République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université M'hamed Bougara - Boumerdès

Faculté des Sciences
Département d'Informatique

Domaine : Mathématiques Informatique
Filière : Informatique
Spécialité : Systèmes d'informatique

Mémoire de fin d'études en vu de l'obtention du Diplôme de Licence Académique

Thème

Android malware detection based on AI and application permission analysis

Présenté par :

AISSAOUI Ikram
BOUARAKI Ikram
HAMADACHE Chemseddine
KITOUS Narimane

Soutenu le 03/06/2026 Devant le jury composé de

Nom Prénom : DJEDDAI Selma
Nom Prénom : BETIT Rahmoune

Dedication

With deepest gratitude, I dedicate this humble work

First of all, I thank Allah the Almighty for blessing me with strength, patience, and courage throughout this journey. Without His guidance and mercy, this achievement would not have been possible.

To my beloved parents, my mother and father, for their endless love, sacrifices, prayers, and unconditional support that have always inspired me to move forward.

To my dear sisters, Yasmine and Amina, for their encouragement, kindness, and for always standing by my side during every step of this journey.

To my wonderful teammates, for the teamwork, collaboration, and memorable moments shared while completing this project.

To our supervisor, Dr. BITIT, for his valuable guidance, patience, support, and advice throughout the realization of this work.

And finally, to everyone who supported me and contributed, directly or indirectly, to the success of this project.

AISSAOUI Ikram

Dedication

With deepest gratitude, I dedicate this humble work

To my beloved parents, whose love, sacrifices, prayers, and endless support have always been the light guiding my path and the strength that helped me overcome every challenge.

Thank you for believing in me, encouraging me, and standing beside me during every difficult moment. No words can truly express the depth of my gratitude and appreciation for everything you have done for me.

To my dear sisters and brother, thank you for your kindness, support, encouragement, and for always bringing comfort and motivation throughout this journey.

To everyone who stood by my side during moments of weakness and difficulty, your presence, support, and sincere words will always remain deeply appreciated and forever remembered.

BOUARAKI Ikram

Dedication

With deepest gratitude, we dedicate this humble work

First of all, we thank Allah the Almighty for giving us strength, patience, and courage throughout this journey. Without His guidance and mercy, this achievement would not have been possible.

To our beloved parents, our fathers and mothers, for their endless love, sacrifices, prayers, and unconditional support that always inspired us to move forward.

To our dear family members, for their encouragement, kindness, and continuous support during every step of this work.

To our dear friends, Samira and Melissa, for their encouragement, support, and the wonderful moments shared throughout this journey.

To our wonderful teammates, for the teamwork, collaboration, and memorable moments shared while completing this project.

To our supervisor, Mr. BITIT, for his valuable guidance, patience, support, and precious advice throughout the realization of this work. Without his help, this work would not have been possible.

And finally, to everyone who supported and contributed, directly or indirectly, to the success of this project.

KITOUS Narimane

Dedication

With deepest gratitude, I dedicate this humble work

Above all, all credit and glory belong to Almighty Allah Taala for His endless blessings and for giving me the strength, patience, and courage to complete this journey.

I dedicate this work to my beloved parents: to my father, who has always been the pillar of my ambition and motivation, and to my mother, who has always been my safe haven and whose prayers constantly guided my path.

I also express my sincere gratitude to my dear cousins, sisters, and brothers — Meriam, Mohamed, Malika, Farida, Anfal, Fatiha, Nadia, and Hamid — for their kindness, warmth, and continuous encouragement during every difficult moment.

My respectful thanks also go to our supervisor, Mr. BITIT, for his patience, valuable guidance, and academic support throughout this work.

Finally, a special thanks to my childhood friend, Akram, for always being there, for his unwavering loyalty, and for sharing countless memories and motivation that helped me throughout this journey.

HAMADACHE Chemseddine

Abstract

Malicious applications on Android represent an increasing threat to user security, as they often exploit app permissions to access sensitive data. Due to the limitations of traditional signature-based methods, this work proposes a detection system based on artificial intelligence and machine learning techniques. The proposed approach is based on analyzing Android permissions as the main features to distinguish between benign and malicious applications. Several models were implemented and evaluated, including LSTM, CNN1D, SVM, Random Forest , and Deep Neural Network (DNN). The experiments were conducted using the NATICUSdroid dataset, which contains both benign and malicious Android applications. The analysis is static and relies only on extracted permissions without executing the applications. The results show that machine learning and deep learning models perform well in general, with Random Forest achieving the best performance in terms of Classification Metrics and Resource Consumption compared to the other tested models. Finally, integrating the selected model into an Android application allows real-time detection directly on the user's device.

Contents

Abstract

List of Abbreviations v

General Introduction 1

1 Android Operating System 2
- 1.1 Introduction . . . 2
- 1.2 Android System Architecture . . . 2
  - 1.2.1 System Apps . . . 2
  - 1.2.2 Java API Framework . . . 2
  - 1.2.3 Native c/c++ Libraries . . . 2
  - 1.2.4 Android Runtime (ART) . . . 3
  - 1.2.5 Hardware Abstraction Layer (HAL) . . . 3
  - 1.2.6 Linux kernel . . . 3
- 1.3 Android Application . . . 4
  - 1.3.1 Application Architecture (Security Perspective) . . . 4
  - 1.3.2 Intents: Inter-Component Communication (ICC) . . . 4
  - 1.3.3 Intent Filters . . . 5
  - 1.3.4 APK Archive Architecture . . . 5
- 1.4 Android Security Mechanisms . . . 6
  - 1.4.1 Linux-Based Security Foundations . . . 6
  - 1.4.2 Application Permissions . . . 6
  - 1.4.3 Application Sandboxing . . . 7
  - 1.4.4 Application Signing . . . 7
- 1.5 Android Malware Categories . . . 8
- 1.6 Malware Analysis Techniques . . . 8
  - 1.6.1 Static Analysis . . . 8
  - 1.6.2 Dynamic Analysis . . . 8
  - 1.6.3 Hybrid Analysis . . . 8
  - 1.6.4 Conclusion . . . 8

2 Permission-Based Malware Detection 9
- 2.1 Introduction . . . 9
- 2.2 Presentation of Android Malware Detection Using Permissions . . . 9
  - 2.2.1 Android Permissions . . . 9
  - 2.2.2 Relationship Between Permissions and Malware Behavior . . . 9
  - 2.2.3 Permissions as Features . . . 10
  - 2.2.4 Convert Permissions into Features . . . 10
  - 2.2.5 Feature Vector Construction (Sparse Representation) . . . 10
  - 2.2.6 Machine Learning and Deep Learning for Malware Detection . . . 10
- 2.3 Related Work . . . 11
  - 2.3.1 Comparison Table . . . 13
- 2.4 Design of Our Approach . . . 13

2.5 Dataset Creation . . . 14
2.5.1 Manifest Extraction . . . 14
2.5.2 Permission Extraction . . . 14
2.5.3 Feature Transformation . . . 14
2.6 Dataset and Preprocessing . . . 15
2.7 Machine and deep Learning Techniques . . . 15
2.7.1 Random Forest (RF) . . . 15
2.7.2 Support Vector Machine (SVM) . . . 16
2.7.3 Convolutional Neural Network (CNN) . . . 16
2.7.4 Long Short-Term Memory (LSTM) . . . 16
2.7.5 Deep Neural Network (DNN) . . . 16
2.8 Classification evaluation Metrics . . . 16
2.8.1 Confusion Matrix . . . 16
2.8.2 Accuracy . . . 16
2.8.3 Precision . . . 17
2.8.4 Recall . . . 17
2.8.5 F1-Score . . . 17
2.9 Resources conception evaluation Metrics . . . 17
2.9.1 Training Time . . . 17
2.9.2 Inference Time . . . 17
2.9.3 Memory Peak Usage . . . 17
2.9.4 Model Size . . . 17
2.10 Conclusion . . . 17

3 Implementation and evaluation **18**
3.1 Introduction . . . 18
3.2 Development environment . . . 18
3.2.1 Hardware Tools . . . 18
3.2.2 Software Tools . . . 19
3.2.3 Libraries Used . . . 20
3.3 Results and Discussion . . . 20
3.3.1 Experimental Setup . . . 20
3.3.2 Dataset Description and Visualization . . . 21
3.3.3 Performance Evaluation . . . 21
3.3.4 Best Model Selection . . . 23
3.4 Application Design and User Interface . . . 24
3.4.1 Use case diagram . . . 24
3.4.2 Class diagram . . . 24
3.4.3 Sequence diagram . . . 25
3.4.4 Application Description and User Interface and Results . . . 26
3.4.5 Application Architecture and Permission Extraction . . . 29
3.4.6 Conclusion . . . 29

General Conclusion **30**

List of Tables

2.1 Comparison of Android malware detection techniques 13

3.1 Computer Hardware Specifications 18
3.2 Software and Tools Used 19
3.3 Libraries Used in the Project 20
3.4 Classification Metrics Comparison 22
3.5 Resource Usage Comparison 23

List of Figures

1.1 Android logo 2
1.2 The Android software stack 3
1.3 APK structure 6
1.4 Sand-boxing architecture 7

2.1 Diagram of the malware detection system. 14

3.1 Class Distribution 21
3.2 Classification Metrics Comparison 22
3.3 Resource Usage Comparison 23
3.4 Use Case Diagram 24
3.5 Class Diagram 25
3.6 Class Diagram 26
3.7 Application Home Interface 27
3.8 Benign Application Scan Result 28
3.9 Malware Detection Alert Screen 29

List of Abbreviations

API	Application Programming Interface
APK	Android Package Kit
AOSP	Android Open Source Project
ART	Android Runtime
CNN	Convolutional Neural Network
CNN1D	1D Convolutional Neural Network
CPU	Central Processing Unit
CSV	Comma-Separated Values
DAC	Discretionary Access Control
DDoS	Distributed Denial of Service
DEX	Dalvik Executable
DNN	Deep Neural Network
DT	Decision Tree
FN	False Negative
FP	False Positive
GA	Genetic Algorithm
GB	Gigabyte
GHz	Gigahertz
GPS	Global Positioning System
GPU	Graphics Processing Unit
HAL	Hardware Abstraction Layer
ICC	Inter-Component Communication
IDE	Integrated Development Environment
JIT	Just-In-Time
KNN	K-Nearest Neighbors
LSTM	Long Short-Term Memory

MAC Mandatory Access Control

MB Megabyte

MLP Multi-Layer Perceptron

MMS Multimedia Messaging Service

ms Milliseconds

NB Naïve Bayes

NDK Native Development Kit

OS Operating System

PDCA Plan-Do-Check-Act

PMAR Permission Mining with Association Rules

PRNR Permission Ranking with Negative Rate

RAM Random Access Memory

RF Random Forest

SELinux Security-Enhanced Linux

SMS Short Message Service

SPR Support Based Permission Ranking

SVM Support Vector Machine

TFLite TensorFlow Lite

TN True Negative

TP True Positive

UI User Interface

UID User Identifier

XML Extensible Markup Language

CIA Confidentiality, Integrity, Availability

AOT Ahead-Of-Time

General Introduction

Smartphones have become essential tools in daily life, used not only as means of communication but also as platforms for work, payment, and access to various services. This omnipresence can be explained by the continuous increase in smartphone sales worldwide, as well as the massive adoption of mobile operating systems. According to StatCounter Global Stats, Android held approximately 70.36% of the global mobile operating system market share as of January 2026, illustrating the dominance of this platform in the global smartphone ecosystem[1]. Therefore, the demand for smartphones has significantly increased over time.

With the daily increase in mobile device users, the Android operating system has become one of the most widely used and prevalent platforms. However, Android based applications are highly vulnerable to various types of malware attacks due to the system's popularity and open nature. These applications face challenges related to sensitive permissions used during installation and runtime [2], which malware developers exploit to launch attacks on users. This situation raises the following key question: How can effective tools be developed to detect malware on the Android platform while taking into account the exploitation of sensitive permissions, ensuring high detection accuracy?

This project is set in this context, with the main objective of designing an effective detection system for Android malware. The specific objectives are to analyze existing malware, assess Android's security weaknesses, develop a machine learning based model, and validate this model on real-world datasets. To achieve these objectives, the adopted methodology combines several key steps: collecting and preprocessing data from Android malware, extracting relevant features, training and testing a machine learning model, followed by a rigorous evaluation of the system's performance. This approach allows a direct link between problem analysis and proposed solutions, ensuring both the reliability and effectiveness of malware detection.

To this end, our work has been divided into three chapters as follows: Chapter 1 introduces the Android system, its architecture, the architecture of an Android application, and Android security mechanisms. It then covers Android malware and concludes with techniques for detecting malicious applications. Chapter 2 reviews permission-based malware detection techniques. It discusses the state of the art, analyzes existing approaches, and presents the proposed model and its design methodology. Chapter 3 describes the implementation and evaluation of the proposed system. It introduces the development environment, provides experimental results, and presents the deployment process using TensorFlow Lite. Finally, a general conclusion summarizes the key findings and highlights perspectives for future work

Chapter 1

Android Operating System

1.1 Introduction

Android is a free and open-source operating system based on the Linux kernel and managed by Google through the Android Open Source Project (AOSP). Google purchased the Android system from its main developers in 2005, while the official announcement of Android was in 2007, and the first Android device appeared on the market in 2008 [3]. Due to its flexibility and large application ecosystem, Android has become the most widely used mobile operating system worldwide. However, its popularity has also made it a major target for malware and security threats. Therefore, understanding the Android system architecture and security mechanisms is essential for studying Android malware detection techniques.



Figure 1.1: Android logo

1.2 Android System Architecture

Android is a Linux-based software stack designed for a diverse range of devices. This stack is composed of multiple layers as illustrated in Figure 1.2, and each layer provides specific services and functionalities required by the upper layers.

1.2.1 System Apps

Located at the top of the stack. Android includes a set of core applications for essential tasks such as SMS, email, and web browsing [4].

1.2.2 Java API Framework

The Java API Framework offers Android OS features through APIs, allowing applications to interact directly with core system services. This layer manages the basic system components, including the View System for UI rendering, the Resource Manager for non-code assets, and the Notification Manager. It also includes the Activity Manager for lifecycle control and Content Providers for secure inter-process data sharing. Because standard applications use the same framework APIs as system apps, this layer serves as the primary interface for system interactions and permission [4].

1.2.3 Native c/c++ Libraries

Many core Android system components and services, such as the Android Runtime (ART) and the Hardware Abstraction Layer (HAL), are built from native code that requires native libraries written in C and C++. The Android platform provides Java framework APIs to offer the functionality of these native libraries to apps. For example, developers can access

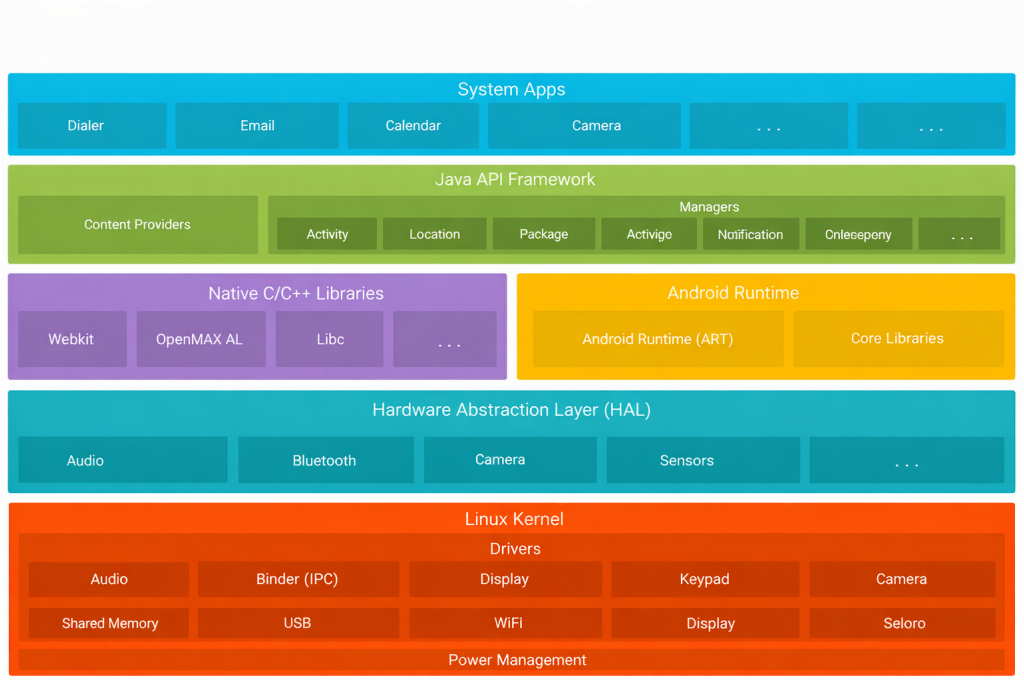


Figure 1.2: The Android software stack

OpenGL ES through the Android framework's Java OpenGL API to add support for drawing and manipulating 2D and 3D graphics in an app. Additionally, if an app requires custom C or C++ code, developers can use the Android Native Development Kit (NDK) to access these native platform libraries directly from their native code [4].

1.2.4 Android Runtime (ART)

Since Android 5.0, each application operates within its own process using an Android Runtime (ART) instance. ART executes DEX files which is bytecode format compiled via build tools like d8 and optimized for minimal memory footprint. Its primary features include Ahead-Of-Time (AOT) and Just-In-Time (JIT) compilation, which convert bytecode into machine code to optimize performance. Additionally, ART manages enhanced garbage collection and core libraries that provide the Java language features required by the higher-level API framework [4].

1.2.5 Hardware Abstraction Layer (HAL)

The hardware abstraction layer (HAL) provides standard interfaces that expose device hardware capabilities to the higher level Java API framework. The HAL consists of multiple library modules, each of which implements an interface for a specific type of hardware component, such as the camera or Bluetooth module. When a framework API makes a call to access device hardware, the Android system loads the library module for that hardware component [4].

1.2.6 Linux kernel

The foundation of the Android platform is the Linux kernel. For example, the Android Runtime (ART) relies on the Linux kernel for underlying functionalities such as threading and low-level memory management. Using a Linux kernel lets Android take advantage of key security features, such as sandboxing and a robust permissions [4].

1.3 Android Application

1.3.1 Application Architecture (Security Perspective)

Android applications are built using a component-based architecture composed of four principal components: Activities, Services, Broadcast Receivers, and Content Providers [5]. Each component functions as an entry point through which the Android system or external applications may interact with the application. While this modular architecture enhances flexibility and inter-application communication, it simultaneously increases the potential attack surface when components are improperly configured or exposed [6].

- **Activity :** An Activity represents a single user interface screen and manages user interaction within an application. Although activities are primarily intended for user engagement, they may also be launched by other applications if explicitly exported in the manifest file. From a security perspective, improperly exported activities can be exploited by malicious applications to trigger sensitive functionality without authorization, a vulnerability commonly referred to as component hijacking or permission re-delegation [7].

- **Services:** A Service is responsible executing background operations without direct user interaction. Services may perform long-running tasks such as data synchronization, network communication, or media playback. Due to their ability to operate in the background, services are frequently abused by malware to conduct hidden operations, including data exfiltration or unauthorized surveillance, without user awareness [8].

- **Broadcast Receivers :** Broadcast Receivers enable applications to respond to system-wide events, such as connectivity changes or incoming messages. They operate passively and are activated when a matching broadcast event occurs. From a security standpoint, exported broadcast receivers may be exploited to inject malicious intents or trigger unintended behaviors. If access controls are not properly enforced, attackers may leverage this mechanism to manipulate application logic or escalate privileges [7].

- **Content Providers :**Content Providers manage structured data and enable controlled data sharing between applications. They expose data through defined interfaces and typically enforce permission-based access control. However, misconfigured content providers may result in capability leaks and unauthorized data access, posing significant security risks [9].

Empirical studies have shown that improper permission enforcement and manifest misconfiguration are among the most common causes of Android application vulnerabilities [9]. Therefore, understanding Android's component-based architecture from a security perspective is essential for analyzing malicious applications and designing effective detection mechanisms

1.3.2 Intents: Inter-Component Communication (ICC)

Intents are asynchronous messaging objects used by the Android system to facilitate communication between components. They function as system-wide signals that notify applications of various events, ranging from hardware changes (e.g., SD card insertion) to data-driven triggers (e.g., the arrival of an SMS) or application-specific actions (e.g., launching an Activity from the main menu). Beyond responding to system events, developers can define custom Intents to trigger specific application logic or broadcast unique state changes—such as alerting the system when a user reaches a specific geographic coordinate [10]. From a security standpoint, Intents are the primary vectors through which malicious data or "malicious commands" are passed between applications.

1.3.3 Intent Filters

When an Intent is dispatched, the Android system determines the appropriate recipient(s) based on the attributes associated with the message. This process follows two distinct mechanisms:

- **Explicit Intents:** The sender explicitly defines the target component (by class name). The message is delivered directly to the specified destination, bypassing the filtering process.

- **Implicit Intents:** The sender specifies an action to be performed without naming a specific recipient. In this case, the system identifies potential candidates by consulting Intent Filters declared in each application's AndroidManifest.xml file [11]. An Intent Filter acts as a declaration of a component's capabilities, specifying the types of Intents it is prepared to handle. This filtering is defined by a set of attributes (Action, Data, and Category) that an incoming Intent must match. For a static analysis method, these filters are critical because they define the visibility of a component; any component with an Intent Filter is automatically "exported" (visible to other apps) unless otherwise restricted by permissions.

1.3.4 APK Archive Architecture

Four different components make up the Android application (app): activity, services, Content Providers, and Broadcast Receivers. Android apps are written in Java and compiled, along with data and resource files, into a single archive file known as Android package or APK. The file format used by the Android operating system to distribute and install mobile applications is the Android Package Kit (APK). An APK file contains all of the code and assets of the applications. With Android Studio, the official integrated development environment (IDE) for building Android apps, APK files are compiled. An APK expansion file is used for storing media files, high-fidelity graphics, or other large assets that would cause an APK to exceed the size limitation [12]. As shown in Fig. 4, an APK usually consists of several components and directories, including:

- **Android Manifest.xml** file contains information about app description like name, version, access rights, referenced application library files, and permission components

- **Classes.dex** file is a Dalvik executable file that contains the methods and classes used by the apps. The Dex file format can be understood by the Dalvik virtual machine and by the Android runtime.

- **Res directory** for indexed resources like icons, images, music etc. Not compiled into Resources.arsc.

- **Lib directory** for compiled code that is specific to the processor's software layer.

- **META-INF directory** holding the app certificate, list of resources, SHA-1 digest, etc.

- **Resources.arsc** is a compiled resource file.

- **Assets directory** containing application assets like maintenance and up-gradation which can be retrieved by the Asset Manager.

Malicious applications can attempt to modify the manifest, inject code into classes.dex, or place harmful files in assets to perform unauthorized actions or bypass permissions.

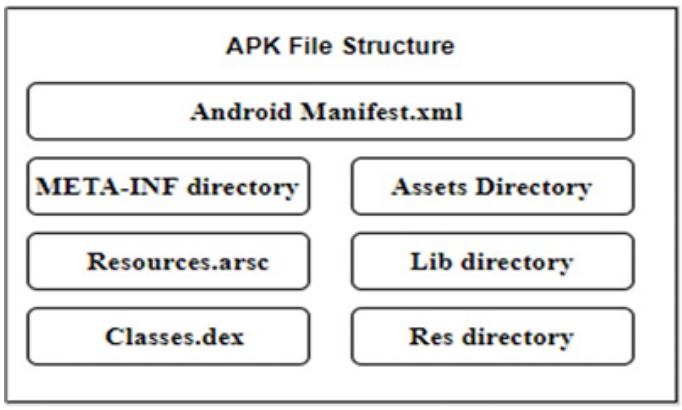


Figure 1.3: APK structure

1.4 Android Security Mechanisms

The Android security architecture is designed to protect user data, application code, and system resources through a multi-layered defense strategy. This section details the fundamental security pillars of the platform, beginning with the low-level protections inherited from Linux and extending to application-specific mechanisms such as sandboxing, permissions, and digital signatures.

1.4.1 Linux-Based Security Foundations

At its most fundamental level, Android's security is built upon the security mechanisms provided by the Linux kernel. By inheriting the Linux Discretionary Access Control (DAC) model, Android treats every application as a separate, isolated user with a unique User ID (UID). This low-level separation ensures that processes are restricted to their own dedicated memory and file space, preventing one application from interfering with the resources of another or the core system.

Furthermore, modern versions of Android incorporate Security-Enhanced Linux (SELinux) to implement Mandatory Access Control (MAC). While DAC allows users to manage their own file permissions[13], SELinux acts as a central authority that enforces global security policies, even over processes with administrative (root) privileges. This dual-layered approach creates a defense-in-depth strategy that hardens the system against privilege escalation and kernel-level exploits[14].

1.4.2 Application Permissions

The Android permission system constitutes the core mechanism for enforcing application-level security. It regulates access to sensitive system resources and protected APIs, ensuring that applications operate within clearly defined privilege boundaries. Applications must explicitly declare the permissions they require in the `AndroidManifest.xml` file. These permissions are evaluated by the Android framework, which enforces access control both at installation time and at runtime.

Since Android 6.0 (Marshmallow), permissions classified as dangerous are granted dynamically during execution, allowing users to approve or deny access while the application is running. Furthermore, users may revoke permissions at any time through system settings. Android defines several protection levels:

- **Normal:** Permissions that pose minimal risk and are automatically granted.
- **Dangerous:** Permissions that provide access to sensitive user data or critical system features such as the camera, contacts, and microphone. These permissions require explicit user approval at runtime.
- **Signature:** Permissions granted only to applications signed with the same developer certificate.

- **Privileged (system-level):** Reserved for system applications installed within the device firmware.

From a security perspective, the permission system plays a central role in limiting unauthorized actions. However, malicious applications often attempt to exploit this model by requesting excessive or unnecessary dangerous permissions, a phenomenon known as over-privileging. In many malware families, combinations of permissions such as network access and access to personal data enable covert data exfiltration.

Permissions are also used to protect application components. As introduced in Section 3.1, Activities, Services, Broadcast Receivers, and Content Providers may specify required permissions in the manifest. This mechanism restricts which external applications may start, bind to, query, or send data to these components. Improper permission configuration or failure to enforce permission checks can result in component hijacking, privilege escalation, or unauthorized data disclosure.

For static malware detection approaches, permission analysis is particularly significant. Extracting and evaluating permission requests from the APK manifest provides insight into an application's intended capabilities and potential risk profile. Numerous empirical studies have shown that abnormal or suspicious permission patterns are strong indicators of malicious behavior.

1.4.3 Application Sandboxing

Android enforces isolation through a kernel-level sandbox, where each application is treated as a unique Linux user identified by a distinct User ID (UID). By assigning every app its own private process and memory space, the system ensures that no application can access another's data by default. For our work, the sandbox is critical because it forces malware to "ask" for access to the system. Because the sandbox restricts unauthorized access, amalicious application must declare permissions in its manifest to perform any sensitive task. Our static analysis exploits this by examining these permission requests, which essentially represent the app's attempt to reach beyond its sandbox boundaries [15].

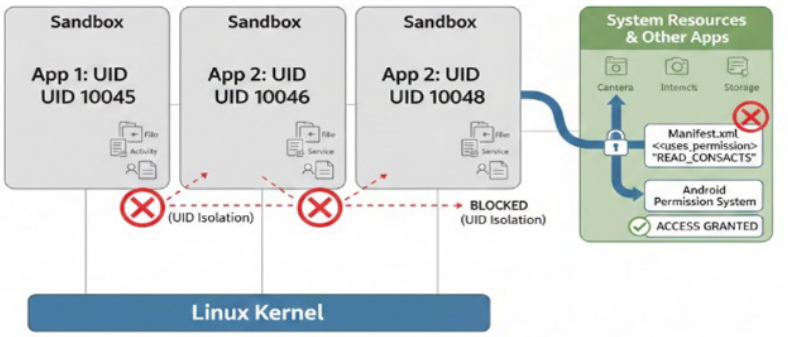


Figure 1.4: Sand-boxing architecture

1.4.4 Application Signing

Application signing is a mandatory security process that ensures the integrity and authenticity of an **APK**. Every application must be digitally signed with a developer's private key before installation to verify that the code has not been altered by third parties. From a security perspective, the Android system uses the signature to enforce Signature-level permissions and to facilitate secure updates, ensuring that an installed application can only be updated by a version signed with the same key. For static malware detection, signature analysis is a vital feature; malicious actors often engage in repackaging attacks, in which legitimate applications are modified and re-signed with different or untrusted certificates to bypass basic security checks [16].

1.5 Android Malware Categories

Android malware includes any type of malicious software developed to compromise user privacy, steal sensitive data, or gain unauthorized privileged access to the device system. Today, attackers use many different categories of threats, and the most common ones are Trojans. These applications disguise themselves as normal or useful applications to trick users into installing them, which allows the malware to request and obtain critical system permissions. Once these applications are running on the device, they can execute other dangerous components such as spyware or Keyloggers to monitor user activities, log keystrokes, and collect sensitive credentials. In addition, some variants operate as Ransomware, which encrypts personal files or locks the screen to demand financial payments from the user. Other advanced threats include Botnets and Rootkits. Botnets connect infected devices into a network controlled remotely by an attacker to launch large-scale attacks, while Rootkits focus on maintaining long-term persistence inside the system to bypass the native Android security mechanisms and hide all malicious actions from being detected [17].

1.6 Malware Analysis Techniques

1.6.1 Static Analysis

Static analysis identifies malicious behavior by examining application code without executing. It is efficient and scalable but may be limited by code obfuscation techniques [18].

1.6.2 Dynamic Analysis

Dynamic analysis evaluates applications in controlled environments to monitor runtime behavior and detect dynamic code loading [14].

1.6.3 Hybrid Analysis

Hybrid analysis combines static and dynamic techniques to enhance detection accuracy and overcome individual limitations [18].

1.6.4 Conclusion

This chapter presented an overview of the Android operating system and its security mechanisms, discussed major categories of Android malware, and reviewed key detection techniques aimed at mitigating mobile security threats.

The next chapter presents a state of the art on Android malware detection using ML/DL techniques and permissions. In addition, it introduces our proposed approach for malware detection.

Chapter 2

Permission-Based Malware Detection

2.1 Introduction

The rapid growth of the Android ecosystem has been accompanied by a significant rise in malicious applications targeting mobile users. As established in the previous chapter, Android malware exploits various system vulnerabilities, with permission abuse representing one of the most common attack vectors. This chapter addresses the problem of Android malware detection through permission analysis and machine learning techniques. We begin by presenting the core concept of permission-based detection, explaining how permissions can reveal malicious behavior and how they are transformed into feature vectors suitable for machine learning models. We also discuss how machine learning algorithms are applied in this context to classify applications as malicious or benign. We then review the most relevant works in the literature through a set of related works, followed by a comparison table summarizing the existing approaches in terms of datasets used, analysis techniques, extracted features, and number of permissions. Finally, we present the design of our proposed system. This includes the selection of the dataset, the preprocessing pipeline covering feature extraction, permission vectorization, and dataset construction, as well as the machine learning techniques employed. We further present the evaluation metrics used to assess model performance, namely the Confusion Matrix, Accuracy, Precision, Recall, and F1-score.

2.2 Presentation of Android Malware Detection Using Permissions

2.2.1 Android Permissions

Android permissions represent one of the main security mechanisms in the Android operating system. They controle how applications access sensitive system resources and user data such as contacts, messages, location, and device information. This mechanism helps protect user privacy by restricting unauthorized access to critical system functionalities. Applications request permissions from users in order to access protected system resources and functionalities. Each Android application must declare the permissions it requires in the `AndroidManifest.xml` file. This manifest specifies the permissions needed to access restricted resources such as hardware components including the camera, GPS, and file system, as well as sensitive system data and specific functionalities provided by other applications.

Through this permission model, Android ensures that applications only access the resources that have been explicitly authorized by the user.

2.2.2 Relationship Between Permissions and Malware Behavior

Although permissions are designed to enhance security, they can also be abused by malicious applications. Some malware requests permissions that are not necessary for the application's intended functionality, a behavior commonly referred to as **over-privileging**.

For example, a simple calculator application requesting access to contacts or SMS messages may appear suspicious. As a result, analyzing application permissions has become an important technique for Android malware detection. By examining the permissions declared in the `AndroidManifest.xml` file [19], researchers can identify abnormal permission requests that may indicate malicious behavior [20]. In many cases, malicious applications tend to request sensitive permissions more frequently or in unusual combinations compared to benign applications. Consequently, permission analysis can reveal patterns that help distinguish malicious applications from legitimate ones.

2.2.3 Permissions as Features

Extraction of Permissions from `AndroidManifest.xml`

The `AndroidManifest.xml` file contains important information about an Android application's requested permissions and application information. In static malware analysis, this file represents a valuable source of permission-related features that can be extracted without executing the application.

Several permission-related features can be extracted from the manifest file. Requested permissions such as `SEND_SMS`, `READ_SMS`, and `ACCESS_FINE_LOCATION` may reveal suspicious access to sensitive resources and user data.

The extracted permissions constitute the initial feature set used to represent Android applications in machine learning and deep learning models. After extraction, the extracted permissions are transformed into numerical representations, such as binary feature vectors, to facilitate the malware detection process [21].

2.2.4 Convert Permissions into Features

After extracting permissions from the `AndroidManifest.xml` file, they are transformed into numerical features that can be processed by machine learning models. Each permission is represented as a feature in a vector space. An Android application is then encoded as a binary vector, where each dimension corresponds to a specific permission: the value is 1 if the permission is requested and 0 otherwise.

This binary representation allows the system to capture patterns and combinations of permissions that may indicate malicious behavior.

2.2.5 Feature Vector Construction (Sparse Representation)

All permission features of an application are combined into a single feature vector. Each dimension corresponds to a specific permission in the overall permission set. Since each application typically requests only a small subset of all possible permissions, the resulting vectors are sparse.

To efficiently handle this sparsity, only the non-zero values (i.e., requested permissions) need to be stored. Sparse representation techniques such as hash-based indexing can be used to reduce memory consumption and improve processing efficiency.

This representation is then used as input for machine learning and deep learning models in the malware detection process.

2.2.6 Machine Learning and Deep Learning for Malware Detection

Machine learning and deep learning have become essential techniques in Android malware detection due to their ability to analyze large-scale datasets and identify patterns associated with malicious behavior. Unlike traditional signature-based detection methods, which rely on a database of known threats, these approaches can generalize from data and detect previously unseen threats, commonly referred to as zero-day malware.

In this context, malware detection is typically formulated as a supervised binary classification problem in which applications are categorized as either benign or malicious. To perform this classification, relevant features must be extracted from Android applications and represented as numerical feature vectors. Several types of features have been explored in the literature, including API calls, system behaviors, and network-related information. Among these, requested permissions are widely used in static analysis because they can

be easily extracted from the `AndroidManifest.xml` file and provide useful insight into application behavior.

A wide range of machine learning and deep learning algorithms have been applied to this problem. Traditional machine learning methods include Random Forest (RF), Support Vector Machines (SVM), and K-Nearest Neighbors (KNN), while deep learning models such as Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), and Long Short-Term Memory (LSTM) networks are capable of automatically learning complex patterns from data.

Although challenges such as dataset imbalance and the continuous evolution of malware remain, these approaches continue to be among the most effective and scalable solutions for Android malware detection.

2.3 Related Work

The study [22] proposes an Android malware detection approach based on **statistical frequency analysis** of permissions extracted from application manifest files. The method relies on examining how frequently specific permissions appear in malicious and benign applications in order to identify suspicious patterns. The dataset used in the study includes several thousand Android applications composed of malicious samples collected from public malware repositories such as Drebin and benign applications obtained from official markets. The features extracted are the permissions declared in the AndroidManifest.xml file. Permission frequency analysis evaluates the occurrence of each permission and identifies those that appear significantly more often in malicious applications. These permissions are then used as indicators to distinguish benign applications from malicious ones. Experimental results show that permission frequency analysis can effectively highlight differences between benign and malicious applications and can support the development of efficient malware detection systems.

The study [23] introduced **SigPID** (Significant Permission IDentification), a malware detection system based on permission usage analysis. Instead of using all available permissions, the authors developed a three-level pruning technique to identify only the most significant permissions capable of distinguishing malicious from benign applications. The three pruning levels are:

- Permission Ranking with Negative Rate (PRNR)
- Support Based Permission Ranking (SPR)
- Permission Mining with Association Rules (PMAR)

This approach reduces the feature set to only 22 significant permissions. The system achieved approximately 90% precision, accuracy, recall, and F-measure using a Support Vector Machine (SVM) classifier.

The study [20] proposed **PermPair**, an Android malware detection model based on permission pair analysis. The authors observed that individual permissions are insufficient for distinguishing malware from benign applications, as the most frequently requested permissions are common to both categories. Instead of analyzing permissions individually, they represented permission pairs extracted from the `AndroidManifest.xml` file as weighted graphs. Separate permission pair graphs were constructed for three malware datasets, namely Genome, Drebin, and Koodous, as well as for normal applications collected from the Google Play Store. These malware graphs were then merged into a single malicious graph using a weighted sum optimization method. A detection algorithm classifies a new application by comparing its malicious score against its normal score, both computed from its permission pairs. The proposed approach achieved an overall accuracy of 95.44% on known malware samples and a detection rate of 84.48% on unknown samples.

The study [24] proposed a permission-based malware detection system using feature selection with a Genetic Algorithm (GA) to identify the most effective permissions. The dataset used includes 1740 applications (1119 malware and 621 benign). Three classifiers—Naïve Bayes (NB), Decision Tree (DT), and Support Vector Machine (SVM)—were tested, achieving the highest accuracy of 98.45% with SVM using 16 selected permissions. GA works by starting with an initial population of random permission subsets, evaluating each subset using accuracy as a fitness function, and iteratively selecting, combining, and mutating the best subsets until the optimal subset is found. The method applies static analysis by analyzing permissions from the `AndroidManifest.xml` file without executing the applications. This approach shows that a small subset of significant permissions is sufficient for high-accuracy malware detection, reducing computational complexity while maintaining robustness.

The study [25] proposed a basic permission feature extraction method for Android malware detection. All requested permissions were extracted and represented as binary features in a vector space. Each application was mapped to a sparse binary vector where **1** indicates the presence of a permission and **0** its absence. These feature vectors were then used to train and evaluate multiple classifiers, including SVM, Random Forest, and Naïve Bayes, achieving a detection accuracy of approximately 87.9% with SVM.

The study [26] proposed **PerDRaML**, a permissions-based Android malware detection system designed to identify malicious applications using a minimal and optimized set of permissions. The authors collected a dataset of 10,000 applications, including 5,000 malware samples from VirusShare and 5,000 benign applications from the Google Play Store. Permissions were extracted from the `AndroidManifest.xml` files using Androguard. These permissions were then filtered using a Random Forest-based feature importance technique to retain only the most significant ones.

The final feature set was reduced to only five features: three permissions (`READ_PHONE_STATE`, `WRITE_EXTERNAL_STORAGE`, and `ACCESS_COARSE_LOCATION`), along with the application smali size and the permission rate metric.

The system was evaluated using four machine learning classifiers with 10-fold cross-validation. The results showed accuracies of 89.70% with SVM, 89.96% with Random Forest, and 89.52% with Naïve Bayes, while reducing the feature set by approximately 77% compared to the benchmark approach.

The study [27] proposed an autonomous Android malware detection system that combines static analysis, feature engineering, and deep learning. The authors used a dataset from the InvesAndMal2019 repository, which includes specialized malware categories such as ransomware, scareware, adware, and SMS malware. Features such as API calls, opcode sequences, and n-grams were extracted from the dataset samples using static analysis techniques and fed into a Long Short-Term Memory (LSTM) deep learning model to distinguish between malicious and benign applications.

The proposed system achieved an overall accuracy of 85%, with a precision of 0.87, recall of 0.86, and an F1-score of 0.87 for malware classification.

2.3.1 Comparison Table

Ref	Dataset	Techniques Used	Extracted Features	Evaluation Metrics
[26]	VirusShare + Google Play (5,000 malware + 5,000 benign)	ML (SVM, Random Forest, Naïve Bayes)	3 permissions + small size + permission rate	Accuracy, Precision, Sensitivity, F-measure
[20]	Genome + Drebin + Koodous + Contagio + PwnZen (7,533 malware + Google Play (1,493 + 4,500 benign))	Graph-based (Weighted Sum)	Permission Pairs	Accuracy, TPR, FPR
[27]	InvesAndMal2019	LSTM	API calls, opcode sequences, n-grams	Accuracy, Precision, Recall, F1 score
[24]	Android Malware Genome Project (1,119 malware + Google Play Store (621 benign))	Feature selection using GA + classifiers (Naïve Bayes, Decision Tree, SVM)	Android Permissions (16 selected using GA)	Accuracy, TNR, TPR, F1 score
[22]	Drebin + AndroZoo (5,000 malware + 5,000 benign)	Permission Frequency Analysis	Android permissions from `AndroidManifest.xml`	Accuracy
[25]	M0Droid dataset (200 malware + 200 benign)	ML (SVM, Random Forest, Naïve Bayes)	Permissions as binary vectors	Accuracy
[23]	Permission-based dataset from APK files (199 malware + 199 benign)	Feature selection + ML (K-NN, Naïve Bayes, Decision Tree, Random Forest)	Android permissions reduced from 323 to 48 important	Accuracy

Table 2.1: Comparison of Android malware detection techniques

2.4 Design of Our Approach

This schema presents the workflow of the proposed Android malware detection system. It starts with collecting benign and malicious APK files, followed by feature extraction based on Android permissions. After data cleaning and preprocessing, the dataset is divided into training and testing sets. Several models are applied, including LSTM, CNN1D, SVM, Random Forest, and DNN. Their performance is evaluated using accuracy, precision, recall, and F1-score, along with resource consumption comparison. Finally, the best-performing model, Random Forest (RF), is selected and deployed in an Android application for real-time malware detection.

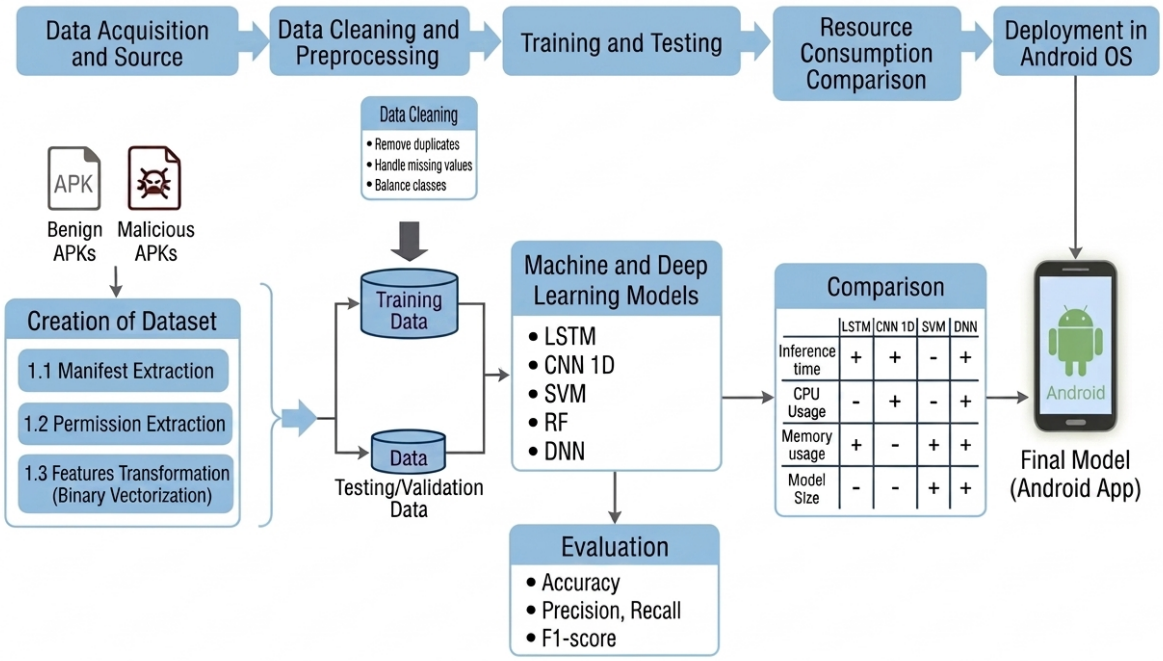


Figure 2.1: Diagram of the malware detection system.

2.5 Dataset Creation

The creation of a dataset is a fundamental step in Android malware detection, as it transforms raw Android applications into structured data that can be processed by machine learning algorithms. Android applications are distributed as APK files, which are compressed archives similar to ZIP files. These files contain various components such as the **AndroidManifest.xml**, compiled code (`classes.dex`), and other resources.

2.5.1 Manifest Extraction

This step involves extracting the **AndroidManifest.xml** file from each **APK**. This file contains essential metadata describing the application, including its components (activities, services, broadcast receivers) as well as many features that can be used for static analysis. Among these features, **the declared permissions** are particularly important, as the manifest file provides a list of all permissions required by the application. Therefore, it is considered a primary source of static information, offering valuable insight into the behavior and capabilities of the application without requiring its execution.

2.5.2 Permission Extraction

Once the manifest file is obtained, the next step is to extract all the permissions requested by the application. Permissions represent the access rights that the application needs to interact with the system or user data. They are particularly useful for malware detection because suspicious applications often request sensitive permissions or an unusually high number of permissions compared to normal applications.

2.5.3 Feature Transformation

Once permissions are extracted, they must be convertedinto a numerical format suitable for machine learning models. Formally, let **R** be the set of all possible permissions. Each application is represented as a binary vector: **Xi = (v1, v2, v3, . . . , vj)**

Where:

- **vj = 1 if the j-th permission is present**
- **vj = 0 otherwise**

This process, known as **binary vectorization**, transforms each application into a structured feature vector that can be directly used by classification models such as SVM and Random Forest.. In addition, not all extracted features are equally useful. Some features may be redundant or irrelevant, which can negatively impact model performance. Therefore, feature reduction techniques are often considered to retain only the most informative features, improving efficiency and reducing model complexity.

In our work, the dataset creation steps described above were not performed manually. Instead, we used **a preprocessed dataset(csv)**, where all features were already extracted and transformed into a suitable format. However, these steps remain essential in the general pipeline of Android malware detection and are necessary to understand how the dataset is constructed.

2.6 Dataset and Preprocessing

In machine learning-based systems, the performance of a model strongly depends on the quality of the dataset and the structure of its features. Preprocessing refers to the set of operations applied to raw data in order to transform it into a clean and structured format suitable for machine learning algorithms. This phase typically includes several key steps, such as data cleaning, feature transformation, feature selection, and dataset splitting.

Data cleaning involves handling missing values, removing duplicate entries, and eliminating inconsistent records that could negatively affect model training. Feature transformation consists of converting raw data into a numerical format compatible with machine learning algorithms. In the context of Android permission-based detection, this is achieved through binary vectorization, where each permission is encoded as 1 if declared by the application and 0 otherwise. Feature selection aims to retain only the most discriminative features, reducing dimensionality and improving model efficiency. Finally, dataset splitting divides the data into training and testing sets, often using a stratified strategy to preserve class distribution.

In our approach, we used the Android_Malware.csv dataset, which contains 29,332 Android applications represented by 86 permission-based binary features, along with a label indicating whether the application is benign (0) or malicious (1). The dataset is notably well-balanced, with 14,632 benign samples (49.9%) and 14,700 malicious samples (50.1%), which is a favorable property for training unbiased classification models. The permissions are already represented in binary format, and no missing values were identified. Therefore, the only preprocessing step applied in our work was the stratified train/test split (80% for training and 20% for testing), ensuring a reliable evaluation of the models on unseen data. The details of this step are presented in the following chapter.

2.7 Machine and deep Learning Techniques

2.7.1 Random Forest (RF)

Random Forest is a supervised machine learning algorithm used for classification and prediction. It constructs multiple decision trees using random subsets of data and aggregates their outputs to produce a final result, reducing overfitting and improving generalization [28]. In this project, Random Forest is used to classify applications as malware or benign based on extracted features such as permissions, improving detection accuracy. Previous studies have shown that Random Forest achieves very high accuracy and low error rates in Android malware detection, confirming its effectiveness [29].

2.7.2 Support Vector Machine (SVM)

Support Vector Machine (SVM) is a supervised learning algorithm that maps data into a highdimensional space using kernels to find an optimal decision boundary between classes. It is effective for separating malware from benign samples [30]. Research shows that SVM classifiers applied to Android malware detection can accurately separate malicious and benign apps using extracted static features, demonstrating the algorithm's practical utility in this domain [31].

2.7.3 Convolutional Neural Network (CNN)

CNNs are deep learning models designed for automatic feature extraction and hierarchical pattern recognition. They consist of convolutional and pooling layers, often enhanced with residual connections or advanced activations to improve accuracy [32]. CNNs are used in Android malware detection to classify apps based on static features like permissions, intents, services, and receivers by transforming these features into images, enabling automatic pattern recognition for accurate malware identification and we are going to use CNN1D in our work [33].

2.7.4 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) specialized in learning long-term dependencies from sequential data. They automatically capture complex patterns without extensive manual feature engineering, making them suitable for analyzing sequences in various domains. In Android malware detection, LSTM models process sequences of app behaviors over time such as permissions, API calls, and events to accurately distinguish benign from malicious applications[34].

2.7.5 Deep Neural Network (DNN)

Deep Neural Networks (DNNs) are multi-layered models capable of learning complex patterns from data. They are widely used in fields such as medical imaging, robotics, and earth observation. Challenges such as lack of transparency, sensitivity to data variations, and overconfident predictions limit their deployment in critical applications. Techniques like uncertainty estimation, including model and data uncertainty, Bayesian methods, and ensemble learning improve reliability [35]. DNNs, including CNNs and LSTMs, are applied to malware detection by learning patterns from features extracted from malware files. Transfer learning leverages pre-trained networks and fine-tunes them on malware datasets, achieving high accuracy in identifying malicious software[36].

2.8 Classification evaluation Metrics

To assess the performance of the proposed malware detection models, we employed five standard evaluation metrics widely used in binary classification tasks.

2.8.1 Confusion Matrix

The confusion matrix is a table that summarizes the prediction results of a classification model. It contains four values:

- **True Positive (TP)**: Malware correctly identified as malware.
- **True Negative (TN)**: Benign apps correctly identified as benign.
- **False Positive (FP)**: Benign apps incorrectly classified as malware.
- **False Negative (FN)**: Malware incorrectly classified as benign.

2.8.2 Accuracy

Accuracy measures the overall proportion of correctly classified samples.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.1}$$

2.8.3 Precision

Precision measures how many of the predicted malware samples are actually malware.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.2}$$

2.8.4 Recall

Recall measures how many actual malware samples were correctly detected.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.3}$$

2.8.5 F1-Score

F1-Score is the harmonic mean of Precision and Recall, providing a balanced measure between the two.

$$\text{F1-Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.4}$$

2.9 Resources conception evaluation Metrics

2.9.1 Training Time

The time required to train the model on the dataset

2.9.2 Inference Time

The time required for the model to make a prediction on a new sample.

2.9.3 Memory Peak Usage

The maximum amount of RAM consumed by the model during execution

2.9.4 Model Size

The storage space required to save the trained model.

2.10 Conclusion

The second chapter presented an overview of Android malware detection using permissions as the primary feature for analysis. It began by explaining the role of Android permissions and their relationship with malicious behavior, highlighting how abnormal permission requests can indicate potential threats. The chapter then described how permissions can be transformed into feature vectors suitable for machine learning models. Various machine learning techniques were introduced, emphasizing their effectiveness in detecting both known and unknown malware. A review of related work demonstrated that permission-based approaches, combined with feature selection and machine learning algorithms, can achieve high detection accuracy while maintaining low computational cost. However, existing approaches still face limitations such as dataset dependency and evolving malware strategies. Based on these observations, the next chapter presents the design and implementation of our proposed system, which aims to improve malware detection performance using an optimized permission-based machine learning approach.

Chapter 3

Implementation and evaluation

3.1 Introduction

In this chapter, we present the development environment adopted for the realization of our project on the detection of malicious Android applications. We first describe the hardware used as well as the software tools utilized during the different stages of the project. Next, we detail the dataset used and the data preparation before training the classification models. We also present the different models used as well as the evaluation criteria chosen to measure their performance. Finally, we present the results obtained, followed by a comparative analysis to identify the advantages and limitations of each model.

3.2 Development environment

3.2.1 Hardware Tools

Computers

Device	Processor	RAM
Computer 1	Intel Core i7-1250U (1.10 GHz)	16 GB
Computer 2	Intel Core i7-1165G7 (2.80 GHz)	16 GB
Computer 3	Intel Core i5-8350U (1.70 GHz)	8 GB
Computer 4	Intel Core i5-8365U (1.60 GHz)	8 GB

Table 3.1: Computer Hardware Specifications

Mobile Devices

- **Phone 1**
  - Name: realme C51
  - RAM: 4 GB
- **Phone 2**
  - Name: Galaxy A54 5G
  - RAM: 256 GB

3.2.2 Software Tools

Software / Tool	**Description**
Kaggle	Kaggle is an online platform used to manage datasets, train models, and test experiments in an interactive cloud environment. It also provides useful resources and community support for data science projects.
Python	Python is an open-source programming language widely used in machine learning and data analysis. It is simple, flexible, and helps speed up development and debugging.
Google Colab	Google Colab is a cloud-based environment used to run Python code with free access to GPU/CPU resources, making it suitable for training deep learning models.
Jupyter Notebook	Jupyter Notebook is an interactive development environment used to write and execute code step by step, and to document experiments clearly.
Android Studio	Android Studio is the official integrated development environment used for building, testing, and debugging Android applications with a rich set of development tools.
Visual Studio Code	Visual Studio Code is a lightweight and powerful code editor used for writing, editing, and managing source code with support for multiple programming languages.
Java	Java is a programming language used for developing Android applications and implementing backend functionalities.
XML	XML is a markup language used to define the structure and design of user interfaces in Android applications.

Table 3.2: Software and Tools Used

3.2.3 Libraries Used

Library	Description
Pandas	Pandas is used to load, organize, clean, and analyze tabular data efficiently. It simplifies handling missing values and different file formats.
scikit-learn	Scikit-learn provides useful tools for data preprocessing, machine learning algorithms, and model evaluation.
TensorFlow	TensorFlow is used to build, train, and test deep learning models efficiently. It supports large-scale computations.
Keras	Keras is a high-level library used with TensorFlow to simplify the creation and training of neural networks.
NumPy	NumPy is used for numerical operations, array processing, and mathematical computations.
os	The os module is used to manage files, folders, and system paths during project development.
Matplotlib	Matplotlib is used to create graphs and plots for result analysis and performance visualization.
Seaborn	Seaborn is used to generate clear and attractive statistical visualizations based on the project data.
LabelEncoder	LabelEncoder (from scikit-learn) is used to convert categorical labels into numerical values suitable for machine learning models.
csv	The csv module is used to read and write data in CSV format for dataset handling and preprocessing.
json	The json module is used to parse, read, and write data in JSON format for structured data exchange.
pickle	The pickle module is used to serialize and save trained models and reload them for later use.
sys	The sys module provides access to system-specific parameters and functions used during program execution.
tracemalloc	The tracemalloc module is used to trace memory allocation and analyze memory usage in Python programs.
psutil	The psutil library is used to monitor system resources such as CPU usage, memory consumption, and running processes.
time	The time module is used to measure execution time and manage time-related operations in the program.

Table 3.3: Libraries Used in the Project

3.3 Results and Discussion

3.3.1 Experimental Setup

Dataset

We utilized the NATICUSdroid dataset comprising 29,332 APK samples with a balanced class distribution (14,632 benign applications and 14,700 malware applications). The dataset was used without additional feature engineering, as it was already preprocessed and structured for machine learning tasks.

Feature Representation

Each application is represented as a binary feature vector, where each feature corresponds to a specific Android permission. A value of 1 indicates the presence of a permission, while 0 indicates its absence.

Train/Test Split

The dataset was divided using a stratified random split with a ratio of 80% for training (23,466 samples) and 20% for testing (5,866 samples). The random seed was fixed at 42 to ensure the reproducibility of results across all experimental runs.

3.3.2 Dataset Description and Visualization

Class Distribution

The dataset contains a total of 29,332 Android applications, including 14,632 benign and 14,700 malware samples. As illustrated in Figure 3.1, the dataset is well-balanced between the two classes, which helps reduce bias during model training and evaluation.

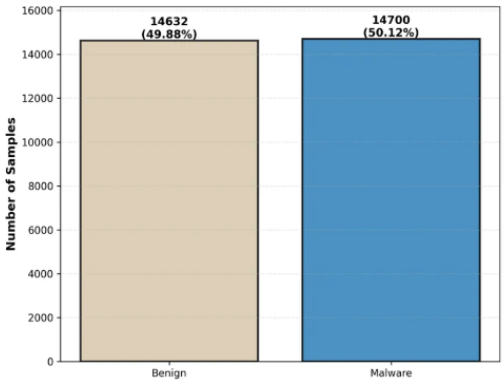


Figure 3.1: Class Distribution

Dataset Characteristics

The dataset consists of 29,332 Android applications described by 86 permission-based features extracted from the AndroidManifest file. These features are represented in a binary format, resulting in a high-dimensional feature space. The dataset is highly sparse, with approximately 89.01% zero values and only 10.99% active features. On average, each application requests 9.45 permissions (±5.82), with values ranging from 0 to 65 permissions per sample.

A class-wise analysis reveals that malware applications request more permissions on average (10.83) compared to benign applications (8.06), indicating distinct behavioral patterns between the two classes. This characteristic difference serves as a strong indicator for classification and validates the use of permissions as discriminative features for malware detection.

3.3.3 Performance Evaluation

Classification Metrics

To comprehensively assess the effectiveness of the implemented machine learning and deep learning models for Android malware detection, we evaluated their performance using four standard classification metrics: Accuracy, Precision, Recall, and F1-Score. These metrics collectively provide a holistic evaluation of model behavior, capturing both overall correctness and class-specific performance characteristics.

Model	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
Random Forest	96.91	97.16	96.67	96.91
SVM	95.36	95.34	95.41	95.38
DNN	96.86	97.62	96.09	96.85
CNN1D	96.80	97.12	96.46	96.79
LSTM	93.76	93.07	94.59	93.83

Table 3.4: Classification Metrics Comparison

Random Forest achieved the highest accuracy (96.91%) with balanced precision (97.16%) and recall (96.67%), demonstrating robust performance across both classes. SVM achieved 95.36% accuracy with nearly identical precision and recall values (95.34% and 95.41%), indicating stable predictions. Among deep learning models, DNN attained the highest precision (97.62%), while CNN1D maintained balanced performance across all metrics (96.80% accuracy). LSTM exhibited the lowest performance (93.76% accuracy and 93.83% F1-Score); this suggests that sequential modeling may not be the most suitable approach for static permission-based features. suitable approach for static permission-based features.

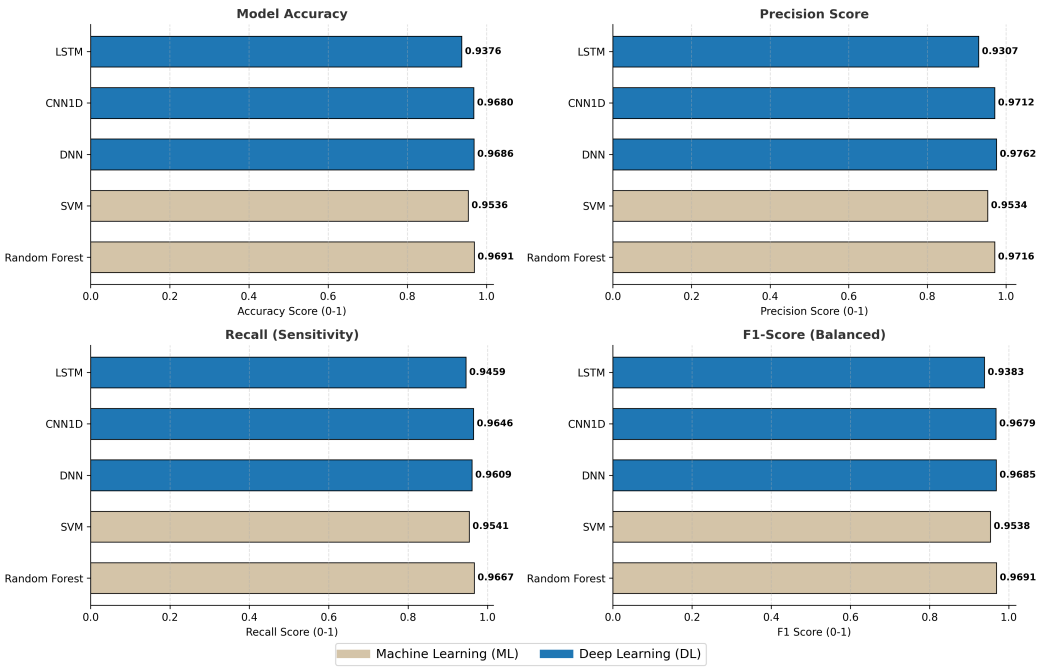


Figure 3.2: Classification Metrics Comparison

Resource Consumption

Beyond classification accuracy, resource Consumption is critical for real-world deployment on resource-constrained mobile devices. Both training efficiency and inference performance must be evaluated to determine practical viability for production systems.

SVM achieved the fastest inference time at 0.22 milliseconds per sample. Deep learning models exhibited substantially longer training times, with LSTM requiring 742.31 seconds, which may limit its suitability for iterative model development. Memory consumption varied significantly, ranging from 12.20 MB (DNN) to 30.98 MB (SVM), with SVM's higher footprint presenting constraints for mobile deployment. CPU utilization during inference ranged from 9.40% (DNN) to 35.45% (Random Forest).

Model	Train Time (s)	Inference (ms)	Memory (MB)	CPU (%)
Random Forest	1.87	39.67	12.37	35.45
SVM	11.53	0.22	30.98	13.40
DNN	34.70	73.48	12.20	9.40
CNN1D	130.86	103.90	12.47	13.05
LSTM	742.31	75.44	14.60	12.00

Table 3.5: Resource Usage Comparison

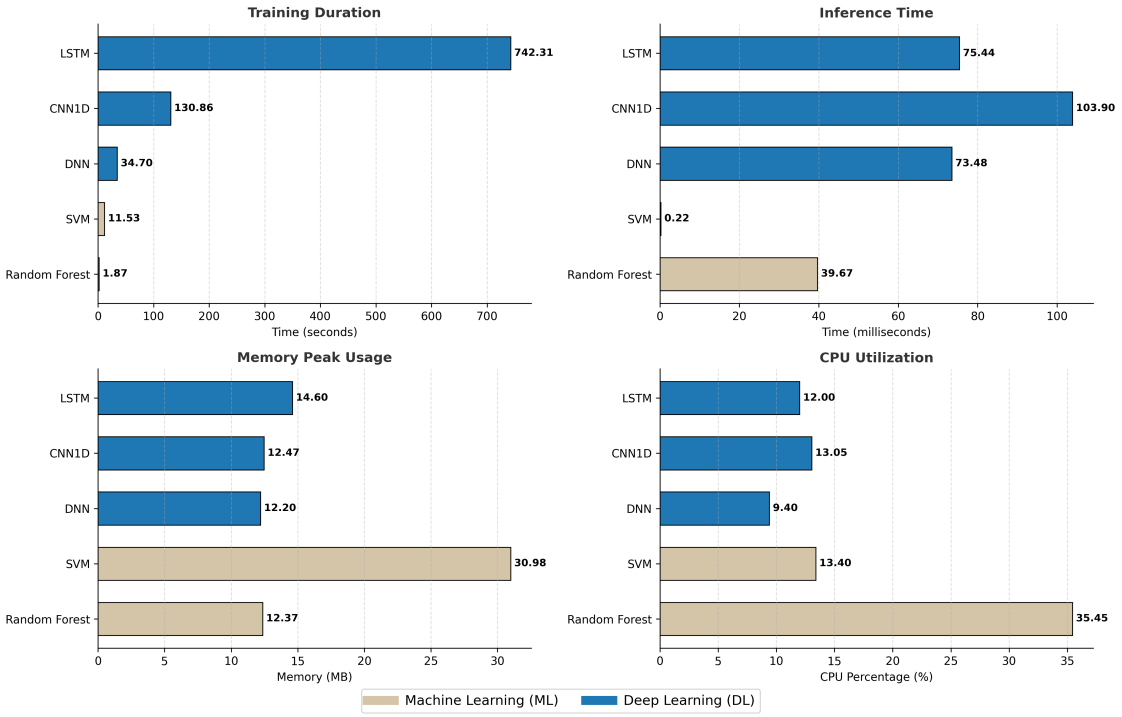


Figure 3.3: Resource Usage Comparison

3.3.4 Best Model Selection

Based on the obtained results, Random Forest was selected as the most suitable model for Android malware detection. In terms of performance, Random Forest achieved the best results among all models, with an accuracy of 96.91% and an F1-score of 96.91%. Its precision (97.16%) and recall (96.67%) are also well balanced, which is important in malware detection to avoid both false positives and false negatives.

Regarding efficiency, the model showed a very fast training time (1.8 seconds) and low prediction time, with an average inference time of 39.67 ms. It also requires a reasonable amount of memory (12.37 MB) and moderate CPU usage (35.45%), which makes it practical for real-world applications. Although deep learning models such as DNN and CNN1D achieved good performance, they require more training time and resource consumption. For these reasons, Random Forest offers a good balance between performance and efficiency, making it the most appropriate choice for this task.

3.4 Application Design and User Interface

3.4.1 Use case diagram

The Use Case Diagram illustrates the interactions between the user and the system, highlighting the main functionalities provided by the application.

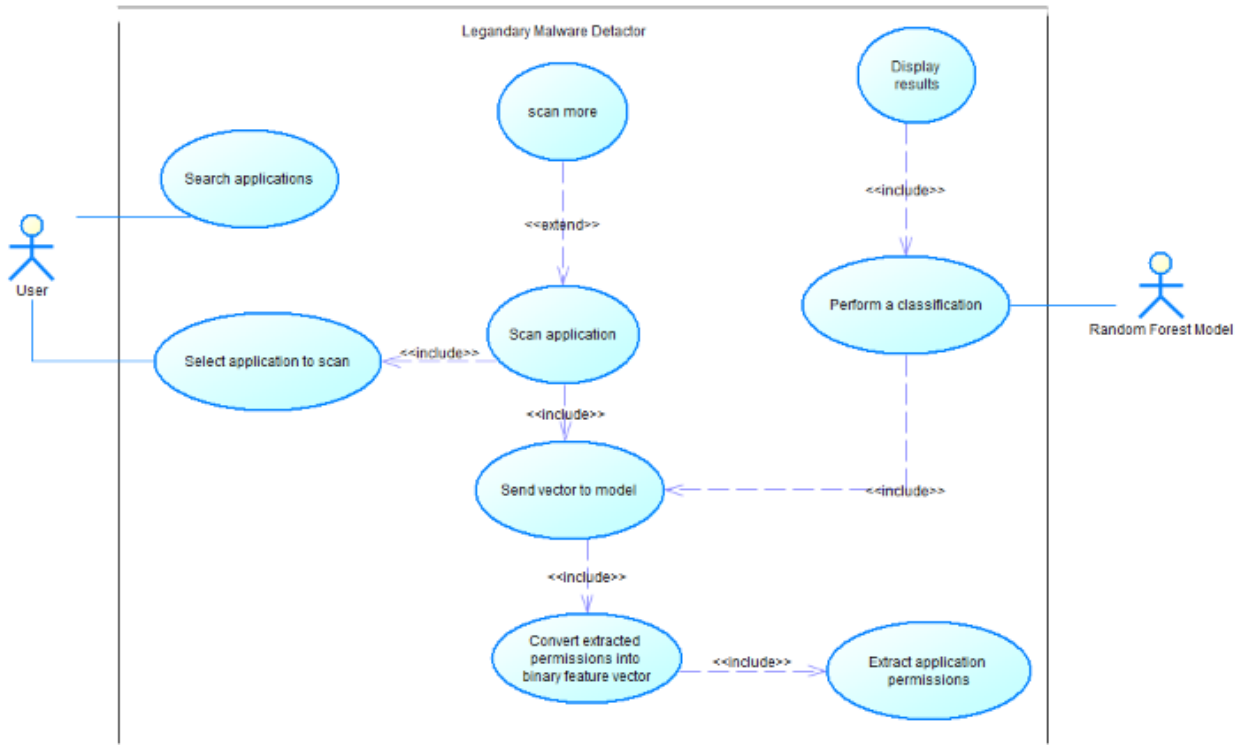


Figure 3.4: Use Case Diagram

3.4.2 Class diagram

The Class Diagram presents the main classes of the application and the relationships between them.

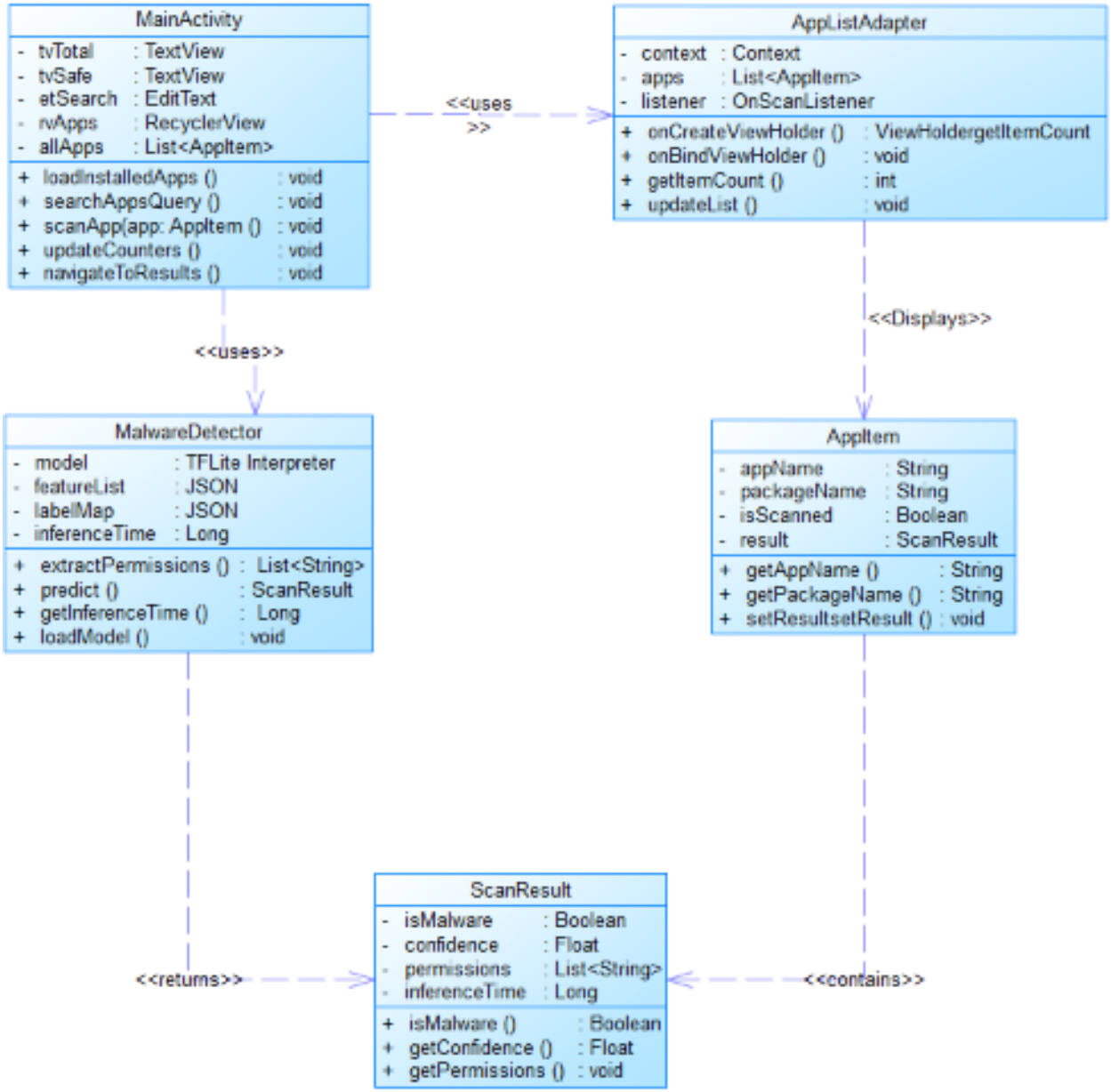


Figure 3.5: Class Diagram

3.4.3 Sequence diagram

The Sequence Diagram describes the sequence of interactions between the different components during the application scanning process. In this case, the Sequence Diagram illustrates the process of scanning an application, from the moment the user selects an app until the classification result is displayed.

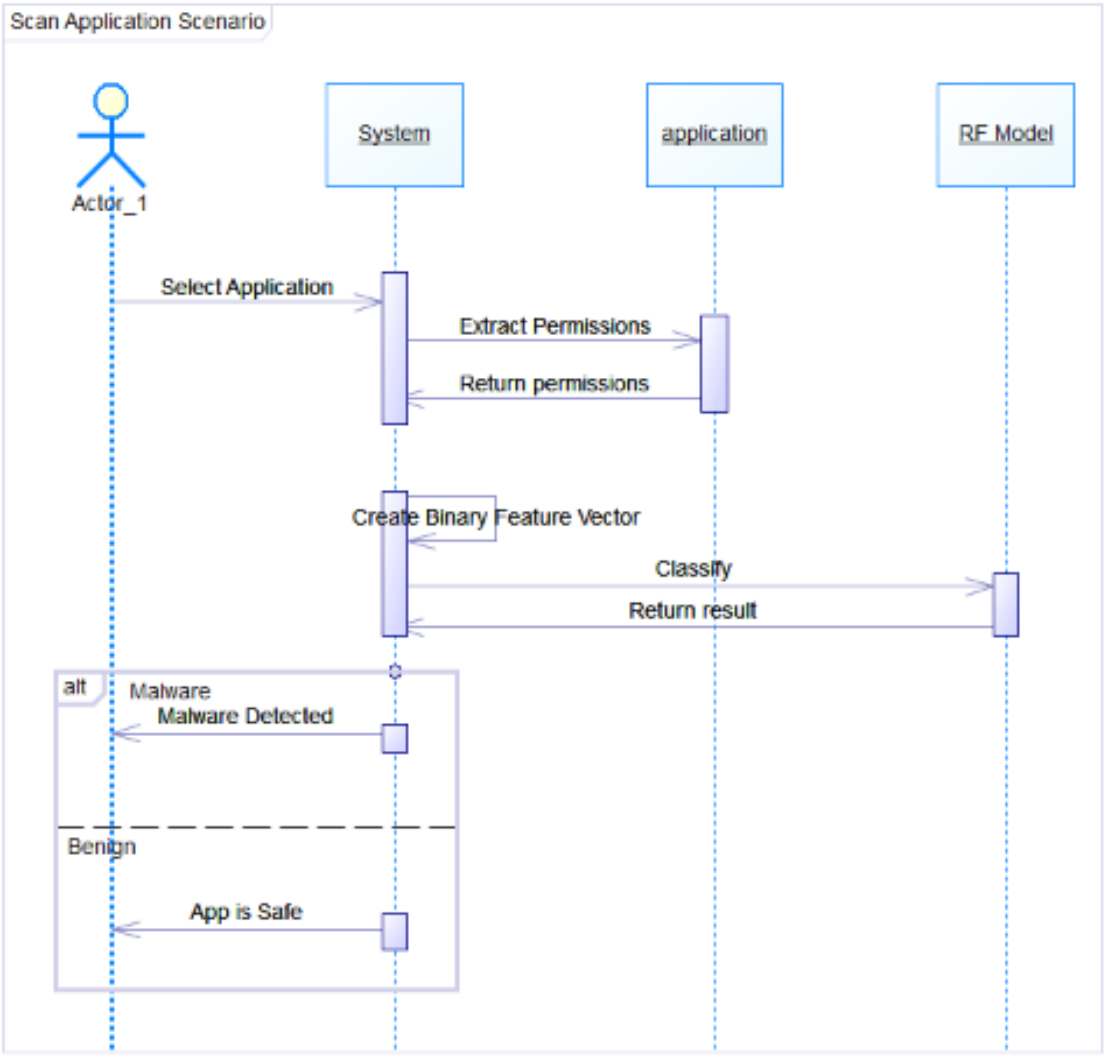


Figure 3.6: Class Diagram

3.4.4 Application Description and User Interface and Results

LegendaryMalwareDetector is an Android mobile application designed to detect malicious software installed on Android devices using a machine learning-based approach. The application integrates a TensorFlow Lite (TFLite) model trained on Android application permissions to classify installed applications as either benign or potentially malicious.

The application features an intuitive and modern user interface that supports both light and dark modes, automatically adapting to the user's system preferences. Upon launching, users are presented with a splash screen before accessing the main interface, which displays a list of installed applications along with a real-time search functionality.

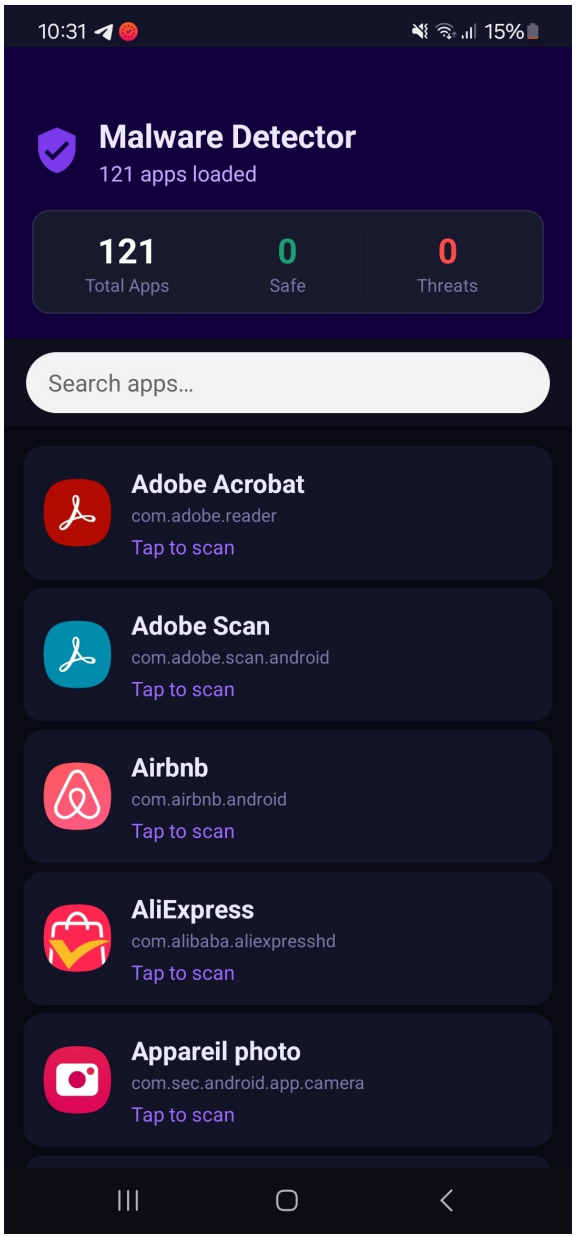


Figure 3.7: Application Home Interface

Users can select one application for analysis through a dedicated selection interface. Once selected, the system extracts the permissions declared by each application and transforms them into a feature vector. This vector is then processed by the on-device TFLite model to generate a classification result, indicating whether the application is safe or potentially malicious, along with a confidence score as shown in Figure 3.8

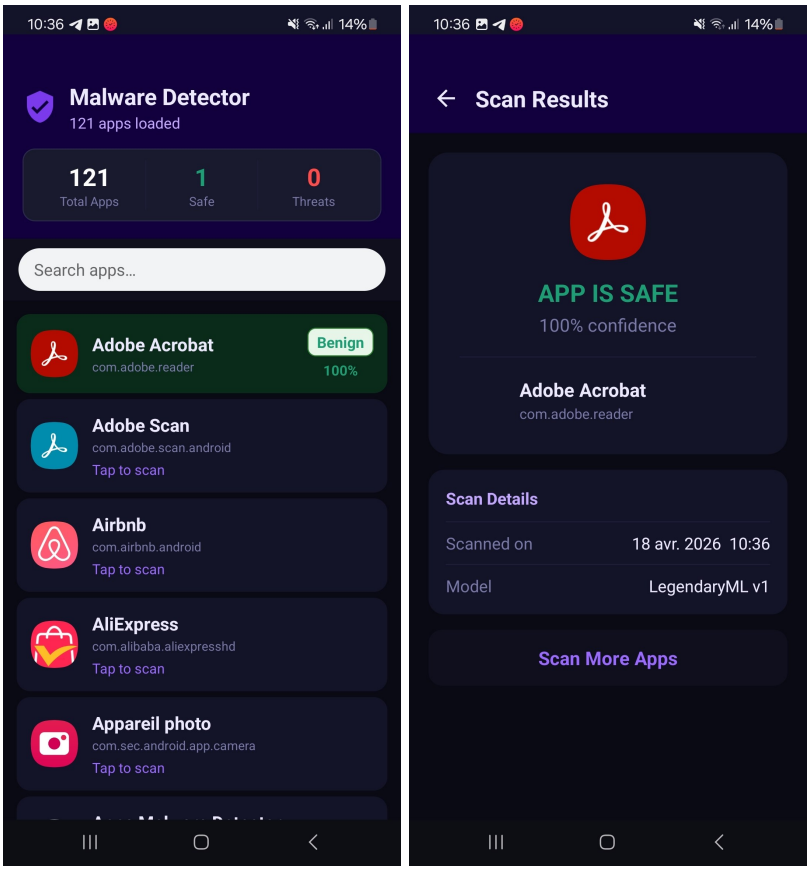


Figure 3.8: Benign Application Scan Result

To further evaluate the application in a realistic scenario, a custom Android application was created using Android Studio and intentionally assigned a set of sensitive permissions commonly associated with spyware behavior. These permissions include access to SMS, contacts, location, microphone, and device state, which are typically considered high-risk in security analysis. When analyzed by the application, the model classified the test application as malware with a confidence score of 87%, as shown in 3.9 This result confirms that the system is capable of identifying suspicious permission patterns even for applications that were not part of the training dataset. It also highlights the effectiveness of permission-based features in detecting potentially malicious behavior. Detection is accompanied by visual warning and haptic feedback, ensuring that the user is immediately alerted when a potential threat is identified.

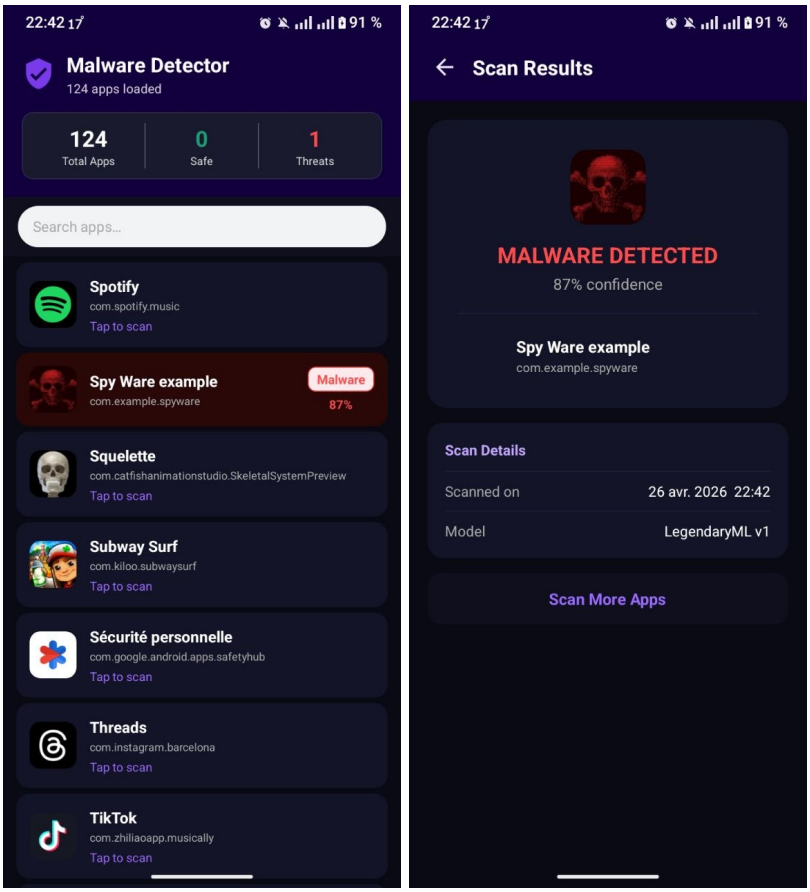


Figure 3.9: Malware Detection Alert Screen

The application was developed using Java and the Android SDK, following a modular architecture that separates the user interface layer from the machine learning inference layer. This design ensures the maintainability, scalability, and efficient integration of the detection model within the mobile environment.

3.4.5 Application Architecture and Permission Extraction

From a technical perspective, the application follows a modular architecture composed of two main layers: the user interface layer and the analysis layer. The permission extraction process is performed by accessing the application's package information through the Android Package Manager, which allows retrieving the permissions declared in the AndroidManifest.xml file without requiring direct file parsing. These permissions are then preprocessed and transformed into a binary feature vector that serves as input to the machine learning model. This design ensures efficient on-device analysis while maintaining a clear separation between data processing and user interaction.

3.4.6 Conclusion

In this chapter, we presented the implementation and evaluation of the proposed Android malware detection system. We described the development environment, dataset preparation, preprocessing steps, and the evaluation of several machine learning and deep learning models. The experimental results showed that the Random Forest model achieved the best balance between detection performance and computational efficiency. Finally, an Android application integrating the selected TensorFlow Lite model was successfully developed to detect malicious applications in a real-device environment.

General Conclusion

The rapid growth of Android applications has increased the spread of Android malware, making mobile security a major challenge. In this project, we proposed a permission-based Android malware detection system using machine learning and deep learning techniques for static analysis.

The first chapter introduced the Android architecture, application components, security mechanisms, malware types, and detection techniques. The second chapter focused on permission-based malware detection methods, machine learning and deep learning algorithms, and related works. The third chapter presented the implementation process, dataset preparation, experimental evaluation, comparative analysis, and the development of the Android application.

The experimental results demonstrated that the Random Forest model provided the best compromise between accuracy and computational efficiency. Moreover, the developed Android application confirmed the effectiveness of the proposed approach in detecting suspicious applications on real devices.

As future work, the system can be enhanced by combining static and dynamic analysis techniques and by using larger and more diverse datasets to improve robustness and detection accuracy.

Bibliography

[1] StatCounter. Mobile operating system market share worldwide – january 2026. https://gs.statcounter.com/os-market-share/mobile/worldwide, 2026. Accessed: January 2026.

[2] Maha Adam Gumaa. Graph approach for android malware detection using machine learning techniques. *Humanities & Natural Sciences Journal*, 2(11):190–203, 2021.

[3] Asim S Yuksel, Abdul H Zaim, and Muhammed A Aydin. A comprehensive analysis of android security and proposed solutions. *International Journal of Computer Network and Information Security*, 6(12):9, 2014.

[4] Google. The android platform. https://developer.android.com/guide/platform, February 2026. Android Developers. Accessed: February 16, 2026.

[5] Android Developers. Application fundamentals. *Google Developers,[Online]. Available: https://developer. android. com/guide/components/fundamentals.[Accessed 14 January 2019]*, 2023.

[6] William Enck, Damien Octeau, Patrick D McDaniel, and Swarat Chaudhuri. A study of android application security. In *USENIX security symposium*, volume 2, pages 1–38, 2011.

[7] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX security symposium*, volume 30, page 88, 2011.

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

[9] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, volume 14, page 19, 2012.

[10] Android Developers. Intents and intent filters, 2009.

[11] Radoniaina Andriatsimandefitra Ratsisahanana. Characterization and detection of android malware based on information flows. 2014.

[12] Google. Apk format. https://developer.android.com/studio/debug/apk-analyzer, February 2026. Android Developers. Accessed: February 19, 2026.

[13] Google. System and kernel security. https://source.android.com/docs/security, 2026. Android Open Source Project. Accessed: 2026.

[14] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. The android platform security model. *ACM Transactions on Privacy and Security (TOPS)*, 24(3):1–35, 2021.

[15] Google. Application sandbox, 2026. Android Open Source Project. Accessed: 2026.

[16] Google. Apk signature scheme v2, 2026. Android Open Source Project.

[17] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, D. Brand, and M. Conti. Android security: A survey of issues, malware and defenses. *IEEE Communications Surveys & Tutorials*, 17(2):998–1022, 2015.

[18] Khaled Bakour, Halil Murat Ünver, and Razan Ghanem. The android malware detection systems between hope and reality. *SN applied sciences*, 1(9):1120, 2019.

[19] Kabakus Abdullah Talha, Dogru Ibrahim Alper, and Cetin Aydin. Apk auditor: Permission-based android malware detection system. *Digital Investigation*, 13:1–14, 2015.

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

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

[22] Juliza Mohamad Arif, Mohd Faizal Ab Razak, Suryanti Awang, Sharfah Ratibah Tuan Mat, Nor Syahidatul Nadiah Ismail, and Ahmad Firdaus. A static analysis approach for android permission-based malware detection systems. *PloS one*, 16(9):e0257968, 2021.

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

[24] Oktay Yildiz and Ibrahim Alper Doğru. Permission-based android malware detection system using feature selection with genetic algorithm. *International Journal of Software Engineering and Knowledge Engineering*, 29(02):245–262, 2019.

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

[26] Fahad Akbar, Mehdi Hussain, Rafia Mumtaz, Qaiser Riaz, Ainuddin Wahid Abdul Wahab, and Ki-Hyun Jung. Permissions-based detection of android malware using machine learning. *Symmetry*, 14(4):718, 2022.

[27] P Sivaprakash, M Sankar, J Vimala Ithayan, R Chithambaramani, RM Dilip Charaan, and D Marichamy. Autonomous android malware detection system based on static analysis. In *2024 International Conference on Integration of Emerging Technologies for the Digital World (ICIETDW)*, pages 1–6. IEEE, 2024.

[28] Hasan Ahmed Salman, Ali Kalakech, and Amani Steiti. Random forest algorithm overview. *Babylonian Journal of Machine Learning*, 2024:69–79, 2024.

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

[30] K. Yasar and F. Tabsharani. What is a support vector machine (svm)?, November 2024. Accessed: 2026-04-17.

[31] Safa J Altaha, Ahmed Aljughaiman, and Sonia Gul. A survey on android malware detection techniques using supervised machine learning. *IEEE Access*, 12:173168–173191, 2024.

[32] Md Himel Reza, Md Noman Biswas Sibly, Shaikh Golam Rabbani, Shafayetul Huda Sadi, Md Faysal Ahamed, Fariya Bintay Shafi, Rusab Sarmun, and Muhammad EH Chowdhury. A comprehensive review of convolutional neural networks: foundations, enhancements and applications. *Neural Computing and Applications*, 38(4):56, 2026.

[33] Ömer Kiraz and İbrahim Alper Doğru. Visualising static features and classifying android malware using a convolutional neural network approach. *Applied Sciences*, 14(11):4772, 2024.

[34] Ahmed Alhussen. Advanced android malware detection through deep learning optimization. *Engineering, Technology & Applied Science Research*, 14(3):14552–14557, 2024.

[35] Jakob Gawlikowski, Cedrique Rovile Njieutcheu Tassi, Mohsin Ali, Jongseok Lee, Matthias Humt, Jianxiang Feng, Anna Kruspe, Rudolph Triebel, Peter Jung, Ribana Roscher, et al. A survey of uncertainty in deep neural networks. *Artificial intelligence review*, 56(Suppl 1):1513–1589, 2023.

[36] Ahmed Bensaoud, Jugal Kalita, and Mahmoud Bensaoud. A survey of malware detection using deep learning. *Machine Learning with Applications*, 16:100546, 2024.