الجمهورية الجزائرية الديمقراطية الشعبية
Peoples Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University May 8, 1945 -Guelma -

Faculty of Mathematics, Computer Science and Material Sciences
Computer Science Department

Masters Degree Thesis
Branch: Computer Science
Option: Informatics Systems

Theme:

Advanced Android Malware Detection: Leveraging Machine Learning for Zero-Day Threat Defense

Presented by:
Yehya Djaghout

Jury Members:

Title	Full Name	Quality
Dr.	Djalila Boughareb	Chairperson
Dr.	Abdelhakim Hannousse	Supervisor
Ms.	Soumia Felkaoui	Examiner

June 2024

To my dear parents, as a testament to my profound gratitude and unwavering appreciation for all the sacrifices they make for me, the trust they place in me, and the boundless love they surround me with.
To my dear brother, for whom I can never find enough words to express my love. I wish you all the success in the world.
To my friends and everyone who supported me throughout this journey. Your encouragement, understanding, and companionship have been invaluable. Thank you for being there for me and for your unwavering support.

Acknowledgements

— First and foremost, I express my deepest gratitude to Allah, who granted me the strength, patience, determination, and, most importantly, the health needed to carry out this work. Without these blessings, the completion of this research would not have been possible.

— I would like to express my deep gratitude to my supervisor Dr. Hannousse Abdelhakim, for his patient guidance, enthusiastic encouragement and useful critiques of this research work.

— I address my thanks to our head of the department Dr. Zineddine Kouahla and all my teachers who contributed to our training. I would also like to thank the members of the jury.

— Finally, I wish to express my heartfelt gratitude to my parents for their unwavering support and encouragement throughout my studies. Their continuous belief in my abilities and their constant encouragement have been a source of immense strength and motivation. Without their love, understanding, and sacrifices, this journey would not have been possible.

" يَرْفَعِ اللَّهُ الَّذِينَ آمَنُوا مِنكُمْ وَالَّذِينَ أُوتُوا الْعِلْمَ دَرَجَاتٍ "

سورة المجادلة - الآية 11 –

ملخص

نظرًا لانتشار استخدام أجهزة الأندرويد على نطاق واسع، أصبحت هدفًا رئيسيًا للبرامج الضارة، مما يؤكد على الحاجة الملحة لآليات كشف فعّالة لحماية المستخدمين وبياناتهم. تقدم هذه الرسالة نهجاً مبتكراً لتحليل الثابت المختلط لكشف برامج الأندرويد الضارة، يستفيد من تقنيات تعلم الآلة، وبالتحديد تقنية التعلم المتكامل، مما يدمج تحليل الإذنات، وفحص الترميز المعلوماتي، وتصور بايت كود، لاستثمار قوة كل منها. يهدف نهجنا الشامل إلى تعزيز دقة الكشف وقدرته على التكيف، لمواجهة بشكل فعال تكتيكات تطوير برامج الضارة المتطورة و إتاحة آلية فعّالة للدفاع ضد تهديدات يوم الصفر. أجريت تجارب واسعة على مجموعتي بيانات تحتوي على عينات لبرامج ضارة من فترات زمنية مختلفة، حيث حققت الطريقة التحليلية الخاصة بنا دقة ملحوظة بنسبة ٩٩.٢٨٪ على مجموعة البيانات التي تتضمن عينات برامج ضارة قديمة، مع الأداء البارز على مجموعة البيانات التي تحتوي على عينات برامج ضارة حديثة بنسبة ٩٦.٠٦٪. تؤكد هذه النتائج فعالية نموذجنا المتكامل في توفير دفاع قوي ضد مجموعة واسعة من سلوكيات البرامج الضارة، مما يسهم بشكل كبير في مجال أمن المعلومات من خلال تقديم حلاً متقدماً ومرناً لكشف برامج الأندرويد الضارة، وتقديم تطبيقات عملية لتعزيز أمان أجهزة الموبايل في الواقع العملي.

الكلمات المفتاحية: كشف برامج الأندرويد الضارة، تحليل ثابت، نهج مختلط، تعلم الآلة.

Résumé

La popularité croissante des appareils Android en fait une cible principale pour les logiciels malveillants, soulignant le besoin critique de mécanismes efficaces de détection pour protéger les utilisateurs et leurs données. Cette thèse présente une approche innovante d'analyse statique mixte qui exploite les techniques d'apprentissage automatique, en particulier l'apprentissage d'ensemble, pour la détection des malwares Android. Cette approche intègre l'analyse des permissions, l'examen des opcodes et la visualisation du bytecode, en capitalisant sur les forces de chaque méthode. Notre approche vise à améliorer la précision de la détection et son adaptabilité, contrecarrant efficacement les tactiques évolutives des développeurs de logiciels malveillants, et fournirnissant un mécanisme de défense efficace contre les menaces Zero-Day. Des expériences approfondies menées sur deux ensembles de données contenant des échantillons de malwares de différentes périodes démontrent la performance supérieure de notre méthode. Nous avons atteint une précision remarquable de 99,82% sur l'ensemble de données comprenant des échantillons de malwares anciens, mettant en valeur la robustesse de notre modèle face aux menaces historiques. Pour l'ensemble de données contenant des échantillons de malwares récents, notre approche a atteint une précision élevée de 96,06%, surpassant significativement les autres méthodes qui ont montré des baisses notables de performance avec les malwares plus récents. Ces résultats soulignent l'efficacité de notre modèle intégré pour fournir une défense robuste contre une large gamme de comportements malveillants. Cette recherche contribue de manière significative à la cybersécurité en proposant une solution avancée et flexible pour la détection des malwares Android, offrant des implications pratiques pour renforcer la sécurité des appareils mobiles dans des applications réelles.

Mots-clés : détection de malwares Android ; analyse statique ; approche mixte ; apprentissage machine.

Abstract

The widespread use of Android devices has made them a prime target for malware, highlighting the critical need for effective detection mechanisms to protect users and their data. This thesis introduces an innovative mixed static analysis approach that leverages machine learning technique, specifically, ensemble learning, for Android malware detection, which integrates permission analysis, opcode examination, and bytecode visualization, capitalizing on the strengths of each method. Our comprehensive approach aims to enhance detection accuracy and adaptability, effectively countering the evolving tactics of malware developers, and providing an effective Zero-Day threat defense mechanism. Extensive experiments conducted on two datasets containing malware samples from different time periods demonstrate the superior performance of our method. We achieved a remarkable accuracy of 99.82% on the dataset comprising older malware samples, showcasing our model's robustness in handling historical threats. For the dataset containing recent malware samples, our approach achieved a high accuracy of 96.06%, significantly outperforming other methods which exhibited notable decreases in performance with newer malware. These findings underscore the effectiveness of our integrated model in providing a robust defense against a wide range of malware behaviors. This research contributes significantly to cybersecurity by proposing an advanced and flexible solution for Android malware detection, offering practical implications for enhancing mobile device security in real-world applications.

Keywords: android malware detection; static analysis; mixed approach; machine learning.

Table of Contents

List of Figures

List of Tables

Introduction

The ubiquitous adoption of mobile devices, particularly Android-powered smartphones, has revolutionized various aspects of modern life [51]. However, this rapid proliferation has concurrently introduced critical cybersecurity challenges. The inherent convenience and accessibility afforded by these mobile technologies have inadvertently created opportunities for malicious actors to exploit vulnerabilities. Given the predominance of Android devices in the global market and their role in managing sensitive personal and financial data, the imperative to secure these platforms cannot be overemphasized. Addressing the security threats posed to Android ecosystems is of paramount importance to safeguard user privacy and mitigate potential financial losses or data breaches.

Historically, Android malware has evolved in complexity and prevalence, posing substantial threats to users and organizations alike. This evolution has been marked by increasingly sophisticated techniques used by attackers to bypass security measures and infiltrate devices. Notable malware attacks, such as the notorious *DroidDream* and the sophisticated *HummingBad*, have highlighted the potential damages, including unauthorized data access, financial loss, and significant privacy breaches. *DroidDream* [30], for instance, infected over 50 applications on the Android Market, leading to widespread data theft and unauthorized control over infected devices. *HummingBad* [14, 31], on the other hand, established a persistent rootkit on devices, enabling the attackers to generate fraudulent ad revenue and exfiltrate sensitive information. These incidents underscore the urgent need for robust detection and prevention mechanisms to safeguard users from ever-evolving malware threats. As malware authors continue to refine their strategies, the challenge for security professionals becomes increasingly complex, necessitating advanced and adaptive solutions to protect the vast and diverse Android ecosystem.

Over the years, various approaches have been developed to tackle Android malware, each with its strengths and limitations [19]. Static methods such as permission-based analysis, which scrutinizes the permissions requested by apps, and opcode-based detection, focusing on the operational codes used in app execution, have been widely utilized. Additionally, visualization techniques that transform code into visual patterns for easier anomaly detection have gained traction. However, these methods often struggle with evolving malware tactics and the need for real-time detection.

In response to these challenges, our research proposes a mixed static analysis approach that integrates permission analysis, opcode examination, and bytecode visualization. This holistic method aims to leverage the strengths of each individual technique to create a more comprehensive and effective malware detection system. Permission analysis scrutinizes the permissions requested by applications, identifying those that seek unnecessary or potentially dangerous access to sensitive data or system functions. Opcode examination delves into the operational codes used during app execution, uncovering patterns and anomalies that may indicate malicious intent. Bytecode visualization transforms the code into visual patterns, making it easier to detect deviations from normal behavior. By combining these approaches, our model enhances its ability to detect a wide range of malware behaviors and adapt to new threats more efficiently. This integrated approach not only improves detection accuracy but also provides a more robust defense against sophisticated malware that might evade single-method detection systems. Through comprehensive analysis, our method addresses the multifaceted nature of modern malware, ensuring a higher security level for Android users, and providing a strong defense mechanism against Zero-day attacks.

The present thesis is structured into four chapters. Chapter 1 highlights the proliferation of Android systems and the structure of Android applications, aiding in the understanding of how malware is injected into Android apps. It also discusses the different types of Android malware and presents some notable examples, along with their impacts. Chapter 2 provides an overview of various Android malware detection methods, offering a comprehensive classification of these techniques and presenting related works for each method. Chapter 3 details the proposed mixed approach for effectively detecting Android malware and describes the experimental setup adopted to validate the proposed system. Finally, Chapter 4 presents and discusses the obtained results, comparing them with state-of-the-art methods. This comparison showcases the advancements provided by the proposed system in detecting Android malware compared to existing approaches.

Chapter 1

Emerging Threats of Android Malware

The proliferation of smartphones, particularly those powered by the Android operating system, has revolutionized the way we interact with technology on a daily basis. With millions of applications available for download from various app stores, Android users enjoy a diverse range of functionalities, from productivity tools to entertainment apps. However, this widespread adoption of mobile technology also brings forth significant challenges, particularly in the realm of cybersecurity. Amidst the convenience and innovation facilitated by mobile applications, there lurks a growing menace: *Android-based malware*. These malicious programs pose a serious threat to the security and privacy of users, potentially leading to financial losses, identity theft, and unauthorized access to sensitive information. In this chapter, we delve into the increasing dependence of users on mobile applications, specifically focusing on those built for the Android platform. We examine the internal structure of Android applications, uncovering their architectural components. Additionally, we discuss the profound implications of this reliance, particularly in light of the prevalent risks posed by Android-based malware. By analyzing the potential consequences of using these applications, we aim to illuminate the security vulnerabilities and threats encountered by users. Our exploration underscores the importance of heightened awareness and the implementation of robust protective measures to mitigate these risks effectively.

1.1 Dependency on Mobile Applications

The prevalence of mobile devices, particularly Android smartphones, has seen a significant increase in recent years. In 2021, there were nearly 15 billion mobile devices in use globally, an increase from just over 14 billion in the previous year. This number is projected to rise to 18.22 billion by 2025, as illustrated in Figure 1.1. This exponential growth reported by Statista [49] is largely driven by the increasing dependency on mobile applications for various tasks.

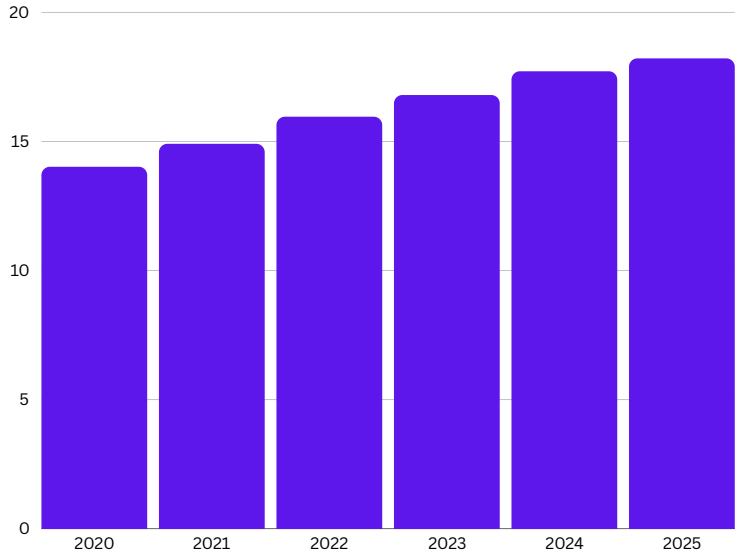


Figure 1.1 – Forecast number of mobile devices worldwide in billions [49].

The meteoric rise of mobile applications has fueled a growing dependency on smartphones for an ever-expanding array of tasks. From managing personal finances to ordering food, from staying connected with friends to accessing critical work documents, mobile applications have become the go-to solution for addressing myriad needs and desires. The convenience and accessibility offered by these apps have revolutionized the way we navigate the modern world, blurring the boundaries between the physical and digital realms.

1.2 Android Applications

Android applications, commonly referred to as *apps*, are software programs designed to run on the Android operating system, the world's most widely used mobile platform. These apps are integral to the mobile ecosystem, enabling users to perform a wide range of tasks directly from their smartphones or tablets. Developed using the Android Software Development Kit (SDK), Android apps are distributed through various channels, with Google Play Store being the most prominent [8].

1.2.1 Role in the Mobile Ecosystem

Android apps play a pivotal role in the mobile ecosystem, acting as the primary interface through which users interact with their devices. They facilitate a multitude of functions, from essential services like communication and navigation to entertainment and education. By providing tailored experiences and leveraging the capabilities of mobile hardware, Android apps enhance the usability and versatility of smartphones, making them indispensable tools in modern life. The Android app ecosystem is marked by its extraordinary diversity and complexity. This vast array of apps can be broadly categorized into several

types based on their functionalities 1.2:

— *Productivity Tools:* These apps help users manage their daily tasks, schedules, and works. Examples include email clients, calendar apps, to-do lists, and document editors. The Productivity category has an average rating of 4.1 stars and approximately 28% of apps have over 50,000 downloads [9].

— *Games:* Ranging from simple puzzles to graphically intensive multiplayer games, Android gaming apps cater to all age groups and preferences, providing immersive entertainment experiences. Games are one of the most popular categories with over 100,000 apps, an average rating of 4.3 stars, and about 20% of apps exceeding 50,000 downloads [9].

— *Social Networking:* Apps like Facebook, Instagram, Twitter, and TikTok enable users to connect, share, and interact with others globally, fostering a digitally interconnected society. This category averages a 4.2-star rating and around 30% of apps have more than 50,000 downloads [9].

— *Communication:* Messaging and calling apps such as WhatsApp, Telegram, and Skype facilitate real-time communication through text, voice, and video. The Communication category boasts an average rating of 4.1 stars, with approximately 25% of apps surpassing 50,000 download [9].

— *Media and Entertainment:* Streaming services like Netflix, Spotify, and YouTube offer on-demand access to movies, music, and videos, transforming media consumption habits. These apps typically have high user engagement, reflected in an average rating of 4.3 stars and 22% of apps achieving over 50,000 downloads [9].

— *E-commerce:* Shopping apps like Amazon, eBay, and Alibaba allow users to browse and purchase products from their mobile devices, revolutionizing retail experiences. The E-commerce category has an average rating of 4.2 stars, with about 18% of apps having more than 50,000 downloads and an average price for paid apps being around $3 [9].

The complexity of these applications varies significantly, from simple utilities with straightforward functionalities to sophisticated programs that integrate advanced technologies like artificial intelligence, augmented reality, and cloud computing.

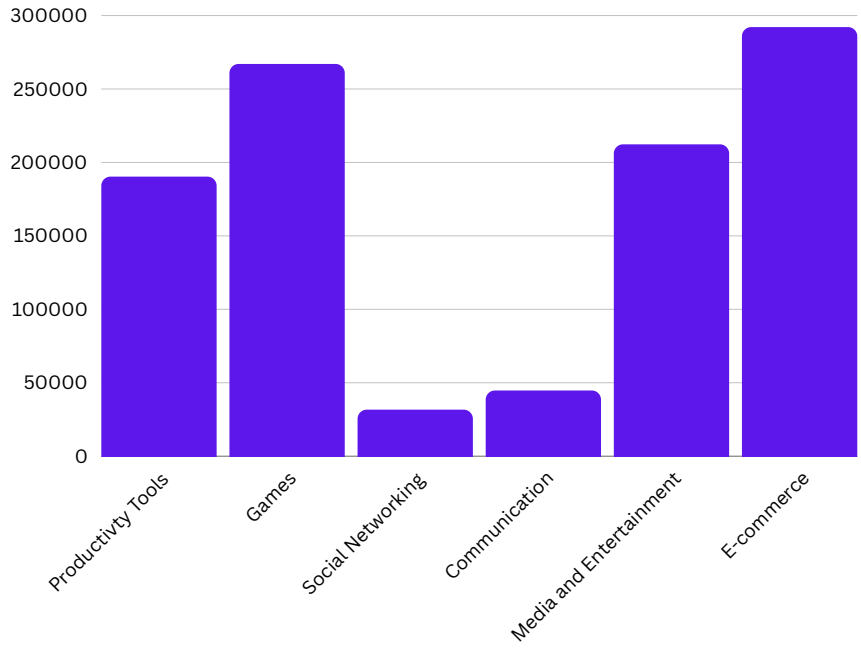


Figure 1.2 – Most popular android applications categories [9].

Figure 1.2 illustrates the distribution of popular Android application categories, revealing significant trends in user preferences and developer focus. Games and e-commerce applications dominate the landscape with the highest number of applications, underscoring a robust demand for entertainment and online shopping solutions. The prominence of these categories can be attributed to their extensive user engagement and the profitability of in-app purchases and e-commerce transactions. In contrast, social networking and communication applications represent the lowest numbers, indicating either a market saturation of existing applications or a reduced demand for new entries in these domains.

1.2.2 Popularity of Android Apps

The popularity of Android apps is underscored by compelling statistics and trends. As of the latest data, the Google Play Store hosts over 1.9 million apps, making it the largest app marketplace globally. The adoption of Android apps continues to surge, driven by the widespread availability of affordable Android devices and the continuous innovation in app development. Several key statistics highlight the dominance and usage patterns of Android apps:

— *Download Volumes:* In 2023, Android apps were downloaded over 100 billion times globally. This immense volume of downloads highlights the widespread adoption and reliance on Android apps across different demographics and regions [39, 40].

— *Market Share:* Android holds a dominant position in the global smartphone market, with over 70% of mobile devices operating on the Android OS. This significant market share ensures a vast and diverse user base for Android applications [39].

— *User Engagement:* Android users exhibit high levels of engagement with their apps, spending several hours daily interacting with various applications. Categories such as social networking, gaming, and streaming media are particularly popular, driving substantial user engagement and interaction [13].

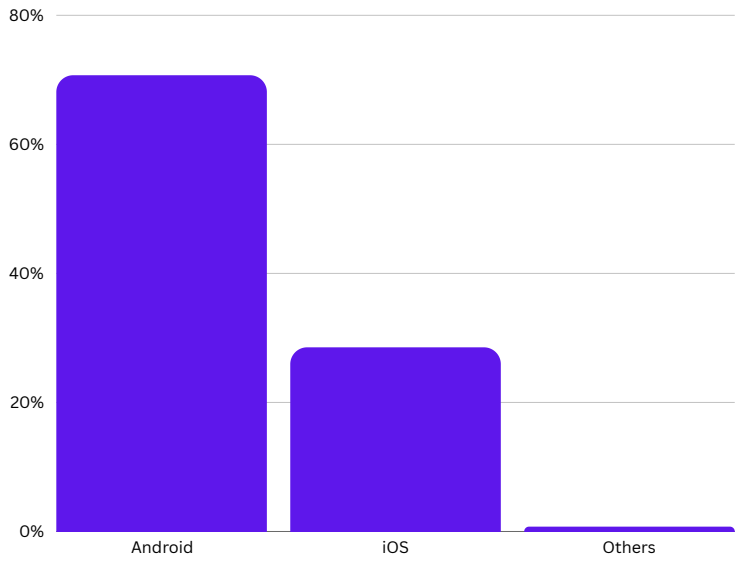


Figure 1.3 – Market share of mobile operating systems worldwide Q1 2024 [51].

Figure 1.3 shows that Android has the highest market share among mobile operating systems worldwide in Q1 2024, with around 70%. iOS comes second with a market share of around 28%, while other operating systems like *Samsung*, *Windows Phone* collectively account for a small portion of around 2% market share. These statistics not only demonstrate the popularity and widespread use of Android apps but also underline their critical role in the daily lives of users. The extensive reach and significant impact of Android applications necessitate ongoing efforts to ensure their security and reliability, particularly in the face of rising malware threats.

1.2.3 Android Apps on Other Devices

Beyond smartphones and tablets, Android applications have expanded their reach to a variety of other devices, significantly enhancing their functionality and user experience. One prominent example is the integration of Android apps with smart TVs [20].

— *Android TV:* Android TV is a version of the Android operating system designed for digital media players and smart TVs. It provides a consistent and user-friendly interface for accessing streaming services, games, and other apps on the larger screen. Android TV supports Google Play Store, allowing users to download and install a wide range of apps optimized for television screens.

— *Wear OS:* Android applications also extend to wearable devices through Wear OS, Google's operating system for smartwatches. These apps offer functionalities such as fitness tracking, notifications, navigation, and music playback, significantly enhancing the utility of wearables. By integrating these capabilities, Wear OS apps provide a seamless and enriched user experience, making smartwatches more versatile and essential in daily life.

— *IoT Devices:* The versatility of Android applications also extends to Internet of Things (IoT) devices. From smart speakers to home automation systems, Android apps empower users to control and monitor various IoT devices, creating a cohesive and interconnected smart home environment. This integration facilitates seamless interaction between devices, enhancing convenience and efficiency in everyday life.

1.3 Structure of Android applications

Android applications are distributed as APK (Android Package Kit) files, which are essentially compressed archives containing all the necessary files for an application to function on an Android device. Understanding the structure of these APK files is crucial for malware detection and analysis, as it allows for a detailed inspection of their contents to identify any malicious components. While typical APK files include components like the *AndroidManifest.xml*, *classes.DEX* file (compiled code), *res* folder (resources), *assets* folder, and *META-INF* directory, deviations such as the presence of multiple *.DEX* files or the absence of certain expected files can indicate anomalous or potentially malicious behavior. Figure 1.4 illustrates the internal structure of typical APK files.

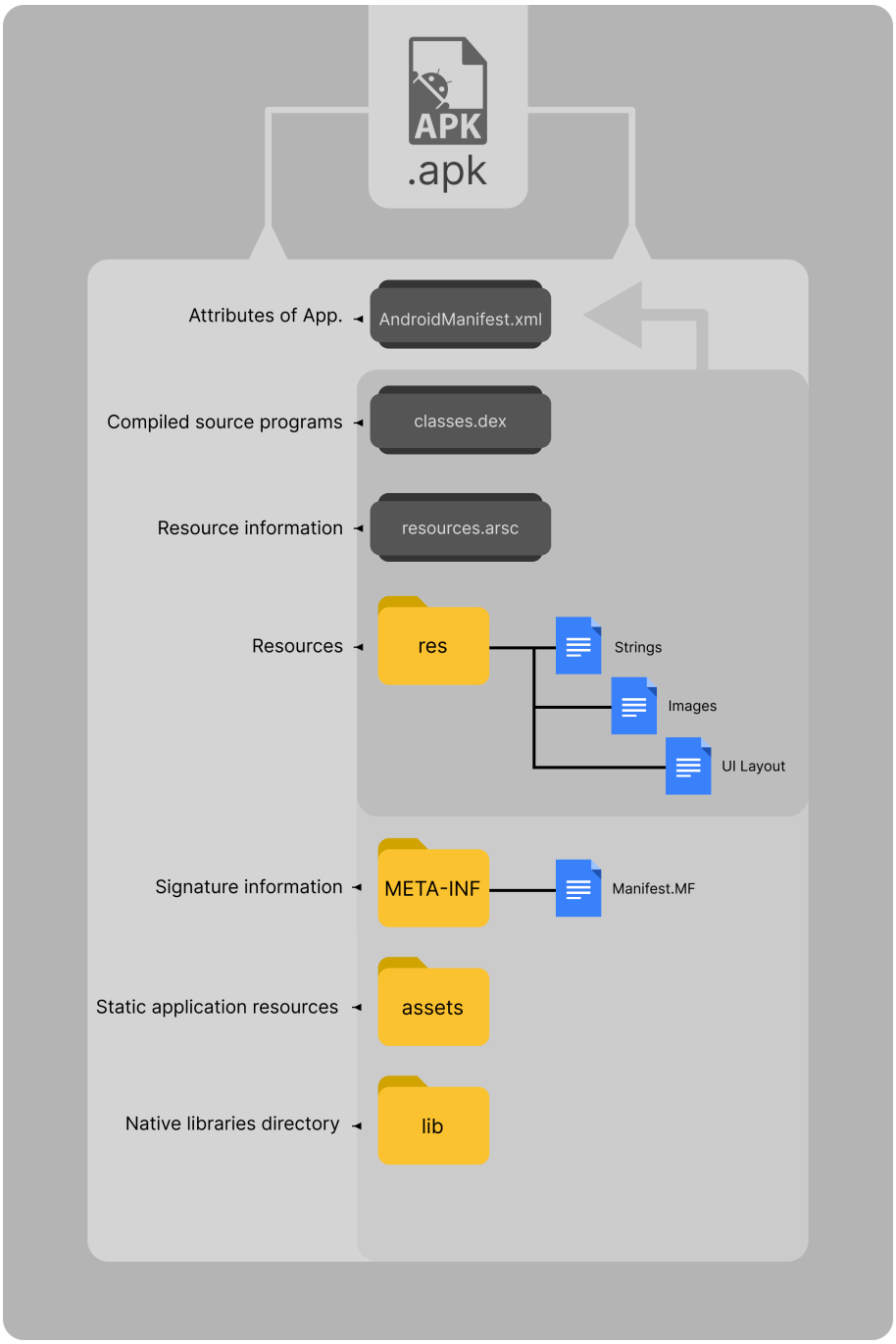


Figure 1.4 – Structure of Android applications.

1.3.1 Manifest File

The Manifest File, known as *AndroidManifest.xml*, serves as a comprehensive guide for an Android application. This file is crucial as it defines key information that the Android operating system needs before executing any of the app's code. It specifies the app's package name, which uniquely identifies the app on the device and in the Google Play Store. Additionally, the manifest file lists the permissions required by the app, such as access to the internet, camera, or location services. These permissions inform users about

the app's potential access and functionality, enhancing transparency and security. The manifest also declares the minimum Android version required for the app to run, ensuring compatibility with the device's operating system. By detailing these vital components, the Manifest File plays a pivotal role in the app's deployment and functionality, acting as a blueprint for the Android OS to correctly manage and execute the app [16].

Figure 1.5 illustrates a code snippet of an Android *AndroidManifest.xml* file, which serves as a declaration for a simple Android application. It specifies the package name *com.example.myapp*, which uniquely identifies the application. The `<uses-permission>` tag requests internet access for the app. The `<application>` tag contains several attributes: *allowBackup* enables data backup, *icon* sets the application's launcher icon, *label* sets the application's name, *supportsRtl* ensures right-to-left text layout support, and theme specifies the application's theme. Within the `<application>` tag, an `<activity>` element defines the main activity (*MainActivity*) and includes an `<intent-filter>` that designates this activity as the application's entry point, launching it when the device's home screen icon is tapped. This setup ensures the proper configuration and behavior of the Android application.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp">
    <uses-permission android:name="android.permission.INTERNET" />
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Figure 1.5 – A snippet of a simple Manifest file.

1.3.2 Application code

The application code, encapsulated within the compiled *classes.dex* file, contains the essential logic and functionality of the application. This file includes the actual programming code, typically written in Java or Kotlin, defining how the application operates and responds to user interactions. It encompasses various components such as activities, services, broadcast receivers, and content providers, each contributing to different aspects of the

application's functionality. Within this code, developers implement algorithms, business logic, user interface elements, and interactions with external resources like databases and network services. The application code serves as the backbone of an Android application, dictating its behavior and ensuring its proper functioning [16].

1.3.3 Resources

The resources folder plays a crucial role in ensuring the smooth functioning of Android applications across various devices and user environments. It encompasses a diverse set of components, including layout files defining the user interface structure, image files for graphical elements, string files containing localized text for different languages, and XML files specifying configuration parameters and other settings. These resources collectively provide the visual and textual elements necessary for the application's user interface and overall experience. By separating these elements from the application code, developers can easily customize and adapt the application's appearance and behavior without modifying the underlying logic, facilitating better maintainability and scalability of the application. The folder also contains a *.arsc* file, often named *resources.arsc*, which includes compiled resources like strings and styles, making them more efficient to access at runtime. This file is generated during the build process and serves as a binary representation of the application's resources [16].

1.3.4 Assets

The Assets folder within an Android application houses raw files that don't fall under the category of traditional resources. These files might include additional fonts, data files, or other resources that the app needs but aren't considered part of the core resource set. Unlike resources stored in the res directory, assets are accessed using a different mechanism and are typically read directly from the APK at runtime [16].

1.3.5 Native Libraries

The *lib* folder within an APK file stores platform-specific libraries that are essential for performance-intensive operations within the application. These libraries, typically written in lower-level languages such as C or C++, provide optimized functionality for resource-intensive tasks, thereby enhancing the overall performance of the application [16].

1.3.6 META-INF

The META-INF directory within an Android application houses metadata about the APK file itself. This metadata includes information about the contents of the APK and signature files used to authenticate the application's integrity and origin. The *MANIFEST.MF*

file, found within this directory, lists the contents of the APK and plays a vital role in helping the Android system understand the structure, components, and authenticity of the installed application [16].

1.4 Rising Concerns about Android Malware

Amidst the proliferation of mobile applications, the threat of Android malware casts a significant shadow over the digital landscape. Malicious actors, driven by financial motives, espionage, or malicious intent, exploit vulnerabilities in the Android ecosystem to introduce a wide range of threats to unsuspecting users [21].

The implications of Android malware extend beyond individual users, affecting organizations, industries, and the broader cybersecurity landscape. For users, falling victim to Android malware can lead to financial losses, identity theft, privacy breaches, and reduced device performance.

Moreover, the impact of Android malware is not limited to mobile devices. With the integration of Android into various IoT devices, such as smart home systems, connected appliances, and industrial IoT deployments, the risks to data privacy, device integrity, and user safety are substantial.

Similarly, Android TV devices and Wear OS smartwatches are also vulnerable to malware. Compromised Android TV devices can be used to launch DDoS attacks, distribute malicious content, or intercept sensitive information, while infected Wear OS devices may jeopardize user privacy, health data, and personal information.

For organizations, the consequences of Android malware can be even more severe, including data breaches, regulatory penalties, reputational damage, and operational disruptions. In an interconnected world where mobile devices serve as gateways to corporate networks and critical infrastructure, the repercussions of Android malware can resonate across entire industries and economies [21].

1.4.1 Causes of Android Malware

The prevalence of Android malware is fueled by several key factors that create opportunities for malicious actors to infiltrate and compromise devices:

— *Open-Source Nature of Android:* Android's open-source framework, while fostering innovation and customization, also makes it more susceptible to exploitation. The flexibility that allows developers to modify the operating system can be exploited by malicious actors to insert malware into apps or even the OS itself [53].

— *Fragmentation of the Android Ecosystem:* The Android ecosystem is highly fragmented, with numerous manufacturers and devices running different versions of the OS. This fragmentation results in inconsistent security updates and patches, leaving

many devices vulnerable to known exploits. Attackers often target older, unpatched versions of Android [53].

— *Lax App Review Processes:* Despite efforts by Google to enhance security, the Google Play Store has historically struggled with lax app review processes, relaxed or lenient procedures employed by app stores or platforms when evaluating and approving applications for distribution to users. Malicious apps can sometimes evade detection and be distributed to millions of users before being identified and removed. This problem is exacerbated by third-party app stores, which often have even less stringent security measures [53].

— *Social Engineering and Phishing:* Attackers often use social engineering tactics to trick users into downloading malicious apps or clicking on harmful links. Phishing emails, malicious SMS messages, and fake websites are common methods used to distribute Android malware [53].

— *Third-Party App Stores:* While Google Play Store is the primary source for Android apps, many users download apps from third-party app stores, which often lack rigorous security checks. These stores can host pirated apps that are infected with malware, significantly increasing [53].

— *Rooting and Jailbreaking:* Users who root or jailbreak their devices to gain more control and remove manufacturer restrictions often disable built-in security features. This practice can expose the device to a higher risk of malware infection as it bypasses many of the security protocols established by the Android OS [53].

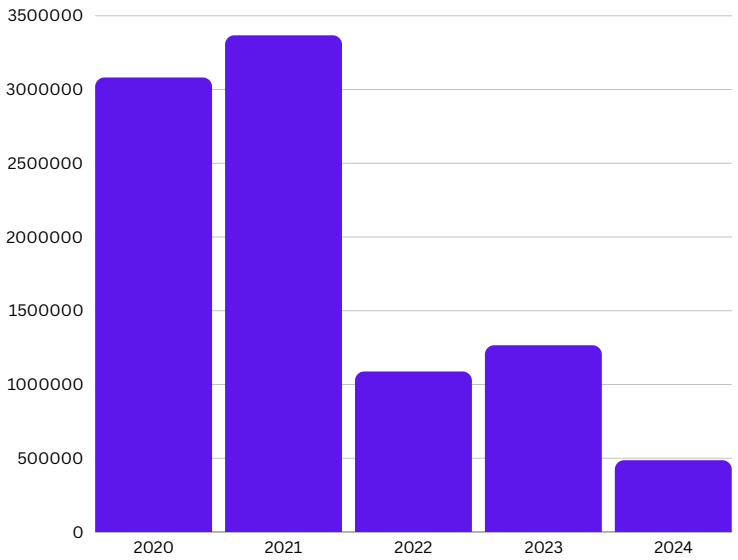


Figure 1.6 – Android malware growth over the years [11].

Figure 1.6 illustrates the growth of Android malware over the years, highlighting a significant spike in the number of malware instances in 2020 and 2021. During these years,

the count of Android malware sharply increased, reaching its peak. This surge could be attributed to several factors, including the rapid expansion of the Android user base, increased internet penetration, and the exploitation of vulnerabilities by cybercriminals amid the global shift to remote work and online activities due to the COVID-19 pandemic. However, in subsequent years, the number of Android malware instances shows a noticeable decline, which may suggest improved security measures, better user awareness, and enhanced detection and prevention technologies within the Android ecosystem.

1.4.2 Definition and Forms of Android Malware

Android malware refers to malicious software specifically designed to target Android operating systems. Various forms of Android malware include:

— *Spyware:* Spyware is a type of malware designed to covertly gather user information without their consent. It can access various types of data, including contact lists, messages, browsing history, and even real-time location data [23]. A significant instance of spyware was the discovery of malicious code in some file manager applications on Google Play, which transmitted sensitive information to servers in China. The widespread impact of this threat was highlighted in a report indicating that spyware affected over 40% of organizations globally [46].

— *Adware:* Adware automatically displays or downloads unwanted advertisements on the users device. Some adware operates stealthily, running in the background to generate revenue through ad clicks [23]. For example, the Goldoson malware, discovered in several popular apps, was found to load hidden ads and harvest user data. According to a study, adware was responsible for 42.55% of all detected mobile threats, highlighting its significant presence in the mobile threat landscape [45].

— *Ransomware:* Ransomware locks the users device or encrypts their data, demanding a ransom to restore access. Android ransomware can be particularly devastating as it can render the device unusable until the ransom is paid [15]. In 2023, a surge in ransomware attacks was observed, with the mobile sector witnessing a 35% increase in ransomware incidents compared to the previous year. This trend underscores the growing threat of ransomware in the mobile ecosystem [2].

— *Trojans:* Trojans are malicious programs that disguise themselves as legitimate applications to trick users into installing them. Once installed, they can perform various malicious activities, such as stealing personal information or subscribing users to premium services without their knowledge [23]. The Fleckpe subscription Trojan, for instance, downloaded additional payloads and subscribed users to unwanted services. In 2023, Trojans accounted for 34% of mobile malware detection, demonstrating their significant presence in the threat landscape [24].

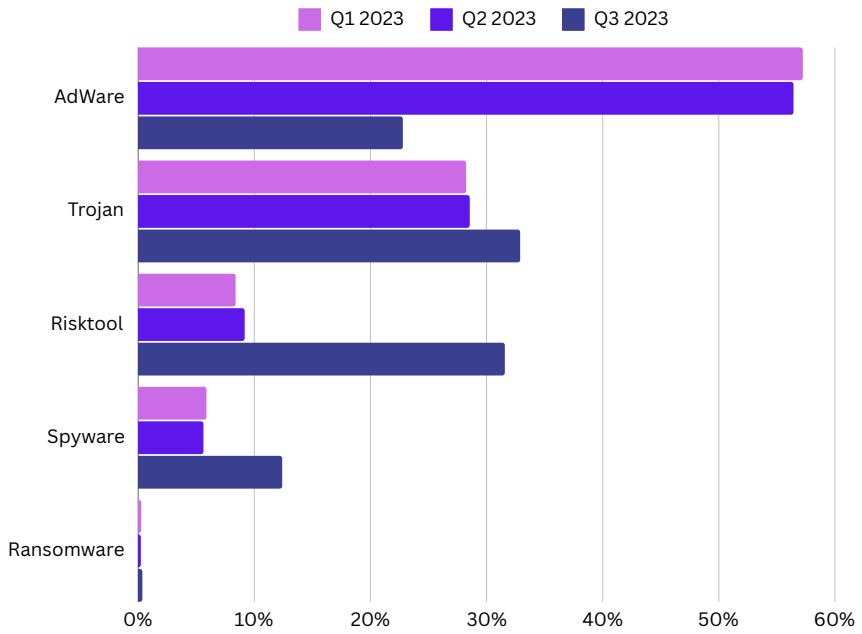


Figure 1.7 – Distribution of detected mobile malware by type, Q1-Q3 2023 [44][50].

Figure 1.7 shows the distribution of detected mobile malware by type from Q1 to Q3 2023. The percentage of AdWare, which was relatively high in Q1 and Q2, dropped significantly in Q3. This decline suggests a reduction in the prevalence or effectiveness of AdWare during this period. In contrast, the percentage of Risktool malware exhibited a substantial increase in Q3 compared to the earlier quarters. This shift indicates a growing trend or focus on Risktool malware, which often includes applications that can be misused for malicious purposes, such as gaining unauthorized access or performing harmful actions. The data highlights changing dynamics in mobile malware threats, emphasizing the need for adaptive security measures.

1.4.3 Notable Android Malware Attacks

Android malware poses a significant threat to the security and privacy of millions of users worldwide. These malicious software programs target vulnerabilities within the Android operating system, often exploiting unsuspecting users through seemingly legitimate applications or websites. In recent years, several high-profile malware campaigns have underscored the severity of this threat, highlighting the need for robust cybersecurity measures and proactive detection strategies. In this section, we delve into four notable instances of Android malware: *Judy Malware*, *HummingBad*, *Gooligan*, and *Xiny*. Each of which has left a significant impact on the Android ecosystem, affecting millions of devices and demonstrating the evolving tactics used by cybercriminals to infiltrate mobile devices.

— *Judy Malware:* One of the largest malware campaigns, Judy infected over 36 million devices by embedding itself in over 40 apps on the Google Play Store. It primarily

functioned as adware, generating fraudulent clicks on advertisements to generate revenue for the attackers. Discovered in May 2017, the malicious apps had been present on the Play Store for an extended period, potentially affecting users for a long time before being detected and removed by Google. These attacks highlight the persistent and evolving nature of Android malware threats. Each incident emphasizes the need for robust security practices and advanced detection mechanisms to protect against the increasing sophistication of cyber threats targeting Android devices [41].

— *HummingBad:* Discovered in February 2016, HummingBad infected around 10 million devices. This malware installed fraudulent apps and generated fraudulent ad revenue. It also attempted to gain root access to devices, allowing it to perform more severe malicious activities. The campaign was linked to a group of Chinese cybercriminals known as Yingmob, and at its peak, it was estimated to generate $300,000 per month in revenue from fraudulent ads. Efforts to resolve it involved extensive updates and security patches from Google and other stakeholders in the Android ecosystem [14].

— *Gooligan:* Gooligan malware compromised over a million Google accounts by targeting older versions of the Android OS. It gained root access to devices and stole authentication tokens that allowed attackers to access users Gmail, Google Photos, Google Docs, and other services. This malware was discovered in November 2016 and was part of a larger campaign affecting at least 86 apps available on third-party app stores. The financial impact included both direct revenue from fraudulent app installs and the potential indirect costs related to data breaches and unauthorized access to sensitive personal information [14].

— *Xiny:* A persistent threat since 2015, Xiny malware is known for its ability to root devices and install unwanted apps. It affected millions of devices and continuously evolved to bypass security measures, making it particularly challenging to eradicate. The malware's capability to re-infect devices even after a factory reset posed significant challenges. Efforts to combat Xiny included continuous updates to Android's security features and collaboration between Google and security researchers to identify and block new variants of the malware[32].

1.5 Conclusion

The meteoric rise of Android applications and the increasing reliance on mobile devices have undoubtedly transformed our daily lives, offering unparalleled convenience and access to a wealth of services. However, this technological revolution comes with an inherent risk: the proliferation of Android malware. As we have explored in this chapter, the threat landscape is rapidly evolving, with cybercriminals continuously developing new tactics

and exploiting vulnerabilities within the Android ecosystem. The implications of Android malware extend far beyond individual users, posing significant risks to organizations, industries, and critical infrastructure. Financial losses, data breaches, privacy violations, and operational disruptions are just a few of the potential consequences of falling victim to these malicious attacks. Addressing this challenge requires a multifaceted approach, combining robust security practices, advanced detection mechanisms, and a heightened awareness among users and developers alike. We delve deeper into the realm of Android malware detection and mitigation strategies, in the next chapter.

Chapter 2

Android Malware Detection Approaches

As Android has emerged as the dominant mobile operating system, malware targeting Android devices has become a growing concern. The widespread presence of malicious apps not only violates user privacy and security but also endangers sensitive personal and financial information. Traditional detection methods that rely on known malware signatures are proving ineffective in keeping up with the rapid evolution of malware. This has spurred the need for more robust and adaptive detection techniques, particularly those powered by machine learning (ML) approaches. Machine learning holds promise by analyzing patterns and behaviors associated with malware, allowing for the identification of previously unidentified threats [28]. By training on vast datasets of both benign and malicious applications, ML models can effectively discern subtle differences and generalize from learned patterns to detect new variants of malware. This paradigm shift towards ML-based detection methods underscores the importance of integrating advanced computational techniques to enhance security measures against sophisticated cyber threats [34]. In this chapter, we offer a thorough presentation of significant research efforts leveraging Machine Learning (ML) techniques for the detection of Android malware. Through a meticulous examination of these studies, our aim is to elucidate the advancements, methodologies, and outcomes attained in the domain of Android malware detection using ML. Notably, the literature encompasses four primary techniques, each offering distinct advantages: *permission-based*, *opcode-based*, *visualization-based*, and *hybrid* approaches [29]. In the following sections of this chapter, we will describe each of these approaches in detail, discussing notable research works and advancements associated with each method. Figure 2.1 illustrates a classification of various approaches documented in the literature for detecting Android malware.

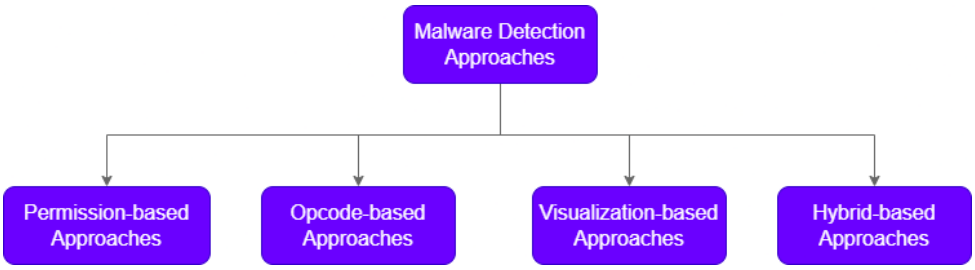


Figure 2.1 – Classification of Android malware detection approaches.

2.1 Permission-based Approaches

Android applications must explicitly declare the permissions they require, such as access to the camera, contacts, or internet, in their manifest files (*AndroidManifest.xml*, refer to Section 1.3 for more details about the structure of the Android application file). This is essential for the operating system to manage application privileges and maintain user security. However, malware often exploits this by requesting excessive or inappropriate permissions to perform malicious activities, such as accessing sensitive data or controlling device functions without the user's consent.

Permission-based detection methods leverage this aspect by meticulously analyzing the permissions requested by an application. These methods involve creating profiles of known benign and malicious applications based on their permission requests. Machine learning models are then trained on these profiles to learn the patterns and correlations between specific permissions and malicious behavior. For instance, an application requesting access to both the camera and the internet might be flagged as suspicious, especially if it doesn't have a clear reason for needing both [26]. By evaluating the permissions requested by new applications and comparing them with the established profiles, trained machine learning models can classify applications as either malicious or benign. This approach not only helps in detecting existing malware but also in identifying new, previously unseen threats by recognizing anomalous permission requests that deviate from normal behavior patterns.

Alsoghyer and Almomani [5] investigated the role of application permissions in detecting Android ransomware. Their study focused on analyzing 115 permissions associated with ransomware behavior. Receiving boot system completion, internet access, network state access, reading phone state, and writing into external storage were found among the top frequently required permissions by Android ransomware. To assess the effectiveness of various machine learning models in ransomware detection, the authors experimented with several classifiers, including Random Forest (RF), Sequential Minimal Optimization algorithm (SMO), Decision Tree (DT), and Naïve Bayes (NB). Utilizing a balanced dataset comprising 500 samples for each category (ransomware and benign apps), the RF model

emerged as the most effective, achieving an impressive accuracy of 96.9%.

Rathore et al. [42] conducted an extensive analysis on Android permissions, identifying a set of 197 permissions and applying three distinct dimensionality reduction techniques: variance threshold, principal component analysis, and auto-encoders, to derive 16 significant permissions. Noteworthy permissions among these included reading phone state, writing into external storage, and Wi-Fi access permissions. The study employed a diverse range of machine learning models, including deep neural networks, to assess performance on a dataset comprising 5560 malware and 5721 benign Android applications. Among the models evaluated, the Random Forest classifier exhibited the best overall performance. With the original feature set, it achieved the highest area under the curve (AUC) score of 98.1%. Furthermore, employing the reduced feature sets obtained from each dimensionality reduction technique, the Random Forest classifier yielded AUC scores of 97%, 97.6%, and 97.7%, respectively, showcasing its robust performance across different data representations.

Todd McDonald et al. [33] conducted a study on the effectiveness of using Android manifest permissions as features for machine learning-based malware detection. They allowed individual machine learning algorithms to assign weights to each feature, focusing on analyzing permissions. Among the most frequently encountered permissions, internet and network state access, as well as reading phone state, were found to be prevalent. The study utilized four machine learning algorithms: Random Forest, Support Vector Machine, Gaussian Naïve Bayes, and K-Means. Random Forest emerged as the top performer, achieving an F1-score of 84.2%. Evaluating their approach on a substantial dataset comprising 4,597 benign and 6,000 malicious Android applications, the authors demonstrated that single-algorithm machine learning techniques utilizing only manifest permissions can surpass commercial antivirus engines in Android malware detection.

2.2 Opcode-based Approaches

Opcode-based techniques focus on the sequences of opcodes (operation codes) executed by an application. Opcodes are the fundamental instructions that a CPU executes to perform various operations, such as arithmetic, data movement, and control flow. Opcodes in Android applications are the low-level instructions that the Android runtime environment either Dalvik Virtual Machine (DVM) [1] or Android Runtime (ART) [56] executes. These opcodes dictate the fundamental operations that the application performs, ranging from arithmetic operations and data handling to control flow management. In an APK file, these opcodes are found within the *classes.DEX* file (refer to Section 1.3 for more details about the structure of the Android application file), which is the compiled bytecode of the application. When developers write code in languages like Java or Kotlin, this code is compiled into the Dalvik Executable format (*.DEX*), which contains the opcode sequences. By decompiling the *classes.DEX* file using tools like JADX or APKTool, one can extract

and analyze these opcodes to understand the application's behavior and identify any malicious patterns.

Figure 2.2 represents a simple method that prints *"Hello, Dalvik!"* to the standard output. The `.registers 2` directive indicates that the method uses two registers, `v0` and `v1`. The `const-string` instruction loads a string constant into register `v0`, `sget-object` loads a static object reference from a class field into register `v1`, and `invoke-virtual` calls the *println* method on the *PrintStream* object referenced by `v1`. Finally, `return-void` terminates the execution of the method.

```
.method public static main([Ljava/lang/String;)V
    .registers 2
    const-string v0, "Hello, Dalvik!"
    sget-object v1, Ljava/lang/System;->out:Ljava/io/PrintStream;
    invoke-virtual {v1, v0}, Ljava/io/PrintStream;->println(Ljava/lang/String;)V
    return-void
.end method
```

Figure 2.2 – A snippet of a simple Opcode.

Analyzing the sequences of these opcodes can reveal characteristic patterns associated with both benign and malicious behavior. For instance, certain opcodes may be used more frequently in malware to exploit system vulnerabilities, access sensitive data, or perform unauthorized actions. By examining these patterns, security researchers can develop profiles of typical opcode sequences for benign applications and compare them to those found in potential malware.

To automate and enhance the detection process, machine learning algorithms can be used. These algorithms can be trained on labeled datasets of known benign and malicious applications to learn the distinctive opcode patterns associated with each class. Once trained, these models can analyze new applications' opcode sequences to identify deviations from normal patterns, flagging those that exhibit suspicious or anomalous behavior indicative of malware. Opcode analysis provides a fine-grained view of an application's functionality, making it a powerful tool for malware detection. Unlike higher-level features, such as permissions, opcodes offer a detailed and granular perspective on the actual operations an application performs. This detailed insight can help uncover sophisticated malware that may evade detection through more superficial analysis methods [26].

Niu et al. [38] proposed a novel method for generating opcode sequences by analyzing Function Call Graphs (FCGs). These FCGs serve as representations of the calling relationships between functions within a program, effectively capturing the program's flow and behavior. The extracted sequences of opcodes are then converted into numerical values or vectors using one-hot encoding. For classification purposes, the authors employed a Long Short-Term Memory (LSTM) neural network, chosen for its suitability in handling

sequential data such as opcode sequences. The parameters of the LSTM model were fine-tuned using grid search and cross-validation techniques to optimize performance. In their experimentation, the LSTM-based neural network, with an embedding layer initialized through Word2Vec, outperformed other classifiers such as Support Vector Machines (SVM). It achieved higher accuracy in malware detection compared to traditional machine learning models, with an F1-score of 97%. This evaluation was conducted on a dataset comprising 1000 benign, 1000 Trojan, and 796 AdWare samples.

Sihag et al. [47] proposed a probabilistic method for classifying Android malware into families based on analyzing their opcodes. They utilized a dataset containing 1010 samples from 15 different Android malware families. The study began by extracting opcode sequences from the *classes.DEX* files of these samples using the APKTool, retaining only the opcode operations and discarding the operands. Similar operations were then grouped together and assigned common, abstract names to standardize the opcode representation. Next, the authors generated n-grams ranging from 1 to 5, which were then filtered using a frequency-based method to select the most representative features. They experimented with different feature vector lengths, including 128, 256, 512, and 1025, to determine the optimal configuration for classification. The classification process utilized a Bloom filter to measure the similarities among samples. This approach allowed the identification of closely related family sets for each malware sample during the training phase. Through extensive experimentation, the best performance was achieved using 128 features of 5-grams, resulting in an accuracy of 94.34%.

Kaleem et al. [37] introduced *Op2Vec*, an innovative technique for converting opcodes of Android applications into meaningful vector representations using the skip-gram model, a neural network-based approach for learning word embeddings. In this method, opcodes are treated similarly to words in natural language processing. Opcode sequences are extracted from *classes.DEX* files, and the skip-gram model is trained on these sequences to learn embeddings where opcodes with similar contexts are mapped to nearly identical vectors. The Op2Vec embeddings are then used to create sequences of vectors for end-to-end Android malware detection, replacing traditional manual feature engineering. These vector sequences are fed into various deep neural networks, including Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), and Long Short-Term Memory (LSTM) networks, to automatically learn and classify malware. This approach leverages the context-based learning capability of skip-gram to capture the semantic relationships between opcodes, enhancing the effectiveness of malware detection. The authors evaluated several recent deep learning techniques on their Op2Vec dataset, demonstrating significantly improved performance. They reported an average detection accuracy of 97.47% on a dataset comprising 16240 benign and 12330 Android malware applications, underscoring the robustness of the Op2Vec approach in accurately identifying malicious applications.

2.3 Visualization-based Approaches

Visualization-based detection methods in Android malware analysis involve converting the application's code, typically stored in binary files like *classes.DEX*, into visual representations, often grayscale or color images, where each byte of the binary file is mapped to a pixel intensity value. This transformative process encodes the bytecode instructions into pixel values, generating images with specific sizes that encapsulate the structural and sequential aspects of the code. The fundamental assumption is that malware and benign software will exhibit distinct visual patterns when rendered as images. Leveraging computer vision and deep learning techniques, researchers aim to discern these patterns within the images [17].

Once the binaries are converted into images, those images can either be fed directly into deep learning models with convolutional layers for feature extraction and analysis. Alternatively, feature extraction techniques such as using local or global image descriptors can be applied to extract distinctive features from these images. These features capture the underlying patterns and characteristics of the binary data that correspond to malicious behavior. By analyzing these visual features, machine learning models can be trained to differentiate between benign and malicious files with high accuracy. The extracted features serve as inputs to various machine learning classifiers, enhancing the detection capability by focusing on the visual signatures of malware [26, 35].

Ding et al. [17] conducted a study utilizing a dataset comprising 3962 malware samples spanning 14 families and 1000 benign samples. The research proposed a novel approach involving the transformation of the content within the *classes.DEX* files into grayscale images with dimensions of 512x512 pixels. These images were subsequently employed as input for a Convolutional Neural Network (CNN). The CNN architecture comprised multiple convolutional and pooling layers designed to automatically learn and extract features from the image data. To identify the optimal model structure, various configurations of the CNN were explored, differing in the number of convolutional and pooling layers utilized. Through systematic experimentation, the study aimed to determine the most effective architecture for malware detection based on image representations of Android application code. The study achieved the best performance by incorporating high-order feature maps into their CNN architecture, resulting in an accuracy of 95.1%.

Mercaldo and Santone [35] proposed an innovative method for detecting mobile malware and classifying them into their respective families and variants by representing Android applications as grayscale images. The images have a fixed width of 256 pixels and a variable length depending on the size of the binary file. They extracted a feature vector of 256 dimensions from the grayscale histogram of each image, which served as input to various traditional classifiers, including J48, Random Tree (RT), Random Forest (RF), AdaBoost, and Bayesian Network (BN), as well as a Deep Neural Network (DNN). The

DNN architecture utilized by the authors consists of an input layer, multiple hidden layers (ranging from 1 to 12), a dropout layer for regularization, a batch normalization layer, and an output layer. This architecture was experimented with to determine the optimal number of hidden layers for the best performance. The proposed approach was tested on three main tasks: malware detection, family identification , and family variant detection. The method was evaluated on a comprehensive dataset comprising 50000 Android applications (24553 malware and 25447 benign). The results demonstrated the effectiveness of the approach, with the DNN achieving the best performance with 10 hidden layers. The F1-scores obtained were 87.5% for malware detection, 91.5% for family identification, and 95.9% for family variant detection.

Almomani et al. [4] propose an automated vision-based Android malware detection (AMD) model that employs 16 different fine-tuned convolutional neural network (CNN) algorithms. The bytecodes of the classes.dex files extracted from the Android benign and malware apps were converted to color and grayscale visual images with different resolutions based on the app sizes. For example, the input image resolution for the Xception CNN algorithm, which achieved the best performance, was 299x299x3. The generated images were resized to meet the input requirements of the employed CNN algorithms. Both balanced (2486 malware and 2486 benign samples) and imbalanced (14733 malware and 2486 benign samples) datasets were used for training and testing. The fine-tuned Xception CNN algorithm achieved the best performance, with a detection accuracy of 99.40% for balanced color images and 98.05% for imbalanced color images, outperforming other CNN algorithms and existing approaches that utilize conventional vision-based algorithms on the same benchmark Android dataset.

2.4 Hybrid Approaches

Hybrid techniques in Android malware detection involve the integration of multiple detection methods to capitalize on the unique advantages of each approach. For instance, one hybrid approach might combine visualization-based and opcode-based techniques, allowing for the simultaneous analysis of both visual patterns and opcode sequences within Android applications. By leveraging both types of information, this hybrid method aims to enhance detection accuracy by capturing a broader range of malware characteristics. Another prevalent hybrid strategy involves integrating permission analysis with opcode analysis. This combined approach offers a comprehensive understanding of an application's behavior by considering both the requested permissions and the underlying opcode sequences. By combining these insights, hybrid techniques can often achieve higher detection rates and lower false-positive rates compared to using a single detection method alone [26].

Geden [18] explored various features extracted through the application of several reverse engineering methods to the *classes.DEX* file, alongside string features extracted from

the *Manifest.MF* meta file and *AndroidManifest.xml* file. The authors employed n-gram analysis of different lengths (3-grams, 4-grams, and 5-grams) and utilized two primary feature selection methodologies: Information Gain (IG), which scores features based on their information gain, and a novel method called Normalized Angular Distance (NAD), which prioritizes features based on their distinguishing level over document frequencies. The system architecture encompasses data collection from reverse engineering files, feature extraction using n-gram models, feature selection through IG and NAD techniques, and classification using various classifiers. Evaluation was conducted on a dataset comprising 1800 benign and 1800 malicious APK files, with a focus on accuracy and resilience against unseen Android malware families. The highest performance was achieved using a combination of 4-grams extracted from the *classes.DEX* file and 5-grams from the AndroidManifest.xml file with the Random Forest (RF) classifier, resulting in an accuracy of 98.33%, outperforming other classifiers such as Support Vector Machine (SVM) and Naïve Bayes (NB).

Alzaylaee et al. [6] introduced DL-Droid, a system that employs a deep neural network (DNN) trained on a combination of static and dynamic features. Static features encompass permissions, while dynamic features include API calls and intents. For feature extraction, DL-Droid leverages DynaLog, an automated platform capable of sequentially running a large number of Android applications on real devices or emulators and logging their dynamic behaviors. DynaLog is further modified to extract permissions before execution. The extracted features are then utilized to train a DNN classifier for detecting malware among benign Android applications. Evaluated on over 31,125 applications (11,505 malware and 19,620 benign), DL-Droid achieved its peak performance with a detection F1-score of 96.27%. The DNN model, comprising 3 hidden layers with 200 neurons each, surpassed traditional machine learning classifiers.

Maryam et al. [48] proposed two hybrid machine learning-based Android malware detection frameworks, HybriDroid and cHybriDroid, which combine static and dynamic analysis techniques. HybriDroid uses a machine learning model trained on static features such as permissions and intents. Applications flagged as suspicious by this model are then subjected to a dynamic analysis phase. This phase involves executing the application in an emulator and extracting dynamic features, such as system calls, usage of external DeXClass, cryptographic activities, and rehashing activities. These dynamically extracted features are used to train a secondary machine learning model to refine the detection of malicious behavior. In contrast, cHybriDroid simultaneously extracts both static and dynamic features and combines them into a single feature set. This comprehensive feature set is then fed into a single machine learning model for malware detection. The authors experimented with several classifiers, including Support Vector Machine (SVM), Decision Tree (DT), Random Forest (RF), Naïve Bayes (NB), and the Tree-based Pipeline Optimization Tool (TPOT), a Python automated machine learning tool that optimizes

machine learning pipelines using genetic programming. Their experiments were conducted on a dataset comprising 2500 benign samples and 2500 malware samples from various families. The best performing model for the proposed system was the TPOT-based technique, achieving an F-score of 97%.

2.5 Conclusion

The detection of Android malware has become critical due to the increasing sophistication of malicious applications. Traditional signature-based methods are insufficient against evolving threats, prompting the development of advanced machine learning (ML) techniques. This chapter explores four primary ML-based approaches: permission-based, opcode-based, visualization-based, and hybrid methods. Permission-based approaches analyze the permissions requested by applications. These methods are effective because malicious applications tend to request a distinct set of permissions compared to benign ones. Opcode-based techniques focus on the sequences of operation codes (opcodes) executed by applications. By leveraging neural networks, these techniques can detect patterns in the opcodes that are indicative of malware, resulting in significant detection rates. Visualization-based methods convert binaries into visual representations. Convolutional neural networks (CNNs) can then be used to detect malware based on visual patterns found within these images. This innovative approach allows for the identification of malicious software by visualizing code as images, revealing patterns that might not be apparent through traditional text-based analysis. Hybrid approaches combine multiple methods to enhance detection effectiveness. By integrating static and dynamic analysis features, hybrid models can leverage the strengths of each individual method, leading to more comprehensive and accurate malware detection.

The integration of ML in Android malware detection signifies a shift towards more adaptive and intelligent security measures, essential for mitigating sophisticated cyber threats and ensuring user safety. However, the research landscape shows that no single approach can comprehensively address all types of malware due to the diverse strategies employed by malicious actors. Therefore, ongoing research and development are focused on creating more robust, versatile, and resilient detection frameworks. This includes exploring more hybrid techniques, continuous feature enhancement, and real-world testing to adapt to new and emerging threats. The evolving nature of cyber threats necessitates a dynamic and proactive approach to Android malware detection, underscoring the importance of advanced machine learning methodologies in safeguarding users and systems.

Chapter 3

A Mixed Static Analysis Approach for Android Malware Detection

The prevalence of Android malware underscores the urgent need for advanced detection techniques to safeguard user data and device integrity. While traditional static analysis methods are valuable, they often struggle to comprehensively identify malicious behaviors due to their reliance on isolated features such as permissions, opcode or bytecode sequences. To address this challenge, hybrid approaches integrate both static and dynamic analysis techniques, leveraging the strengths of each to offer a multifaceted perspective on an application's behavior. Dynamic analysis involves executing the application in a controlled environment to monitor its behavior, hence it can be time-consuming and resource-intensive, requiring sophisticated sandbox environments that can be circumvented by malware capable of detecting emulation. In contrast, static analysis examines the application's code and resources without execution, enabling a faster and more scalable examination of a large number of applications. Given these considerations, we opt for a mixed static analysis approach that integrates multiple static analysis techniques, including permissions analysis, opcode examination, and bytecode visualisation. This integration offers a robust solution, unveiling potential actions an application can execute, exposing underlying logic and potential obfuscation techniques, and detecting visual similarities with known malware.

This synergistic approach may not only enhances detection accuracy by reducing false positives and negatives but also bolsters resilience against sophisticated evasion tactics employed by malware developers. This approach is critical for timely threat detection and mitigation in environments where new malware variants emerge rapidly and need to be detected instantly. By favorizing static analysis techniques, this mixed approach aims to offer a more efficient and practical solution for analyzing Android applications while maintaining a high level of detection accuracy. In this chapter, we delve into the details of the proposed mixed static approach for the detection of Android malware.

3.1 Overview of the proposed approach

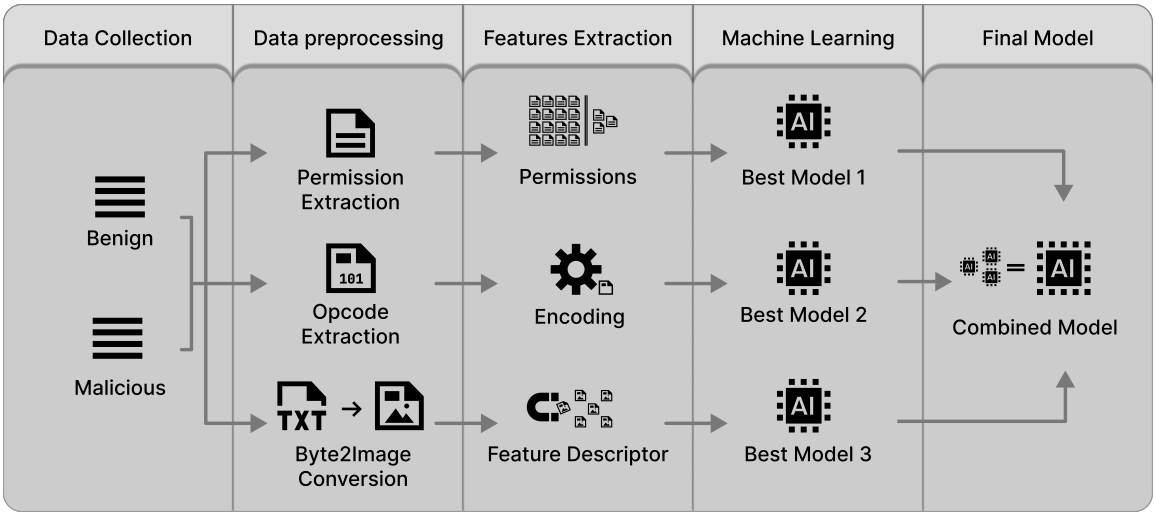


Figure 3.1 – Architecture of our system.

The proposed approach shown in Figure 3.1 integrates comprehensive stages, starting with data collection from diverse sources such as *APKCombo*, *Drebin*, and *Androzoo*. Following this, data preprocessing involves extracting permissions, opcodes, and converting byte data into images. The extracted features are then utilized to train three distinct machine learning models, each specialized in permissions, opcodes, and images. Finally, these models are combined to form a robust and unified final model, ensuring a holistic and effective analysis. In the subsequent sections, we provide a detailed overview of each step in our proposed approach. This includes a thorough explanation of each incorporated technique in the mixed approach.

3.2 Permissions Analysis

In this section, we focus on the permission-based detection method incorporated in our proposed Mixed Static Analysis Approach. The goal is to extract, analyze, and refine the list of permissions from the Android application package (APK) files in our dataset to enhance the accuracy of our malware detection model.

The first step in our permissions analysis is to extract the permissions declared in each APK file. We scan each APK file in our dataset to retrieve the list of permissions it requests. This process involves parsing the *AndroidManifest.xml* file, where permissions are declared. The extraction process is automated to handle large datasets efficiently, ensuring a thorough and consistent analysis across all applications in our dataset. Once we have extracted the permissions from all APK files, we compile a comprehensive list of

all unique permissions. This involves aggregating the permissions from all APK files and removing duplicates to ensure that each permission is only listed once, regardless of how many applications request it. The resulting list is then saved into a Comma-Separated Values file (.CSV), where each row represents an individual APK file, and each column corresponds to a unique permission. The cells in the .CSV file indicate whether a particular permission is requested by a given APK.

To evaluate the effectiveness of permission-based detection, we use several machine learning classifiers to analyze the list of permissions. These classifiers are trained on the permissions data to distinguish between malicious and benign APK files. We systematically assess the performance of these classifiers to determine their ability to accurately identify malware based on the permissions requested by each application. This thorough analysis helps us refine our detection model, ensuring that it leverages the most relevant permissions to improve the accuracy and robustness of our malware detection approach.

3.3 Opcode Examination

In our experimentation with Android malware detection, we focus on extracting and analyzing opcodes from Android application files, specifically from the Dalvik Executable (DEX) format found within APK files. This process begins by loading the APK file, which is the package format used by Android for the distribution and installation of apps. Inside the APK, the executable content is stored in a DEX file, which contains the compiled code running on the Android Runtime (ART). We make use of the Androguard tool [52] to parse the DEX file and access the Dalvik bytecode, comprising the low-level instructions executed by the Android system. We then systematically iterate through all the classes and methods defined in the DEX file, extracting the bytecode for each method that contains it. From the extracted bytecode instructions, we focus on selecting only the operation codes (opcodes), which represent the actions performed by the application, such as arithmetic operations, data movement, or control flow changes. The operands, which provide the data or references needed by the opcodes, are ignored in this process. The extracted opcodes are recorded sequentially in a file for subsequent analysis.

To transform these opcodes into a format suitable for machine learning, we utilize the Term Frequency-Inverse Document Frequency (TF-IDF) method [43]. TF-IDF is a statistical measure used to evaluate the importance of a word in a document relative to a collection of documents. In our context, each opcode sequence is treated as a document, and the entire collection of opcode sequences forms the corpus. This method helps in converting the textual opcode sequences into numeric vectors, where each dimension represents the importance of an opcode n-gram in the sequence. We experimented with different n-grams (combinations of consecutive opcodes) and vector sizes to optimize the representation of the opcode sequences for our machine learning models.

3.4 Bytecode Visualization

Bytecode visualization involves converting the bytecode of an application into a visual format, typically an image, to leverage image processing techniques for malware detection. To achieve this, we conducted an extensive experimentation process comprising six steps: *transforming binary files to squared grayscale images*, *resizing*, *feature extraction*, *evaluation*, *combination*, and *performance enhancement via varying interpolation*. By following these steps, we aimed to leverage the visual characteristics of bytecode for robust and accurate Android malware detection, demonstrating the potential of this approach across different datasets and configurations. In the following, we detail each of these steps.

1. **Transformation to squared grayscale images:** We converted the binary assets of the APK file*AndroidManifest.xml*, *classes.dex*, and *resources.arsc*into individual squared grayscale images. Each file was transformed based on its size to preserve structural integrity. Adopting a grayscale and squared image format ensures uniformity and compatibility with image processing algorithms. Grayscale reduces computational complexity by simplifying the color channel to one, advantageous for large datasets. Squaring the images standardizes their dimensions, facilitating easier model input and visual pattern comparison.
2. **Image resizing:** The generated images were resized to uniform dimensions of 512x512 pixels. This standardization ensures consistent input for machine learning models and facilitates the comparison of visual patterns. The choice of 512x512 pixels, resized using default interpolation techniques, strikes a balance between preserving detail and maintaining computational efficiency, as commonly practiced in previous studies [19].
3. **Feature extraction:** We applied several local feature descriptors to extract salient features from the images. The descriptors used were Scale-Invariant Feature Transform (SIFT), Adaptive and Generic Accelerated Segment Test (AGAST), Maximally Stable Extremal Regions (MSER), and Features from Accelerated Segment Test (FAST). These four algorithms were selected for their robustness in capturing comprehensive visual features, each contributing unique strengths to the feature extraction process:
   - *Scale-Invariant Feature Transform (SIFT)*: SIFT is renowned for its ability to detect and describe local features in images. It is particularly effective in recognizing patterns regardless of scale and rotation, making it highly suitable for diverse malware samples [36].
   - *Adaptive and Generic Accelerated Segment Test (AGAST)*: AGAST is an efficient corner detection algorithm that excels in identifying interest points

in an image. Its adaptability and speed make it valuable for processing large datasets [36].

— *Maximally Stable Extremal Regions (MSER)*: MSER detects stable regions within images that remain consistent across different thresholds. This characteristic is essential for identifying distinct regions of bytecode that may indicate malicious activity [36].

— *Features from Accelerated Segment Test (FAST)*: FAST is a high-speed corner detection method that identifies key points based on the intensity contrast of pixels. Its computational efficiency and accuracy are crucial for real-time malware detection applications [36].

By utilizing these diverse feature descriptor algorithms, we ensured that the bytecode's visual representations were rich in information. This comprehensive feature extraction enhances the capability of machine learning models to accurately distinguish between benign and malicious Android applications.

4. **Classifier evaluation:** The features extracted by the local descriptors were fed into various machine learning classifiers. This step assessed the classifiers' ability to distinguish between malware and benign files. We utilized the same five classifiers from previous analyses (CF, RF, LR, SVM, and KNN) to maintain consistency and comparability. Each classifier's performance was rigorously evaluated to identify the most effective model for distinguishing malware from benign samples.

5. **Combination analysis:** We explored pairwise combinations of the files and their features, as well as the combination of all three files. This was done to identify which combination provided the best classification performance. To handle the size variations of individual files, we used the maximum size for each file type, padding smaller files with zeros on the right-hand side. This approach ensured that the bytes of each file type remained distinct, avoiding mix-ups and preserving data integrity.

6. **Performance enhancement via interpolation:** For the best-performing combination identified in the previous step, we experimented with different image sizes ranging from 64x64 to 1024x1024 pixels. This allows us to assess how image resolution impacts feature descriptor performance and model accuracy. Additionally, various interpolation techniques available in OpenCV [22] were used to resize the images, aiming to determine the optimal technique for extracting keypoints and improving classification results. We experimented with different image sizes and interpolation techniques (INTER_LINEAR, INTER_CUBIC, INTER_NEAREST, and INTER_LANCZOS4) to optimize keypoint extraction and classification results. Techniques available in OpenCV were applied, such as nearest-neighbor, bilinear, and bicubic interpolation, to find the most effective method for our goals.

— *Nearest-Neighbor (INTER_NEAREST)*: This simplest interpolation method selects the nearest pixel value, resulting in a blocky image but fast processing.
— *Bilinear Interpolation (INTER_LINEAR)*: This method computes the output pixel value using a weighted average of the four nearest pixels, producing smoother images than nearest-neighbor.
— *Bicubic Interpolation (INTER_CUBIC)*: This technique uses 16 nearest pixels to determine the output value, resulting in even smoother images and better edge preservation.
— *Lanczos Interpolation (INTER_LANCZOS4)*: This high-quality resampling technique uses a larger window of pixels, providing the most accurate and smoothest images, but at a higher computational cost.

3.5 Mixed approach

In this section, we integrate the results from the three different analysis approaches permission-based, opcode-based, and visualization-based to develop two ensemble machine learning models: *Stacking* and *Voting*. These ensemble techniques aim to leverage the strengths of individual models to improve the overall accuracy and robustness of our malware detection system.

Stacking, also known as stacked generalization [55], is an ensemble learning technique that involves training a meta-model to combine the predictions of several base models. The base models are trained on the entire training dataset, and their predictions are used as input features for the meta-model, which is then trained to make the final prediction. This method allows the meta-model to learn how to best combine the strengths and mitigate the weaknesses of each base model, potentially leading to improved predictive performance.

Voting [27] is another ensemble learning technique where multiple models are trained independently, and their predictions are aggregated to determine the final output. There are two main types of voting: *hard voting* and *soft voting*. In hard voting, the final prediction is determined by the majority vote of the base models, while in soft voting, the predicted probabilities from each model are averaged, and the class with the highest average probability is selected. In this work, we consider the equal importance of all the incorporated static approaches and hence we consider soft voting.

By employing stacking and voting ensembles, we aim to create a more accurate and robust Android malware detection system. These techniques help in leveraging the complementary strengths of the different analysis approaches, thus enhancing the overall detection capability of our models.

3.6 Conclusion

In this chapter, we have explored a comprehensive mixed static analysis approach for Android malware detection, leveraging multiple techniques to enhance detection accuracy and robustness. Our methodology integrates permission-based analysis, opcode examination, and bytecode visualization to provide a multifaceted view of an application's behavior. This integration enables the identification of malicious patterns that might be missed when using a single analysis technique. The use of permissions analysis allows for the identification of suspicious applications based on the permissions they request, which can be indicative of potential malicious intent. Opcode examination provides deeper insight into the application's executable logic, uncovering hidden malicious activities. Bytecode visualization transforms the application's binary data into images, allowing for the application of advanced image processing techniques to detect malware. To further enhance our detection capabilities, we employed two ensemble learning techniques: stacking and voting. Stacking combines the predictions of several base models using a meta-model, learning how to best integrate their strengths and mitigate weaknesses. Voting aggregates the predictions of multiple models, either through majority voting or averaging predicted probabilities, to make a final decision. Both approaches leverage the complementary strengths of our individual models, resulting in a more robust and accurate malware detection system. In the following chapter we provide the results of conducted experiments demonstrating that the adopted ensemble technique significantly improve the performance of our malware detection models.

Chapter 4

Experimentation and Analysis

In this chapter, we delve into the evaluation of the proposed mixed approach and compare it with state-of-the-art methods. For the evaluation, we collect and utilize two distinct datasets of benign and malware APK files from different periods. We conduct extensive experiments to determine the optimal configuration for each static analysis method: *permission-based*, *opcode-based*, and *visualization-based* approaches. We test each method with various classifiers and parameter values tailored to its specific requirements. After identifying the best configuration for each approach, we explore different combination methods to integrate these approaches and select the most effective technique. The combination methods are assessed to ensure that the integrated approach leveraged the strengths of each individual method, leading to improved performance. For comparison with state-of-the-art methods, we replicate recent studies in the field and benchmarked their results against our findings. This comparison aim to highlight the advantages and limitations of our approach. By juxtaposing our results with those from existing studies, we will be able to underscore the improvements our mixed method brings, particularly in terms of accuracy, robustness, and applicability across different datasets and time periods. This thorough evaluation demonstrates the efficacy of our approach and its potential for enhancing malware detection capabilities in real-world scenarios.

4.1 Data Collection

To conduct effective experimentation and analysis for Android malware detection using machine learning, we assembled a comprehensive dataset consisting of both malicious and benign samples. The data sources and collection methods are described in detail below:

— *Malicious samples*: For our malicious samples, we utilized two primary sources: Androzoo [3] and Drebin [10]. The Androzoo repository [3], a large collection of Android application (APK) files from various sources, was filtered to include samples from 2020 onwards to maintain relevance and up-to-dateness. To select the

most harmful applications, we utilized VirusTotal [54], a widely respected online service that aggregates the results of over 70 antivirus engines, website scanners, and other tools to provide a comprehensive analysis of files and URLs for potential threats. VirusTotal's robust detection capabilities allow us to accurately identify and rank malicious applications based on the number of detections reported by its suite of analysis tools. To create a focused and effective dataset for our research, we sorted the APK files by the number of VirusTotal detections. This process involved analyzing each APK file with VirusTotal and recording the number of antivirus engines that flagged the file as malicious. By prioritizing files with the highest number of detections, we ensured that our dataset prominently featured the most harmful and clearly malicious samples available. From this sorted list, we downloaded the first 2000 files, starting with those that had the highest number of VirusTotal detections. This approach ensured the inclusion of highly malicious samples in our dataset, providing a rich source of data for training and evaluating our malware detection models. The Drebin dataset [10], on the other hand, is a well-known benchmark in Android malware detection research. It offers a set of 5560 malicious APK files collected between August 2010 and October 2012, each manually labeled by experts and categorized based on their behaviors. For our study, we randomly selected samples from the Drebin dataset to ensure a comprehensive representation of various malware types and families. This selection process was designed to encompass a broad spectrum of malware characteristics, providing a well-rounded foundation for evaluating and enhancing our detection models. By incorporating the Drebin dataset, we aimed to leverage its rich and varied malware samples to thoroughly test and refine our detection techniques, thereby improving their robustness and generalizability.

— *Benign samples*: For benign samples, we manually downloaded APK files from the APKCombo website [7], a trusted source for legitimate Android applications. Our selection criteria were stringent to ensure the benign nature of the samples: each application had to have a minimum of 50,000 downloads, indicating popularity and user trust, and a rating of 3 stars or higher, suggesting user satisfaction and reducing the likelihood of problematic apps. These criteria were slightly relaxed for applications developed by well-known and reputable companies, acknowledging the inherent trust and security associated with established developers. Additionally, each of these files was subjected to a VirusTotal check, and only those reported as safe were selected. This multi-tiered approach ensured a high level of confidence in the benign nature of the samples, providing a reliable foundation for training and evaluating our malware detection models. By combining these stringent selection criteria with VirusTotal verification, we aimed to create a robust and trustworthy dataset of benign applications to complement our malware samples, enhancing the

overall accuracy and reliability of our research findings.

Dataset	**#Benign**	**#Malicious**
APKComboAndrozoo	698	698
APKComboDrebin	1097	1097

Table 4.1 – Distribution of benign and malicious samples in experimented datasets

The collected samples were used to form two distinct and balanced datasets: the *APKComboAndrozoo* dataset and the *APKComboDrebin* dataset. As their names imply, the *APKComboAndrozoo* dataset comprises malicious samples sourced from *Androzoo* and benign samples obtained from the *APKCombo* repository. Conversely, the *APKCombo-Drebin* dataset consists of malicious samples selected from the original *Drebin* dataset and benign samples sourced from *APKCombo*. Given that the malicious samples in the *Drebin* dataset date back to 2012, while those from *Androzoo* are from 2020 onwards, analyzing these datasets separately allows for the observation and comparison of malware evolution over different periods. This comparative analysis provides valuable insights into how malware has adapted and evolved, facilitating the refinement and enhancement of our detection methodologies accordingly. Table 4.1 offers detailed information about the two compiled datasets used in the study, highlighting the number of benign and malicious samples in each for reference.

4.2 Evaluation process

In our evaluation, we assessed the performance of several classifiers: Cascade Forest (CF), Random Forest (RF), Linear Regression (LR), Support Vector Machine (SVM), and K-Nearest Neighbors (KNN). Particularly, CF is a sophisticated ensemble learning technique that constructs multiple layers of random forests to capture intricate patterns and dependencies within the data. This cascading architecture combines the benefits of both shallow and deep learning approaches, making it particularly effective for complex tasks such as malware detection [57]. We adopted a 10-fold cross-validation methodology [25] where each classifier underwent rigorous training and testing. This method entails dividing the dataset into ten subsets, training the model on nine subsets, and validating it on the remaining one. This iterative process is repeated ten times, ensuring each subset is utilized for both training and validation, thus providing a more reliable estimate of model performance by reducing variance. To gauge the effectiveness of our models, we employed four key metrics: *Recall*, *Precision*, *Accuracy*, and *F1-Score*.

— **Recall**: measures the proportion of malware samples that correctly identified by

the model. It is defined as:

$$\text{Recall} = \frac{\text{correctly predicted malware samples}}{\text{total number of malware samples}}$$

High recall indicates that the model effectively identifies most of the malware samples.

— **Precision**: measures the proportion of malware predictions that are actually correct. It is defined as:

$$\text{Precision} = \frac{\text{correctly predicted malware samples}}{\text{correctly predicted malware} + \text{incorrectly predicted benign}}$$

High precision indicates that the model has a low false positive rate, meaning it rarely misclassifies benign samples as malware.

— **Accuracy**: measures the proportion of malware and benign samples that are predicted correctly. It is defined as:

$$\text{Accuracy} = \frac{\text{correctly predicted malware} + \text{correctly predicted benign}}{\text{total number of samples}}$$

This metric provides a general sense of how often the model is correct.

— **F1-Score**: is the harmonic mean of precision and recall, providing a balance between the two. It is particularly useful when the class distribution is imbalanced. It is defined as:

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

A high F1-Score indicates that the model has both high precision and high recall, making it effective in detecting malware while minimizing false positives.

4.3 Permission-based detection

To assess the efficacy of permission-based features in detecting Android malware, we evaluated five distinct machine learning classifiers: Cascade Forest (CF), Random Forest (RF), Linear Regression (LR), Support Vector Machine (SVM), and K-Nearest Neighbors (KNN). Employing 10-fold cross-validation across all models ensured the robustness and generalizability of the results. Table 4.2 provides a summary of performance metrics for each model on both datasets, encompassing recall (R), precision (P), accuracy (A), and F1-score (F1).

For the *APKComboDrebin* dataset, *Cascade Forest* achieved the best result with an F1-score of 97.17%, indicating high precision and accuracy in distinguishing between malicious and benign samples. *Random Forest* closely followed with an F1-score of 97.10%. SVM, Logistic Regression, and KNN also performed well but did not surpass the ensemble methods. For the *APKComboAndrozoo* dataset, the *Random Forest* model outperformed

Dataset	Model	R (%)	P (%)	A (%)	F1 (%)
APKComboDrebin	**CF**	98.36	**96.05**	**97.13**	**97.17**
	RF	**98.91**	95.38	97.04	97.10
	LR	96.63	90.94	93.48	93.69
	SVM	97.17	96.20	96.63	96.66
	KNN	97.81	91.60	94.39	94.59
APKComboAndrozoo	**CF**	84.67	**87.71**	86.32	86.05
	RF	**86.25**	87.05	**86.61**	**86.55**
	LR	74.07	84.18	79.95	78.69
	SVM	85.53	86.51	85.90	85.89
	KNN	84.25	83.92	83.82	83.93

Table 4.2 – Performance comparison on permission-based detection

the others, achieving an F1-score of 86.55%, demonstrating its robustness in handling the diverse characteristics of this dataset. However, *Cascade Forest* was not far behind, with an F1-score of 86.05%, showcasing its strong potential in this context as well. The performance of Logistic Regression, SVM, and KNN was comparatively lower, indicating that ensemble methods are more effective for this datasets.

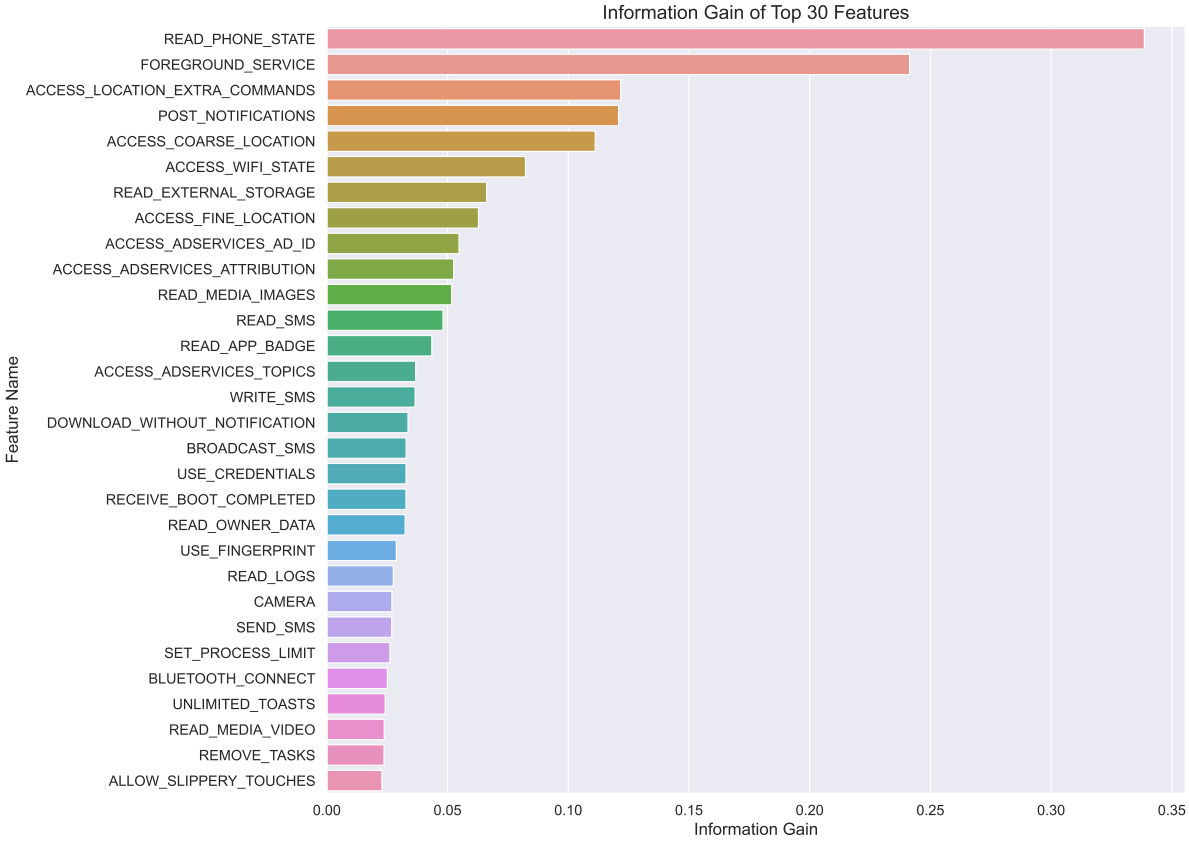


Figure 4.1 – Top 30 important permissions in the *APKComboDrebin* dataset.

Overall, the results presented in Table 4.2 highlight the effectiveness of ensemble methods, notably *Cascade Forest* and *Random Forest*, in harnessing permission-based features for Android malware detection. The high F1-scores attained by these models

underscore their potential as reliable tools for cybersecurity applications. However, the disparities in detection performance observed between the two datasets, *APKComboDrebin* and *APKComboAndrozoo*, suggest a shift in tactics among recent malware samples obtained from *APKComboAndrozoo*. This evolution in tactics is evidenced by variations in the types of permissions sought by malicious applications over time. Further analysis is warranted to discern and adapt to these evolving trends, ensuring the ongoing effectiveness of our detection methodologies in combatting emerging malware threats.

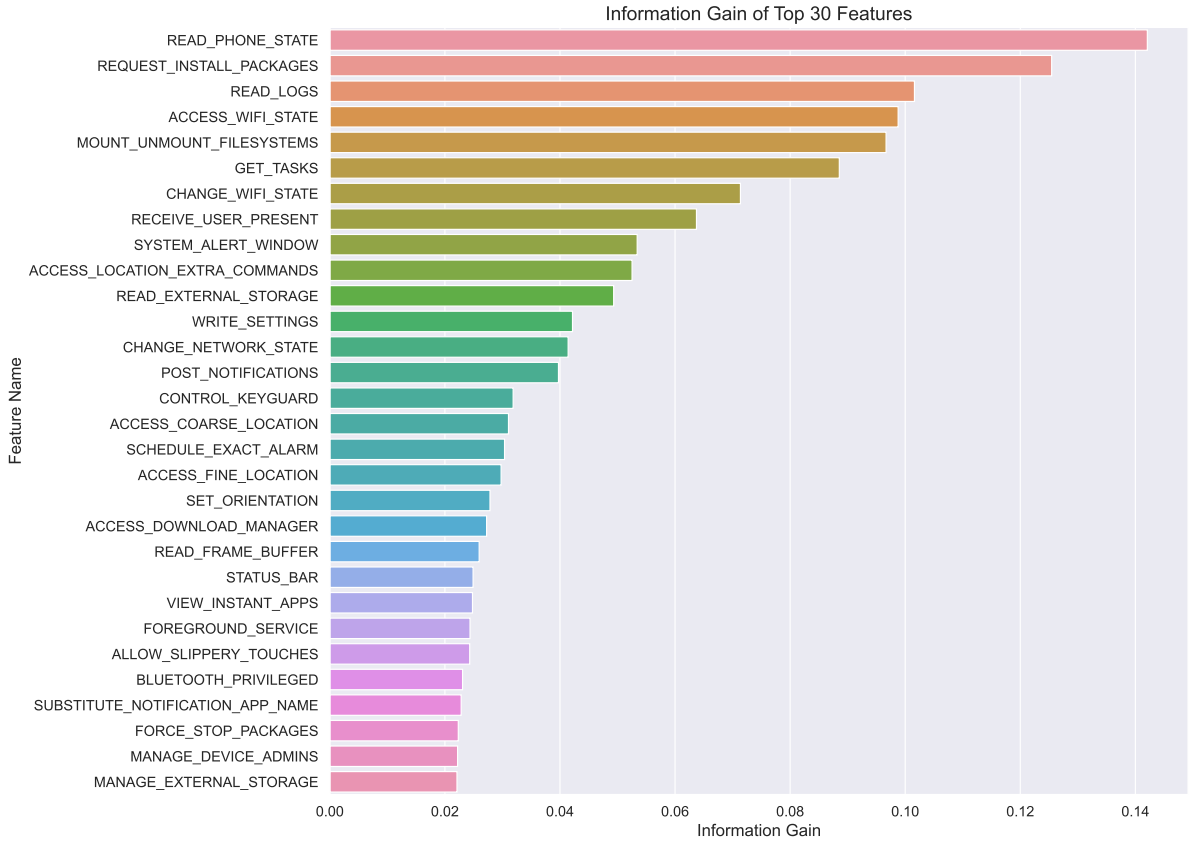


Figure 4.2 – Top 30 important permissions in the *APKComboAndrozoo* dataset.

Given that not all permissions are equally informative for distinguishing between benign and malicious applicationssome permissions are rarely requested and do not offer significant information for malware detectionwe used information gain [12] to gain insights into the most critical permissions for discerning malicious from benign APK files. Information gain quantifies the reduction in entropy, or uncertainty, achieved by understanding the value of a particular feature. In this context, it measures how much knowledge of a specific permission enhances the prediction of whether an application is benign or malicious. By computing the information gain for each permission, we can pinpoint and retain only those permissions that make the most substantial contribution to our model's predictive capabilities. Figure 4.1 displays the top 30 significant features based on information gain

for the *APKComboDrebin* dataset, while Figure 4.2 presents the top 30 important features based on information gain for the *APKComboAndrozoo* dataset.

The comparison of the lists of permissions provided in Figure 4.1 and Figure 4.2, respectively, vividly illustrates the evolution of tactics in permission usage over the years. The *READ_PHONE_STATE* permission's sustained prominence, albeit with a significant decrease in information gain from 34% to approximately 14% in newer samples, underscores its enduring relevance for malicious actors. This permission likely remains attractive due to its access to sensitive device information, such as device identifiers and network connectivity status, enabling various nefarious activities, including identity theft and targeted attacks. Additionally, the emergence of permissions like *REQUEST_INSTALL_PACKAGES* in the *APKComboAndrozoo* dataset highlights the shifting strategies adopted by malware developers. This permission, which facilitates the installation of applications without user consent, reflects a growing trend towards more stealthy and aggressive tactics. Malware strains leveraging this permission can install additional malicious software or unwanted applications, expanding their foothold on infected devices and enhancing their persistence.

4.4 Opcode-based detection

For opcode-based detection, we use our two collected datasets (*APKComboDrebin* and *APKComboAndrozoo*) to train and test the same five machine learning classifiers used in the permission-based detection. Our experimentation involved using Term Frequency-Inverse Document Frequency (TF-IDF) to convert opcode sequences into numerical representations. We tested different n-grams ranging from 1-1 to 3-3 to identify the best configuration that captures the most relevant patterns in the data with a default vector size set to 256. After determining the optimal n-gram configuration, we further explored the impact of varying vector sizes on the performance of our models. The vector sizes tested ranged from 64 to 512. This step aimed to understand how the dimensionality of the feature space affects the classifiers' ability to distinguish between benign and malicious applications.

To ensure the reliability and generalizability of our results, we employed a 10-fold cross-validation approach. Through this thorough evaluation process, we aimed to determine the most effective combination of n-gram configuration and vector size for opcode-based malware detection. The use of two different datasets also helped in assessing the consistency and adaptability of our models across varied samples, ensuring that our findings are robust and applicable to real-world scenarios. Table 4.3 presents the results of our experiments using different n-gram configurations for opcode sequences. Meanwhile, Table 4.4 displays the performance outcomes for various vector sizes, based on the optimal n-gram configuration identified from Table 4.3.

The results in Table 4.3 highlight the impact of using different n-grams on the performance of our classifiers. For the *APKComboDrebin* dataset, CF achieved the highest

N-gram	Model	APKComboDrebin dataset				APKComboAndrozoo dataset			
		R (%)	P (%)	A (%)	F1 (%)	R (%)	P (%)	A (%)	F1 (%)
1	**CF**	99,18	98,64	98,91	98,91	**93.14**	97.62	**95.43**	**95.31**
	RF	99,36	97,68	98,50	98,51	92.00	97.90	95.00	94.83
	LR	99,09	97,59	98,31	98,33	86.43	95.25	91.00	90.54
	SVM	98,91	95,96	97,36	97,40	78.00	96.39	87.50	86.11
	KNN	99,36	96,82	98,04	98,07	89.57	92.40	91.00	90.87
2	**CF**	99,45	**98,91**	99,18	99,18	92.00	98.18	95.14	94.96
	RF	99,73	98,13	98,91	98,92	91.29	**98.62**	95.00	94.77
	LR	99,27	98,47	98,86	98,87	92.14	92.80	92.43	92.40
	SVM	98,63	96,61	97,58	97,61	77.43	96.21	87.14	85.66
	KNN	99,09	97,41	98,22	98,24	89.14	93.64	91.43	91.19
3	**CF**	99,64	98,83	**99,23**	**99,23**	92.00	98.04	95.07	94.87
	RF	**99,73**	98,12	98,91	98,92	90.71	97.74	94.29	94.04
	LR	99,64	98,39	99,00	99,01	91.71	93.41	92.57	92.50
	SVM	99,09	96,97	98,00	98,02	77.14	96.39	87.07	85.56
	KNN	99,36	97,50	98,41	98,42	89.14	93.10	91.21	90.98
1-2	**CF**	99,45	98,73	99,09	99,09	92.29	97.63	95.00	94.85
	RF	99,64	98,21	98,91	98,92	91.14	98.04	94.64	94.42
	LR	99,36	98,13	98,72	98,73	91.57	93.21	92.43	92.37
	SVM	99,00	95,95	97,40	97,44	77.71	95.58	87.00	85.59
	KNN	99,36	97,16	98,22	98,24	89.43	93.16	91.36	91.17
1-3	**CF**	99,36	98,64	99,00	99,00	92.29	97.89	95.14	94.97
	RF	99,64	98,30	98,95	98,96	91.57	97.22	94.43	94.24
	LR	99,45	98,12	98,77	98,78	91.43	94.39	92.93	92.83
	SVM	99,00	96,12	97,49	97,53	77.71	95.75	87.07	85.66
	KNN	99,45	97,60	98,50	98,51	89.00	92.70	90.93	90.73

Table 4.3 – Opcode-based performance results using different n-grams

F1-score and accuracy, both at 99.23%, with 3-grams. CF also delivered the best precision score of 98.91% with 2-grams. Meanwhile, RF excelled in recall, reaching 99.73% with 3-grams. With 1-gram, CF still performed commendably, securing the fourth-best F1-score and accuracy at 98.91%. In the *APKComboAndrozoo* dataset, CF led in F1-score, accuracy, and recall, achieving 95.31%, 95.43%, and 93.14% respectively with 1-grams. RF, on the other hand, achieved the highest precision at 98.62% with 2-grams. When using 3-grams, CF maintained strong performance, with the third-best F1-score of 94.87%, accuracy of 95.07%, the best precision at 98.04%, and the fourth-best recall score at 92%. These findings indicate that 3-grams are particularly effective for the *APKComboDrebin* dataset, especially for achieving high F1-scores and accuracy with CF, while 1-grams are more suitable for the *APKComboAndrozoo* dataset to maximize F1-scores, accuracy, and recall.

This suggests that the choice of n-grams should be tailored to the specific dataset to optimize the performance of malware detection classifiers. To develop a machine learning model that performs well across both the *APKComboDrebin* and *APKComboAndrozoo* datasets, it's essential to prioritize a configuration that balances high performance on both datasets. Accordingly, using CF with 2-grams strikes a balance, offering excellent precision on both datasets (98.91% and 98.18% respectively) while maintaining robust performance across all the remaining key metrics. This configuration leverages the strengths

V. size	Model	APKComboDrebin dataset				APKComboAndrozoo dataset			
		R (%)	P (%)	A (%)	F1 (%)	R (%)	P (%)	A (%)	F1 (%)
512	**CF**	99.69	**99.20**	**99.44**	**99.44**	92.84	97.48	**95.20**	**95.06**
	RF	99.69	98.35	99.01	99.01	91.84	97.71	94.84	94.64
	LR	98.51	97.79	98.14	98.14	**94.27**	87.86	90.54	90.91
	SVM	99.25	95.64	97.36	97.41	77.38	96.22	87.11	85.57
	KNN	99.25	97.21	98.20	98.22	88.55	92.74	90.76	90.49
256	**CF**	99,45	98,91	99,18	99,18	92.00	98.18	95.14	94.96
	RF	**99,73**	98,13	98,91	98,92	91.29	98.62	95.00	94.77
	LR	99,27	98,47	98,86	98,87	92.14	92.80	92.43	92.40
	SVM	98,63	96,61	97,58	97,61	77.43	96.21	87.14	85.66
	KNN	99,09	97,41	98,22	98,24	89.14	93.64	91.43	91.19
128	**CF**	99.38	98.95	99.16	99.16	91.41	98.05	94.77	94.55
	RF	99.50	98.17	98.82	98.83	89.98	98.28	94.19	93.85
	LR	99.13	97.34	98.20	98.22	89.26	94.94	92.19	91.92
	SVM	98.76	95.50	97.05	97.10	77.24	96.06	86.96	85.42
	KNN	99.13	97.09	98.07	98.10	88.12	92.20	90.26	89.99
64	**CF**	99.25	98.41	98.82	98.83	90.12	97.84	94.05	93.75
	RF	99.44	97.63	98.51	98.52	90.55	96.95	93.84	93.59
	LR	98.88	95.68	97.20	97.25	84.96	95.57	90.47	89.88
	SVM	98.63	95.33	96.89	96.95	77.38	96.22	87.11	85.57
	KNN	99.07	97.14	98.07	98.09	87.69	91.73	89.83	89.55
32	**CF**	99.07	98.22	98.63	98.64	89.97	**98.48**	94.27	93.97
	RF	99.50	97.69	98.57	98.59	89.83	96.94	93.48	93.18
	LR	97.95	94.84	96.30	96.37	80.53	93.71	87.46	86.42
	SVM	98.39	95.16	96.68	96.74	77.24	96.21	87.03	85.48
	KNN	98.63	96.72	97.64	97.66	87.54	91.94	89.90	89.61

Table 4.4 – Opcode-based performance results using 2-gram and varying the vector size.

observed in both datasets, ensuring that the model is not overly specialized to one dataset's characteristics but performs reliably across different types of malware samples.

When exploring the impact of vector sizes and the use of 2-grams, the results described in Table 4.4 indicate that increasing the vector size to 512 provides the best accuracy and F1-score for both datasets. For the *APKComboDrebin* dataset, this adjustment results in an F1-score increase of 0.21%, reaching an impressive 99.44%. Similarly, for the *APKComboAndrozoo* dataset, the F1-score improves by 0.29%, reaching 95.06%. These findings suggest that larger vector sizes, coupled with the use of 2-grams, enhance the model's ability to capture and utilize more detailed feature representations, leading to more accurate and effective classification of APK files across both datasets. These findings underscore the potential of the CF classifier in leveraging opcode-based features for robust and accurate Android malware detection, showcasing its superior performance across different datasets and TF-IDF vector sizes.

4.5 Visualization-based detection

For the visualization-based model, we used the same datasets and machine learning models mentioned in the previous sections. Initially, we generated squared grayscale images from

the datasets for individual and combined files with 512x512 size each. Figure 4.3 shows an example of those images for two different applications, benign and malware.

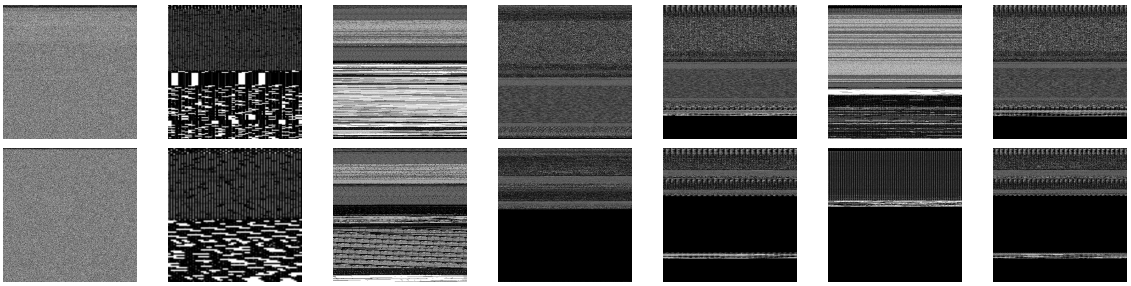


Figure 4.3 – Grayscale images generated from individual and combined files. First row represent images of a benign app while the second row represents the images generated for a malware app. A: *classes.dex*, B: *AndroidManifest.xml*, C: *resources.arsc*

4.5.1 Feature Extraction and Classifier Evaluation

We evaluated the performance of four local feature descriptor algorithmsSIFT, AGAST, MSER, and FASTby by extracting features from the generated images and then feeding these features into five different classifiers. Tables 4.5-4.11 present the results of these classifiers for the *APKComboDrebin* and *APKComboAndrozoo* datasets, using individual and combined assets of APK files: *classes.dex*, *AndroidManifest.xml*, and *resources.arsc*. The best performance scores for each scenario are highlighted in bold.

When analyzing the performance of various feature descriptors and classifiers across the two experimented datasets, several key insights emerge. Considering only *classes.dex* files within the *APKComboDrebin* dataset, the SIFT descriptor combined with the SVM model achieved the highest F1-score of 95.24%. In contrast, for the *APKComboAndrozoo* dataset, the best F1-score of 94.64% was obtained using the FAST descriptor with the CF model. A configuration that performs well across both datasets is the CF classifier with the AGAST descriptor, which provided an F1-score of 94.63% on the *APKComboAndrozoo* dataset and a close performance of 94.84% on the *APKComboDrebin* dataset. For *resources.arsc* files, lower scores were observed across both datasets. The *APKComboDrebin* dataset saw its highest F1-score of 86.38% with the SIFT descriptor and RF model. In the *APKComboAndrozoo* dataset, the best F1-score was 80.60% using AGAST and RF. The most balanced approach for both datasets was AGAST with CF, achieving the second-best score of 84.49% for *APKComboDrebin* and the third-best score of 78.88% for *APKComboAndrozoo*.

Desc.	Model	APKComboDrebin dataset				APKComboAndrozoo dataset			
		R (%)	P (%)	A (%)	F1 (%)	R (%)	P (%)	A (%)	F1 (%)
SIFT	**CF**	96.90	93.55	95.08	95.17	88.54	93.07	90.90	90.67
	RF	96.62	93.39	94.85	94.95	84.96	92.57	89.04	88.57
	LR	95.62	91.61	93.39	93.55	77.79	87.07	83.09	82.11
	SVM	96.62	**93.93**	**95.17**	**95.24**	83.24	92.16	88.04	87.42
	KNN	96.90	91.52	93.94	94.12	86.67	88.19	87.46	87.32
AGAST	**CF**	96.99	92.83	94.71	94.84	**94.55**	**94.78**	**94.63**	94.63
	RF	96.26	92.55	94.21	94.34	93.84	94.84	94.34	94.32
	LR	97.72	91.52	94.30	94.50	90.24	93.99	92.19	92.02
	SVM	97.81	91.06	94.07	94.30	92.55	94.34	93.48	93.42
	KNN	97.54	91.38	94.12	94.33	91.41	92.19	91.83	91.78
MSER	**CF**	95.53	91.38	93.20	93.37	92.69	94.38	93.55	93.50
	RF	95.26	91.69	93.25	93.40	92.27	93.82	93.05	93.01
	LR	94.25	90.65	92.20	92.37	90.53	91.67	91.11	91.06
	SVM	95.53	91.00	92.98	93.17	92.40	94.54	93.48	93.43
	KNN	95.17	89.14	91.75	92.04	91.27	91.99	91.62	91.60
FAST	**CF**	97.35	92.37	94.62	94.77	**94.55**	**94.78**	**94.63**	**94.64**
	RF	97.08	92.37	94.49	94.64	94.41	**94.78**	94.55	94.56
	LR	**98.09**	92.48	95.03	95.19	88.53	93.28	91.04	90.80
	SVM	97.72	91.15	94.07	94.30	90.97	94.59	92.84	92.70
	KNN	97.44	91.42	94.12	94.32	92.84	93.40	93.12	93.09

Table 4.5 – Visualization-based performance results using *classes.dex* files.

Desc.	Model	APKComboDrebin dataset				APKComboAndrozoo dataset			
		R (%)	P (%)	A (%)	F1 (%)	R (%)	P (%)	A (%)	F1 (%)
SIFT	**CF**	91.07	91.37	91.20	91.16	70.35	**83.64**	78.15	**76.15**
	RF	**93.17**	87.33	89.70	90.07	69.77	80.80	76.50	74.72
	LR	88.52	89.88	89.24	89.14	72.21	76.94	75.29	74.44
	SVM	92.53	90.63	91.43	91.52	73.21	82.27	**78.65**	77.40
	KNN	84.42	90.80	87.88	87.39	69.64	77.19	74.50	73.11
AGAST	**CF**	90.80	93.26	92.07	91.93	71.48	77.61	75.36	74.32
	RF	90.16	92.55	91.43	91.27	72.20	76.52	74.93	74.21
	LR	90.97	90.40	90.66	90.66	71.62	73.80	72.93	72.60
	SVM	91.43	**94.20**	**92.89**	**92.76**	68.19	77.94	74.36	72.68
	KNN	90.43	92.82	91.71	91.56	71.05	71.41	71.21	71.16
MSER	**CF**	87.43	92.13	89.97	89.64	68.91	73.60	71.99	71.08
	RF	88.33	90.00	89.24	89.10	68.05	70.81	69.91	69.35
	LR	87.60	89.25	88.47	88.33	67.76	69.03	68.41	68.25
	SVM	85.96	91.44	88.93	88.53	65.62	72.04	69.92	68.55
	KNN	86.42	89.34	87.97	87.76	70.49	66.78	67.62	68.49
FAST	**CF**	90.43	94.00	92.30	92.12	73.78	78.12	76.44	75.79
	RF	89.61	92.53	91.16	90.99	73.93	77.92	76.44	75.82
	LR	89.89	91.01	90.47	90.40	**75.21**	75.44	75.29	75.26
	SVM	90.52	93.13	91.89	91.74	71.63	79.64	76.58	75.36
	KNN	89.52	91.95	90.80	90.64	71.91	74.43	73.49	73.09

Table 4.7 – Visualization-based performance results using *AndroidManifest.xml* files.

Desc.	Model	APKComboDrebin dataset				APKComboAndrozoo dataset			
		R (%)	P (%)	A (%)	F1 (%)	R (%)	P (%)	A (%)	F1 (%)
SIFT	**CF**	**91.14**	82.57	95.75	86.29	78.37	76.81	77.36	77.56
	RF	89.04	**84.29**	**95.84**	**86.38**	77.51	76.00	76.50	76.71
	LR	88.33	78.42	94.55	82.57	72.93	73.06	72.99	72.92
	SVM	90.15	80.76	95.28	84.83	79.09	76.02	77.00	77.46
	KNN	73.77	83.18	93.83	77.70	57.16	74.23	68.62	64.34
AGAST	**CF**	88.46	81.64	95.46	84.49	78.91	79.18	78.91	78.88
	RF	86.76	82.17	95.37	84.05	80.76	**80.75**	**80.63**	**80.60**
	LR	89.08	76.74	94.32	81.92	79.92	76.52	77.62	78.10
	SVM	87.42	76.98	94.24	81.10	80.63	76.72	77.91	78.49
	KNN	76.11	73.45	93.02	73.66	78.90	75.99	76.90	77.31
MSER	**CF**	90.56	78.34	94.91	83.91	77.36	75.35	75.93	76.26
	RF	91.67	79.98	95.40	85.30	74.35	74.70	74.43	74.43
	LR	80.98	70.68	92.32	75.31	80.37	69.01	71.99	74.18
	SVM	79.90	74.09	92.89	76.57	81.81	71.58	74.36	76.19
	KNN	72.68	73.14	92.16	72.23	76.51	69.55	71.42	72.79
FAST	**CF**	87.32	81.03	95.21	83.85	79.06	79.62	79.27	79.16
	RF	83.89	82.94	95.13	83.00	78.90	79.15	78.91	78.88
	LR	90.20	73.62	93.91	80.72	79.05	74.28	75.75	76.49
	SVM	85.56	79.10	94.64	81.66	**80.91**	76.82	77.98	78.58
	KNN	73.20	76.74	93.34	73.76	79.62	76.21	77.26	77.72

Table 4.6 – Visualization-based performance results using *resources.arsc* files.

When focusing on *AndroidManifest.xml* files, the overall scores were lower for the *APKComboAndrozoo* dataset, with the highest F1-score of 76.15% achieved using the SIFT descriptor and CF classifier. For the *APKComboDrebin* dataset, AGAST with SVM performed better, achieving an F1-score of 92.76%. The optimal configuration here appears to be FAST with CF, yielding the second-best F1-score of 92.12% for *APKComboDrebin* and the third-best F1-score of 75.79% for *APKComboAndrozoo*.

4.5.2 Combination Analysis

When combining *classes.dex* and *AndroidManifest.xml* files, the *APKComboDrebin* dataset's best score slightly increased to 95.56% using SIFT and CF, while the *APKComboAndrozoo* dataset's best score increased to 94.98% with FAST and CF. The most balanced configuration maintaining the highest scores for both datasets is using AGAST with CF achieving the second highest F1-score 94.49% for the *APKComboAndrozoo* dataset and a reasonably high F1-score 94.81% for the *APKComboDrebin* dataset. Combining *classes.dex* and *resources.arsc* files yielded a slight improvement for *APKComboDrebin*, reaching 96.59% with SIFT and CF, but no improvement for *APKComboAndrozoo*.

Desc.	Model	APKComboDrebin dataset				APKComboAndrozoo dataset			
		R (%)	P (%)	A (%)	F1 (%)	R (%)	P (%)	A (%)	F1 (%)
SIFT	**CF**	96.90	**94.27**	**95.49**	**95.56**	88.39	93.40	91.04	90.77
	RF	**97.80**	91.99	94.62	94.79	84.52	92.07	88.61	88.07
	LR	96.90	91.99	94.21	94.37	77.65	88.43	83.74	82.63
	SVM	97.35	92.81	94.89	95.02	82.95	93.40	88.53	87.78
	KNN	96.26	92.04	93.94	94.08	87.10	90.25	88.82	88.59
AGAST	**CF**	97.35	92.43	94.67	94.81	94.27	94.76	94.48	94.49
	RF	96.99	92.61	94.58	94.71	93.84	94.85	94.34	94.32
	LR	97.35	91.42	94.07	94.27	89.82	94.22	92.12	91.92
	SVM	97.63	91.21	94.07	94.29	92.83	94.55	93.70	93.65
	KNN	97.08	90.82	93.57	93.80	92.70	93.65	93.19	93.15
MSER	**CF**	96.17	91.40	93.52	93.70	93.12	94.30	93.69	93.67
	RF	96.08	91.37	93.48	93.64	91.41	94.63	93.05	92.95
	LR	95.80	90.12	92.61	92.85	88.26	92.45	90.47	90.24
	SVM	96.80	91.00	93.57	93.78	91.11	93.92	92.55	92.43
	KNN	95.62	90.38	92.66	92.89	90.40	91.29	90.83	90.80
FAST	**CF**	97.08	93.08	94.90	95.01	**94.84**	**95.17**	**94.98**	**94.98**
	RF	97.08	93.15	94.94	95.05	94.13	94.85	94.48	94.47
	LR	97.63	91.85	94.44	94.62	89.82	93.61	91.83	91.64
	SVM	97.45	91.67	94.26	94.44	91.83	94.91	93.41	93.31
	KNN	97.17	90.74	93.57	93.81	92.98	93.17	93.05	93.05

Table 4.8 – Visualization-based performance results using *classes.dex* and *AndroidManifest.xml*.

Desc.	Model	APKComboDrebin dataset				APKComboAndrozoo dataset			
		R (%)	P (%)	A (%)	F1 (%)	R (%)	P (%)	A (%)	F1 (%)
SIFT	**CF**	90.04	92.67	97.18	91.20	79.75	77.96	78.54	78.79
	RF	91.00	92.90	97.34	91.76	80.76	78.00	78.90	79.31
	LR	89.57	94.15	97.34	91.59	77.89	74.37	75.46	76.00
	SVM	91.45	93.44	97.42	92.13	80.47	77.43	78.47	78.84
	KNN	91.93	92.72	97.42	92.11	74.17	78.51	76.90	76.18
AGAST	**CF**	98.00	97.09	98.03	97.53	80.37	**82.47**	**81.59**	**81.31**
	RF	**98.43**	96.96	**98.15**	**97.67**	78.94	80.62	79.88	79.69
	LR	97.72	95.72	97.36	96.69	79.08	75.93	76.94	77.43
	SVM	97.29	96.39	97.47	96.81	79.22	76.51	77.37	77.77
	KNN	97.29	95.71	97.19	96.46	75.79	76.08	75.94	75.88
MSER	**CF**	91.05	92.26	97.46	91.47	78.22	78.52	78.37	78.30
	RF	91.58	93.75	97.78	92.46	79.51	77.31	78.01	78.34
	LR	86.26	92.40	96.82	88.82	**81.37**	71.02	74.00	75.79
	SVM	87.89	93.66	97.22	90.27	79.66	75.20	76.65	77.33
	KNN	85.76	93.97	96.98	89.48	75.21	73.63	74.00	74.32
FAST	**CF**	97.57	**97.20**	97.92	97.38	78.52	80.55	79.66	79.45
	RF	98.29	96.66	97.98	97.46	79.37	79.93	79.66	79.61
	LR	96.29	96.07	96.97	96.15	79.51	74.83	76.29	77.05
	SVM	97.14	96.89	97.64	96.99	80.07	76.55	77.73	78.21
	KNN	97.86	96.25	97.64	97.03	75.93	76.30	76.08	76.04

Table 4.10 – Visualization-based performance results using *AndroidManifest.xml* and *resources.arsc* files

Desc.	Model	APKComboDrebin dataset				APKComboAndrozoo dataset			
		R (%)	P (%)	A (%)	F1 (%)	R (%)	P (%)	A (%)	F1 (%)
SIFT	**CF**	96.99	**96.25**	**96.58**	**96.59**	93.41	93.94	93.62	93.62
	RF	97.36	95.57	96.39	96.43	90.69	93.56	92.19	92.05
	LR	97.45	94.90	96.08	96.14	90.41	91.55	90.97	90.95
	SVM	**98.27**	94.91	96.49	96.55	91.55	93.51	92.55	92.45
	KNN	96.90	95.01	95.89	95.93	91.39	93.57	92.47	92.36
AGAST	**CF**	95.99	95.51	95.71	95.72	**94.41**	94.21	94.27	94.28
	RF	96.53	95.61	96.03	96.05	93.99	94.04	93.98	93.97
	LR	96.63	94.12	95.26	95.33	90.98	93.70	92.41	92.30
	SVM	96.72	94.86	95.71	95.76	86.82	93.85	90.54	90.13
	KNN	96.63	94.17	95.30	95.37	90.12	91.10	90.62	90.55
MSER	**CF**	97.54	95.07	96.21	96.27	94.12	**94.84**	**94.48**	**94.46**
	RF	97.26	95.48	96.30	96.35	93.98	93.76	93.84	93.85
	LR	98.18	94.05	95.94	96.05	92.41	91.81	92.05	92.08
	SVM	97.99	94.95	96.35	96.43	94.13	92.81	93.41	93.45
	KNN	97.54	94.90	96.12	96.18	93.42	90.70	91.90	92.02
FAST	**CF**	96.62	95.89	96.21	96.23	92.98	94.37	93.69	93.65
	RF	97.08	95.55	96.26	96.29	93.41	93.87	93.62	93.62
	LR	96.54	93.77	95.03	95.11	90.83	93.09	92.05	91.93
	SVM	97.45	94.58	95.90	95.97	87.40	94.31	91.04	90.66
	KNN	96.81	94.48	95.53	95.60	90.12	92.09	91.19	91.05

Table 4.9 – Visualization-based performance results using *classes.dex* and *resources.arsc*.

Desc.	Model	APKComboDrebin dataset				APKComboAndrozoo dataset			
		R (%)	P (%)	A (%)	F1 (%)	R (%)	P (%)	A (%)	F1 (%)
SIFT	**CF**	97.81	96.20	96.94	96.98	93.55	**94.84**	94.20	94.16
	RF	97.90	95.77	96.76	96.81	90.55	93.67	92.19	92.02
	LR	97.72	95.50	96.53	96.58	90.54	91.29	90.90	90.88
	SVM	98.09	95.59	96.76	96.81	92.11	94.23	93.19	93.10
	KNN	97.72	95.84	96.71	96.76	92.41	93.57	92.98	92.93
AGAST	**CF**	97.17	96.17	96.63	96.65	94.27	94.44	94.34	94.34
	RF	97.72	95.41	96.49	96.53	94.13	94.47	94.27	94.27
	LR	97.63	94.14	95.76	95.84	90.54	93.07	91.83	91.72
	SVM	98.17	94.06	95.94	96.05	88.25	94.78	91.69	91.36
	KNN	97.72	93.86	95.62	95.72	89.54	91.78	90.76	90.63
MSER	**CF**	97.90	**96.45**	97.13	97.16	94.12	94.43	94.27	94.26
	RF	97.90	**96.45**	97.13	97.16	94.27	94.57	94.41	94.41
	LR	**98.54**	95.40	96.85	96.92	92.26	92.42	92.33	92.33
	SVM	98.45	96.15	**97.22**	**97.27**	92.84	94.07	93.48	93.44
	KNN	97.99	95.98	96.90	96.95	91.69	91.41	91.47	91.50
FAST	**CF**	97.54	94.90	96.13	96.19	94.27	94.72	**94.48**	**94.48**
	RF	97.63	95.08	96.26	96.32	**94.55**	94.35	94.41	94.43
	LR	97.63	94.23	95.81	95.89	90.97	92.32	91.69	91.61
	SVM	97.44	94.66	95.94	96.01	88.11	93.91	91.19	90.85
	KNN	97.90	93.61	95.58	95.69	89.69	91.94	90.90	90.78

Table 4.11 – Visualization-based performance results using *classes.dex*, *AndroidManifest.xml* and *resources.arsc* files.

Notably, combining *classes.dex* and *resources.arsc* files for the *APKComboDrebin* dataset significantly boosted the F1-score to 97.67% using AGAST and RF, though the *APKComboAndrozoo* dataset only reached 81.31%. Finally, integrating all three file types, the *APKComboDrebin* dataset achieved its second-highest F1-score of 97.27% with MSER and SVM, while the *APKComboAndrozoo* dataset maintained the same score of 94.48% as when combining *classes.dex* and *AndroidManifest.xml*. A balanced configuration using MSER and RF yielded the second-best score of 97.17% for *APKComboDrebin* and the third-best score of 94.41% for *APKComboAndrozoo*.

4.5.3 Performance enhancement via interpolation

After identifying the best Descriptor-Classifier combination, we explored the potential for performance enhancement by applying different resize interpolation techniques beyond the default *INTER_LINEAR* method. To this end, we experimented with three other interpolation techniques available in OpenCV: *INTER_CUBIC*, *INTER_NEAREST*, and *INTER_LANCZOS4*. The goal was to determine whether these alternative resizing methods could further optimize the feature extraction process and subsequently improve classification performance. Each technique offers a different approach to resizing: *INTER_CUBIC* uses a cubic convolution for higher quality at the cost of processing time, *INTER_NEAREST* employs nearest-neighbor interpolation for faster but potentially less accurate resizing, and *INTER_LANCZOS4* uses a high-quality Lanczos filter to achieve better accuracy for resizing, especially in downscaling scenarios. By comparing the performance outcomes of these methods, we aimed to identify the optimal resizing strategy that enhances the classifier's ability to accurately detect and classify features within the APK files, ultimately improving the overall effectiveness of the Descriptor-Classifier combination.

Interpolation	**APKComboDrebin dataset**				**APKComboAndrozoo dataset**			
	R (%)	**P (%)**	**A (%)**	**F1 (%)**	**R (%)**	**P (%)**	**A (%)**	**F1 (%)**
Cubic	97.54	96.09	96.76	96.80	93.84	93.46	93.62	93.63
Nearest	97.99	95.52	96.67	96.72	89.55	93.02	91.40	91.23
Lanczos4	98.18	96.03	97.04	97.08	94.55	93.65	94.05	94.08
Linear	**97.90**	**96.45**	**97.13**	**97.16**	**94.27**	**94.57**	**94.41**	**94.41**

Table 4.12 – Performance using different interpolation methods using MSER-RF.

The results of the experiment are shown in Table 4.12, which illustrates that no significant improvement was achieved over the default *INTER_LINEAR* method. However, the *INTER_LANCZOS4* method demonstrated performance that closely matched the results obtained with *INTER_LINEAR*. Specifically, while *INTER_LINEAR* remained the best performing interpolation technique for most metrics, *INTER_LANCZOS4* showed comparable accuracy and F1-scores, suggesting it as a viable alternative for scenarios where

a higher-quality resize might be beneficial without compromising performance. On the other hand, *INTER_CUBIC* and *INTER_NEAREST* did not yield competitive results, indicating that their interpolation approaches were either too computationally intensive without corresponding accuracy gains (*INTER_CUBIC*) or too simplistic to maintain feature integrity (*INTER_NEAREST*). This experiment underscores the robustness of *INTER_LINEAR* for resizing in feature extraction but also highlights *INTER_LANCZOS4* as a close alternative, potentially useful in specific contexts where its nuanced interpolation might offer marginal benefits.

Size	**APKComboDrebin dataset**				**APKComboAndrozoo dataset**			
	R (%)	**P (%)**	**A (%)**	**F1 (%)**	**R (%)**	**P (%)**	**A (%)**	**F1 (%)**
64	92.00	86.33	93.46	89.01	79.94	83.79	82.23	81.78
128	96.81	90.46	93.25	93.50	83.54	89.94	87.03	86.49
256	**98.30**	93.94	95.98	96.04	85.89	91.64	88.93	88.58
512	97.90	**96.45**	**97.13**	**97.16**	**94.27**	**94.57**	**94.41**	**94.41**

Table 4.13 – Performance using different image sizes within MSER-RF configuration.

After identifying the best interpolation technique, we tested our optimal Descriptor-Classifier configuration (MSER-RF) with various image sizes ranging from 64x64 to 512x512 pixels. The findings, as presented in Table 4.13, highlight a clear trend: as the image size increases, the performance of the models improves. The best results were obtained with the 512x512 pixel size, where the increased resolution allowed for more detailed feature extraction, enhancing the classifier's accuracy and F1-score. Although we attempted to use images sized at 1024x1024 pixels, our hardware was unable to process them due to memory constraints and computational limits. This experiment underscores the importance of image resolution in model performance, indicating that larger image sizes can significantly enhance the quality of feature extraction and, consequently, the effectiveness of classification, provided the hardware capabilities are sufficient to handle the increased computational load.

4.6 Mixed Apporoach-based detection

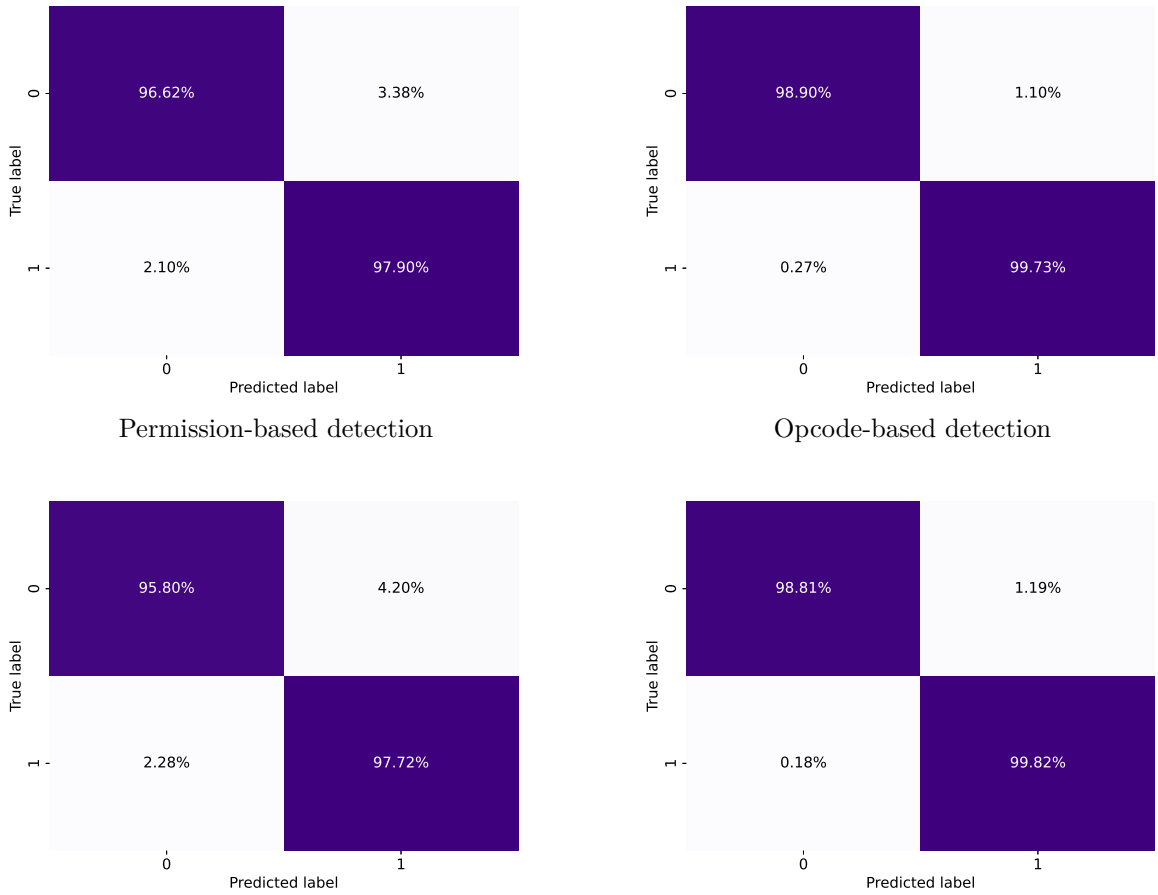
Table 4.14 showcases the performance metrics of various mixed models across the *APKComboDrebin* and *APKComboAndrozoo* datasets. Among these models, the voting ensemble method emerges as the top performer, achieving the highest F1-score on both datasets. Specifically, voting attains an impressive F1-score of 99.82% for the *APKComboDrebin* dataset and 96.01% for the *APKComboAndrozoo* dataset, indicating its effectiveness in accurately distinguishing between malicious and benign applications. On the other hand, the stacking ensemble method with RF metamodel exhibits superior performance on the *APKComboDrebin* dataset, yielding an F1-score of 98.72%. However, when applied to the

APKComboAndrozoo dataset, the stacking approach with LR metamodel achieves an F1-score of 94.33%, which falls short compared to the performance achieved by the individual models utilizing opcodes and byte-visualization techniques. These findings underscore the efficiency of the voting ensemble technique compared to stacking for Android malware detection. The voting ensemble technique demonstrates its effectiveness by providing a notable increase in performance compared to the best-performing model obtained from previous experiments. In the *APKComboDrebin* dataset, the voting ensemble achieves an improvement of 0.38% over the best individual model (CF with opcode-based detection). Similarly, in the *APKComboAndrozoo* dataset, the voting ensemble showcases a significant enhancement of 0.95% compared to the same individual model. This improvement suggests that the ensemble method effectively leverages the diverse insights of multiple classifiers, resulting in enhanced performance in distinguishing between benign and malicious Android applications.

Mixed	**APKComboDrebin dataset**				**APKComboAndrozoo dataset**			
Model	**R (%)**	**P (%)**	**A (%)**	**F1 (%)**	**R (%)**	**P (%)**	**A (%)**	**F1 (%)**
Vote	**99.82**	**98.83**	**99.82**	**99.82**	**94.99**	97.06	**96.06**	**96.01**
Stack-CF	98.26	97.34	97.76	97.78	90.41	96.79	93.62	93.39
Stack-RF	98.63	98.82	98.72	98.72	90.69	97.14	93.98	93.76
Stack-LR	98.72	97.32	97.99	98.00	92.98	95.78	94.41	94.33
Stack-SVM	98.26	98.50	98.36	98.36	91.42	97.31	94.34	94.11
Stack-KNN	97.90	98.22	98.04	98.04	90.55	**97.64**	94.13	93.88

Table 4.14 – Performance of the mixed approach.

The comparison between individual static approaches and the mixed approach, as illustrated in Figures 4.4 and 4.5, offers valuable insights into their performance characteristics on the *APKComboDrebin* and *APKComboAndrozoo* datasets. Notably, the mixed approach exhibits a remarkable reduction in false negatives compared to individual static approaches, with only 0.18% for the *APKComboDrebin* dataset and 5.01% for the *APKComboAndrozoo* dataset. This reduction in false negatives suggests that the mixed approach effectively captures more instances of malicious applications that would have been missed by individual static approaches alone, thereby enhancing the overall detection capability. However, it's worth noting that the mixed approach shows a slight increase in false positives compared to opcode-based detection, with an increment of 0.09% in the *APKComboDrebin* dataset and 0.44% in the *APKComboAndrozoo* dataset. While false positives can lead to unnecessary alerts, this marginal increase indicates a trade-off for the improved detection of malicious applications, achieved by the mixed approach. This suggests that while the mixed approach may produce slightly more false positives, its ability to significantly reduce false negatives outweighs this drawback, resulting in a net improvement in overall detection accuracy.



Permission-based detection

Opcode-based detection

Visualization-based detection

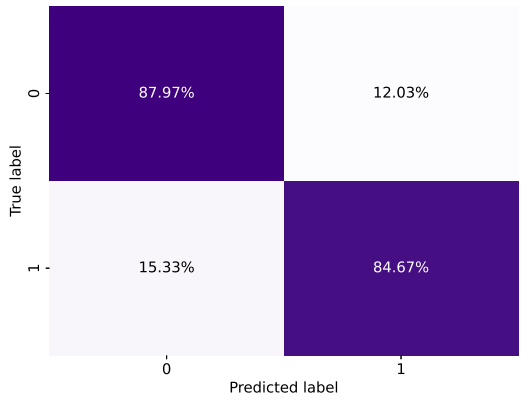
Mixed Approach-based detection

Figure 4.4 – Confusion matrix of static approaches - *APKComboDrebin* dataset.

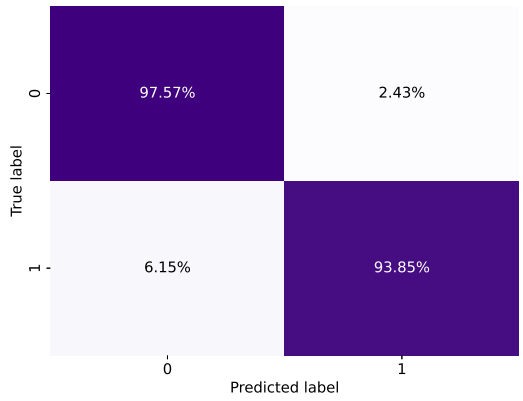
To sum up, the mixed approach demonstrates its efficacy in enhancing the detection of Android malware by substantially reducing false negatives, thereby improving the overall detection capability. Although there is a slight increase in false positives compared to opcode-based detection, this trade-off is justifiable given the significant improvement in detection accuracy achieved by the mixed approach. These findings underscore the importance of leveraging a combination of static approaches to achieve more robust and effective malware detection in Android applications.

4.7 Comparison with the State-of-the-art works

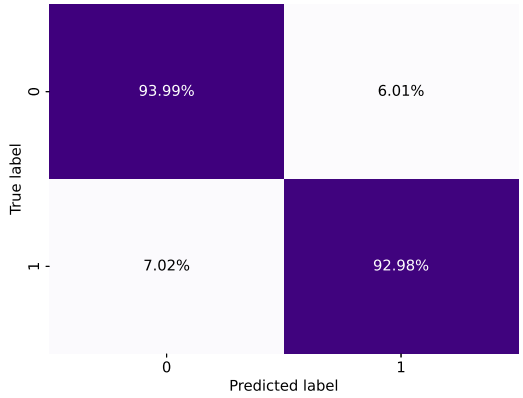
Table 4.15 compares our obtained results with those of state-of-the-art methods in terms of performance on two datasets: *APKComboDrebin* and *APKComboAndrozoo*. Our mixed model (*Vote*) significantly outperforms all others on the *APKComboDrebin* dataset, achieving remarkable scores across all metrics: 99.82% for Recall, Precision, Accuracy, and F1-score. The second-best performance on this dataset comes from the CNN model trained on visualized DEX files proposed by Ding et al. [17], which also shows high performance



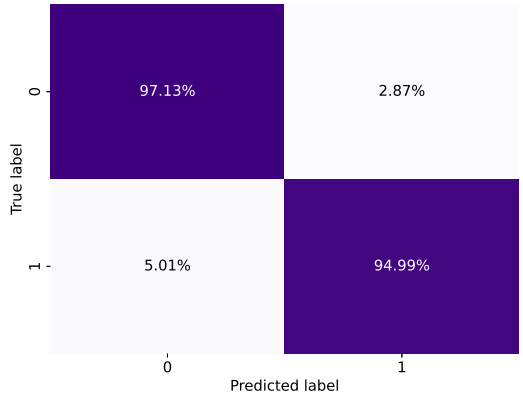
Permission-based detection



Opcode-based detection



Visualization-based detection



Mixed Approach-based detection

Figure 4.5 – Confusion matrix of static approaches - *APKComboAndrozoo* dataset.

but falls short compared to our model. Similarly, on the *APKComboAndrozoo* dataset, our model leads with the highest scores: 94.99% Recall, 97.06% Precision, 96.06% Accuracy, and 96.01% F1-score. The closest competitor is the DNN model proposed by Mercaldo et al. [35], which is trained on opcodes, yet it does not surpass our model in any of the metrics.

Ref.	**Model**	**APKComboDrebin**				**APKComboAndrozoo**			
		R (%)	**P (%)**	**A (%)**	**F1 (%)**	**R (%)**	**P (%)**	**A (%)**	**F1 (%)**
[5]	RF	96.45	90.94	93.34	93.57	75.05	79.21	77.08	76.63
[42]	RF	90.97	90.64	90.75	90.76	82.81	81.19	81.71	81.92
[17]	CNN	99.54	93.97	96.58	96.67	84.46	96.15	90.00	89.93
[35]	DNN	88.42	97.97	92.14	92.95	94.33	89.87	91.79	92.04
[47]	Bloom	92.86	91.53	95.01	93.24	91.17	93.03	89.05	91.00
Our	**Vote**	**99.82**	**98.83**	**99.82**	**99.82**	**94.99**	97.06	**96.06**	**96.01**

Table 4.15 – Comparison with state-of-the-art works.

4.8 Conclusion

In this chapter we concluded by underscoring the efficacy of the proposed mixed approach for Android malware detection. The experimental results demonstrate that combining permission-based, opcode-based, and visualization-based methods significantly enhances detection accuracy and robustness. The mixed approach effectively leverages the strengths of each individual analysis technique, leading to a more comprehensive detection system that can identify a broader range of malware behaviors. Notably, the ensemble learning methods, including stacking and voting, play a critical role in improving the model's overall performance by integrating the complementary strengths of the different analysis approaches. The evaluation also highlights the superiority of the proposed model compared to state-of-the-art methods. The model achieves remarkable scores across various metrics, including recall, precision, accuracy, and F1 score, particularly excelling on the *APKComboDrebin* and *APKComboAndrozoo* datasets. This indicates that the proposed approach not only enhances detection capability but also offers a robust solution adaptable to different datasets and evolving malware tactics. Overall, the findings affirm that the mixed static analysis approach, supported by ensemble learning techniques, provides a significant advancement in the field of Android malware detection.

Summary and conclusions

The research presented in this thesis addresses the pressing issue of Android malware detection through an innovative and comprehensive approach. The increasing complexity and prevalence of malware attacks on Android devices necessitate advanced detection mechanisms that can adapt to evolving threats. Our proposed mixed static analysis approach, which integrates permission analysis, opcode examination, and bytecode visualization, offers a robust solution to this challenge. By leveraging the strengths of each individual technique, our model significantly enhances detection accuracy and robustness, providing a more effective defense against a wide range of malware behaviors. The experimental results affirm the superiority of our approach compared to traditional methods and existing state-of-the-art techniques. Our model demonstrates high performance across two datasets, with malware samples collected from different periods, showcasing its ability to adapt to various malware types and detection scenarios. Specifically, the *Drebin* malware samples date back to 2012, whereas the *Androzoo* malware samples are relatively recent, dating from 2020 onwards. While state-of-the-art techniques and individual analysis methods achieve an impressive detection rate within the *APKComboDrebin* dataset, which comprises older malware samples, they exhibit a notable decrease in performance when applied to the *APKComboAndrozoo* dataset, which includes newer malware samples. This highlights the robustness and adaptability of our model, as it consistently outperforms other methods regardless of the age and type of malware, ensuring reliable detection across diverse and evolving malware landscapes. This research findings not only contributes to the academic field of cybersecurity but also has practical implications for improving the security of Android devices. As mobile technology continues to advance and proliferate, the importance of maintaining robust security measures cannot be overstated. These findings of this study underscore the potential of integrated analysis methods to offer a more resilient solution for malware detection, ultimately contributing to the security of users in the digital age, and providing a way to defend against Zero-day attacks.

As future work, we plan to enhance the model's capabilities by integrating additional static and dynamic analysis techniques, demonstrating their resilience against the continuous evolution of malware developers' tactics. Additionally, we aim to expand the dataset to include a broader range of benign and malicious samples. By continually refining and expanding our approach, we intend to stay ahead of emerging threats and ensure that

our detection system remains effective in the face of evolving malware strategies. Our ultimate goal is to provide a comprehensive, adaptable, and reliable solution that can be implemented in real-world applications, thereby significantly contributing to the protection of the Android ecosystem. This proactive stance will help us deliver a robust security framework that safeguards users and devices from the ever-changing landscape of cyber threats.

References

[1] Absar, J. *Programming for the Android Dalvik Virtual Machine*, 1st ed. Springer Publishing Company, Incorporated, 2017.

[2] Albin Ahmed, A., Shaahid, A., Alnasser, F., Alfaddagh, S., Binagag, S., and Alqahtani, D. Android ransomware detection using supervised machine learning techniques based on traffic analysis. *Sensors 24*, 1 (2024).

[3] Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. Androzoo: collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR'16)* (New York, NY, USA, 2016), MSR '16, Association for Computing Machinery, p. 468471.

[4] Almomani, I., Alkhayer, A., and El-Shafai, W. An automated vision-based deep learning model for efficient detection of android malware attacks. *IEEE Access 10* (2022), 2700–2720.

[5] Alsoghyer, S., and Almomani, I. On the effectiveness of application permissions for android ransomware detection. In *2020 6th Conference on Data Science and Machine Learning Applications (CDMA)* (2020), pp. 94–99.

[6] Alzaylaee, M. K., Yerima, S. Y., and Sezer, S. Dl-droid: Deep learning based android malware detection using real devices. *Computers & Security 89* (2020), 101663.

[7] Apkcombo. Apkcombo - download apks for android. https://apkcombo.com/. Accessed: 2024-06-19.

[8] AppBrain. Number of android apps on google play. https://www.appbrain.com/stats/number-of-android-apps, May 2024. Accessed: May 22, 2024.

[9] AppBrain. Top categories on google play. https://www.appbrain.com/stats/android-market-app-categories, May 2024. Accessed: May 20, 2024.

[10] Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., and Rieck, K. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS* (2014), The Internet Society.

[11] AV-ATLAS. Total amount of malware and pua under android. https://portal.av-atlas.org/malware/statistics/, 2024. Accessed: May 23, 2024.

[12] Azhagusundari, B., Thanamani, A. S., et al. Feature selection based on information gain. *International Journal of Innovative Technology and Exploring Engineering (IJITEE) 2*, 2 (2013), 18–21.

[13] BuildFire. App statistics. https://buildfire.com/app-statistics/, 2024. Accessed: May 20, 2024.

[14] Cinar, A. C., and Kara, T. B. The current state and future of mobile security in the light of the recent mobile security threat reports. *Multimedia Tools and Applications 82* (2023), 18425–18444.

[15] CrowdStrike. Mobile malware. https://www.crowdstrike.com/cybersecurity-101/malware/mobile-malware/, 2024. Accessed: May 20, 2024.

[16] Developers, A. Developer guide. https://developer.android.com/guide, 2024. Accessed: May 23, 2024.

[17] Ding, Y., Zhang, X., Hu, J., and et al. Android malware detection method based on bytecode image. *Journal of Ambient Intelligence and Humanized Computing 14* (2023), 6401–6410.

[18] Geden, M. *Ngram and Signature Based Malware Detection in Android Platform*. PhD thesis, University College London, 09 2015.

[19] Haidros Rahima Manzil, H., and Manohar Naik, S. Detection approaches for android malware: Taxonomy and review analysis. *Expert Systems with Applications 238* (2024), 122255.

[20] Jackson, W. The future of android iot apis: Android tv, glass, auto, and wear. In *Pro Android Wearables*. Apress, Berkeley, CA, 2015, pp. 325–340.

[21] Jiang, F., Wu, W., Chen, X., Wang, J., Luo, X., and Lin, Y. Android malware analysis in a nutshell. *PloS one 17*, 7 (2022), e0270647.

[22] Kaehler, A., and Bradski, G. R. *Learning OpenCV 3: computer vision in C++ with the OpenCV library*, first edition, second release ed. O'Reilly Media, Sebastopol, CA, 2017.

[23] Kaspersky. Malware in google play 2023. https://usa.kaspersky.com/blog/malware-in-google-play-2023/29356/, 2023. Accessed: May 21, 2024.

[24] Kaspersky. Global mobile banking malware grows 32 percent in 2023. https://www.kaspersky.com/about/press-releases/2024_global-mobile-banking-malware-grows-32-percent-in-2023, 2024. Accessed: Mai 21, 2024.

[25] Kohavi, R. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International Joint Conference on Artificial Intelligence* (03 2001), vol. 14.

[26] Kouliaridis, V., and Kambourakis, G. A comprehensive survey on machine learning techniques for android malware detection. *Information 12*, 5 (2021).

[27] Kuncheva, L. I. *Combining Pattern Classifiers*. John Wiley & Sons, Ltd, 2004.

[28] Liu, K., Xu, S., Xu, G., Zhang, M., Sun, D., and Liu, H. A review of android malware detection approaches based on machine learning. *IEEE Access 8* (2020), 124579–124607.

[29] Liu, K., Xu, S., Xu, G., Zhang, M., Sun, D., and Liu, H. A review of android malware detection approaches based on machine learning. *IEEE Access 8* (2020), 124579–124607.

[30] Martinelli, F., Mercaldo, F., Nardone, V., and Santone, A. Twinkle twinkle little droiddream, how i wonder what you are? In *2017 IEEE International Workshop on Metrology for AeroSpace (MetroAeroSpace)* (2017), pp. 21–25.

[31] Martinelli, F., Mercaldo, F., Nardone, V., Santone, A., and Vaglini, G. Model checking and machine learning techniques for hummingbad mobile malware detection and mitigation. *Simulation Modelling Practice and Theory 105* (2020), 102169.

[32] McAfee. Android game malware: Stealthy and widespread. https://www.mcafee.com/blogs/mobile-security/android-game-malware/, 2024. Accessed: 2024-06-11.

[33] McDonald, J. T., Herron, N., Glisson, W. B., and Benton, R. K. Machine learning-based android malware detection using manifest permission. In *Proceedings of the 54th Hawaii International Conference on System Sciences* (2021).

[34] Meijin, L., Zhiyang, F., Junfeng, W., Luyu, C., Qi, Z., Tao, Y., Yinwei, W., and Jiaxuan, G. A systematic overview of android malware detection. *Applied Artificial Intelligence 36*, 1 (2022), 2007327.

[35] Mercaldo, F., and Santone, A. Deep learning for image-based mobile malware detection. *Journal of Computer Virology and Hacking Techniques 16* (2020), 157–171.

[36] Miksik, O., and Mikolajczyk, K. Evaluation of local detectors and descriptors for fast feature matching. In *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)* (2012), pp. 2681–2684.

[37] Nazir, S., Khan, K. N., Ullah, N., Ali, S., Khan, M. S., Nauman, M., and Ghani, A. Op2vec: An opcode embedding technique and dataset design for end-to-end detection of android malware. *Security and Communication Networks 2022* (2022), 3710968.

[38] Niu, W., Cao, R., Zhang, X., Ding, K., Zhang, K., and Li, T. Opcode-level function call graph based android malware classification using deep learning. *Sensors 20*, 13 (2020).

[39] of Apps, B. App statistics. https://www.businessofapps.com/data/app-statistics, 2024. Accessed: May 20, 2024.

[40] of Apps, B. Google play statistics. https://www.businessofapps.com/data/google-play-statistics, 2024. Accessed: May 20, 2024.

[41] Point, C. Judy malware possibly largest malware campaign found on google play, 2024.

[42] Rathore, H., Sahay, S. K., Rajvanshi, R., and Sewak, M. Identification of significant permissions for efficient android malware detection. In *Broadband Communications, Networks, and Systems. BROADNETS 2020. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, H. Gao, R. J. D. Barroso, P. Shanchen, and R. Li, Eds., vol. 355. Springer, Cham, 2021, pp. 38–54.

[43] Sammut, C., and Webb, G. I., Eds. *Encyclopedia of Machine Learning*. Springer US, Boston, MA, 2010.

[44] Securelist. It threat evolution q2 2023 mobile statistics. https://securelist.com/it-threat-evolution-q2-2023-mobile-statistics/110427/, 2023. Accessed: May 20, 2024.

[45] Seraj, S., Pavlidis, M., Trovati, M., and Polatidis, N. Maddroid: malicious adware detection in android using deep learning. *Journal of Cyber Security Technology* (2023), 1–28.

[46] Shahzad, M., Arshad, S. M. U., Ahmad, H. A., Rafique, M. A., Mazzara, M., and Distefano, S. A comprehensive survey on malware detection approaches. *Sensors 22*, 15 (2022).

[47] Sihag, V., Mitharwal, A., Vardhan, M., and Singh, P. Opcode n-gram based malware classification in android. In *2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)* (2020), pp. 645–650.

[48] Srivastava, G., Maryam, A., Ahmed, U., Aleem, M., Lin, J. C.-W., Arshad Islam, M., and Iqbal, M. A. chybridroid: A machine learning-based hybrid technique for securing the edge computing. *Security and Communication Networks 2020* (2020), 8861639.

[49] Statista. Forecast number of mobile devices worldwide from 2020 to 2025 (in billions)*. https://www.statista.com/statistics/245501/multiple-mobile-device-ownership-worldwide, 2021. Accessed: April 27, 2024.

[50] Statista. Distribution of mobile malware worldwide in 2nd quarter 2023 and 3rd quarter 2023, by type. https://www.statista.com/statistics/653688/distribution-of-mobile-malware-type/, 2023. Accessed: Mai 21, 2024.

[51] Statista. Market share of mobile operating systems worldwide from 2009 to 2024, by quarter. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/, 2024. Accessed: Mai 21, 2024.

[52] Team, A. Androguard: Reverse engineering and pentesting for android applications. https://github.com/androguard/androguard, 2024. Accessed: 2024-06-11.

[53] Threatpost. 38 android devices infected with malware preinstalled in supply chain. https://threatpost.com/38-android-devices-infected-with-malware-preinstalled-in-supply-chain/124275/, 2017. Accessed: May 21, 2024.

[54] Virustotal. Virustotal. https://www.virustotal.com/, 2024. Accessed: 2024-06-11.

[55] Wolpert, D. H. Stacked generalization. *Neural Networks 5*, 2 (1992), 241–259.

[56] Yadav, R., and Bhadoria, R. S. Performance analysis for android runtime environment. In *2015 Fifth International Conference on Communication Systems and Network Technologies* (2015), pp. 1076–1079.

[57] Zhou, Z.-H., and Feng, J. Deep forest: Towards an alternative to deep neural networks. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17* (2017), pp. 3553–3559.