of Discernibility Matrices corresponding to information system shown in Table 3.1-Table 3.4, respectively

Table 3.5 Instance of permission discernibility

	A_1	A_2	A_3	A_4	A_5
A_1					
A_2	P_1				
A_3		P_1, P_3			
A_4		P_1, P_2, P_3			
A_5	P_1, P_3		P_1	P_1, P_2	

Table 3.6 Instance of API calls discernibility.

	A_1	A_2	A_3	A_4	A_5
A_1					
A_2	AP_1, AP_2				
A_3	AP_1, AP_2, AP_3				
A_4		AP_2, AP_3	AP_2		
A_5	AP_1, AP_2, AP_3			AP_2	

Table 3.7 Instance of system call discernibility

	A_1	A_2	A_3	A_4	A_5
A_1					
A_2					
A_3	S_3	S_1, S_3			
A_4	S_2, S_3	S_1, S_2, S_3			
A_5			S_1	S_1, S_2	

Table 3.8 Instance of opcode discernibility

	A_1	A_2	A_3	A_4	A_5
A_1					
A_2	O_1				
A_3	O_2, O_3				
A_4		O_2, O_3	O_1		
A_5		O_3	O_1, O_2		

For each of the selected minimal permission set, API call feature set, system command feature set, and opcode feature set, we create the Permission Discernibility Matrix, API Call Discernibility Matrix, System Call Discernibility Matrix, and Opcode Discernibility Matrix, respectively. Algorithm 2 depicts the whole process. The algorithm creates a Discernibility Matrix for each minimal feature set obtained in the previous step and further calls the rough set ranking algorithm described in the next section.

3.2.2.3 Rough Set-Based Feature Ranking

After creating each of these Discernibility Matrices, a rough set-based feature ranking methodology, summarized in Algorithm 3, is applied on each matrix to rank each of the Permission, API Call, System Command, and Opcode features separately. The algorithm takes the Discernibility Matrix as an input and initializes the weight of each feature in the corresponding minimal feature set to zero. Then, the Discernibility Matrix is traversed, and each entry in the Discernibility Matrix, which consists of one or more features, receives the updated weight of the features as per Equation (2).

$$w(x_k) = w(x_k) + |min\_f_i|/|C_{ij}| \qquad (2)$$

In the above equation, C_{ij} is the entry in the Discernibility Matrix corresponding to applications A_i and A_j, and the entry C_{ij} may contain one or more features. Hence, $|C_{ij}|$ represents the count of features in the entry. min_f_i is the minimal feature set corresponding to the Discernibility Matrix, and $|min\_f_i|$ is the count of features in the minimal feature set. $w(x_k)$ is the weight of k^{th} feature in the entry C_{ij}, which may contain x_1, x_2, x_3, . . . and x_n as the features in the entry.

Algorithm 2 Feature Ranking

1. **Input:** Minimal feature space, i.e., min_f_p, min_f_a, min_f_s, and min_f_o obtained as output of Algorithm 1.
2. **Output:** For each of the minimal feature space, ranked minimal feature list L_p, L_a, L_s, and L_o, respectively, sorted in decreasing order as per the importance of the features.
3. **for each** of the minimized feature space min_f_i in min_f_p, min_f_a, min_f_s, and min_f_o **do**

4. create Discernibilty Matrix for min_f_i
5. **end for**
6. Let D_p, D_a, D_s, and D_o be Discernibility Matrix corresponding to min_f_i in min_f_p, min_f_a, min_f_s, and min_f_o, respectively.
7. **for each** of Discernibility Matrix D_i in D_p, D_a, D_s, and D_o **do**
8. call Algorithm 3 for each D_i in order to perform rough set ranking of each of the features in D_i
9. Let L_p, L_a, L_s, and L_o be be the sorted list of important features for each of min_f_i in min_f_p, min_f_a, min_f_s, and min_f_o, respectively
10. **end for**

This ranking is obtained by arranging each of these features in descending order in terms of their importance. The Rough Set-based feature ranking embodies the following idea [49].

1. The more times an attribute appears in the discernibility, the more important is the attributes.
2. The shorter the entry is, the more important the attribute is in the entry.

3.2.3 Rough Set Reduct Computation Phase

This phase focuses on reducing the feature space so that, with as few features as possible, i.e., a reduced feature space, the classification algorithms could be applied to detect an application as benign or malicious. The reduced feature space obtained using the underlying principles of rough set theory is called reduct in rough set theory. The Discernibility Matrix and rough set feature ranking obtained in the previous phase are used to attain reducts for each of the permission, API call, system command, and opcode feature spaces. Hence, after this phase, for each feature space, i.e., permission, API call, system command, and opcode, we get a reduced feature space, which we call a reduct in rough set theory. Figure 3.4 depicts the current phase under discussion. Algorithm 4 describes the whole process in pseudo-code form. The algorithm first calculates the net weight $w(net_{ij})$ of each of the entry C_{ij} in the Discernibility Matrix D_i, containing features x_1, x_2, x_3, . . . and x_n by summing their individual weights $w(x_1)$,

$w(x_2)$, $w(x_3)$, ... and $w(x_n)$, respectively. Then, all the entries C_{ij} in the Discernibility Matrix D_i are copied in the list LD_i, and then LD_i is sorted as per the net weight calculated in the previous step. Initially, Red_i is assumed to be an empty set. For each entry C_z of the sorted list LD_i containing features x_1, x_2, x_3, ... and x_n, we check weather the Red_i contains any common feature in C_z. If no common feature exists, we select the feature x_i with maximal $w(x_i)$ in C_z; otherwise, we skip the entry C_z. The set Red_i is the reduct computed for min_f_i.

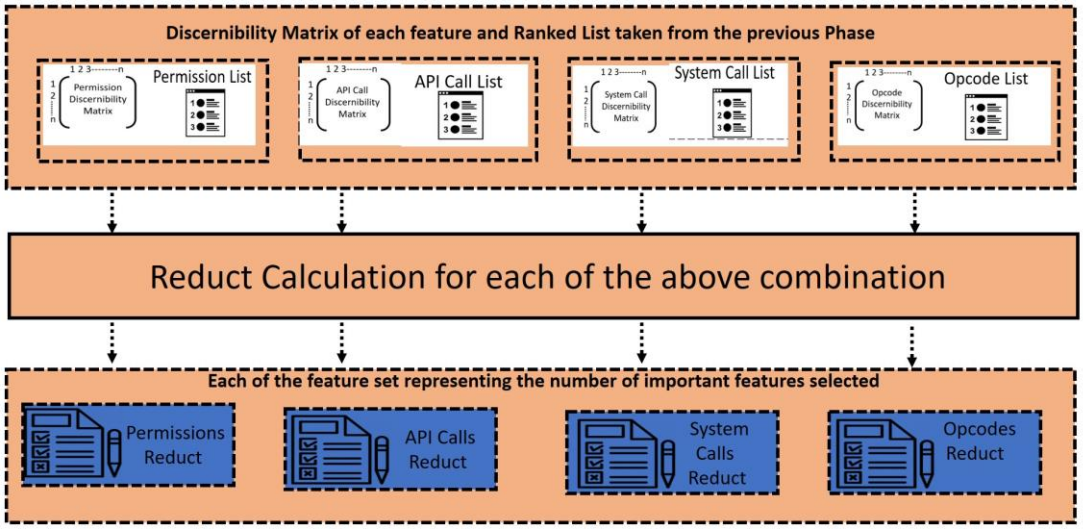

Figure 3.4 Rough Set Reduct Computation Phase

Algorithm 4 Rough Set Reduct Computation

1. **Input:** Discernibility Matrix D_i with dimensions nXn and weight $w(x_i)$ of every feature x_i in the minimal feature space min_f_i in min_f_p, min_f_a, min_f_s, and min_f_o corresponding to D_i in D_p, D_a, D_s and D_o .
2. **Output:** Red_i as the reduct of minimal feature space min_f_i corresponding to D_i.
3. Let LD_p, LD_a, LD_s, and LD_o be the empty list corresponding to permission, API call, system command and opcode feature space.
4. **for each** D_i in D_p, D_a, D_s, and D_o **do**
5. Let $Red_i = \Phi$ denote the empty reduct set corresponding to minimal feature space min_f_i.
6. **for each** $i : 1 \rightarrow N$ **do**

7. **for each** $j : 1 \rightarrow i$ **do**
8. Let C_{ij} be the entry in Discernibility Matrix D_i containing features x_1, x_2, x_3, . . . and x_n
9. Let $w(net_{ij})$ be the cumulative weight of entry C_{ij} having features as x_1, x_2, x_3, . . . and x_n .
10. $w(net_{ij}) = w(x_1) + w(x_2) + w(x_3) \ldots\ldots\ldots + w(x_n)$
11. $LD_i = append(C_{ij})$
12. **end for**
13. **end for**
14. $Sort(LD_i)$ based on $w(net)$ calculated previously.
15. **for each** $z : 1 \rightarrow |LD_i|$ **do**
16. Let C_z be the entry in List LD_i containing features x_1, x_2, x_3, . . . and x_n .
17. **if** $C_z \cap Red_i = \Phi$ **then**
18. Select attribute x_i with maximal $w(x_i)$ in C_z
19. $Red_i = Red_i \cup x_i$
20. **end if**
21. **end for**
22. **end for**

3.2.4 Detection Phase

For building our Android Malware detection system, we experimented with four machine learning algorithms, i.e., the Support Vector Machine (SVM), Random Forest, Logistic Regression, and K-nearest neighbour algorithms, to train and test the dataset. We also performed training and testing on two deep learning models, i.e., Artificial neural network (ANN) and Convolution Neural Network (CNN). Figure 3.5 portrays the overall purpose of the current phase, i.e., with the help of the machine learning models mentioned above and the different reducts, i.e., permission reduct, API call reduct, system call reduct, and opcode reduct calculated in the previous phase; the machine learning models are trained to build the system capable of detecting an Android application as benign or malware.

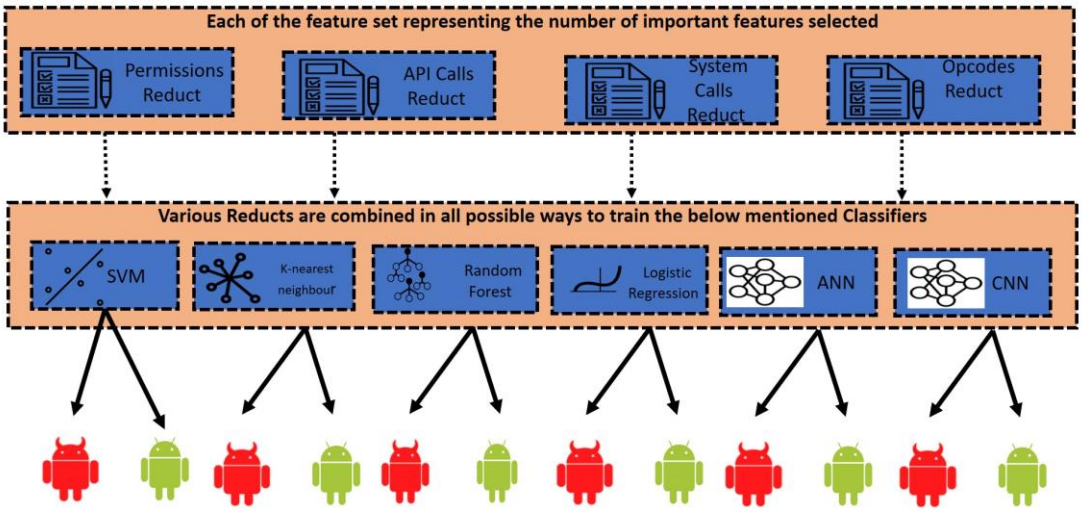

Figure 3.5 Detection Phase

3.3 Results and Discussions

In the current section, we present the results of the evaluation carried out on the proposed malware detection approach.

3.3.1 Results of Ranking Phase

Table 3.9 shows the top 10 important permissions for malware detection. Similarly, the Table 3.10– Table 3.12 represent the top 10 important opcodes, API calls, and system calls, respectively.

Table 3.9 Top ten important permissions.

Rank	Permission Name	Score
1	*READ_PHONE_STATE*	1.56E9
2	*ACCESS_WIFI_STATE*	1.33E9
3	*WRITE_EXTERNAL_STORAGE*	1.26E9
4	*WAKE_LOCK*	1.2E9
5	*ACCESS_COARSE_LOCATION*	1.03E9
6	*ACCESS_NETWORK_STATE*	1.02E9
7	*ACCESS_FINE_LOCATION*	1.01E9
8	*GET_TASKS*	9.52E8
9	*RECEIVE_BOOT_COMPLETED*	8.82E8
10	*GET_ACCOUNTS*	8.4E8

Table 3.10 Top ten important opcodes.

Rank	Opcode	Score
1	*OPCODE − xor − int*	3.09E8
2	*OPCODE − rem − float*	3.07E8
3	*OPCODE − rem − float/2addr*	2.98E8
4	*OPCODE − float − to − long*	2.82E8
5	*OPCODE − and − long*	2.78E8
6	*OPCODE − aget − short*	2.7E8
7	*OPCODE − iget − byte*	2.67E8
8	*OPCODE − aput − short*	2.66E8
9	*OPCODE − iget − short*	2.65E8
10	*OPCODE − rem − double/2addr*	2.65E8

Table 3.11 Top ten important API calls

Rank	API Call	Score
1	*APICALL − android.app.ActionBar*	2.2E8
2	*APICALL − android.widget.PopupWindow*	2.2E8
3	*APICALL − android.widget.BaseAdapter*	2.17E8
4	*APICALL − android.view.ScaleGestureDetector*	2.15E8
5	*APICALL − android.widget.CheckBox*	2.12E8
6	*APICALL − android.widget.AbsListView*	2.1E8
7	*APICALL − android.widget.ListPopupWindow*	2.1E8
8	*APICALL − android.content.res.XmlResourceParser*	2.1E8
9	*APICALL − android.graphics.Path*	2.09E8
10	*APICALL − android.webkit.MimeTypeMap*	2.09E8

Table 3.12 Top ten important system commands

Rank	System Command	Score
1	*SYSTEMCOMMAND − top*	4.27E8
2	*SYSTEMCOMMAND − id*	3.75E8
3	*SYSTEMCOMMAND − start*	3.7E8
4	*SYSTEMCOMMAND − service*	3.58E8
5	*SYSTEMCOMMAND − gzip*	3.54E8
6	*SYSTEMCOMMAND − date*	3.44E8
7	*SYSTEMCOMMAND − log*	3.17E8
8	*SYSTEMCOMMAND − stop*	3.1E8
9	*SYSTEMCOMMAND − mv*	3.06E8
10	*SYSTEMCOMMAND − input*	3.02E8

3.3.2 Detection Results with Individual Features

Table 3.13 displays four different permission sets: full set of permissions from dataset; reduced permissions based on correlation feature elimination, which can be termed permission correlation; the permission set acquired after applying the Chi-square test (permission chi); and lastly, the final reduced permission set acquired after the application of rough set reduct to the permission chi set (permission reduct). All these permission sets have been utilized for training six classifiers, which comprise Support Vector Machines, K-Nearest Neighbors, Random Forest, Logistic Regression, ANN, and CNN. The results show how the performance in terms of accuracy becomes consistently better with the progression of features from the feature set of all permissions to permission correlation, then permission chi, and finally permission reduct. This means that reduct features are the most efficient for malware system development.

Likewise, Tables Table 3.14–Table 3.16 summarize the results for the three types of features, i.e., opcode feature, API call feature, and system command feature, respectively. The same phenomenon is observed in these three tables as was observed in Table 3.13, i.e., for each type of classifier, as the feature set is changed from all feature to feature correlation then to feature chi and finally to feature reduct, the accuracy increases, and training and testing time gets reduced drastically.

Table 3.13 Detection results based on permission.

Classifier	**Feature Set Used**	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**
SVM	All Permissions	78	78	79	78
	Permissions Correlation	79	78	79	78
	Permissions Chi	79	79	79	79
	Permissions Reduct	80	79	79	80
K-Nearest Neighbor	All Permissions	77	77	79	78
	Permission Correlation	78	78	79	78
	Permissions Chi	80	81	78	80
	Permissions Reduct	82	81	82	79
Random Forest	All Permissions	82	84	80	82
	Permissions Correlation	82	84	80	82
	Permissions Chi	83	82	80	81
	Permissions Reduct	83	84	80	81
ANN	All Permissions	76	76	76	77
	Permission Correlation	78	78	79	78
	Permissions Chi	79	79	78	79
	Permissions Reduct	80	80	80	79

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
CNN	All Permissions	77	77	79	78
	Permission Correlation	78	78	79	78
	Permissions Chi	79	80	79	79
	Permissions Reduct	81	80	81	79
Logistic Regression	All Permissions	79	79	77	78
	Permissions Correlation	79	79	77	78
	Permissions Chi	79	80	78	79
	Permissions Reduct	80	80	77	79

Table 3.14 Detection results based on opcode

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	All Opcodes	80	88	82	82
	Opcodes Correlation	80	75	87	82
	Opcodes Chi	81	75	87	81
	Opcodes Reduct	81	79	89	81
K-Nearest Neighbor	All Opcodes	84.50	86	83	84
	Opcodes Correlation	85	86	83	84
	Opcodes Chi	85	85	83	84
	Opcodes Reduct	85	85	83	84
Random Forest	All Opcodes	86	88	85	87
	Opcodes Correlation	86.80	88	86	87
	Opcode Chi	87	88	86	87
	Opcode Reduct	87	88	86	87
ANN	All Opcodes	78	77	86	81

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
ANN	Opcodes Correlation	79	77	85	82
	Opcodes Chi	80	73	82	80
	Opcodes Reduct	79	73	83	78
CNN	All Opcodes	79	75	83	82
	Opcodes Correlation	80	74	85	80
	Opcodes Chi	80	74	84	78
	Opcodes Reduct	79	74	84	78
Logistic Regression	All Opcodes	79	76	86	81
	OpcodesCorrelation	80	76	86	81
	Opcodes Chi	80	75	84	79
	Opcodes Reduct	80	74	85	79

Table 3.15 Detection results based on API calls

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	All API calls	84	83	88	85
	API calls Correlation	85	83	88	85
	API calls Chi	85	83	88	86
	API calls Reduct	86	85	89	83
K-Nearest Neighbor	All API calls	85	86	85	85
	API Calls Correlation	85.70	87	85	86
	API Calls Chi	86	87	85	86
	API Calls Reduct	86	87	84	86

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
Random Forest	All API Calls	88	90	88	90
	API Calls Correlation	89	88	90	88
	API Calls Chi	89	90	88	89
	API Calls Reduct	90	90	88	89
ANN	All API Calls	83	82	86	83
	API Calls Correlation	83	82	86	83
	API Calls Chi	83	82	86	83
	API Calls Reduct	84	81	86	83
CNN	All API Calls	84	82	87	84
	API Calls Correlation	84	82	87	84
	API Calls Chi	84	82	87	84
	API Calls Reduct	85	82	87	84
Logistic Regression	All API Calls	85	83	88	85
	API Calls Correlation	85	83	88	85
	API Calls Chi	85	83	88	85
	API Calls Reduct	86	83	88	85

Table 3.16 Detection results based on system commands.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	All Sys cmd	62	59	86	70
	Sys cmd Correlation	62.50	59	86	70
	Sys cmd Chi	62.80	59	86	70
	Sys cmd Reduct	63	59	86	70

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
K-Nearest Neighbor	All Sys cmd	79	77	82	80
	Sys cmd Correlation	79	77	82	80
	Sys cmd Chi	79	77	83	80
	Sys cmd Reduct	79	77	83	80
Random Forest	All Sys cmd	82	81	85	83
	Sys cmd Correlation	82	81	86	83
	Sys cmd Chi	83	82	86	83
	Sys cmd Reduct	83	82	86	83
ANN	All Sys cmd	63	59	80	68
	Sys cmd Correlation	63	60	80	68
	Sys cmd Chi	64	60	81	70
	Sys cmd Reduct	64	60	81	71
CNN	All Sys cmd	64	60	81	70
	Sys Cmd Correlation	65	60	81	70
	Sys cmd Chi	65	62	82	70
	Sys cmd Reduct	65	61	82	71
Logistic Regression	All Sys cmd	65	61	82	70
	Sys cmd Correlation	65.13	61	82	70
	Sys cmd Chi	65.98	62	83	71
	Sys cmd Reduct	66	62	83	71

3.3.3 Detection Results with Combinations of Two Features

Table 3.17 shows that Random Forest emerges as the best algorithm in terms of accuracy, precision, recall, and F1-score for permissions and opcodes reduct as the feature set.

Table 3.17 Detection results based on permissions and opcodes.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	Permission + Opcode Reduct	85	86	83	84
K-Nearest Neighbor	Permission + Opcode Reduct	87	88	86	86
Random Forest	Permission + Opcode Reduct	90	92	88	90
ANN	Permission + Opcode Reduct	82	83	82	82
CNN	Permission + Opcode Reduct	83	84	82	83
Logistic Regression	Permission + Opcode Reduct	84	85	83	84

Table 3.18 shows results of permission and API calls reduct as feature set used by SVM, K-nearest neighbour, Random Forest, and Logistic regression. Random Forest emerges as the best, with an accuracy of 92%.

Table 3.19 shows that Random Forest with an accuracy of 88% is proved to be the best detection model among all the other three detection models using permissions and system command reduct as the feature set.

Table 3.20 shows that the detection model with classifier as Random Forest and feature set as a combination of opcodes reduct and API calls reduct outperforms the other three detection models with an accuracy of 93%.

Table 3.21 shows the detection results of the models that used opcode reduct and system command reduct as the feature set. The Random Forest classifier performed best with an accuracy of 90%.

Table 3.18 Detection results based on permissions and API calls.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	Permission + API calls Reduct	87	87	85	87

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
K-Nearest Neighbor	Permission + API calls Reduct	88	88	87	88
Random Forest	Permission + API calls Reduct	92	92	90	91
ANN	Permission + API calls Reduct	85	84	84	84
CNN	Permission + API calls Reduct	86	85	85	85
Logistic Regression	Permission + API calls Reduct	87	86	86	86

Table 3.19 Detection results based on permissions and system commands.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1- Score (%)
SVM	Permission + Sys Cmd Reduct	83	84	81	83
K-Nearest Neighbor	Permission + Sys Cmd Reduct	84	84	84	84
Random Forest	Permission + Sys Cmd Reduct	88	91	85	88
ANN	Permissions+ Sys Cmd Reduct	80	82	79	81
CNN	Permissions + Sys Cmd Reduct	81	83	80	82
Logistic Regression	Permission + Sys Cmd Reduct	82	84	81	83

Table 3.20 Detection results based on opcodes and API calls.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall(%)	F1-Score (%)
SVM	Opcode + API call Reduct	88	88	86	88
K-Nearest Neighbor	Opcode + API call Reduct	90	89	88	89
Random Forest	Opcode + API call Reduct	93	93	92	93
ANN	Opcode + API call Reduct	86	85	85	86
CNN	Opcode + API call Reduct	87	86	86	87
Logistic Regression	Opcode + API call Reduct	88	87	87	88

Table 3.21 Detection results based on opcodes and system commands

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	Opcode + Sys Cmd Reduct	84	85	82	84
K-Nearest Neighbor	Opcode + Sys Cmd Reduct	85	87	85	86
Random Forest	Opcode + Sys Cmd Reduct	90	90	88	89
ANN	Opcode + Sys Cmd Reduct	81	83	84	83
CNN	Opcode + Sys Cmd Reduct	82	84	85	84
Logistic Regression	Opcode + Sys Cmd Reduct	83	85	86	85

Table 3.22 shows that the detection model formed with the help of Random Forest as the classifier and API calls reduct and system command reduct as the feature set attains an accuracy of 91%, which is best among all the models in the table.

Table 3.22 Detection results based on API calls and system commands

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	API Call + Sys Cmd Reduct	85	86	84	86
K-Nearest Neighbor	API Call + Sys Cmd Reduct	86	88	86	87
Random Forest	API Call + Sys Cmd Reduct	91	91	88	90
ANN	API Call + Sys Cmd Reduct	82	84	86	83
CNN	API Call + Sys Cmd Reduct	83	85	87	84
Logistic Regression	API Call + Sys Cmd Reduct	84	86	88	85

3.3.4 Detection Results with Combinations of Three Features

Table 3.23 shows that when permission reduct, opcode reduct, and system command reduct are combined to form a single feature set, that feature set, when used with Random Forest, gives the highest accuracy of 95%.

Table 3.24 shows that when permission reduct, opcode reduct, and system command reduct are combined to form a single feature set, that feature set, when used with Random Forest, gives the highest accuracy of 93%.

Table 3.25 shows that when permission reduct, opcode reduct, and system command reduct are combined to form a single feature set, that feature set, when used with Random Forest, gives the highest accuracy of 93%.

Table 3.23 Detection results based on permissions, API calls, and opcodes.

Classifier	**Feature Set Used**	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**
SVM	permissions + API Call + opcode + Reduct	90	90	88	89
K-Nearest Neighbor	permissions + API Call + opcode + Reduct	92	91	90	91
Random Forest	permissions + API Call + opcode + Reduct	95	94	93	95
ANN	permissions + API Call + opcode + Reduct	88	89	89	88
CNN	permissions + API Call + opcode + Reduct	90	91	90	91
Logistic Regression	permissions + API Call + opcode + Reduct	90	89	89	90

Table 3.24 Detection results based on permissions, API calls, and system commands

Classifier	**Feature Set Used**	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**
SVM	permissions + API Call + sys Cmd + Reduct	86	87	85	87
K-Nearest Neighbor	permissions + API Call + sys Cmd + Reduct	87	89	87	88
Random Forest	permissions + API Call + sys Cmd + Reduct	93	92	90	92

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
ANN	permissions + API Call + sys Cmd + Reduct	84	85	84	84
CNN	permissions + API Call + sys Cmd + Reduct	85	87	87	85
Logistic Regression	permissions + API Call + sys Cmd + Reduct	85	87	88	86

Table 3.25 Detection results based on permissions, opcodes, and system commands.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	permissions + Opcode + sys Cmd + Reduct	86	87	83	85
K-Nearest Neighbor	permissions + Opcode + sys Cmd + Reduct	87	87	86	87
Random Forest	permissions + Opcode + sys Cmd + Reduct	93	92	89	91
ANN	permissions + Opcode + sys Cmd + Reduct	84	86	86	86
CNN	permissions + Opcode + sys Cmd + Reduct	84	86	86	86
Logistic Regression	permissions + Opcode + sys Cmd + Reduct	84	86	86	86

Table 3.26 shows combining the opcodes, API calls, and system command reducts and applying all four classifiers the detection model with the Random Forest as the classifier is best among all, with an accuracy of 94%.

Table 3.26 Detection results based on opcodes, API calls, and system commands

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	Opcode + API calls + sys Cmd + Reduct	90	91	88	90
K-Nearest Neighbor	Opcode + API calls + sys Cmd + Reduct	92	90	90	91
Random Forest	Opcode + API calls + sys Cmd + Reduct	94	94	93	94
ANN	Opcode + API calls + sys Cmd + Reduct	87	87	86	87
CNN	Opcode + API calls + sys Cmd + Reduct	88	88	87	88
Logistic Regression	Opcode + API calls + sys Cmd + Reduct	89	88	88	89

3.3.5 Detection Results with Combinations of all Four Features

Table 3.27 shows that all the feature set reducts, i.e., permissions, opcodes, API calls, and system commands reducts, used together with Random Forest emerge as the best classifier, with an accuracy of 97%.

Table 3.27 Detection results based on permissions, opcodes, API calls, and system commands.

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
SVM	Permissions + Opcode + API Call + Sys Cmd Reduct	92	91	90	91
K-Nearest Neighbor	Permissions + Opcode + API Call + Sys Cmd Reduct	93	93	92	92
Random Forest	Permissions + Opcode + API Call + Sys Cmd Reduct	97	95	95	95

Classifier	Feature Set Used	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
ANN	Permissions + Opcode + API Call + Sys Cmd Reduct	89	88	87	89
CNN	Permissions + Opcode + API Call + Sys Cmd Reduct	90	89	88	89

3.4 Discussion and Findings

We describe in this subsection the rationale behind the detections generated using our proposed model. The highest accuracy attained by the model is 97%, which occurs when all four categories of feature type are used. What we observed was that the four types of features actually improve detection considerably if combined together. This is because each of these sets of features captures a different aspect of the malicious application behavior. Since the model will be incorporating various feature types, there will always be more significant and better understanding of the traits that characterize this application. This means that combinations of several feature types will increase the entire correctness in the detection process.

The second thing that was discovered is that API calls are more accurate than the rest of the single features. The malware developers might obfuscate their code in an attempt to avoid detection, but it is a lot harder for them to hide the usage of certain API calls. Detection of those calls can disclose some concealed malware. Therefore, it seems that API call-based analysis is more accurate than the others.

Thirdly, the Random Forest performed best compared to other classifiers. This is because Random Forest makes predictions using the ensemble of decision trees. One of the ways it reduces variance is through the minimization of the risk of overfitting that may occur due to an individual decision tree. The aggregation of the predictions that result from several trees gives a consistent improvement in the overall performance of the model.

Last but not least, we have six different possible combinations of two feature sets consisting of permission, opcodes, API calls, or system commands. We tried all these combinations and found the pair of opcodes and API calls to work best for us,

resulting in the best accuracy. Perhaps it might be because of the reason that an opcode is some low-level instruction which a processor executes. The pattern of such instructions within the code of an application may represent some malicious behavior. Meanwhile, analyzing the API calls of an app will tell what that app does, as some API calls are indicators of malicious activity. The integration of opcodes and API calls allows for a better view into the behavior of an application-in other words, its code level, which could prove useful for identifying sophisticated malware capable of using obfuscation techniques. In contrast, permissions give you a more general view based on declared capability that may not be specific to the same level.

We combined all four features-permissions, opcodes, API calls, and system commands-together in sets of three, giving us four combinations. We tested out the combinations and found that the combination of permissions, opcodes, and API calls got us the best result. This might have been because of permissions providing important information about an app's declared capabilities. Putting together an opcode analysis with API call details really enhances our ability to understand not only how the app's intended usage but also how it may be misused. Specific combinations of permissions can signal potential issues even before analyzing what code is actually being executed in the application. Opcodes and API calls provide a deep, low-level insight into what the application is actually doing, but it's an additional layer of permissions that helps give it a more wide-ranging view which throws even more light on what an application might be trying to achieve and some relevant risks.

3.5 Comparison with Other Related Work

We conducted a thorough comparison between the detection outcomes achieved by our suggested approach and those of other studies found in the existing literature regarding the detection of Android malware. We implemented several other state-of-the-art techniques on our data sets and to facilitate this comparison; we present a concise summary of the findings in Table 3.28, which encompasses the results obtained by various works that have utilized certain or all components of the manifest file for detection purposes. By examining these results, it becomes evident that our proposed methodology surpasses all of the aforementioned related works in terms of

detection accuracy, signifying its superior performance in comparison to existing approaches.

Table 3.28 Comparison of proposed model with related works.

Detection Technique	**Feature set Used**	**Detection Accuracy**	**No. of Applications**	**Feature Ranking Method**	**Feature Selection Method**
SIGPID [23]	Permissions	92 %	5494 malicious & 310,926 benign apps	Negative Rate & support	Sequential Forward Selection(SFS) & Principal Component Analysis (PCA)
PermPair [50]	Permissions	94.60 %	5993 benign & 7533 malicious applications	Ranked Permission-pairs	Not used
Proposed Approach	Permissions , Opcode, API Calls and System Calls	97%	15,000 benign & 15,000 malicious	Rough Set based Ranking	Chi-Square Test, Pearson Correlation & Rough Set Reduct

3.6 Limitations

The work performed in this research paper is based on static analysis. Static Android malware analysis has shortcomings, such as not capturing the run time behavior of applications like data leakage and network communications. Due to obfuscation techniques employed by malware writers, static analysis may not be able to capture the true intention of the code. With these limitations in the picture, static analysis may miss the malicious behaviour of Android applications, which may show its actual hostile conduct at run time.

Additionally, the current proposed model is an off-device model, and hence it can not be installed on smartphones for real-time detection.

3.7 Summary

We presented in this paper a novel Android malware detection model based on rough set theory. We utilized a hybrid of static features that are of four categories: permissions, opcodes, API calls, and system commands. In the first place, we preprocessed the data and eliminated features that had a high correlation and those not correlated at all with the class variable. The significance of each feature was determined using the ranking score assigned to it with a concept from the rough set theory, which is called the Discernibility Matrix. Further, an algorithm was then utilized to compute the rough set reducts. Here, the number of features in each category were reduced based on the ranking scores. After feature reduction, machine learning algorithms were also applied to evaluate the detection accuracy using the refined feature sets. Conclusion Results: Comparison with other advanced detection models found the proposed model to be superior to many state-of-the-art techniques.

The next chapter is dedicated in pairing permissions which is a static feature with the dynamic feature such as system calls. The pairing of static feature with dynamic feature is done to form a hybrid malware detection technique. The hybrid techniques contain advantages of both static and dynamic in order to form more robust malware detection model.

Chapter Four:
COVALENT BOND BASED ANDROID MALWARE DETECTION USING PERMISSION AND SYSTEM CALL PAIRS

In this chapter we propose a technique for Android malware detection using rough set theory. In section 4.1, we highlight the motivation behind the work done and briefly explained the overview of the proposed technique. In section 4.2 we explain in detail the methodology of the proposed technique. In section 4.3 the details of results are discussed and presented. The section 4.4 presents results and discussions. In section 4.5 the proposed approach is compared with other related works. The section 4.6 highlights the limitation of the approach. The section 4.7 summarizes the chapter with future directions.

4.1 Introduction

The Android operating system has maintained a dominant position in the smartphone industry for the past decade. Within the Android API framework, functions grant access to sensitive system resources. Unfortunately, this feature has allowed cyber attackers to develop and disseminate harmful applications through alternative app stores or social media advertisements. Furthermore, an attacker may introduce malicious components in the installed Android application. These malevolent applications empower attackers to perform various operations, including information theft, SMS transmission, and remote device control. Consequently, safeguarding smartphones from these malicious applications is imperative [11, 12, 13].

Malware detection methods currently fall into three primary categories: static, dynamic, and hybrid analysis. Static analysis is capable of discerning malicious behavior by examining an application's source code without executing it [113]. On the other hand, dynamic analysis identifies malicious behavior by analyzing the runtime information generated during the application's execution, such as system calls [114] . The strength of static analysis lies in its ability to pinpoint malicious components directly from the source code, resulting in high code coverage [115]. Dynamic analysis

excels in uncovering exploits within the runtime environment [116]. Therefore, by merging the strengths of static and dynamic analysis, a hybrid analysis approach can be formulated to enhance malware detection accuracy [117, 118].

Several static works have been proposed in the literature for Android malware detection. For instance, in [35], Talha et.al extracted application permissions. They then assign a score to each permission, determined by the ratio of malware instances containing that specific permission to the total number of malware instances. In [50], the study utilized pairs of permissions extracted from the manifest file, resulting in an overall accuracy of 95.44%. IPDroid, as discussed in [49], incorporated both permissions and intents from the manifest file in their analysis. They achieved a notable accuracy of 94.73% by employing a Random Forest classifier.

The TaintDroid model [119] employed dynamic taint analysis to monitor the movement of privacy-sensitive data within third-party applications. Yang et.al. [71] expanded upon the TaintDroid model to not only identify data leaks from applications but also ascertain whether these leaks are a result of user intention or not. In [85], the authors introduced a proficient and automated approach for detecting malware by leveraging the textual semantics of network traffic. Specifically, they treated each HTTP flow produced by mobile applications as a textual document, allowing them to apply natural language processing techniques to extract features at the text level.

Some of the works have combined static and dynamic features to propose a hybrid Android malware detector. MADAM [95] is a host-based malware detection system designed for Android devices. It conducts concurrent analysis and correlation of attributes across four tiers: kernel, application, user, and package. This comprehensive approach aims to identify and thwart malicious activities effectively. Monet [97] consists of a module on the user side, an application responsible for analyzing malicious activity and signatures. Conversely, the module installed on the server side is responsible for detecting malicious applications based on analysis on the client side. In [98] authors developed AppAudit which employs a combination of static and dynamic analysis to deliver highly effective real-time app auditing. It introduces an innovative dynamic analysis approach that leverages this combination to reduce false positives generated by an efficient yet conservative static analysis.

4.1.1 Motivation

Identifying dangerous combinations of permissions and system calls is instrumental in spotting malicious behavior. Hence, this study endeavors to scrutinize permissions and system calls in pairs and introduces a novel methodology to identify such pairs that can differentiate between benign and malicious samples. To the best of our knowledge, we are the first to use permissions and system call pairs to detect Android malware. Pairing permissions and system calls has several key benefits. Firstly, permissions are static features, and system calls are dynamic features; pairing both of them will combine the advantages of static analysis and dynamic analysis to form a hybrid analysis technique. Second, this combination allows for a more detailed examination of an application's behavior. Permissions provide a high-level overview of what resources an app may access, while system calls offer a finer-grained view of actual interactions with the system. By correlating permissions with system calls, we can better understand how an application uses the permissions it requests. This context is crucial in distinguishing legitimate behavior from potentially malicious actions. It enables the detection of anomalies or suspicious activities. For example, if an app with camera access permission unexpectedly starts making network-related system calls, it may raise a red flag. The app requests access to the camera (Android.permission.CAMERA). Additionally, it asks permission to access the internet (Android.permission.INTERNET). Based on permissions alone, the app seems legitimate. Camera apps naturally require camera access and internet access could be justified for features like cloud storage of images. During runtime, if the app makes system calls such as open(), read(), write(), and connect(). This observation may establish suspicious behavior as the app is accessing files unrelated to image storage and making network connections to unusual domains. Hence, this study endeavors to scrutinize permissions and system calls in pairs and introduces a novel methodology to identify such pairs that can differentiate between benign and malicious samples.

4.1.2 Contributions

We present a covalent bond-based Android malware detection model using permissions and system call pair. We use the analogy of covalent bonds between two atoms in chemistry to form covalent bonds between every permission and system call.

We also calculate bond strengths between permission and system call pairs to denote the strength of the bond they create between them. The estimated bond strength helps detect an Android application as malicious or benign. Our detection results demonstrate an overall accuracy of 97.5%, better than many state-of-the-art detection techniques proposed in the literature. The main contributions of the paper are summarized below.

- ✓ We build the permission and system call covalent bond pairs to identify and analyze the impact of these pairs.
- ✓ We proposed a novel approach to calculate the Covalent bond strength score for the permissions and system calls bond pair. Two scores, i.e., malicious and benign, are computed for each bond pair.
- ✓ We designed a technique for identifying Android applications as malicious or benign based on the malicious and benign scores of permission and system call pairs.
- ✓ We conducted a comparative analysis between our proposed model and other state-of-the-art detection techniques. Our findings demonstrate that the proposed model surpasses similar state-of-the-art models in terms of performance.

4.2 Methodology

In this section, we present our novel Covalent Bond Pair-based model for detecting malicious Android applications. The proposed model is depicted in Figure 4.1.

4.2.1 Data set Description

KronoDroid [120], a meticulously structured Android dataset, holds the distinction of being the largest in its category. It is distinguished by its amalgamation of static and dynamic features and the notable inclusion of timestamps. This dataset meticulously accounts for the unique characteristics of dynamic data sources, encompassing samples from over 209 distinct Android malware families. Its creation involved the fusion of diverse sources of benign and malware data, resulting in a comprehensive collection spanning a significant period. The dataset comprises 41,382

instances of malware belonging to 240 distinct malware families, along with 36,755 benign applications.

The dataset predominantly comprises permissions as static features, represented as binary indicators of whether the app requested the standard Android permissions (1) or not (0). There are a total of 166 distinct permissions in the dataset. In contrast, the dynamic feature set mainly consists of system calls, represented by the absolute frequency of each system call issued by the app at runtime. The system call set comprises 288 features. Hence, the total number of features under consideration amounts to 454.

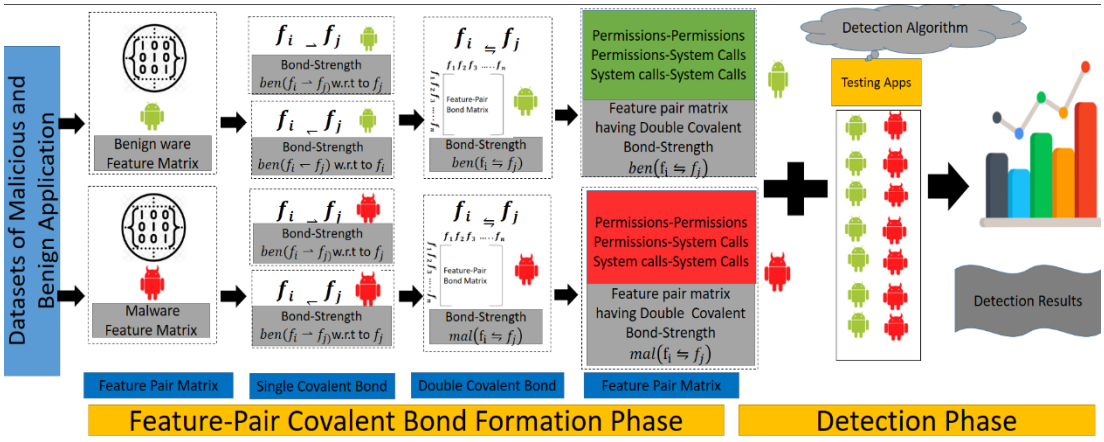

Figure 4.1 Proposed Covalent Bond Pair Detection Model

4.2.2 Feature Space Transformation

As previously stated, the KronoDroid dataset is well-organized and accessible in CSV file format. These files contain information on both malware and benign applications. The feature vectors within these CSV files are represented as combinations of 0's and 1's. A 0 in the feature vector signifies the absence of a particular feature in an application, while a 1 indicates its presence. Table 4.1 and Table 4.2 provide a visual representation of the feature spaces for benign and malicious applications respectively.

Table 4.1 Instance of Benign CSV

Benign CSV	P_1	P_2	P_3	.	.	.	.	.	P_n	S_1	S_2	S_3	.	.	.	.	.	S_m
A_{1B}	1	0	1	1	0	0	1	1	1	0	0	1	1	1	1	1	0	1
A_{2B}	1	0	1	1	0	0	1	0	1	0	0	1	0	1	0	1	0	1
.	0	0	1	1	0	0	1	1	0	0	0	0	1	1	1	0	0	1
.	0	0	1	1	0	0	1	1	0	1	0	0	1	1	1	0	0	0
A_{xB}	1	0	1	0	0	0	1	1	0	0	0	0	1	0	1	0	0	0

Within both the instances of benign and malicious CSV files as represented in Table 4.1 and Table 4.2 respectively, the labels P_1, P_2, P_3, ..., and P_n represent the n permissions, while S_1, S_2, S_3, ..., and S_m denote the m system calls. In our specific dataset, n is set at 166 and m at 288. The benign applications are denoted as A_{1B}, A_{2B}, ..., and A_{xB}, where x represents the total number of benign applications. Similarly, the malicious applications are labeled A_{1M}, A_{2M}, ..., and A_{yM}, with y indicating the total number of malicious applications.

Table 4.2 Instance of Malicious CSV

Malicious CSV	P_1	P_2	P_3	.	.	.	.	.	P_n	S_1	S_2	S_3	.	.	.	.	.	S_m
A_{1M}	0	0	1	1	0	1	1	1	1	0	0	0	1	1	0	1	0	1
A_{2M}	1	1	1	0	0	0	1	1	1	0	1	1	0	0	0	1	0	0
.	0	0	1	1	0	0	1	1	0	0	0	0	1	1	1	0	0	1
.	0	1	1	1	0	0	1	1	0	0	0	0	1	1	0	1	1	1
A_{yM}	1	1	1	0	0	0	1	1	0	0	1	1	1	0	1	0	1	0

4.2.3 Covalent Bond Pair Formation Phase

The concept of feature pair covalent bond formation is based on the concepts of the covalent bond theory of chemistry [36]. A covalent bond arises from the mutual sharing of electrons between the involved atoms. This pair of electrons engaged in this form of bonding is referred to as a shared pair or bonding pair. Additionally known as

molecular bonds, covalent bonds facilitate the attainment of outer shell stability, resembling the configuration of noble gases, by enabling the sharing of these bonding pairs. Covalent bonds are normally categorized into three types: single covalent bonds, double covalent bonds, and triple covalent bonds. We will restrict our proposed methodology to single covalent bonds and double covalent bonds only.

A single bond is established through the sharing of only one pair of electrons between the two involved atoms, symbolized by a single dash (-). Despite having lower density and strength than double and triple bonds, this type of covalent bond is the most stable.

A double bond is created when two pairs of electrons are shared between the participating atoms, denoted by two dashes (=). Double covalent bonds exhibit significantly greater strength than single bonds, although comparatively less stable.

In the case of our proposed methodology, we calculated single covalent bond strengths and double covalent bond strengths between two arbitrary features f_i and f_j, and formed feature pair f_{ij}. We separately calculated these bond strengths from two perspectives: w.r.t benign applications and w.r.t malicious applications. Hence, the concept of covalent bond strengths helps to calculate benign and malicious feature pair scores between every possible feature pair in the dataset. This notion of covalent bond strengths gives us a perspective of separately viewing any arbitrary feature pair regarding the role played for benign and malicious applications. Algorithm 1 depicts the whole phase of Feature Pair Covalent Bond Formation.

Algorithm 1: Feature Pair Covalent Bond Formation

1. ***Input:*** *benign feature matrix* $ben[A_{xB}][f_n]$ *where* x *is the number of benign applications and* n *is the number of features, malicious feature matrix* $mal[A_{yB}][f_n]$ *where* y *is the number of malicious applications and* n *is the number of features.*
2. ***Output:*** *benign feature pair matrix having double covalent bond strengths* $ben_{\leftrightharpoons}[f_n][f_n]$ *and malicious feature pair matrix having double covalent bond strengths* $mal_{\leftrightharpoons}[f_n][f_n]$.

3. ***for each*** $\boldsymbol{i}: \mathbf{1} \rightarrow \boldsymbol{n}$
4. ***for each*** $\boldsymbol{j}: \boldsymbol{i} + \mathbf{1} \rightarrow \boldsymbol{n}$
5. $n(f_{ij}) = 0$
6. $n(f_i) = 0$
7. $n(f_j) = 0$
8. ***for each*** $\boldsymbol{k}: \mathbf{1} \rightarrow \boldsymbol{x}$
9. ***If*** $(ben[A_{kB}][f_i] == 1 \;\&\&\; ben[A_{kB}][f_j] == 1)$
10. $n(f_{ij}) = n(f_{ij}) + 1$
11. ***end if***
12. ***If*** $(ben[A_{kB}][f_i] == 1)$
13. $n(f_i) = n(f_i) + 1$
14. ***end if***
15. ***If*** $(ben[A_{kB}][f_j] == 1)$
16. $n(f_j) = n(f_j) + 1$
17. ***end if***
18. ***end for***
19. $ben_{\rightarrow}[f_i][f_j] = n(f_{ij})/n(f_j)$
20. $ben_{\leftarrow}[f_i][f_j] = n(f_{ij})/n(f_i)$
21. ***end for***
22. ***end for***
23. ***for each*** $\boldsymbol{i}: \mathbf{1} \rightarrow \boldsymbol{n}$
24. ***for each*** $\boldsymbol{j}: \boldsymbol{i} + \mathbf{1} \rightarrow \boldsymbol{n}$
25. $n(f_{ij}) = 0$
26. $n(f_i) = 0$
27. $n(f_j) = 0$
28. ***for each*** $\boldsymbol{k}: \mathbf{1} \rightarrow \boldsymbol{y}$
29. ***If*** $(mal[A_{kB}][f_i] == 1 \;\&\&\; mal[A_{kB}][f_j] == 1)$
30. $n(f_{ij}) = n(f_{ij}) + 1$

31. ***end if***

32. ***If*** $(mal[A_{kB}][f_i] == 1)$

33. $n(f_i) = n(f_i) + 1$

34. ***end if***

35. ***If*** $\left(mal[A_{kB}][f_j] == 1\right)$

36. $n(f_j) = n(f_j) + 1$

37. ***end if***

38. ***end for***

39. $mal_{\rightharpoonup}[f_i][f_j] = n(f_{ij})/n(f_j)$

40. $mal_{\leftharpoondown}[f_i][f_j] = n(f_{ij})/n(f_i)$

41. ***end for***

42. ***end for***

43. ***for each*** $\boldsymbol{i}: \mathbf{1} \rightarrow \boldsymbol{n}$

44. ***for each*** $\boldsymbol{j}: \boldsymbol{i} + \mathbf{1} \rightarrow \boldsymbol{n}$

45. $ben_{\leftrightharpoons}[f_i][f_j] = (ben_{\rightharpoonup}[f_i][f_j] + ben_{\leftharpoondown}[f_i][f_j])/2$

46. $mal_{\leftrightharpoons}[f_i][f_j] = (mal_{\rightharpoonup}[f_i][f_j] + mal_{\leftharpoondown}[f_i][f_j])/2$

47. ***end for***

48. ***end for***

The data set is assumed to have benign and malicious feature matrices in which each of the application feature vectors in the form of 0's and 1's is represented, respectively. Then, the feature vs. the feature matrix is calculated from these feature matrices, holding single covalent bond strengths. If f_i and $f_{j,}$ are two arbitrary features, then we calculate two single covalent bond strengths for the feature pair f_{ij}, one w.r.t f_i and other w.r.t $f_{j,}$. Calculating single bond strength is done from benign and malicious perspectives. The single covalent bond strength of feature vs. feature matrices is combined to form new feature vs. feature matrices holding double covalent bond strengths for both benign and malicious perspectives.

Let us suppose an instance of benign and malicious information systems, as shown in Table 4.3 and Table 4.4 P_1, P_2, and P_3 denote permissions as features in both instances. Similarly, S_1, S_2, and S_3 denote system calls as features. A_{1B}, A_{2B}, A_{3B}, A_{4B}, and A_{5B} denote the benign applications in the supposed instance of benign information systems. Similarly, A_{1M}, A_{2M}, A_{3M}, A_{4M}, and A_{5M} denote the malicious applications in the supposed instance of a malicious information system.

After assuming the benign and malicious instances of the information systems, now we show how to calculate the single bond strengths of every feature pair. As discussed earlier, single bond strengths of two arbitrary features are calculated from two perspectives, i.e., benign and malicious. For each perspective, the single bond strengths are calculated again from two aspects, i.e., w.r.t f_i and w.r.t f_j. The formulas for this are evident from Eq. 1, 2, 3, and 4.

Table 4.3 Supposed Instance of Benign Information Systems

Benign	P_1	P_2	P_3	S_1	S_2	S_3
A_{1B}	0	1	1	1	0	0
A_{2B}	0	0	1	1	1	0
A_{3B}	1	0	1	0	1	1
A_{4B}	0	0	1	0	1	0
A_{5B}	1	0	1	0	1	0

Table 4.4 Supposed Instance of Malicious Information Systems

Malicious	P_1	P_2	P_3	S_1	S_2	S_3
A_{1M}	1	0	0	1	1	1
A_{2M}	1	1	0	1	0	1
A_{3M}	1	1	0	0	1	1
A_{4M}	0	1	0	0	1	0
A_{5M}	0	0	1	1	0	0

Eq. 1 denotes the single benign bond strength of the feature pair f_{ij} w.r.t feature f_j. As discussed earlier, the single bond is established by sharing only one pair of electrons between the two involved atoms, symbolized by a single dash (-). The same phenomenon is established in our concept represented by equation 1 as $ben_{\rightharpoonup}[f_i][f_j]$. Here, the ($\rightharpoonup$) represents a single covalent bond w.r.t. to the feature at the right side of the arrow, simulating the sharing of only one electron pair. It gives us the benign score of the single covalent bond between f_i and f_j w.r.t. f_j, where $n(f_{ij})$ is the number of applications for which both features were present simultaneously in the benign feature matrix. In addition, $n(f_j)$ is defined as the number of applications for which the feature f_j is present. The value for equation 1 will be lying in the set [0, 1]. A value of 1 indicates a strong single covalent bond while a value of 0 indicates a weak bond. The ratio of $n(f_{ij})$ w.r.t $n(f_j)$ denotes the the probability that the association between two features f_i and f_j in the is strong or weak w.r.t to the feature f_j i.e., higher the ratio greater the association.

$$ben_{\rightharpoonup}[f_i][f_j] = n(f_{ij})/n(f_j) \quad (1)$$

$$ben_{\leftharpoondown}[f_i][f_j] = n(f_{ij})/n(f_i) \quad (2)$$

$$mal_{\rightharpoonup}[f_i][f_j] = n(f_{ij})/n(f_j) \quad (3)$$

$$mal_{\leftharpoondown}[f_i][f_j] = n(f_{ij})/n(f_i) \quad (4)$$

Eq. 2 denotes the single benign bond strength of the feature pair f_{ij} w.r.t feature f_i. Here ($\leftharpoondown$) represents a single covalent bond w.r.t. to the feature at the left side of the arrow, simulating the sharing of only one electron pair. It gives us the benign score of the single covalent bond between f_i and f_j w.r.t. f_i, where $n(f_{ij})$ is the number of applications for which both features were present simultaneously in the benign feature matrix. In addition, $n(f_i)$ is defined as the number of applications for which the feature f_i is present. The value for equation 2 will be lying in the set [0, 1]. A value of 1 indicates a strong single covalent bond while a value of 0 indicates a weak bond.

Similarly, with the help of equations 3 and 4, we can calculate $mal_{\rightharpoonup}[f_i][f_j]$ and $mal_{\leftharpoondown}[f_i][f_j]$ where the former is the single malicious bond strength of the feature pair f_{ij} w.r.t feature f_j while later is the single malicious bond strength of the feature pair f_{ij} w.r.t feature f_i. They are both calculated from the malicious feature pair matrix. The value for equation 5 and 6 will be lying in the set [0, 1]. A value of 1 indicates a strong double covalent bond while a value of 0 indicates a weak bond. Since the single covalent bonds are calculated from two perspectives i.e., w.r.t f_i and f_j separately, taking their average will give the strength of association between the two features w.r.t both the perspectives. Higher the average value greater the association between both the features w.r.t both the perspectives.

$$ben_{\leftrightharpoons}[f_i][f_j] = (ben_{\rightharpoonup}[f_i][f_j] + ben_{\leftharpoondown}[f_i][f_j])/2 \quad (5)$$

$$mal_{\leftrightharpoons}[f_i][f_j] = (mal_{\rightharpoonup}[f_i][f_j] + mal_{\leftharpoondown}[f_i][f_j])/2 \quad (6)$$

Eq. 5 and 6 calculate double covalent bond strength for the feature pair f_{ij}. $ben_{\leftrightharpoons}[f_i][f_j]$ denotes the double covalent benign bond strength, and $mal_{\leftrightharpoons}[f_i][f_j]$ denotes the double covalent malicious bond strength. As discussed, the double covalent bond is created when two pairs of electrons are shared between the participating atoms, denoted by two dashes (=). We used ($\leftrightharpoons$) to denote a double covalent bond for the feature pair f_{ij}. The benign single covalent bond strengths calculated in equations 1 and 2 are used to calculate double covalent bond strength in eq. 5, simulating the sharing of two pairs of electrons between the participating atoms. Similarly, the malicious covalent bond strengths calculated in equations 3 and 4 are used to calculate double covalent bond strengths in eq. 6.

Table 4.5 and Table 4.6 depict benign and malicious feature pair matrices representing benign and malicious double feature pair covalent bond strengths, respectively. Table 4.5 and Table 4.6 are calculated from Table 4.3 and Table 4.4 using equations 1, 2, 3, 4, 5, and 6.

Table 4.5 Instance of Benign Feature Pair Matrix

Benign	P_1	P_2	P_3	S_1	S_2	S_3
P_1		0	0.7	0	0.7	0.75
P_2			0.6	0.75	0	0
P_3				0.7	0.6	0.6
S_1					0.37	0
S_2						0.33
S_3						

Table 4.6 Instance of Malicious Feature Pair Matrix

Benign	P_1	P_2	P_3	S_1	S_2	S_3
P_1		0.66	0	0.66	0.66	0.5
P_2			0	0.33	0.66	0.66
P_3				0.6	0	0
S_1					0.33	0.66
S_2						0.66
S_3						

4.2.4 Detection Phase

The double covalent benign and malicious bond strength calculated in the previous phase is used to detect an arbitrary application as malicious or benign. The whole process of the detection phase is depicted in Algorithm 2.

The testing application is first analyzed to form all possible distinct feature pairs. After this, the net benign and malicious scores are calculated based on the feature pairs formed for the test application. The net benign and malicious scores are calculated from the double covalent benign and malicious strengths stored in benign and malicious feature pair matrices, respectively.

Let us take an instance of the test Android application as A_t, then the net benign score and net malicious score of the application are calculated with the help of Eq. 7 and 8 respectively.

$$net_{ben}(A_t) = net_{ben}(A_t) + ben_{\leftrightharpoons}[f_i][f_j] \quad (7)$$

$$net_{mal}(A_t) = net_{mal}(A_t) + mal_{\leftrightharpoons}[f_i][f_j] \quad (8)$$

In Eq. 7 the $net_{ben}(A_t)$ represents the net benign score of application A_t whereas in Eq. 8 the $net_{mal}(A_t)$ represents the net malicious score. Both equations sum up the benign and malicious feature pair scores of all the distinct feature pairs respectively. If $net_{mal}(A_t)$ score is greater than $net_{ben}(A_t)$ then we can deduce that the test application A_t is detected as malicious otherwise benign.

Algorithm 2: Feature Pair Covalent Bond Formation

1. ***Input:*** *benign feature pair matrix having double covalent bond strengths* $ben_{\leftrightharpoons}[f_n][f_n]$ *and malicious feature pair matrix having double covalent bond strengths* $mal_{\leftrightharpoons}[f_n][f_n]$*. Set of Applications* $(A_1, A_2, \ldots\ldots, A_n)$ *to be Tested*
2. ***Output:*** *Each of the applications is Malicious or Benign.*
3. ***for each application*** $(\boldsymbol{A_t})$ $\boldsymbol{t}: \mathbf{1} \rightarrow \boldsymbol{n}$
4. $net_{ben}(A_t) = 0$
5. $net_{mal}(A_t) = 0$
6. ***for each*** $\boldsymbol{i}: \mathbf{1} \rightarrow \boldsymbol{n}$
7. ***for each*** $\boldsymbol{j}: \boldsymbol{i} + \mathbf{1} \rightarrow \boldsymbol{n}$
8. ***if*** $(\boldsymbol{f_i} \in \boldsymbol{A_t}$ && $\boldsymbol{f_j} \in \boldsymbol{A_t})$
9. $net_{ben}(A_t) = net_{ben}(A_t) + ben_{\leftrightharpoons}[f_i][f_j]$
10. $net_{mal}(A_t) = net_{mal}(A_t) + mal_{\leftrightharpoons}[f_i][f_j]$
11. ***end if***
12. ***end for***
13. ***end for***
14. ***If*** $(net_{mal}(A_t) > net_{ben}(A_t))$
15. ***Return*** A_t *as Malicious*
16. ***Else***

17. ***Return*** A_t *as Benign*

18. ***end if***

19. ***end for***

4.3 Results and Discussions

This section reports results obtained from each of the covalent bond pair models. Three types of detection models are formed with the help of covalent bonds pair: permissions-permissions, system calls-system calls, and permissions-system calls.

4.3.1 Feature Pair Analysis

Table 4.7 shows the top ten highest-scoring permission pairs based on both malicious and benign covalent bond strengths between them. The maximum malicious permissions pair is INTERNET and READ_PHONE_STATE, with the malicious colavent bond strength score of 0.96. This behavior seems evident because pairing INTERNET and READ_PHONE_STATE permissions in an Android app may pose privacy and security risks. The INTERNET permission allows access to online resources, while READ_PHONE_STATE grants access to device details like phone numbers and network information. These permissions could enable an app to collect and transmit sensitive user data without consent, potentially indicating malicious intent. Similarly, the reason for other pairs could also be inferred.

Table 4.8 shows system call- system call covalent bond pairs with their malicious and benign score arranged in descending order of covalent bond strengths. The top pair in this table with the highest malicious score is “getuid32-ioctl”. The getuid32 system call retrieves the effective user ID of a process in Linux-based operating systems. On the other hand, the ioctl system call, which stands for "Input/Output Control," is employed in Unix-like systems to control devices beyond standard read and write operations. When used together, these system calls could be leveraged in a potentially malicious manner. For instance, a malicious program might use getuid32 to ascertain if the current user possesses administrative privileges. If

affirmative, it could then utilize ioctl to manipulate a system device or resource, potentially resulting in a security breach or compromise.

Table 4.7 Top Ten highest scoring Permissions pair from both malicious and benign perspectives.

Malicious		**Benign**	
Permissions-Permissions pair	**Malicious score**	**Permissions-Permissions pair**	**Benign score**
INTERNET-READ_PHONE_STATE	0.96	READ_SYNC_SETTINGS-WRITE_SETTINGS	0.98
ACCESS_NETWORK_STATE-INTERNET	0.93	BROADCAST_PACKAGE_REMOVED-BROADCAST_STICKY	0.96
ACCESS_COARSE_LOCATION-ACCESS_FINE_LOCATION	0.92	BROADCAST_PACKAGE_REMOVED-RESTART_PACKAGES	0.84
ACCESS_NETWORK_STATE-READ_PHONE_STATE	0.92	BIND_WALLPAPERS-BLUETOOTH	0.79
INTERNET-WRITE_EXTERNAL_STORAGE	0.91	QUERY_ALL_PACKAGES-WRITE_APN_SETTINGS	0.75
READ_PHONE_STATE-WRITE_EXTERNAL_STORAGE	0.89	ACCESS_MEDIA_LOCATION-INTERACT_ACCROSS_PROFILES	0.72
ACCESS_NETWORK_STATE-WRITE_EXTERNAL_STORAGE	0.88	ACCESS_NETWORK_STATE-WRITE_EXTERNAL_STORAGE	0.72

Malicious		Benign	
Permissions-Permissions pair	**Malicious score**	**Permissions-Permissions pair**	**Benign score**
ACCESS_NETWORK_STATE-ACCESS_WIFI_STATE	0.87	INTERNET-READ_PHONE_STATE	0.71
ACCESS_WIFI_STATE-READ_PHONE_STATE	0.84	INTERNET-WRITE_EXTERNAL_STORAGE	0.70
ACCESS_WIFI_STATE – INTERNET	0.82	ACCESS_NETWORK_STATE-INTERNET	0.70

Table 4.8 Top Ten highest-scoring system call pair from both malicious and benign perspectives.

Malicious		Benign	
System Calls-System Calls pair	**Malicious score**	**System Calls-System Calls pair**	**Benign score**
getuid32-ioctl	0.998	prctl-madvise	0.998
prctl– madvise	0.998	close-SYS_305	0.994
fstat64-SYS_305	0.998	fstat64-SYS_305	0.993
prctl-fstat64	0.997	getuid32-ioctl	0.993
close-SYS_305	0.997	close-fstat64	0.992
prctl-SYS_305	0.996	ioctl-writev	0.992
mmap2-SYS_305	0.996	madvise-mmap2	0.992
mprotect-fstat64	0.996	mprotect-SYS_305	0.991
prctl-mprotect	0.996	prctl-mmap2	0.991
close-mmap2	0.996	read-ioctl	0.991

Table 4.9 Top Ten highest-scoring system call and permission pairs from both malicious and benign perspectives.

Malicious		Benign	
System call-Permissions pair	**Malicious score**	**System call-Permissions pair**	**Benign score**
clock_gettime-INTERNET	0.98	clock_gettime-INTERNET	0.90
getuid32-INTERNET	0.97	ioctl-INTERNET	0.895
ioctl-INTERNET	0.97	getuid32-INTERNET	0.894
close-INTERNET	0.968	read-INTERNET	0.892
futex-INTERNET	0.968	writev-INTERNET	0.892
fadvise64_64 - INTERNET	0.968	write-INTERNET	0.889
SYS_305-INTERNET	0.967	close-INTERNET	0.877
fstat64-INTERNET	0.967	fadvise64_64-INTERNET	0.877
mprotect-INTERNET	0.966	fstat64-INTERNET	0.876
prctl -INTERNET	0.965	SYS_305-INTERNET	0.875

Table 4.9 shows system call and permission pair covalent bonds arranged in descending order of their malicious and benign bond strength score, respectively. One of the top system call and permission pairs in malicious and benign pairs is clock_gettime and INTERNET. An application may use the precise timing obtained from clock_gettime with the internet access granted by the INTERNET permission to perform covert communication. The combination of precise timing and internet access could allow an application to engage in stealthy activities, making it harder to detect malicious behavior. The malicious score of this pair is 0.98, while the benign score is 0.90. Hence, its malicious intent is more in our case than normal intent. Still, one could not rule out that many legitimate applications use these functionalities for legitimate purposes, such as measuring performance or synchronizing with online services.

4.3.2 Detection Results

Table 4.10 shows the performance of various detection models. The proposed models are evaluated on five parameters, i.e., True Positive Rate (TPR), False Positive Rate (FPR), Precision, Accuracy, and F1-Score. The permissions-permissions model is static as it considers only permission-permission covalent bond score for detecting Android Malware applications. The system call – system call covalent bond pair model is dynamic and has better results in the evaluation parameters, which is evident from the fact that dynamic features consider the run time behavior of the application while static feature does not. Hence, those malicious behavior that are activated at run time uncovers hidden insights that are helpful in the identification of malicious application. The next model is the permissions–system call model, a hybrid model in which a covalent bond pair is formed among permissions and system calls, and their bond strengths are used to detect malicious applications. This model, which is a hybrid, has even better evaluation parameters than the system call- system call detection model. The apparent reason seems to be the uncovering of system calls and permissions bonding. The permission requested by the application is not alone responsible for malicious behavior because benign applications may also use the same permission. The combination of permission with system calls allows a more detailed examination of an application's behavior. Permissions provide a high-level overview of what resources an app may access, while system calls offer a finer-grained view of actual interactions with the system. The Permissions-System calls model shown is the best of all. This model is a hybrid model and achieves an overall accuracy of 97.50%, which is better than both static and dynamic models. The confusion matrix for the permissions-system call model is given in Table 4.11.

Table 4.10 Performance of Proposed Detection Models

Detection Model	**TPR**	**FPR**	**Precision**	**Accuracy**	**F1-Score**
Permissions-Permissions	92.27%	5.80%	94.98%	93.39%	93.84%
System calls-System Calls	95.97%	4.43%	96.06%	95.78%	96.01 %

Detection Model	TPR	FPR	Precision	Accuracy	F1-Score
Permissions- System calls	97.77%	2.81%	97.49%	97.50%	97.63%

Table 4.11 Confusion Matrix of Proposed Detection Model

		Predicted	
		Malicious	Benign
Actual	Malicious 12415	True Positive 12104	False Positive 311
	Benign 11027	False Negative 275	True Negative 10752

4.3.3 Detection Results on Unknown Samples

We comprehensively evaluate our proposed model on unknown samples. The sample is taken from the CICAndMal2017 [121] data set. A total of 1800 samples were taken, of which 1000 were malicious, and 800 were benign. These are the unseen samples as they are in the form of apks. We first installed these applications in a virtual environment, and then random clicks were done on installed applications for nearly a minute. The generated system calls are captured with the help of a strace script. The permissions were extracted from the Android manifest file of each application after unpacking each application using the apk tool. The observed result shows an accuracy of 96.20 %. The details of the results are represented in Table 4.12 And Table 4.13.

Table 4.12 Performance of Proposed Detection Models on Unknown Samples.

Detection Model	TPR	FPR	Precision	Accuracy	F1-Score
Permissions-Permissions	93.32%	7.57%	94.%	92.88%	93.62%
System calls-System Calls	94.30%	5.11%	96%	94.55%	95.14 %
Permissions- System calls	95.33%	2.5%	98.%	96.20%	96.64%

Table 4.13 Confusion Matrix of Proposed Detection Model on Unknown Samples

		Predicted	
		Malicious	Benign
Actual	Malicious 1000	True Positive 980	False Positive 20
	Benign 800	False Negative 48	True Negative 752

4.4 Comparison with other related works

We comprehensively evaluate the detection results obtained from our proposed method, comparing them with findings from previous studies in the literature focusing on Android malware detection. We implemented several state-of-the-art techniques on our data set and to facilitate this comparison, we provide a concise summary in Table 4.14. Upon examination of these results, it becomes apparent that our proposed methodology outperforms all the aforementioned related works regarding detection accuracy, demonstrating its superior performance compared to

existing approaches. Moreover, the data set used by all the approaches was old and outdated. The data set used by us is the latest, and it chronicles the entire history of Android, covering the years from 2008 to 2020 while also accounting for the distinct dynamic data sources.

Table 4.14 Comparison of Proposed Model with Related Works.

Methodology	**Approach**	**Features set used**	**No. of Applications**	**TPR**	**Accuracy**
PermPair [50]	Static	Permissions	5993 benign and 7533 malicious	94.11%	94.54 %
Guerra-Manzanares et al. [122]	Dynamic	System Call	28343 Malicious and 34981 Benign	93.60%	94.40%
Guerra-Manzanares et al. [123]	Static	Permissions	4174 Malicious and 37020 Benign	92.80 %	93.50%
Guerra-Manzanares et al. [124]	Dynamic	System Call	41382 Malicious and 36755 Benign	95.50%	95.90%
Proposed Approach	**Hybrid**	**Permissions and System Call**	**41382 Malicious and 36755 Benign**	**97.77%**	**97.50%**

4.5 Limitations

In this subsection, we address certain ambiguities in our proposed approach. Specifically, our model relies on feature pairs to assess applications. Some malware samples with a limited number of features may go undetected. To bypass detection, attackers may incorporate commonly used features into the malware, thereby generating a more significant number of ordinary feature pairs. Additionally, we've observed that when a feature pair appears only once in the malicious samples, and both individual features have a frequency of one for a specific application, it results in a malicious covalent bond strength of one. This misrepresents the actual strength of the bond, potentially elevating the significance of an otherwise insignificant feature pair and leading to misclassification. We plan to address these limitations by exploring the potential of incorporating additional components like intent filters, hardware specifications, and API call logs for more efficient detection alongside the existing focus on permissions and system calls.

4.6 Summary

This study established covalent bonds between permissions and system calls to evaluate their combined impact. We introduced a novel methodology for calculating these pairs' Covalent Bond Strength Score, resulting in both malicious and benign scores. These scores were then utilized in our Android malware detection technique.

We thoroughly compared our proposed model and other advanced detection methods. Our results indicate that our model outperforms similar state-of-the-art models in performance.

The next chapter is based on developing an Android malware detection model using process memory dump files. The process memory dump files capture the dynamic behaviour of the Android malware application under execution. Some stealthy malwares hide their true malicious intent and shows their malicious behaviour only at run time. The technique proposed in the next chapter tries to capture the dynamic behaviour of malicious application.

Chapter Five:
A VISUAL ANDROID MALWARE DETECTION TECHNIQUE BASED ON PROCESS MEMORY DUMP FILES

In this chapter we propose a technique for Android malware detection using rough set theory. In section 5.1, we highlight the motivation behind the work done and briefly explained the overview of the proposed technique. In section 5.2 we explain in detail the methodology of the proposed technique. In section 5.3 the details of results are discussed and presented. The section 5.4 concludes the chapter with future directions.

5.1 Introduction

Android-based smartphones are the most popular among other smartphones, which is evident from the fact that Android-based smartphones have approximately 70% of the total share. The popularity of these devices is the primary reason for the growth of malicious attacks on these devices. Malware writers attack millions of these devices as they target these devices because of the financial and personal data they contain. Hence, there is an urgent need to develop malware detection techniques to counter these attacks.

Malware detection techniques are based on malware analysis techniques. Malware analyses are divided into static, dynamic, and hybrid methods. Each static, dynamic, and hybrid analysis differentiates itself from another in the way it gathers features for performing malware analysis, which is used to develop a malware detection model based on the extracted features.

The static malware analysis technique is one in which the characteristics of Android applications are analyzed without actually executing the file. Such techniques are fast and more scalable in the sense that no execution of the application is re-quired.

In dynamic malware analysis, the characteristics of the Android application are analyzed by running the application's functionality in a controlled environment. Various characteristics of the applications are considered in execution based on which application is characterized as benign or malicious.

In the case of hybrid analysis, both static characteristics, i.e., those features of the application, are derived without executing the application, and dynamic characteris-tics, i.e., those features that are derived while the application is under execution, are considered.

5.1.1 Motivation

The one research methodology that has been explored less in the realm of Android malware analysis is one that uses visual techniques. This methodology is based on representing Android application-related data as an image. The image generated is further used to derive features and, hence, develop a classification model based on those features for malware detection. In [125], authors converted the source files of Android applications into grayscale images, and further local and global features were extracted from these images to train machine learning classifiers. The EfficientNet CNN-based Android malware detection model was based on converting Android Dex files into images. These images are then fed 26 state-of-the-art CNN models, out of which the EfficientNet-B4 CNN-based model gave the best results [126]. In [127], the authors converted the non-intuitive malware features into grayscale images and used machine learning classifiers on a softmax layer of CNN to analyze the generated grayscale images. Ding et al. [128] converted the byte code of Android applications into images and used CNN to train on those images to build an effective malware detection model.

In the current chapter, we developed a visual malware detection technique based on process memory dump files. An Android process memory dump, referred to as a memory dump or core dump, captures the memory snapshot of a running process on an Android device at a specific instance. It encompasses details concerning memory allocation, variables, registers, and other pertinent data structures linked to the process.

Extracting strings from memory is a valuable approach in malware analysis for various reasons. For instance: Unveiling Hardcoded Strings: Malicious software often embeds hardcoded strings to facilitate command and control (C2) communication, encryption keys, or URLs. Delving into memory strings can uncover these hardcoded values, providing insights into the malware's functionality and potential indicators of compromise (IOCs).

Uncovering Obfuscated or Encrypted Strings: Certain malware employs tactics like string obfuscation or encryption to evade detection by conventional static analysis methods. Scrutinizing memory strings may expose these obfuscated or encrypted strings in their plain text form, thereby facilitating deeper analysis.

Exposing Command and Control (C2) Infrastructure: Malware often communicates with remote command and control servers to receive instructions or exfiltrate data. Memory-extracted strings may harbor URLs, IP addresses, or domain names linked with the C2 infrastructure, empowering analysts to identify and potentially disrupt malicious communications.

Discovering Indicators of Compromise (IOCs): Analysts can pinpoint common patterns or unique signatures that serve as indicators of compromise (IOCs) by analysing memory string dumps across multiple malware samples or infected systems. These IOCs are pivotal in detecting and mitigating similar threats in the future.

5.1.2 Contributions

We introduce an innovative approach for Android malware detection leveraging visual techniques. Our method involves the transformation of Android process memory dump files into grayscale images. The memory dump files are meticulously read, with each byte converted into uniform binary representations. These binary sequences are then utilized to generate grayscale images. Subsequently, features such as color histograms, Hu moments, and Haralick textures are extracted from these grayscale images. These derived features train machine learning classifiers, distinguishing between benign and malicious Android applications.

5.2 Methodology

This section explains the technique used to classify an Android application as benign or malicious using memory dumps. An Android process memory dump, also known as a memory dump or core dump, is a snapshot of the memory state of a running process on an Android device at a particular moment in time. It contains information about the memory allocation, variables, registers, and other data structures associat-ed with the process. The proposed model for classifying an Android application as benign or malicious using a memory dump is depicted in Figure 5.1.

The Proposed model is further divided into to number of sub-phases such as the visual representation of Android Process Memory Dump files represented in sub-section 5.2.1. Sub-Section 5.2.2 is dedicated to explaining feature extraction from visual representations.

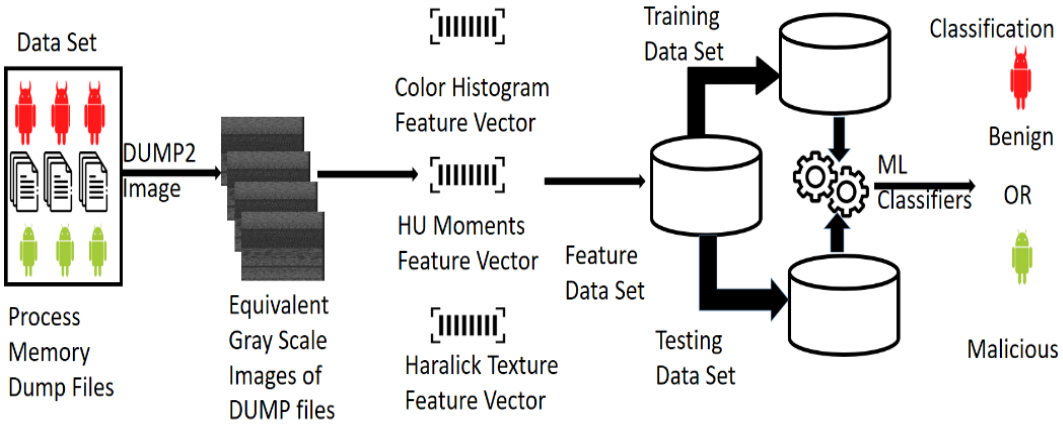

Figure 5.1 Proposed Model for classifying an Android Application as Benign or Malicious from Memory DUMP files.

5.2.1 Visual representation of Android Process Memory Dump files

We have taken the contents of the required Android process memory dump as a bi-nary bit stream and arranged it into a byte matrix. Within the generated datasets, the malware source is depicted as a grayscale image for the sake of this study. Pixel values in the grayscale image range from 0 to 255, where 0 denotes black and 255 denotes white. The length of each image in the datasets varies according to the file size, but all of them have a fixed width of 256 pixels. With values ranging from 0 to

255, each byte in the created byte matrix represents a pixel in the final grayscale image. Following the conversion of the byte matrix into a matrix of values within the 0 to 255 range, the resulting matrix is preserved as a grayscale image. Figure 5.2 visually depicts transforming a malware sample into a grayscale image.

5.2.2 Feature Extraction from Visual Representations

Three types of features are extracted from visual representations of memory dump files i.e., grayscale images created in the previous step. These features are color histogram feature, Hu moments, and Haralick texture.

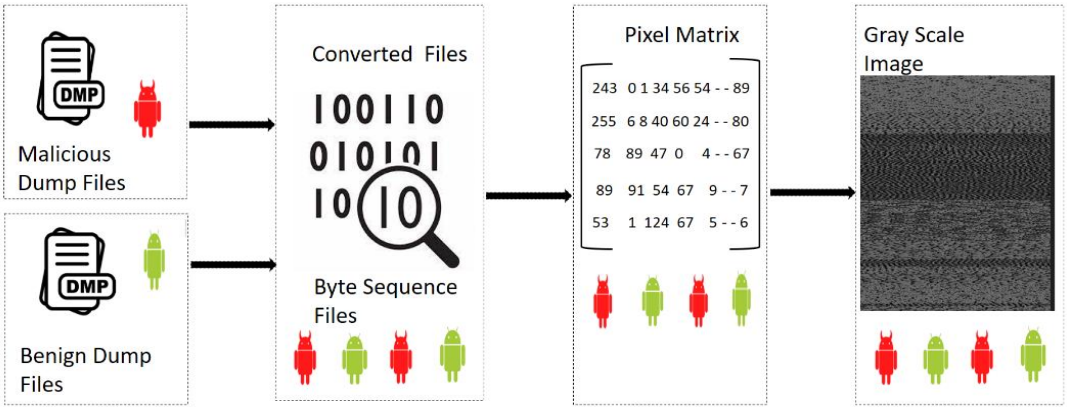

Figure 5.2 Process of converting Memory Dump files into grayscale images

Color histograms

They are commonly employed in the analysis of color images, focusing on the distribution of colors like red, green, and blue across various channels. Conversely, in grayscale imagery, which contains only one intensity channel, the concept of a color histogram is not directly applicable. Instead, we generate a histogram based on pixel intensities for grayscale images. This histogram delineates the distribution of pixel values (ranging from 0 to 255 in an 8-bit grayscale image) throughout the image. We segment the intensity spectrum into 256 bins, each representing a possible intensity value. Subsequently, we tally the frequency of pixels associated with each intensity value and visualize the resulting histogram. Normalizing the histogram values to sum up to 1 facilitates comparison across grayscale images of varying dimensions. We utilize the histogram as a feature vector for the grayscale image by considering each bin that signifies the occurrence frequency of a specific intensity value within the

image. This feature vector is input to machine learning algorithms for subsequent analysis and processing.

5.2.2.1 Hu moments

They represent a set of seven invariant image descriptors utilized in shape analysis and recognition. These descriptors stem from raw moments, which mathematically capture an image's shape, allowing for the description of objects regardless of their position, size, or orientation. Initially, grayscale images undergo conversion into binary images employing thresholding techniques such as Otsu's method or adaptive thresholding, given Hu moments' primary application in binary images. Subsequently, we compute the central moments of the binary image, ensuring translation invariance crucial for Hu moments' robustness. The normalization of central moments follows suit to achieve scale invariance, entailing the division of each central moment by an appropriate power of the zeroth moment, indicative of the object's total mass or area. Ultimately, the computation of Hu moments ensues using the normalized central moments. These seven moments, denoted as Hu1 through Hu7, arise from specific combinations of the normalized central moments. As a feature vector, Hu moments encapsulate the object's shape within the grayscale image. Notably, these moments act as features invariant to translation, rotation, and scale changes, rendering them invaluable in various image analysis tasks.

5.2.2.2 Haralick texture

These features, are also called grey-level co-occurrence matrix (GLCM) features, constitute a collection of statistical metrics for characterizing the texture of grayscale images. These metrics capture the spatial correlations among pixel intensities within an im-age and find widespread application in image analysis and classification endeavours. Initially, we computed the grayscale image's grey-level co-occurrence matrix (GLCM). This matrix tabulates the occurrences of pixel intensity pairs at specific spatial relationships, such as distance and direction, within the image. Subsequently, we selected particular properties or characteristics for extraction from the GLCM. These chosen properties encompass contrast, correlation, energy, and homogeneity. Following this, normalization was employed to scale the computed texture features within a standardized range of 0 to 1, ensuring uniformity across

diverse images. Lastly, a feature vector was constructed to serve as input for machine learning classifiers.

5.3 Results and Discussions

The last section explained how the visual representation in the form of a grayscale image was created from the Android memory dump files. It also explained how features such as color histograms, Hu moments, and Haralick texture were extracted from those grayscale images. This section discusses the experimentation to classify an application as benign or malicious based on the features extracted from the grayscale images of memory dump files.

We used four machine learning classifiers: SVM, Random Forest, K nearest neighbors, and Logistic Regressions. The Data set used for experimentation is the Android Process Memory String Dump [129]. The data set is built using AndroZoo's APK files [109]. The data set mainly consists of strings related to individual processes running at the time of APK usage. Each file in the data set corresponds to an APK. There are 2375 samples of Android process memory string dump files. Of the 2375 files, 1,188 samples correspond to malicious applications, while the remaining 1,187 samples correspond to benign applications.

We trained the model using the combined feature vector formed from all three types of features. We used a train-test split ratio of 70:30. The results are measured on the following parameters: accuracy, precision, recall, and F1-score. The results of the experiments are depicted in Table 5.1.

We observed that Random Forest performs better than the rest of the machine learning classifiers as Random Forest outperforms other classifiers due to its unique methodology. Unlike individual decision trees, Random Forest utilizes an ensemble approach by aggregating multiple decision trees for prediction. This ensemble technique effectively reduces variance and minimizes overfitting risks present in standalone decision trees. By combining predictions from numerous trees, Random Forest consistently improves overall performance.

Table 5.1 Experimental Results

ML Techniques	Accuracy	Precision	Recall	F1-Score
K-Nearest Neighbor	0.89	0.87	0.88	0.88
Logistic Regression	0.89	0.88	0.87	0.89
SVM	0.92	0.91	0.90	0.91
Random Forest	0.94	0.95	0.95	0.94

5.4 Summary

In this work, we have proposed a novel Android malware detection mechanism based on visual techniques. The mechanism is based on converting Android process memory dump files into grayscale images. The memory dump files are read byte by byte, and each byte is converted into equal binary representations. These binary representations are used to form grayscale images. These grayscale images derive features such as colour histograms, Hu moments, and Haralick textures; these derived features per image train machine learning classifiers to predict an Android application as benign and malicious.

Chapter Six:
CONCLUSIONS AND FUTURE SCOPE

Smartphones have surpassed desktop systems in popularity due to their wide array of feature-rich applications. They are now an integral part of daily life. They offer access to numerous services such as online shopping, gaming, and location-based services, making them more powerful than early personal computers. However, as smartphones have gained widespread use, particularly Android devices, there has been a significant increase in malware attacks targeting these platforms. Malicious applications, or malware, can infiltrate smartphones through various channels including SMS, MMS, Bluetooth, internet downloads, and both official and third-party app stores. These attacks pose serious risks, including system damage, financial loss, and data breaches. As a result, the issue of detecting Android malware has attracted significant attention from researchers in recent years, given the rising frequency of attacks on the Android platform. This thesis focuses on addressing the challenge of malware detection on Android smartphones.

This chapter presents the conclusions of the dissertation. It begins by summarizing the key contributions made throughout the thesis, reviewing the proposed models for Android malware detection and evaluating how they fulfil the established objectives. Following this, we highlight several unresolved challenges in the existing literature, discussing areas that require further research and attention in future work.

6.1 Conclusions

1. In line with the objective, we successfully developed a novel Android malware detection technique utilizing static features extracted from both Android manifest files and .dex files. The proposed model is based on rough set theory, where we combined four static features: permissions, opcodes, API calls, and system commands. The development process began with a comprehensive data pre-processing phase, where we eliminated correlated features and those that had no relevance to the class variable. This ensured that the model focused only on the most impactful features, improving overall efficiency and accuracy. A

key innovation in the work was the application of Discernibility Matrices from rough set theory to assign a ranking score to each feature, reflecting its significance in malware detection. The rough set reducts algorithm was then employed to further reduce the number of features based on the ranking score, ensuring a more streamlined and focused analysis. After the feature reduction step, various machine learning algorithms were applied to evaluate the model's detection accuracy. The experimental results demonstrated that the proposed approach outperformed other advanced malware detection models. The superior performance, as evidenced by the comparative analysis, validates the effectiveness of the proposed technique in detecting Android malware, meeting the initial objective of designing an efficient and novel static feature-based detection model.

2. In accordance with the objective, we successfully developed a hybrid malware analysis technique that optimally combines static and dynamic features for Android devices. This study specifically explored the interaction between permissions (a static feature) and system calls (a dynamic feature), establishing a novel methodology for analyzing their combined effect on malware detection. We introduced the concept of a Covalent Bond Strength Score, which quantifies the strength of the relationship between these feature pairs. By calculating both malicious and benign scores, we were able to effectively assess how these features interact in different contexts, contributing to the accuracy of malware detection. The proposed model harnesses the combined power of static and dynamic features, ensuring a comprehensive analysis that addresses the limitations of relying on only one type of feature. By integrating the Covalent Bond Strength Score, the model was able to differentiate between benign and malicious behaviours with greater precision. A detailed comparative analysis was conducted between our hybrid model and other advanced malware detection techniques. The results of this comparison demonstrated the superior performance of our model, surpassing similar state-of-the-art approaches in terms of detection accuracy and overall effectiveness. This validation highlights the success of the proposed hybrid analysis

technique in meeting the objective of optimally combining static and dynamic features to enhance malware detection on Android devices.

3. In line to develop a memory forensics-based technique for malware detection, we introduced an innovative Android malware detection mechanism that leverages visual analysis techniques. Our approach focuses on analyzing memory dump files from Android processes, converting them into grayscale images for further examination. The memory dump files were processed byte by byte, with each byte being converted into its binary representation. These binary representations were then used to form grayscale images, enabling a visual interpretation of the memory data. Key features were extracted from these grayscale images, including colour histograms, Hu moments, and Haralick textures, which capture important visual patterns that differentiate between benign and malicious applications. These features were subsequently used to train machine learning classifiers, which then classified Android applications as either benign or malicious based on the patterns present in the memory dumps. The proposed memory forensics-based technique successfully incorporates visual methods into malware detection, offering a novel approach that effectively combines memory analysis with machine learning. Our experimental results demonstrate the capability of this method to accurately classify applications, meeting the objective of utilizing memory forensics for Android malware detection. The visual analysis approach also opens new avenues for further exploration in the field of memory-based malware detection, offering a unique and powerful tool for classifying Android applications

6.2 Future Work

The emergence of increasingly sophisticated and stealthy Android malware continues to create significant challenges for the research community, with several critical areas still requiring further investigation. This section outlines some of the key unresolved issues that demand additional research.

1. The proposed models are an off-device model, and hence they can not be installed on smartphones for real-time detection. In future, we will propose a client‑server-based model to integrate in smartphones.
2. In the future, our research will analyze additional components of the manifest file, such as intent filters and hardware specifications, to further enhance detection accuracy.
3. We aim to build more robust techniques by considering other types of features that can be derived from the grayscale image of Android process memory dump files, which will strengthen the machine learning classifiers. We will also use deep-learning models to know their effectiveness in classifying malware using visual techniques.

REFERENCES

[1] H. Liu, D. Pardoe, K. Liu, M. Thakur, F. Cao and C. Li, "Audience expansion for online social network advertising," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[2] K. Patel and B. Buddadev, "Detection and mitigation of android malware through hybrid approach," in *Security in Computing and Communications: Third International Symposium, SSCC 2015, Kochi, India, August 10-13, 2015. Proceedings 3*, 2015.

[3] B. D. Prasad, "RTI-TRAPS: An Adaptive Vehicle Tracking Methodology for Public Transportation," *International Journal of Computer Applications*, vol. 975, p. 8887, 2013.

[4] R. Trestian, P. Shah, H. Nguyen, Q.-T. Vien, O. Gemikonakli and B. Barn, "Towards connecting people, locations and real-world events in a cellular network," *Telematics and Informatics*, vol. 34, p. 244–271, 2017.

[5] A. Kaushik and D. P. Vidyarthi, "A cooperative cell model in computational mobile grid," *International Journal of Business Data Communications and Networking (IJBDCN)*, vol. 8, p. 19–36, 2012.

[6] S. K. Singh and D. P. Vidyarthi, "A heuristic channel allocation model with multi lending in mobile computing network," *International Journal of Wireless and Mobile Computing*, vol. 16, p. 322–339, 2019.

[7] A. Omorinoye, Q.-T. Vien, T. A. Le and P. Shah, "On the resource allocation for D2D underlaying uplink cellular networks," in *2019 26th International Conference on Telecommunications (ICT)*, 2019.

[8] V. V. Paranthaman, Y. Kirsal, G. Mapp, P. Shah and H. X. Nguyen, "Exploring a new proactive algorithm for resource management and its application to wireless mobile environments," in *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*, 2017.

[9] V. S. Pendyala and J. Holliday, “Performing intelligent mobile searches in the cloud using semantic technologies,” in *2010 IEEE International Conference on Granular Computing*, 2010.

[10] M. Malik and D. P. Agrawal, “Secure web framework for mobile devices,” in *2012 IEEE Globecom Workshops*, 2012.

[11] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti and M. Rajarajan, “Android security: a survey of issues, malware penetration, and defenses,” *IEEE communications surveys & tutorials*, vol. 17, p. 998–1022, 2014.

[12] A. P. Felt, M. Finifter, E. Chin, S. Hanna and D. Wagner, “A survey of mobile malware in the wild,” in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 2011.

[13] R. Surendran, T. Thomas and S. Emmanuel, “Detection of malware applications in android smartphones,” in *WORLD SCIENTIFIC REFERENCE ON INNOVATION: Volume 4: Innovation in Information Security*, World Scientific, 2018, p. 211–234.

[14] E. Chin, A. P. Felt, V. Sekar and D. Wagner, “Measuring user confidence in smartphone security and privacy,” in *Proceedings of the eighth symposium on usable privacy and security*, 2012.

[15] D. Puthal, S. P. Mohanty, P. Nanda and U. Choppali, “Building security perimeters to protect network systems against cyber threats [future directions],” *IEEE Consumer Electronics Magazine,* vol. 6, p. 24–27, 2017.

[16] Z. Cheng, “Mobile malware: Threats and prevention,” *McAfee Avert,* 2007.

[17] Y. Zhou and X. Jiang, “Dissecting android malware: Characterization and evolution,” in *2012 IEEE symposium on security and privacy*, 2012.

[18] Q. Huang, V. K. Singh and P. K. Atrey, “On cyberbullying incidents and underlying online social relationships,” *Journal of Computational Social Science,* vol. 1, p. 241–260, 2018.

[19] V. K. Singh, Q. Huang and P. K. Atrey, “Cyberbullying detection using probabilistic socio-textual information fusion,” in *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2016.

[20] N. Vishwamitra, X. Zhang, J. Tong, H. Hu, F. Luo, R. Kowalski and J. Mazer, “MCDefender: Toward effective cyberbullying defense in mobile online social networks,” in *Proceedings of the 3rd ACM on International Workshop on Security and Privacy Analytics*, 2017.

[21] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han and X. Zhang, “Exploring permission-induced risk in android applications for malicious application detection,” *IEEE Transactions on Information Forensics and Security,* vol. 9, p. 1869–1882, 2014.

[22] M. Alazab, M. Alazab, A. Shalaginov, A. Mesleh and A. Awajan, “Intelligent mobile malware detection using permission requests and API calls,” *Future Generation Computer Systems,* vol. 107, p. 509–521, 2020.

[23] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-An and H. Ye, “Significant permission identification for machine-learning-based android malware detection,” *IEEE Transactions on Industrial Informatics,* vol. 14, p. 3216–3225, 2018.

[24] N. Milosevic, A. Dehghantanha and K.-K. R. Choo, “Machine learning aided Android malware classification,” *Computers & Electrical Engineering,* vol. 61, p. 266–274, 2017.

[25] A. K. Singh, C. D. Jaidhar and M. A. Kumara, “Experimental analysis of Android malware detection based on combinations of permissions and API-calls,” *Journal of Computer Virology and Hacking Techniques,* vol. 15, p. 209–218, 2019.

[26] C. Wang, Q. Xu, X. Lin and S. Liu, “Research on data mining of permissions mode for Android malware detection,” *Cluster Computing,* vol. 22, p. 13337–13350, 2019.

[27] A. Case, R. D. Maggio, M. Firoz-Ul-Amin, M. M. Jalalzai, A. Ali-Gombe, M. Sun and G. G. Richard III, “Hooktracer: Automatic detection and analysis of keystroke loggers using memory forensics,” *Computers & Security,* vol. 96, p. 101872, 2020.

[28] P. Fernández-Álvarez and R. J. Rodríguez, “Module extraction and DLL hijacking detection via single or multiple memory dumps,” *Forensic Science International: Digital Investigation,* vol. 44, p. 301505, 2023.

[29] J. Liu, Y. Feng, X. Liu, J. Zhao and Q. Liu, “MRm-DLDet: a memory-resident malware detection framework based on memory forensics and deep neural network,” *Cybersecurity,* vol. 6, p. 21, 2023.

[30] I. Kara, “Fileless malware threats: Recent advances, analysis approach through memory forensics and research challenges,” *Expert Systems with Applications,* vol. 214, p. 119133, 2023.

[31] P. O'Kane, S. Sezer and K. McLaughlin, “Obfuscation: The hidden malware,” *IEEE Security & Privacy,* vol. 9, p. 41–47, 2011.

[32] O. Or-Meir, N. Nissim, Y. Elovici and L. Rokach, “Dynamic malware analysis in the modern era—A state of the art survey,” *ACM Computing Surveys (CSUR),* vol. 52, p. 1–48, 2019.

[33] M. C. Grace, W. Zhou, X. Jiang and A.-R. Sadeghi, “Unsafe exposure analysis of mobile in-app advertisements,” in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[34] W. Enck, M. Ongtang and P. McDaniel, “On lightweight mobile phone application certification,” in *Proceedings of the 16th ACM conference on Computer and communications security,* 2009.

[35] K. A. Talha, D. I. Alper and C. Aydin, “APK Auditor: Permission-based Android malware detection system,” *Digital Investigation,* vol. 13, p. 1–14, 2015.

[36] J. Choi, W. Sung, C. Choi and P. Kim, “Personal information leakage detection method using the inference-based access control model on the Android platform,” *Pervasive and Mobile Computing,* vol. 24, p. 138–149, 2015.

[37] J. Song, C. Han, K. Wang, J. Zhao, R. Ranjan and L. Wang, “An integrated static detection and analysis framework for android,” *Pervasive and Mobile Computing,* vol. 32, p. 15–25, 2016.

[38] Z. Wang, C. Li, Z. Yuan, Y. Guan and Y. Xue, "DroidChain: A novel Android malware detection method based on behavior chains," *Pervasive and Mobile Computing,* vol. 32, p. 3–14, 2016.

[39] S. K. Sasidharan and C. Thomas, "ProDroid—An Android malware detection framework based on profile hidden Markov model," *Pervasive and Mobile Computing,* vol. 72, p. 101336, 2021.

[40] V. Moonsamy, J. Rong and S. Liu, "Mining permission patterns for contrasting clean and malicious android applications," *Future Generation Computer Systems,* vol. 36, p. 122–132, 2014.

[41] F. Idrees and M. Rajarajan, "Investigating the android intents and permissions for malware detection," in *2014 IEEE 10th international conference on wireless and mobile computing, networking and communications (WiMob),* 2014.

[42] Y. Zhou, Z. Wang, W. Zhou and X. Jiang, "Hey, you, get off of my market: detecting malicious apps in official and alternative android markets.," in *NDSS*, 2012.

[43] J. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, Y. Wang and Y. Xiang, "A3CM: automatic capability annotation for android malware," *IEEE Access,* vol. 7, p. 147156–147168, 2019.

[44] R. Taheri, M. Ghahramani, R. Javidan, M. Shojafar, Z. Pooranian and M. Conti, "Similarity-based Android malware detection using Hamming distance of static binary features," *Future Generation Computer Systems,* vol. 105, p. 230–247, 2020.

[45] H. Bai, N. Xie, X. Di and Q. Ye, "Famd: A fast multifeature android malware detection framework, design, and implementation," *IEEE Access,* vol. 8, p. 194729–194740, 2020.

[46] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon and K. R. Drebin, "Effective and explainable detection of android malware in your pocket," in *Network and distributed system security symposium*.

[47] M. V. Varsha, P. Vinod and K. A. Dhanya, "Identification of malicious android app using manifest and opcode features," *Journal of Computer Virology and Hacking Techniques,* vol. 13, p. 125–138, 2017.

[48] A. Mahindru and A. L. Sangal, “FSDroid:-A feature selection technique to detect malware from Android using Machine Learning Techniques: FSDroid,” *Multimedia Tools and Applications,* vol. 80, p. 13271–13323, 2021.

[49] K. Khariwal, J. Singh and A. Arora, “IPDroid: Android malware detection using intents and permissions,” in *2020 Fourth world conference on smart trends in systems, security and sustainability (WorldS4),* 2020.

[50] A. Arora, S. K. Peddoju and M. Conti, “Permpair: Android malware detection using permission pairs,” *IEEE Transactions on Information Forensics and Security,* vol. 15, p. 1968–1982, 2019.

[51] M. Kumaran and W. Li, “Lightweight malware detection based on machine learning algorithms and the android manifest file,” in *2016 IEEE MIT Undergraduate Research Technology Conference (URTC),* 2016.

[52] S. Feldman, D. Stadther and B. Wang, “Manilyzer: automated android malware detection through manifest analysis,” in *2014 IEEE 11th International Conference on Mobile Ad Hoc and Sensor Systems,* 2014.

[53] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, J. Nieves, P. G. Bringas and G. Álvarez Marañón, “MAMA: manifest analysis for malware detection in android,” *Cybernetics and Systems,* vol. 44, p. 469–488, 2013.

[54] C. Li, K. Mills, D. Niu, R. Zhu, H. Zhang and H. Kinawi, “Android malware detection based on factorization machine,” *IEEE Access,* vol. 7, p. 184008–184019, 2019.

[55] R. Sato, D. Chiba and S. Goto, “Detecting android malware by analyzing manifest files,” *Proceedings of the Asia-Pacific Advanced Network,* vol. 36, p. 17, 2013.

[56] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee and K.-P. Wu, “Droidmat: Android malware detection through manifest and api calls tracing,” in *2012 Seventh Asia joint conference on information security,* 2012.

[57] K. O. Elish, X. Shu, D. D. Yao, B. G. Ryder and X. Jiang, “Profiling user-trigger dependence for Android malware detection,” *Computers & Security,* vol. 49, p. 255–273, 2015.

[58] M. Zhang, Y. Duan, H. Yin and Z. Zhao, “Semantics-aware android malware classification using weighted contextual api dependency graphs,” in *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, 2014.

[59] Y. Feng, S. Anand, I. Dillig and A. 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*, 2014.

[60] W. Wang, Z. Gao, M. Zhao, Y. Li, J. Liu and X. Zhang, “DroidEnsemble: Detecting Android malicious applications with ensemble of string and structural static features,” *IEEE Access,* vol. 6, p. 31798–31807, 2018.

[61] H. Zhang, S. Luo, Y. Zhang and L. Pan, “An efficient Android malware detection system based on method-level behavioral semantic analysis,” *IEEE Access,* vol. 7, p. 69246–69256, 2019.

[62] H. Zhu, H. Wei, L. Wang, Z. Xu and V. S. Sheng, “An effective end-to-end android malware detection method,” *Expert Systems with Applications,* vol. 218, p. 119593, 2023.

[63] Y. Fang, Y. Gao, F. A. N. Jing and L. E. I. Zhang, “Android malware familial classification based on dex file section features,” *IEEE Access,* vol. 8, p. 10614–10627, 2020.

[64] Y.-S. Yen and H.-M. Sun, “An Android mutation malware detection based on deep learning using visualization of importance from codes,” *Microelectronics Reliability,* vol. 93, p. 109–114, 2019.

[65] Z. Xu, K. Ren, S. Qin and F. Craciun, “CDGDroid: Android malware detection based on deep learning using CFG and DFG,” in *Formal Methods and Software Engineering: 20th International Conference on Formal Engineering Methods, ICFEM 2018, Gold Coast, QLD, Australia, November 12-16, 2018, Proceedings 20*, 2018.

[66] X. Xiao, S. Zhang, F. Mercaldo, G. Hu and A. K. Sangaiah, “Android malware detection based on system call sequences and LSTM,” *Multimedia Tools and Applications,* vol. 78, p. 3979–3999, 2019.

[67] X. Qin, F. Zeng and Y. Zhang, “MSNdroid: the Android malware detector based on multi-class features and deep belief network,” in *Proceedings of the ACM Turing Celebration Conference-China*, 2019.

[68] M. Ahmadi, A. Sami, H. Rahimi and B. Yadegari, “Malware detection by behavioural sequential patterns,” *Computer Fraud & Security*, vol. 2013, p. 11–19, 2013.

[69] M. Sun, T. Wei and J. C. S. Lui, “Taintart: A practical multi-level information-flow tracking system for android runtime,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[70] L. K. Yan and H. Yin, “{DroidScope}: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis,” in *21st USENIX security symposium (USENIX security 12)*, 2012.

[71] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning and X. S. Wang, “Appintent: Analyzing sensitive data transmission in android for privacy leakage detection,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[72] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer and Y. Weiss, ““Andromaly”: a behavioral malware detection framework for android devices,” *Journal of Intelligent Information Systems,* vol. 38, p. 161–190, 2012.

[73] A. Reina, A. Fattori and L. Cavallaro, “A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors,” *EuroSec, April*, 2013.

[74] V. M. Afonso, M. F. de Amorim, A. R. A. Grégio, G. B. Junquera and P. L. de Geus, “Identifying Android malware using dynamically obtained features,” *Journal of Computer Virology and Hacking Techniques,* vol. 11, p. 9–17, 2015.

[75] I. Burguera, U. Zurutuza and S. 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*, 2011.

[76] M. Zheng, M. Sun and J. C. S. Lui, “DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability,” in *2014 international wireless communications and mobile computing conference (IWCMC)*, 2014.

[77] M. Almeida, M. Bilal, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Varvello and J. Blackburn, “Chimp: Crowdsourcing human inputs for mobile phones,” in *Proceedings of the 2018 World Wide Web Conference*, 2018.

[78] J.-w. Jang, H. Kang, J. Woo, A. Mohaisen and H. K. Kim, “Andro-Dumpsys: Anti-malware system based on the similarity of malware creator and malware centric information,” *computers & security*, vol. 58, p. 125–138, 2016.

[79] A. Arora, S. Garg and S. K. Peddoju, “Malware detection using network traffic analysis in android based mobile devices,” in *2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies*, 2014.

[80] Z. Chen, H. Han, Q. Yan, B. Yang, L. Peng, L. Zhang and J. Li, “A first look at android malware traffic in first few minutes,” in *2015 IEEE Trustcom/BigDataSE/ISPA*, 2015.

[81] M. Aresu, D. Ariu, M. Ahmadi, D. Maiorca and G. Giacinto, “Clustering android malware families by http traffic,” in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, 2015.

[82] S. Wang, Z. Chen, X. Li, L. Wang, K. Ji and C. Zhao, “Android malware clustering analysis on network-level behavior,” in *Intelligent Computing Theories and Application: 13th International Conference, ICIC 2017, Liverpool, UK, August 7-10, 2017, Proceedings, Part I 13*, 2017.

[83] H. F. Alan and J. Kaur, “Can Android applications be identified using only TCP/IP headers of their launch time traffic?,” in *Proceedings of the 9th ACM conference on security & privacy in wireless and mobile networks*, 2016.

[84] M. Conti, L. V. Mancini, R. Spolaor and N. V. Verde, “Analyzing android encrypted network traffic to identify user actions,” *IEEE Transactions on Information Forensics and Security*, vol. 11, p. 114–125, 2015.

[85] S. Wang, Q. Yan, Z. Chen, B. Yang, C. Zhao and M. Conti, “Detecting android malware leveraging text semantics of network flows,” *IEEE Transactions on Information Forensics and Security*, vol. 13, p. 1096–1109, 2017.

[86] A. Shabtai, L. Tenenboim-Chekina, D. Mimran, L. Rokach, B. Shapira and Y. Elovici, "Mobile malware detection through analysis of deviations in application network behavior," *Computers & Security,* vol. 43, p. 1–18, 2014.

[87] S. Wang, Z. Chen, L. Zhang, Q. Yan, B. Yang, L. Peng and Z. Jia, "Trafficav: An effective and explainable detection of mobile malware behavior using network traffic," in *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*, 2016.

[88] S. Wang, Z. Chen, Q. Yan, B. Yang, L. Peng and Z. Jia, "A mobile malware detection method using behavior features in network traffic," *Journal of Network and Computer Applications,* vol. 133, p. 15–25, 2019.

[89] Y. Pang, Z. Chen, X. Li, S. Wang, C. Zhao, L. Wang, K. Ji and Z. Li, "Finding Android malware trace from highly imbalanced network traffic," in *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, 2017.

[90] P. Borges, B. Sousa, L. Ferreira, F. B. Saghezchi, G. Mantas, J. Ribeiro, J. Rodriguez, L. Cordeiro and P. Simoes, "Towards a hybrid intrusion detection system for android-based PPDR terminals," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2017.

[91] R. Chen, Y. Li and W. Fang, "Android malware identification based on traffic analysis," in *International conference on artificial intelligence and security*, 2019.

[92] A. Feizollah, N. B. Anuar, R. Salleh and F. Amalina, "Comparative study of k-means and mini batch k-means clustering algorithms in android malware detection using network traffic analysis," in *2014 international symposium on biometrics and security technologies (ISBAST)*, 2014.

[93] J. Li, L. Zhai, X. Zhang and D. Quan, "Research of android malware detection based on network traffic monitoring," in *2014 9th IEEE Conference on Industrial Electronics and Applications*, 2014.

[94] A. Liu, Z. Chen, S. Wang, L. Peng, C. Zhao and Y. Shi, "A fast and effective detection of mobile malware behavior using network traffic," in *International Conference on Algorithms and Architectures for Parallel Processing*, 2018.

[95] A. Saracino, D. Sgandurra, G. Dini and F. Martinelli, “Madam: Effective and efficient behavior-based android malware detection and prevention,” *IEEE Transactions on Dependable and Secure Computing,* vol. 15, p. 83–97, 2016.

[96] Q. Han, V. S. Subrahmanian and Y. Xiong, “Android malware detection via (somewhat) robust irreversible feature transformations,” *IEEE Transactions on Information Forensics and Security,* vol. 15, p. 3511–3525, 2020.

[97] M. Sun, X. Li, J. C. S. Lui, R. T. B. Ma and Z. Liang, “Monet: a user-oriented behavior-based malware variants detection system for android,” *IEEE Transactions on Information Forensics and Security,* vol. 12, p. 1103–1112, 2016.

[98] M. Xia, L. Gong, Y. Lyu, Z. Qi and X. Liu, “Effective real-time android application auditing,” in *2015 IEEE Symposium on Security and Privacy*, 2015.

[99] M. Grace, Y. Zhou, Q. Zhang, S. Zou and X. Jiang, “Riskranker: scalable and accurate zero-day android malware detection,” in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, 2012.

[100] M. Lindorfer, M. Neugschwandtner and C. Platzer, “Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis,” in *2015 IEEE 39th annual computer software and applications conference*, 2015.

[101] S. Arshad, M. A. Shah, A. Wahid, A. Mehmood, H. Song and H. Yu, “SAMADroid: a novel 3-level hybrid malware detection model for android operating system,” *IEEE Access*, vol. 6, p. 4321–4339, 2018.

[102] X. Wang, Y. Yang, Y. Zeng, C. Tang, J. Shi and K. Xu, “A novel hybrid mobile malware detection system integrating anomaly detection with misuse detection,” in *Proceedings of the 6th International Workshop on Mobile Cloud Computing and Services*, 2015.

[103] Z. Yuan, Y. Lu and Y. Xue, “Droiddetector: android malware characterization and detection using deep learning,” *Tsinghua Science and Technology,* vol. 21, p. 114–123, 2016.

[104] Y. Liu, Y. Zhang, H. Li and X. Chen, “A hybrid malware detecting scheme for mobile Android applications,” in *2016 IEEE International Conference on Consumer Electronics (ICCE),* 2016.

[105] H.-Y. Chuang and S.-D. Wang, “Machine learning based hybrid behavior models for Android malware analysis,” in *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015.

[106] T. Chakraborty, F. Pierazzi and V. S. Subrahmanian, “Ec2: Ensemble clustering and classification for predicting android malware families,” *IEEE Transactions on Dependable and Secure Computing*, vol. 17, p. 262–277, 2017.

[107] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh and L. Cavallaro, “The evolution of android malware and android analysis techniques,” *ACM Computing Surveys (CSUR)*, vol. 49, p. 1–41, 2017.

[108] A. Martín, R. Lara-Cabrera and D. Camacho, “Android malware detection through hybrid features fusion and ensemble classifiers: The AndroPyTool framework and the OmniDroid dataset,” *Information Fusion*, vol. 52, p. 128–142, 2019.

[109] K. Allix, T. F. Bissyandé, J. Klein and Y. Le Traon, “Androzoo: Collecting millions of android apps for the research community,” in *Proceedings of the 13th international conference on mining software repositories*, 2016.

[110] A. Martín, R. Lara-Cabrera and D. Camacho, “A new tool for static and dynamic Android malware analysis,” in *Data Science and Knowledge Engineering for Sensing Decision Support: Proceedings of the 13th International FLINS Conference (FLINS 2018)*, 2018.

[111] Z. Pawlak, “Rough set theory and its applications to data analysis,” *Cybernetics & Systems*, vol. 29, p. 661–688, 1998.

[112] Z. Han, Q. Zhang and F. Wen, “A survey on rough set theory and its application,” *Control theory and applications*, vol. 16, p. 153–157, 1999.

[113] D. Wagner and R. Dean, “Intrusion detection via static analysis,” in *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, 2000.

[114] B. B. Kang and A. Srivastava, *Dynamic Malware Analysis.*, 2011.

[115] G. Fraser and A. Arcuri, “Automated test generation for java generics,” in *Software Quality. Model-Based Approaches for Advanced Software and Systems Engineering:*

6th International Conference, SWQD 2014, Vienna, Austria, January 14-16, 2014. Proceedings 6, 2014.

[116] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software.," in *NDSS*, 2005.

[117] R. Zhang, S. Huang, Z. Qi and H. Guan, "Combining static and dynamic analysis to discover software vulnerabilities," in *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 2011.

[118] R. Zhang, S. Huang, Z. Qi and H. Guan, "Static program analysis assisted dynamic taint tracking for software vulnerability discovery," *Computers & Mathematics with Applications,* vol. 63, p. 469–480, 2012.

[119] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS),* vol. 32, p. 1–29, 2014.

[120] A. Guerra-Manzanares, H. Bahsi and S. Nõmm, "Kronodroid: time-based hybrid-featured dataset for effective android malware detection and characterization," *Computers & Security,* vol. 110, p. 102399, 2021.

[121] A. H. Lashkari, A. F. A. Kadir, L. Taheri and A. A. Ghorbani, "Toward developing a systematic approach to generate benchmark android malware datasets and classification," in *2018 International Carnahan conference on security technology (ICCST),* 2018.

[122] A. Guerra-Manzanares, M. Luckner and H. Bahsi, "Concept drift and cross-device behavior: Challenges and implications for effective android malware detection," *Computers & Security,* vol. 120, p. 102757, 2022.

[123] A. Guerra-Manzanares, H. Bahsi and M. Luckner, "Leveraging the first line of defense: A study on the evolution and usage of android security permissions for enhanced android malware detection," *Journal of Computer Virology and Hacking Techniques,* vol. 19, p. 65–96, 2023.

[124] A. Guerra-Manzanares, M. Luckner and H. Bahsi, “Android malware concept drift using system calls: detection, characterization and challenges,” *Expert Systems with Applications,* vol. 206, p. 117200, 2022.

[125] H. M. Ünver and K. Bakour, “Android malware detection based on image-based features and machine learning techniques,” *SN Applied Sciences,* vol. 2, p. 1299, 2020.

[126] P. Yadav, N. Menon, V. Ravi, S. Vishvanathan and T. D. Pham, “EfficientNet convolutional neural networks-based Android malware detection,” *Computers & Security,* vol. 115, p. 102622, 2022.

[127] J. Singh, D. Thakur, F. Ali, T. Gera and K. S. Kwak, “Deep feature extraction and classification of android malware images,” *Sensors,* vol. 20, p. 7013, 2020.

[128] Y. Ding, X. Zhang, J. Hu and W. Xu, “Android malware detection method based on bytecode image,” *Journal of Ambient Intelligence and Humanized Computing,* vol. 14, p. 6401–6410, 2023.

[129] I. Homem and P. Papapetrou, “Android Process Memory String Dumps Dataset,” *Stockholm University Dataset,* 2017.