Android Malware Detection using LLM

University of Turku
Department of Computing
MDP in Information and Communication Technology: Cyber Security
July 2025
Madura Bashana Rajapakshe

Supervisors:
Antti Hakkala (University of Turku)
Tahir Mohammad (University of Turku)

UNIVERSITY OF TURKU
Department of Computing

Madura Bashana Rajapakshe: Android Malware Detection using LLM

MDP in Information and Communication Technology: Cyber Security, 84 p., 11 app. p.

July 2025

Android's widespread adoption has made it a prime target for mobile malware, posing significant threats to user privacy, financial security and device security. Traditional malware detection approaches, such as signature-based and heuristic algorithms, frequently fail to keep up with the dynamic nature of Android malware. To solve this issue, this thesis provides a static analysis-based malware detection system that uses fine-tuned transformer models, notably BERT, to categorize Android apps. The system extracts permission ermissions from AndroidManifest.xml and API call sequences from smali code, which are then treated as textual features suitable for language model input. Three different BERT-based classifiers were trained: one with permissions, one with API calls, and one with a combine feature set. The final categorization decision is determined using an ensemble majority-voting approach. Experimental results from the CIC-AndMal2017 dataset indicate that the combined-feature model outperformed both single-feature models and traditional baselines, with an accuracy of 92% and an F1-score of 0.915. The system was deployed as a real-time detection service with a FastAPI backend and a React-based web frontend, allowing for easy malware investigation. This research illustrates the feasibility of using large language models for static malware detection and provides a scalable framework for incorporating future dynamic or hybrid analysis methods.

Keywords: Android Malware, APK, LLM, Malware Analysis

Contents

1 Introduction **2**

1.1 Background and Motivation 2

1.2 Problem Statement 4

1.3 Research Questions 5

1.4 Research Objectives 6

1.5 Scope 7

1.6 Organization 8

2 Literature Review **9**

2.1 Android Malware Landscape 10

2.1.1 Growth of Android Eco System 10

2.1.2 Evolution of Android Malware 12

2.1.3 Common Malware Infection Vectors 13

2.1.4 Techniques Used by Malware Developers 14

2.1.5 Impact of Android Malware 15

2.2 Traditional Malware Detection Techniques 16

2.2.1 Signature Based Detection 16

2.2.2 Heuristic Based Detection 17

2.2.3 Behavior Based Detection 17

2.2.4 Limitations of Traditional Detection Techniques 17

2.3 Machine Learning for Android Malware Detection 19
2.3.1 Traditional Machine Learning Approaches 20
2.3.2 Deep Learning Approaches 24
2.4 Large Language Models for Malware Detection 29
2.4.1 LLM-based existing solutions for malware detection 30
2.5 Research Gaps 36

3 Methodology **38**
3.1 Overview of the Proposed System 38
3.1.1 Initial Assessment 39
3.1.2 How does the proposed system work 41
3.2 Data Collection and Pre-processing 44
3.2.1 Data Collection 44
3.2.2 Data Preparation 46
3.3 Model Architecture 49
3.3.1 Rationale for LLM Selection 49
3.3.2 Fine Tuning Process 51
3.4 Training and Evaluation 54
3.4.1 Training Protocol 54
3.4.2 Evaluation Metrics 55
3.5 Ethical and Legal Considerations 57
3.6 Limitations 58

4 Implementation **60**
4.1 System Architecture 60
4.1.1 Backend Inference Engine (FastAPI) 60
4.1.2 Frontend Interface (React.js) 61
4.1.3 Model Ensemble Framework 61

4.2 APK Processing and Feature Extraction 62
4.3 Real-Time Inference with Pretrained Models 63
4.4 Web-Based User Interface 63
4.5 Deployment Considerations 67
4.6 Monitoring and Updates 68

5 Results Analysis and Discussion **69**
5.1 Quantitative Analysis 69
5.2 Qualitative Analysis 77
5.3 Insights and Observations 78
5.4 Interpretation of Findings 79
5.5 Limitations 79
5.6 Implications for Research and Practice 80

6 Conclusion and Future Work **82**
6.1 Summary of Contributions 82
6.2 Recommendations 83
6.3 Future Research Directions 83

References **85**

Appendices

A Implemented Code Snippets **A-1**

Usage of AI

AI was used to suggest alternative wordings, expressions, and correct spelling and grammar.

Code suggestions and debugging support were done using AI-based assistants.

Notebook Google LLM helped summarize research articles and generate section drafts.

AI has been utilized in brainstorming design strategies, validating technical approaches, and suggesting improvements for system architecture and model evaluation methods.

List of Figures

3.1 Initial Workflow Diagram 41
3.2 Static Feature Extraction from APKs 42
3.3 Training three distinct models 43
3.4 System Diagram 44
3.5 API calls extraction for the creation of API calls Dataset 47
3.6 Permission extraction for the creation of Permission only Dataset . . 48
3.7 Combined Model Test Accuracy 56
3.8 Combined Model Metrics 56
3.9 Fine Tuning Process 57

4.1 Front End of the System 64
4.2 Middle of File Analysis Process 65
4.3 Malicious APK 65
4.4 Extracted Features and Voting Process for Malicious APK 66
4.5 Benign APK 66
4.6 Extracted Features and Voting Process for Benign APK 67
4.7 Loading pre-trained model files 67
4.8 File Storage Paths 68

5.1 Model Performance Comparison 72
5.2 Permission Only Model Metrics 73

5.3 Permission Only Model Test Accuracy 73
5.4 API Only Model Metrics 74
5.5 API only Model Test Accuracy 74
5.6 Combined Model Test Accuracy 75
5.7 Combined Model Metrics 75
5.8 Model HeatMap 76
5.9 Data Distribution in the training Data Set 77

A.1 API Code 1 . . . . . A-1
A.2 API Code 2 . . . . . A-2
A.3 API Code 3 . . . . . A-3
A.4 API Code 4 . . . . . A-3
A.5 API Code 5 . . . . . A-4
A.6 Permission Code 1 . . . . . A-4
A.7 Permission Code 2 . . . . . A-5
A.8 Permission Code 3 . . . . . A-6
A.9 Permission Code 4 . . . . . A-7
A.10 Combine Code 1 . . . . . A-8
A.11 Combine Code 2 . . . . . A-9
A.12 Combine Code 3 . . . . . A-10
A.13 Combine Code 4 . . . . . A-10
A.14 Combine Code 5 . . . . . A-11

List of Tables

2.1 Comparative Analysis of Traditional ML Methods 24
2.2 Comparative Analysis of Traditional DL Methods 28
2.3 Comparative Analysis of LLM-Based Malware Detection Methods . . 36

3.1 Selected hyperparameters used for BERT model fine-tuning 53
3.2 Model performance metrics for different feature sets 54

5.1 Accuracy and F1 score of some comparative models 69

List of Acronyms

GSMA - Global System for Mobile Communications Association

UI - User Interface

SVM - Support Vector Machine

APK - Android Package Kit

OS - Operating System

ML - Machine Learning

LLM - Large Language Model

NLP - Natural Language Processing

API - Application Programming Interface

DL - Deep Learning

DNN - Deep Neural Network

MLP - Multi-Layer Perceptron

LoRA - Low-Rank Adaptation

CFG - Control Flow Graphs

OOD - Out-of-Distribution

BERT - Bidirectional Encoder Representations from Transformers

1 Introduction

1.1 Background and Motivation

Since the early 2000s, there has been a substantial shift in the mobile malware landscape. Initially, mobile devices were considered secure due to their limited capabilities. However, with the advent of advanced technology, the attack surface has expanded, allowing malware to target mobile devices more effectively. The introduction of smartphones provided attackers with new opportunities, further broadening the exploitation environment.

The evolution of Android malware has been rapid and sophisticated. A comprehensive analysis [1] of the development of mobile malware from 2000 to 2020 reveals the variety of malware kinds and their methods of infection. In the early stages, Adware and SMS fraud at premium rates were the primary targets of malware. However, the categories have evolved rapidly, and current malware categories now include ransomware, banking Trojans, spyware, and rootkits. Malware developers have increasingly used social engineering techniques, privilege escalation, and code obfuscation to gain unauthorized access, as well as to evade detection [1] [2]. According to hands-on experience, malicious applications often request permissions such as "READ_SMS", "SEND_SMS", "ACCESS_FINE_LOCATION", or "READ_CONTACTS" etc., while hiding the actual purpose.

Android has become the most popular and widely used mobile operating system

in the world, with more than 5.6 billion unique smartphone users at the beginning of 2024 [3]. According to GSMA Intelligence, 69.4% of the global population uses mobile phones, and of this, around 72% use the Android OS [4]. According to Kaspersky's findings, there has been a significant increase in cyberattacks compared to previous years. In 2023, the number of attacks reached 33.8 million, representing a 50% increase from previous years. Among these attacks, adware was the most prevalent threat, accounting for 40.8% of the total [5].

However, despite its popularity, Android is also the most attacked Operating System (OS), according to Kaspersky [6]. Both Android and iOS face security challenges, but Android is particularly targeted due to its open-source nature, extensive app ecosystem, and diverse device landscape. This makes it more vulnerable to various security threats and malware attacks.

The Android Permission System features several security hardening measures that make it difficult for malware to be installed on the Android system. Every application should specifically request the user's permission during installation to perform specific functions on the device, such as sending an SMS message. However, due to unawareness, most users carelessly grant rights to unidentified apps, which undermines the intent of the permission mechanism. As a result, malicious apps can be easily installed on devices [2]. Additionally, another issue on Android is that the Android ecosystem allows third-party app stores and easy side-loading of APKs, which has eventually led to the proliferation of malicious APKs on the platform.

Traditional malware detection methods, such as heuristic-based, behavioral-based, and signature-based, form the foundations of Malware detection. However, as malware becomes more sophisticated, these approaches face significant limitations. Identifying well-known patterns or "Signatures" of malicious code is where **signature-based detection** relies on. This can be effective against previously identified malware, but it has significant drawbacks. **Heuristic analysis** tries to

identify the malware by evaluating the code structure and the behavior which aims to detect previously unknown threats. However, this also presents challenges to the strategy. Then the third detection technique, **Behavior Based detection**. In behavior-based detection, specific malware programs may not function effectively in a protected environment, such as a virtual system or sandbox. Malware samples may be falsely identified as benign.

1.2 Problem Statement

According to the latest research, 59.99% of the total global web traffic comes from mobile phones, while 37.78% from laptops and computers [7]. Hence, concentrating on mobile malware is the most exciting and time-bound method in contemporary times. The increasing complexities and amount of malware pose a significant and expanding threat to mobile users and their sensitive data. Zero-day malware, Novel malware, and variants utilizing sophisticated obfuscation and anti-detection technologies are challenging to detect using conventional Android malware detection techniques, such as signature-based and heuristic-based approaches [8]. Furthermore, most ML-based methods require manual feature engineering, which can be time-consuming and may overlook the intricate and dynamic nature of malicious apps. Although deep learning methods have demonstrated potential in malware detection [9], there is increasing interest in utilizing the capabilities of Large Language Models (LLMs), especially Transformer based models such as BERT, which have attained cutting edge outcomes in Natural Language Processing (NLP) by deciphering intricate contextual relationships in textual data [10]. Without requiring extensive manual feature engineering BERT has the potential to learn the complex patterns which indicate the malicious behavior due to Android features such as manifest files and API calls [11].

Further research is needed to determine the effectiveness of how well-pretrained

BERT models work when applied directly and modified for Android malware detection, utilizing different data sources (such as manifest files, API call sequences, or combinations of these). Hence, this thesis is prepared to address the problem mentioned above effectively and possibly classify Android malware by investigating the capabilities of the BERT LLM in analyzing Android application data.

1.3 Research Questions

To address the growing threat posed by Android malware, this study explores the following research questions:

1. **RQ1** - How accurately can a fine-tuned BERT model identify Android APKs using features?

   The primary emphasis here is on determining the accuracy of the proposed approach.

2. **RQ2** - How does combining API calls and permissions improve the identification compared to using any one feature alone?

   Will it be possible for extracted permissions alone identify APKs effectively, or is it better to use them in combination with other features? Alternatively, would combining the results from individual features into a new factor improve the overall accuracy?

3. **RQ3** - Is it possible to use contemporary web frameworks to implement a BERT-based detection pipeline for real time analysis?

   Will it be possible to implement a real time analysis with using the web frameworks such as FastAPI?

1.4 Research Objectives

The primary objective of this research is to develop a robust Android malware detection system leveraging the capabilities of Bidirectional Encoder Representations from Transformers (BERT). The primary objective is to determine whether an Android APK is benign or malicious by leveraging BERT's contextual understanding to classify applications.

Primary Objectives:

1. Extracting features from APK: Extracting features such as API calls and permissions would be the main target and main input for the LLM. These features are the main identifiers in distinguishing between malicious and benign applications.

2. Transforming extracting features into a BERT-compatible input : extracted features need to be converted to a structured textual format, which should be suitable for BERT input.

3. Fine Tuning BERT: Fine-tune the BERT using the prepared dataset to classify the APKS correctly

4. Deploying a real-time backend for automated APK analysis: Develop a backend system that can evaluate APK files in real-time.

Secondary Objectives

1. Evaluation using proper metrics: Assess the performance of the developed system using metrics such as accuracy, precision, recall, F1-score.

2. Benchmarking against Baseline Classifiers: Comparing the developed system with other traditional classifiers.

3. Identification of Significant Features Contributing to Malware Classification : Examine how particular features, like particular permissions or API calls, affect the classification result.

1.5 Scope

This research is fundamentally focused on feature extraction and based on the features, fine-tuning a pre-trained LLM model. However, it also presents certain limitations, which are acknowledged in this study.

Scope

1. Analysis of APK files: This research focuses exclusively on Android APKs, the standard format for distributing and installing applications on Android devices. This study does not cover other mobile file formats.

2. Using publicly available datasets to train the BERT model: To train and assess the proposed malware detection model, the study utilizes publicly accessible datasets, specifically the Android malware dataset (CIC-AndMal2017) [12]. Between 2015 and 2017, 5,065 benign APKs and 26 malware samples from 42 distinct malware families were collected for the CIC-AndMal2017 dataset [12].

3. Feature extraction: The study's main goal is to extract static features from APK files, like permissions and API calls. After that, these features are converted into a BERT-compatible format for categorization.

Delimitations

Although dynamic analysis methods, such as behavioral sandboxing, can shed light on how programs behave during runtime, their scalability and operational limitations prevent them from being included in this study. Additionally, since this study primarily focuses on characteristics, dynamic analysis is not required for this training.

1.6 Organization

The thesis is organized as follows: the literature review, which follows the introduction, will provide specific details regarding the Android malware ecosystem, additional information about conventional malware detection methods, and other relevant topics. The methodology section, which outlines the techniques used in this study, follows the literature review. The next chapter focuses on implementation, detailing all deployment and implementation procedures. This is followed by the outcomes and analysis section, which presents the results and their evaluation. The discussion section comes next, leading into the conclusion and future work sections.

2 Literature Review

Extensive studies on mobile security have been triggered by Android-based malware, leading to both traditional and machine learning-based detection methods. This chapter presents a comprehensive overview of the existing literature relevant to Android malware analysis and detection. Moreover, this chapter will begin with the landscape of Android malware, including its evolution, typical attack vectors, and prominent malware families. This chapter then explores traditional detection techniques, including static and dynamic analysis, emphasizing their respective limitations in handling modern malware threats, such as obfuscation and zero-day exploits. The combination of Machine Learning (ML) and Deep Learning (DL) approaches in malware detection is examined in the subsequent sections by focusing on classification performance, feature representation, and generalization capabilities. This is followed by a focused discussion on Large Language Models (LLMs), particularly BERT and transformer-based architectures, and their emerging applications in code understanding and cybersecurity.

Furthermore, this study also focuses on investigating mobile malware feature extraction techniques, including the use of permissions and API calls, which are foundational to static analysis workflows. And finally, the chapter concludes by outlining the current research gaps, highlighting the limited use of context-aware models, such as BERT, in Android malware detection, and sets the stage for the novel contributions of this thesis.

2.1 Android Malware Landscape

Android, as the most widely adopted mobile operating system, has become a significant target for malicious actors due to its openness, extensive user base, and application distribution methodology [13]. With mobile apps serving as intermediaries in personal, financial, and organizational activities, the proliferation of Android malware represents a significant and evolving cybersecurity threat [13]. This section explores the broader landscape of Android malware, examining the ecosystem's evolution, the development of malware over time, typical attack channels, adversarial tactics employed by threat actors, and the impact of such malware on users and systems.

2.1.1 Growth of Android Eco System

Andy Rubin, Rich Miner, Nick Sears, and Chris White founded Android Inc. in Palo Alto, California, in October 2003 [14]. The initial objective was to develop a cutting-edge operating system for digital cameras. But by recognizing the broader potential of mobile devices, the focus turned to creating a flexible mobile operating system. Google acquired Android Inc. in 2005, with the intention of launching a robust, open-source platform into the mobile phone industry. This acquisition established the groundwork for Android's development under Google's direction.

Android has grown at an exponential rate since its inception, becoming the most popular mobile operating system globally. As of January 2025, Android has earned over 72.15% share from the global market, a significant increase from just 12% in 2010 [15]. Due to Android's open-source nature, which makes the process easier for numerous device manufacturers to adopt and customize the operating system, it has become widely used, meeting a diverse range of consumer demands and price points.

Due to its versatility and adaptability, the Android operating system has established a dominant presence across various markets, particularly in regions where

affordability and customization are key considerations. Moreover, its expansive user base has attracted a substantial developer community, contributing to the development of a robust ecosystem of services and applications. Android's open architecture allows users to install applications from third-party sources, in addition to the Google Play Store. While this open-source nature allows more flexibility for developers and consumers, it also poses serious security risks. Applications that are side-loaded frequently evade the security checks of legitimate app stores, increasing the risk of malware infections and data breaches [16].

Furthermore, third-party app stores lack security controls, making them a fertile ground for malicious actors to distribute compromised applications. The possibility of security flaws inside the Android ecosystem is increased by the lack of uniform security protocols across different platforms [17]. The Android Operating system consists of multiple layers, including the Linux kernel, native libraries, the application framework, and apps, which are some of its key components. Although this layered architecture facilitates modularity and flexibility, it presents a wide attack surface for security threats [18].

According to Ahamed Shibly [18], vulnerabilities can exist at various levels of the Android architecture, from the kernel to user-installed applications [18]. For example, research has shown that malicious programs can obtain sensitive data without authorization by taking advantage of security holes in permission models and inter process communication protocols. Furthermore, the fragmentation of the Android ecosystem, encompassing numerous device manufacturers and various custom operating system versions, has made the timely deployment of security updates significantly more challenging. Due to the delay in patching known vulnerabilities, a significant portion of devices are vulnerable to exploitation [18].

2.1.2 Evolution of Android Malware

With the advent of mobile devices and new technology, security challenges have arisen, including mobile malware. The first known mobile malware was "Cabir," identified in 2004, which targeted devices running the Symbian OS. According to Arttu Reijonen [19], this incident marked the beginning of a new era in cybersecurity, where threat actors began to focus on mobile devices as well. Since then, the creation of new mobile malware has kept pace with the development of mobile technology, with increasingly complex and dangerous malware strains emerging in tandem with each improvement in device capabilities.

The development of Android malware has been characterized by a shift from simple threats to extremely complex attacks. According to Zhou *et al.* [20], in the early stages, Simple SMS Trojans malware primarily consisted of SMS Trojans constructed to send premium-rate messages without the user's authorization [20]. One notable example was the Fakeplayer Trojan [21], which was discovered in 2010 and masqueraded as a media player to exploit users.

Furthermore, with the rise of smartphone adoption between 2010 and 2012, malware production became increasingly commercialized, aided by the expansion of black markets for malware generation tools, system vulnerabilities, and stolen data. According to Tam *et al.* [13], reports indicate that attackers are earning up to 12,000 USD per month using mobile malware campaigns. According to F-Secure [22], by 2012, the increasing popularity of Android had earned it 79% of all mobile malware. Advanced features, including overlay assaults for phishing credentials, real-time key logging, dynamic payload downloads, and remote device control, are displayed by contemporary malware families like Cerberus and Vultur [23] [24]. To circumvent conventional security protections and thoroughly integrate themselves into the Android architecture, these recent attacks frequently employ privilege escalation techniques to obtain root access [13]. This complexity increases the need

for more robust and context-aware malware detection methods, specifically in the Android environment.

Android malware can be categorized into several types, each defined by its method of operation and the intention of harm. According to Kaur *et al.* [25], the most well-known categories are Adware, Backdoor, File Infector, Potentially Unwanted Applications or else PUA, Ransomware, Riskware, Scareware, Spyware, Trojan, Trojan-SMS, Trojan-Spy, Trojan-Banker, and Trojan-Dropper. Often, or most of the time, these groups overlap to increase their effectiveness and avoid detection. Contemporary malware samples commonly combine activities from several classes. Due to these kind of scenarios, a need for a comprehensive and adaptive security solution must be implemented.

2.1.3 Common Malware Infection Vectors

The variety of techniques used by attackers to compromise devices is heavily linked to the spread of Android malware. Nowadays, infection vectors are exploiting human and technological weaknesses and then spread the payloads. Among them, the most notable infection vectors are third-party App Stores, Social Engineering, Phishing, and Exploitation of OS vulnerabilities, as well as privilege escalation.

As mentioned earlier, the open ecosystem of Android allows third-party applications to be installed on the Operating System, which also raises security risks. According to Wang *et al.*[26] malicious applications are common in alternative app stores since those app stores lack security controls and verification methods. For example, compared to Google Play [26], which examined more than 6 million Android apps from 16 Chinese app stores, found a significantly greater prevalence of malware, fake, and duplicated apps. According to Kotzias *et al.* [27] Even though Google play was responsible for 87% of all app installations, only accounted for 67% of unwanted app installs, whereas alternative markets, which also accounted for 5.7% of all app

installations, contributed contributed to more than 10% of unwanted app installs.

The next infection vector is Social Engineering and phishing, where one of the most common ways to distribute Android malware. According to Adu-Manu *et al.* [28], psychological manipulation is involved in phishing attacks, which increases the effectiveness of the attack by compromising the user's security. These attacks are extremely difficult to defend against because they take use of human characteristics rather than technological flaws.

Typically, Android malware exploits operating system vulnerabilities to gain elevated privileges, allowing unauthorized access to user data and system resources. When malicious apps take use of OS vulnerabilities to run code with higher permissions than intended, this can lead to privilege escalation attacks. The viability of such attacks has been demonstrated by studies, underscoring the importance of addressing these flaws to enhance Android security.

2.1.4 Techniques Used by Malware Developers

Malware developers employ various techniques to evade detection. One of the famous methods is Code obfuscation and Encryption, which most malware developers use to hide the code and evade reverse engineering the malware. According to Dong *et al.* [29], the Obfuscation method includes control flow alteration, string encryption, and identifier renaming, which make static analysis difficult. According to Elsersy *et al.* [30], these strategies significantly lower the effectiveness of traditional detection methods, making sophisticated analytic tools necessary.

Reflection and dynamic code loading is another method that allows malware to load and execute code at runtime [31]. These methods enable malicious apps to adjust their behavior dynamically, complicating detection. Peoplau *et al.* [32] have highlighted the security implications of these techniques, which have emphasized the need for strong detection procedures.

More sophisticated malware can recognize when it is being analyzed or executed in a sandbox and change its behavior to evade detection. Among the methods include requesting user participation, postponing execution, and looking for emulation artifacts. These evasion techniques make analysis more difficult and make automated detection methods less effective.

2.1.5 Impact of Android Malware

Android malware has had a significant effect on operational, privacy, and economic aspects. Financial loss is a major concern among all other issues, as malware campaigns generate substantial amounts of illicit revenue through identity theft, ad fraud, and unauthorized premium SMS payments. According to Tynan *et al.* [33], one such instance is the HummingBad malware, which affected millions of devices and generated substantial revenue for its attackers by secretly installing malicious software and simulating ad clicks. Android malware also results in privacy invasion and data leaks in addition to financial loss. Zhiyuan Chen [34] reveals that malicious apps have been discovered to capture private user data without consent, including contact lists, text messages, call logs, and geolocation data.

As a conclusion to this section, as the Android ecosystem has expanded, so too have the sophisticated malware threats that exploit numerous infection pathways and employ clever evasion strategies. Creating effective countermeasures requires an understanding of this changing threat landscape. The next section will discuss traditional malware detection approaches, their potential use in addressing such issues, and why these methods are no longer suitable in the current era.

2.2 Traditional Malware Detection Techniques

Early efforts to counter malware threats heavily relied on traditional detection techniques, which continue to form the foundation of many security systems today. These traditional methods are focused on identifying malicious software using known characteristics, behaviors, or manually generated patterns. Signature-based detection remains to be one of the oldest and most widely used techniques, giving rapid identification of known threats [35]. According to Aslan *et al.* [35], this traditional method is effective against known malware, but does not perform well against zero-day malware. However, as malware evolved to use obfuscation, polymorphism, and dynamic behavior changes, heuristic-based and behavior-based detection approaches arose to detect suspicious activity in addition to static signatures.

The following section discusses the foundational principles of signature-based, heuristic-based, and behavior-based detection, followed by inherent limitations that motivate the exploration of more advanced, learning based detection systems.

2.2.1 Signature Based Detection

A signature is a characteristic in malware that encapsulates the program structure and identifies each malware instance uniquely. Typically, when a malware is written, a signature is embedded into its code, which can be used to determine the malware family [35]. Then, when an antivirus identifies the malware, it typically breaks down the malicious file code and looks for patterns that indicate the presence of malware. This method works well and quickly to identify known malware, however, it is not enough to identify novel malware. Additionally, by employing obfuscation techniques, malware from the same family can readily evade detection using signatures. Malware signatures are stored in a database and subsequently compared throughout the detection process. Another name for this type of detection method is matching or scanning for strings or patterns. It may also be hybrid, dynamic, or static [36].

2.2.2 Heuristic Based Detection

According to Aslan *et al.* [35] in recent years, heuristic-based detection has been utilized numerous times. This is a complex detection method that leans on experiences and different techniques, such as rules and ML techniques. Using this method, it can identify even zero-day malware with a higher degree of accuracy, but it is unable to identify complex malware. According to Rabia Tahir [36], heuristic-based detection identifies or differentiates between normal and unusual behavior of a system, allowing for the identification and resolution of both known and unknown malware attacks. This detection method consists of two steps, where the first step involves monitoring the system's behavior when there is no attack and documenting crucial information that can be examined and validated in the event of one. To identify the differences within a specific malware family, this method will be employed. Despite being an effective technique, heuristic based detection has drawbacks, including a high false positive rate and the requirement for additional resources [36].

2.2.3 Behavior Based Detection

Behavioral malware detection techniques observe program behaviors using monitoring tools and determine whether the program is malware or not [35]. According to Aslan *et al.* [35], even when the program codes are being altered, the behavior of the program will be similar, and thus this approach can identify the majority of new malware. However, some malware binaries do not work in a protected environment (virtual machine, sandbox environment). Thus, malware samples might be misclassified as benign [35].

2.2.4 Limitations of Traditional Detection Techniques

Traditional malware detection methods, including signature-based, heuristic-based, and behavior-based approaches, have been foundational in cybersecurity; however,

they have severe limitations in the face of emerging threats. According to Natsos *et al.* [37], the limitations of Signature-based detections are:

- Ineffectiveness Against Zero-Day Threats: Recent or otherwise unknown malware variants can be easily bypassed by this method, as they will not be cataloged until the malware enters the community.
- Polymorphic and Metamorphic Malware: Malware writers normally use code obfuscation techniques, including encryption or polymorphism, to change the malware's appearance without affecting its operation. This will make it difficult to understand its original behavior since the code was obfuscated.
- Maintenance Overhead: Adding newly created malware strains to the signature database on a regular basis requires a significant amount of resources, and it will likely struggle to keep up with the pace.

Then, when it comes to Heuristic Based detections, according to Shaukat *et al.* [38], the limitations of Signature-based detections are:

- Higher false positive rate: Due to the heuristic rules, legitimate Apps can be identified as malicious, which increases the false positive rate [38].
- Evasion Techniques: Advanced malware can evade heuristic checks by imitating benign behavior or obfuscating malicious actions to bypass heuristic checks [38].
- Limited Scope: Heuristic approach may not cover the full spectrum of the malware behavior, which could result in insufficient detection [38].

Lastly, when it comes to Behavior-Based detections, the identified limitations are as follows, according to Aslan *et al.* [35].

- High-dimensional and noisy feature sets: Approaches related to feature extraction, such as n-gram, n-tuple, and graph models, can provide a huge number of features, but most of them are redundant or irrelevant, complicating detection and raising false positives.

- Challenges in similarity measurement: When behaviors are partially overlapping, algorithms such as Hellinger distance, cosine similarity, and chi-square struggle to distinguish between benign and malicious behavior.

- Sandbox evasion by malware: Some of the advanced malware samples are coded to know they are executing inside an analysis lab in a sandbox or virtual machine and will crash, delay execution, or execute harmlessly to avoid detection.

- Obfuscation hinders runtime analysis: Code obfuscation techniques such as encryption or dynamic code loading make it difficult to track actual behavior at run time and thus reduce the effectiveness of behavioral logging.

It is evident that conventional detection methods alone are insufficient for effective security, given the increasing complexity and frequency of Android malware attacks. The limitations of the above-mentioned methods underscore the need for a novel approach to detecting malware, with a focus on the Android OS.

2.3 Machine Learning for Android Malware Detection

Signature-based detection approaches have become ineffective as the complexity and scale of Android malware have grown. To resolve this, researchers have employed machine learning (ML) techniques that can generalize from known samples and

detect previously unknown threats. To differentiate between benign and malicious apps, ML-based techniques utilize several factors, including permissions, API calls, control flow, and behavioral patterns. These approaches are effective for identifying obfuscated or zero-day malware and may be used on both static and dynamic data sets.

This section presents a comprehensive review of ML strategies used for Android malware detection. Subsection 2.3.1 illustrates conventional ML techniques, which often use handmade features and classical classifiers like SVMs and decision trees, while Subsection 2.3.2 investigates deep learning-based models.

2.3.1 Traditional Machine Learning Approaches

Traditional Machine Learning (ML) methods have dominated Android malware detection by relying on static information retrieved from Android application packages (APKs), such as permissions, API calls, hardware components, and network addresses. Such methods make decisions based on manual feature engineering, where domain experts manually select and gather features suspected of being related to malicious behavior. These methods were effective in the early stages of detecting malware, but they struggle against advanced adversary practices, such as code obfuscation, dynamic loading, and polymorphic malware. Foundational works such as Drebin [2], DroidAPIMiner [39] and MAMADroid [40].

Drebin

Drebin [2] was introduced as a lightweight approach for detecting Android malware that runs directly on smartphones. It combines static analysis with machine learning. The core process involves statically analyzing applications, representing them in a format suitable for machine learning-based detection, and providing explanations for the results. Drebin follows four main steps to detect Android malware. One

of the key steps is Static Feature Extraction, where Drebin performs lightweight static analysis on Android applications by extracting features from the manifest and dex code, converting them into sets of strings [2]. The next step is Feature Vectorization, where these features are transformed into a high-dimensional, sparse binary vector space, allowing for pattern-based comparisons between applications [2]. In the Machine Learning Detection phase, a linear SVM is trained offline to differentiate between benign and malicious applications using their feature vectors. This trained model is then deployed on devices for malware detection [2]. Finally, in the Explainability step, Drebin identifies and explains the most influential features contributing to a detection decision using sentence templates, enabling users to understand why an application has been classified as malicious [2].

According to Arp *et al.* [2], Drebin extracts features from Android applications, and out of that, four are from Manifest-based features and the other four are from Code-based features. Manifest-based features are Hardware components, requested permissions, App components, and filtered intents [2], and code-based features are restricted API calls, Used permissions, Suspicious API calls, and Network addresses [2]. These extracted features are vectorized into binary representations (presence/absence) and categorized by a linear Support Vector Machine (SVM). The strength of Drebin relies on its interpretability, where features are human-readable, and SVM weights reflect their relevance, especially in an example such as REQUEST_INSTALL_PACKAGES, which is highly weighted as suspicious. Nevertheless, due to its reliance on static analysis, this approach is vulnerable to Code Obfuscation and Dynamic Loading (Malware that fetches payloads at runtime leaves no static traces) [2].

DroidAPIMiner

DroidAPIMiner [39] is an effective and lightweight Android Application malware classifier that utilizes API level data within the bytecode to distinguish malware from benign applications. DroidAPIMiner uses general data mining methods to automatically learn malware patterns. This process involves three main phases: feature extraction, feature refinement, and model learning and generation. In Feature Extraction, APK files are statically examined to extract API calls and package-level information from bytecode, as well as requested permissions. In Feature Refinement, third-party APIs are filtered out to avoid false positives. APIs used in malware are selected. For APIs used by both malicious and benign apps, data flow analysis is employed to examine parameter values, and then features are selected based on the prevalence of risky values in malware. In Model Learning and Generation, trained classifiers (ID3, C4.5, KNN, and SVM) using purified feature vectors, with a 2/3 training and 1/3 testing split for evaluation.

According to Aafer et al. [39], DroidAPIMiner uses Information Gain (IG) and Chi-Squared (χ^2) metrics to rank API calls based on their discriminative strength. A decision tree classifier then utilizes these ranked APIs for classification. However, DroidAPIMiner has certain limitations, including semantic blindness and permission ignorance. Semantic blindness refers to the treatment of API calls as isolated keywords—e.g., getSubscriberId() is flagged as risky regardless of its actual context [39]. Permission ignorance means the system fails to correlate API usage with the corresponding permission declarations—for example, using getLastLocation() without checking for the ACCESS_FINE_LOCATION permission [39].

MAMADroid

MAMADroid [40] is based on application behavior, especially the sequence of abstracted API calls made by an app. Then these sequences are turned into a Markov

chain with probabilistic modeling of API family transitions (e.g., android.app → android.telephony → java.net). Then, a Random Forest Classifier detects deviations from benign app behavior. The system process involves four main operational phases:

1. Call Graph Extraction: This will perform static analysis using tools such as Soot to extract call graphs from APKs.

2. Sequence Extraction: API call sequences are formed by following pathways from the call graph's entry nodes.

3. Markov Chain Modeling: Then, the API calls are abstracted into packages or families.

4. Classification: Classifiers are trained using feature vectors (Random Forest, 1-NN, 3-NN). The use of PCA achieves dimensionality reduction. SVM was tested but rejected owing to low performance.

Even though this approach analyzes malware based on API calls, it has notable limitations. One such limitation is the loss of granularity, where APIs are abstracted to their package level, which discards important method-level context that could be critical for accurate detection [40]. Another issue is permission-API decoupling, meaning the system does not evaluate whether the permissions requested by an app are justified by its API usage. For instance, an application might request the READ_SMS permission without invoking any SMS related APIs, which could be a sign of malicious intent [40].

Even though traditional machine learning approaches form the foundation for malware detection, they are limited by their reliance on static data, lack of contextual reasoning, and inability to provide actionable insights. Emerging techniques, such as obfuscation and dynamic code loading, further exacerbate these limitations, underscoring the need for models that can understand semantics and make

explainable decisions. While some of these shortcomings can be mitigated by deep learning, it also introduces new challenges related to computational overhead and interpretability. These aspects will be discussed in the next subsection. A comprehensive summary of machine learning-based approaches is presented in Table 2.1.

Tool	**Technique**	**Features Used**	**Strengths**	**Limitations**
Drebin	Linear SVM	Permissions, APIs, URLs	Interpretable, lightweight	Static-only analysis, Lacks code semantics
Droid-APIMiner	Decision Trees	Ranked API calls	Efficient API prioritization	Ignores permissions, Lacks contextual analysis
MAMA-Droid	Markov Chains + Random Forest	API call sequences	Resilient to code changes	Over-abstracts APIs, Lacks permission checks

Table 2.1: Comparative Analysis of Traditional ML Methods

2.3.2 Deep Learning Approaches

Deep Learning (DL) has transformed the Malware detection process by automating feature extraction from raw APK components, hence lowering reliance on manual engineering. Most of the time, traditional Machine learning depends on manually chosen features, such as permission and API requests, curated by domain experts. DL models, on the other hand, ingest raw or minimally processed APK data, e.g., Dalvik bytecode, Smali instructions, or manifest files, and automatically learn hier-

archical representations of code structure, behavior, and resource usage. Also, DL models can identify complex malware, such as zero-day and polymorphic malware, due to learning from a large and diverse dataset. Despite its advantages, identifying malware using deep learning is not always feasible due to certain limitations, as discussed in this section. Several instances of deep learning applications in malware identification are outlined below.

Deep-Droid

Deep-Droid [41] is an Android detection framework based on Deep Learning, which uses Convolutional Neural Networks (CNNs) and Long Short Term Memory (LSTM) networks to analyze static and dynamic characteristics. The Deep-Droid methodology is built on static analysis, which means studying an Android application without executing it. This utilizes a sequential neural network design, comprising three layers: the input layer, the hidden layer, and the output layer. The input layer contains 215 neurons, representing the number of application characteristics employed. The hidden layer consists of 25 neurons, and the output layer contains only one neuron for binary classification (benign or malicious). The model employs static analysis elements generated from categories such as Manifest Permissions, Intents, Commands, and API Calls [41].

Although this approach offers strengths such as higher detection rates and resilience to code obfuscation [41], its limitations include computational overhead and the black-box nature of models that fail to explain why specific features are flagged as malicious.

MalDozer

Maldozer which is another automated android malware detection and family attribution framework and this was designed to address the rising demand for sophisticated,

automatic, and light weight malware detection tools due to the growing use of the Android OS on mobile and Internet of Things (IoT) platforms, which makes them soft targets for malware applications. According to Karbab *et al.* [42], Maldozer processes raw API call sequences in DEX files, automating feature extraction using method embedding (e.g., word2vec/GloVe), and converting APIs into semantic vectors. These vectors are then input into a neural network (convolutional layers, global max pooling, etc.) to learn malicious/benign patterns automatically, with minimal operator intervention.

According to Karbab *et al.* [42], Maldozer is resistant to some of the obfuscation techniques used in API calls, efficient for apps written in Java/Kotlin, and can accurately identify, detect, and classify malware into relevant families. However, all of these strengths are overshadowed by the limitations, as MalDozer is not resilient to dynamic code loading and reflection obfuscation. Furthermore, it ignores native code, which native payloads may evade, and lastly, it does not support non-DEX applications. In brief, MalDozer provides effective automated static analysis but remains susceptible to advanced evasion mechanisms that circumvent static inspection.

R2-D2

According to Huang *et al.* [43], R2-D2 is also a deep learning-based Android malware detection system that employs CNNs to examine Android application bytecode. The most notable innovation is the conversion of the classes.dex bytecode into a fixed-size RGB color picture using hexadecimal values to color codes [43]. The image representation enables the system to treat malware detection as an image classification problem. After the transformation, an Inception v3 CNN model was used to analyze the picture, which automatically extracts features and determines if the APK is malicious or benign. This technique eliminates the need for manual

feature engineering, allowing for a complete learning process [43].

According to Huang *et al.* [43], R2-D2 has outstanding accuracy and performance, indicating over 98% accuracy. Additionally, this has the potential to identify previously undisclosed malware and detect visual similarities across malware families, enabling quick categorization. Despite its strengths, R2-D2 suffers from several shortcomings. The transformation of bytecode into an image may introduce artifacts, allowing malware to go undetected if uncorrelated bytecode happens to present itself as correlated in the image by chance. Although the use of pooling layers is effective, it can potentially lower detection accuracy by stripping away semantic context from the image representation of the code. Furthermore, the CNN model is too huge to execute solely on the device, necessitating a client-server architecture for detection. Finally, since malware changes, R2-D2 requires frequent sample collection and model upgrades to remain accurate. While it works well on big, realistic datasets, some assessment metrics may be slightly worse than those trained on smaller, more controlled datasets. A comprehensive summary of Deep learning approaches can be found in Table 2.2.

Application	**Technique**	**Features Used**	**Strengths**	**Limitations**
Deep-Droid	CNN + LSTM	Static + dynamic features	High detection rate, robust to obfuscation	High computation cost, lacks interpretability
MalDozer	LSTM + method embeddings	API calls (DEX)	Accurate malware family detection, some obfuscation resistance	Fails on dynamic loading, ignores native code, DEX-only support
R2-D2	CNN	Runtime API sequences	Detects unknown malware, good accuracy	API over-abstraction, artifact risk, lower precision

Table 2.2: Comparative Analysis of Traditional DL Methods

Traditional ML and DL approaches have undeniably increased Android malware detection to a certain extent, and their limitations reveal a critical mismatch between their capabilities and the evolving complexity of modern threats. For instance, Drebin [2] and Maldozer [42] excel at detecting known malware patterns, considering permissions, APIs, and code components as distinct keywords or sequences. This "contextual blindness" keeps individuals from answering basic questions: Why is a calculator software seeking access to SMS messages? Is the application's intended purpose sufficient justification for using sensitive APIs like Runtime.exec()? These semantic dependencies remain neglected, leaving models vulnerable to attacks like obfuscation or dynamic code loading.

Deep learning's "black box" nature adds another layer of complexity to this. Models such as LSTM and CNN are less transparent in their decision-making processes. Businesses and regulators are increasingly seeking explainability, not only to be able to trust the system but to meet the requirements of regulations that mandate why an application has been identified as malicious. Therefore, gaps such as those mentioned above can be addressed using large language models, which will be discussed in the section below.

2.4 Large Language Models for Malware Detection

In the context of malware detection, Large Language Models (LLMs) provide transformational solutions. Unlike traditional methods, LLMs are primarily designed to understand context. LLMs are trained on large code and text corpora to understand bidirectional correlation between features, such as correlating permissions with their actual function. This contextual awareness enables LLMs to asking questions like, "Is this app's behavior compatible with its declared purpose?" Furthermore, LLMs can address the issue of explainability. Attention methods are used in the LLM server as a built-in highlighting tool, indicating which permissions, APIs, or code snippets were most influential in a choice.

Finally, LLMs are also excellent at adaptability. LLMs are pre-trained on diverse datasets, which requires minimal fine-tuning to react to new threats, thereby decreasing the dependency on tagged malware samples. Their ability to analyze obfuscated code helps to bridge the gap left by static and sequential models.

In summary, the improvement of LLM has revolutionized malware detection by combining deep learning with its features, as expected by humans.

2.4.1 LLM-based existing solutions for malware detection

This section discusses four currently developed malware detection systems that use Large Language Models (LLMs), highlighting their key approaches and limitations. It also describes the unique characteristics and benefits of the proposed system, emphasizing how it addresses current gaps and enhances detection capabilities. These four solutions are AppPoet [44], Android Malware Detection Using BERT [11], LAMD [45] and LLM-MalDetect [46].

AppPoet

Zhao *et al.* [44] proposed AppPoet, a large language model based system for Android malware detection. According to Zhao *et al.* [44], this implementation aims to overcome the limitations of classic learning-based approaches, particularly their inability to extract behavioral semantic data and generate human-readable, interpretable reports. AppPoet leverages the success of LLMs in natural language comprehension and complex reasoning, and utilizes these skills to extract and analyze semantic links between application characteristics. AppPoet is a multi-view system supported by an LLM. This utilizes a static analysis approach to extract features from the APK file, such as APIs, permissions, URLs, etc. After that, the extract features are divided into three separate observation views, such as Permission View, API View, and URL, and uses feature View [44].

One of the main aspects of AppPoet is its multi-view prompt engineering method. This method guides an LLM (gpt-4-1106-preview) to do two core activities for each view mentioned above.

1. Generate function description for individual features within the view.

2. Generate behavioral overviews of the overall perspective.

According to Zhao *et al.* [44], text outputs from LLM are then turned into

machine-readable representation vectors by utilizing an embedding model (text-embedding-ada-002). These vectors are then concatenated and fed into the DNN classifier to determine whether the APK is malicious or benign. According to Zhao *et al.* [44], a DNN classifier was used instead of relying solely on an LLM, in order to mitigate potential "hallucinations" that LLMs may produce in specific domain tasks.

AppPoet consists of several strengths and weaknesses. One of its strengths is the usage of LLM to perform deep semantic analysis on retrieved static characteristics [44]. Additionally, AppPoet is capable of capturing both explicit and implicit behavioral links in app data, which increases detection accuracy and interpretability. These results are higher than those obtained when compared to Drebin [2] and MaMaDroid [40]. To enhance the quality of LLM output, a multi-view observation technique has been implemented, categorizing aspects such as permissions, APIs, and URLs. Another notable feature is its ability to create human-readable reports. Additionally, AppPoet features a memory module that enhances feature interpretation speed and consistency, and it makes final decisions using a Deep Neural Network (DNN) classifier, specifically an MLP [44].

However, AppPoet also has some critical drawbacks, which highlight the limitations of its usability. AppPoet is a static-only analysis tool, which results in a higher false positive rate and is unable to capture dynamic behaviors, such as runtime code execution. Additionally, another drawback is that the use of LLM API (e.g., GPT-4 and embedding models) introduces new issues, such as network dependency, cost overhead, and vulnerability to hallucinations, in which the model produces inaccurate or contrived outputs [44]. Due to this, there is a need for an error-checking mechanism to ensure factual correctness. Furthermore, due to the changing nature of malware, regular updates to datasets are necessary to keep the model effective. Furthermore, AppPoet does not provide a publicly available model for external par-

ties to use, as the proposed system will be introduced with the above-mentioned issues [44].

Furthermore, AppPoet does not provide a publicly available model for external use, whereas the proposed system aims to address the aforementioned limitations while offering an accessible solution for third parties.

LAMD

LAMD is a Context-Driven Android Malware Detection and Classification with LLMs [45] primarily addresses two significant challenges, as identified by Qian et al. [45]: the excessive amount of support code in APKs that exceeds the context window limitations of large language models (LLMs), and the complex interdependencies between program components that hinder linear reasoning. LAMD is composed of two main components designed to tackle these issues. The first component, Key Content Extraction, analyzes APK files to identify suspicious APIs, particularly those involving the access and transmission of sensitive data. A proprietary backward slicing technique is then applied to extract relevant code, variables, and dependencies, resulting in a refined Control Flow Graph (CFG). This process suppresses irrelevant benign code while preserving the semantic context necessary to understand potentially malicious behavior [45]. The second component, Tier-wise Code Reasoning, employs a three-tiered hierarchical reasoning approach to manage LLM token limitations and enhance semantic understanding effectively. Together, these components enable LAMD to perform more context-aware and explainable malware detection.

When considering the strengths and weaknesses, overcoming traditional limitation such as inadequate adaptation to changing assaults, dataset bias, and restricted interpretability is one of its core. Then, strengths such as utilizing LLM for reasoning, handling the complex code by simplifying the structural complexity using hierarchical analysis, providing explainable results are some of its advantages [45].

Despite its strengths, this also has some serious drawbacks, such as LLM hallucination risk, a higher false positive rate in OOD due to minimal task-specific training. Furthermore, the approach is based solely on static analysis, which limits its ability to detect behaviors that emerge only during program execution. Additionally, since the approach is primarily based on static analysis, which limits its ability to detect behaviors that emerge only during program execution [45]. Due to the static-only approach in LAMD [45], this may result in an insufficient understanding of specific malware. LAMD's dependence on sophisticated external LLMs, such as GPT-4, incurs enormous computational expenses and resource demands; for example, testing over 9,000 APKs requires around $1800 in API usage [45]. Moreover, since the system is relying on network connectivity for API access, it is susceptible to communication failures caused by network instability. Furthermore, general-purpose LLMs may also lack the fine-grained expertise required for detailed analysis of Android behavior. Finally, while the system has a factual consistency check, it lacks a comprehensive error repair mechanism, which is critical for assuring the accuracy and dependability of its output.

LLM-MalDetect

LLM-Maldetect is another framework for detecting Android malware using LLM. In this method, it improves semantic understanding of APK files by modeling feature dependencies and utilizing structured prompt engineering for more precise identification [46]. According to Feng *et al.* [46] this framework has three major stages;

1. **Data Pre-processing:** At this stage, APK files are being decompiled utilizing tools such as APKTool to extract the Android Manifest, Smali code, and resource files, with a special emphasis on resource files for feature extraction [46].

2. **Feature Engineering:** According to Feng *et al.* [46], there are two sub

processes at this stage, where one is feature extraction and the second one is feature selection. In the feature extraction sub-process, API calls, permissions, and string features need to be extracted. String features improve the interpretability by exposing the inconsistencies.

Then, at the feature selection level, features are filtered out based on mutual information by selecting the most appropriate permissions and API calls while minimizing computational cost. The glm-4-flash model is used to summarize string characteristics, and permission groups are condensed for increased efficiency.

3. **Model Training:** According to Feng *et al.* [46], this consists of three subprocesses, prompt construction, Tokenization, and Embedding, and lastly LLM Fine-tuning. As mentioned by Feng *et al.*, Natural Language Prompts are constructed from extracted features to guide the LLM's reasoning and also to prevent hallucinations and refusals. As usual in the tokenization stage, standard tokenization techniques have been used to turn textual inputs into LLM-compatible vectors. Then at the last stage, LLM-Maldetect finetune using the Mistral 7B model and LoRA (Low Rank Adaptation). This technique enables efficient adaptation by modifying only specified parameters, resulting in a final LLM capable of identifying applications as either dangerous or benign.

When considering the advantages of this framework, it addresses significant shortcomings of traditional learning-based methods by enhancing flexibility, semantic thinking, and behavioral understanding. According to Feng *et al.* [46], one of its notable contributions is the extraction of string features from sources like string.xml, Smali code, and resource files, which improves interpretability and facilitates forensic analysis. The system utilizes LLM reasoning skills to examine complex feature associations, overcoming the limitations of restricted or restrictive feature sets. Prompt

engineering is efficiently used to aid the LLM in analyzing app behavior and fine-tuning using LoRA [46].

Nevertheless, LLM-Maldetect is not flawless. When compared to the traditional ML methods, LLM-Maldetect's inference latency is relatively high, making it less appealing in real-time deployment scenarios. Despite being compact in terms of LLM standards, this still requires a significant amount of computational resources, making it unfeasible in resource-constrained environments. Additionally, the use of third-party dependencies, such as glm-4-flash, introduces potential network dependencies and availability complications. Again, the model lacks fine-grained domain-specific analysis, primarily handling complex logical code structures. Also, there is still room to grow on polymorphic or dynamically loading malware. A comprehensive summary of Large Language models can be found in Table 2.3.

Method	**Technique**	**Features Used**	**Strengths**	**Limitations**
AppPoet	LLM, DL	Static features	Deep semantic analysis via LLM, captures behavioral links, generates human-readable reports, uses memory module, final decision via DNN	Static-only analysis, LLM API issues, requires frequent dataset updates, no public model
LAMD	LLM, slicing, static analysis	API calls, CFG	Handles complex code via hierarchical reasoning, provides explainable results, simplifies structure using slicing	LLM hallucinations, high false positives, static-only scope, high resource usage, limited malware understanding

Method	Technique	Features Used	Strengths	Limitations
LLM-MalDetect	LLM	Feature extraction	Uses LLM to analyze complex feature relations, string-based extraction from source code	High inference time, resource-heavy, network dependency, limited support for dynamic/polymorphic malware

Table 2.3: Comparative Analysis of LLM-Based Malware Detection Methods

2.5 Research Gaps

Traditional ML and DL techniques often rely on limited feature sets, such as API call frequencies or permissions, which fail to capture the deeper semantic relationships or provide interpretability. Due to the limited insights provided and the tendency of these models to produce binary classifications, forensic analysis becomes challenging, as such outputs can directly impact decision-making. Even some of the new LLM-based solutions, such as LAMD and LLM-MalDetect, offer better contextual awareness; they are using single-pass classification pipelines or process features simultaneously. In contrast, the **proposed system employs a distinct, fine-tuned LLM for API requests and permissions**, enabling independent assessment and granular analysis of every feature set. The **two-model structure allows the system to determine which part (permissions or APIs) is initiating a potential threat**, and this **boosts the interpretability**. Additionally, **proposed method improves adaptation to invisible malware by capturing hidden syntactic and semantic patterns through the use of majority voting and code-aware LLMs**. Compared to existing LLM methods, **this approach is more**

appropriate for scalable, real-world deployment than current LLM techniques as it places a higher priority on local inference, feature modularity, and decision transparency.

3 Methodology

This chapter outlines the complete methodology adopted to develop the Android malware detection system based on transformer-based language models. The approach consists of several phases, beginning with the conceptualization and assessment of the system design, followed by data acquisition and preprocessing, model architecture selection and fine-tuning, training and evaluation strategies, and finally ethical considerations and limitations. All the components of the pipeline are to assist in producing a reproducible and scalable system with the capability of identifying malicious APKs using static attributes and contextual information. The following subsections present a step-by-step description of the technical foundation of the proposed system.

3.1 Overview of the Proposed System

This section introduces the procedure for developing the suggested malware detection system. Section 3.1.1 describes the initial decisions, including dataset selection, static feature extraction, and model training. Section 3.1.2 describes the system's operational workflow, from APK upload and feature extraction to real-time inference using fine-tuned BERT models via a REST API.

3.1.1 Initial Assessment

The proposed Android malware detection system leverages large pretrained transformer models for categorization in conjunction with static analysis of application features. Specifically, the proposed system utilizes fine-tuned transformer-based models for categorization after extracting security-sensitive aspects, such as permissions and API calls, from Android application packages (APKs) through static analysis.

The overall implementation pipeline comprises four distinct stages, which are also illustrated graphically in Figure 3.1. The first stage, which is Data Collection, has been completed using publicly available datasets, particularly the CIC-AndMal2017 dataset [12], an extensive collection of Android APK files. These datasets provide comprehensive coverage of both behaviors and patterns necessary for efficient model training, encompassing both benign applications and various malware types.

Next stage is the **Static Feature Extraction** where APKTool was then used to decompile APK files and extract source-level data, specifically Smali code and XML manifests. Static analysis techniques were applied to extract meaningful security features, including declared permissions from the AndroidManifest.xml file and invoked API calls extracted from Smali files through regular expression (regex) parsing. On the same stage, features unrelated to security behaviors, such as user interface or graphics-related API calls, were carefully filtered out.

Then **Model Training with Fine-tuned Transformer Models** comes as the third stage. Several different models were initially tested to determine the most suitable one for the proposed approach. The first model examined was StarCoder, selected for its specialized capabilities in evaluating code-based activities. However, its high processing demands exceeded the available hardware capacity, rendering the deployment ineffective (see Figure 3.3).

The Random Forest Classifier was then tested using basic machine learning meth-

ods. Although it achieved perfect accuracy on the training data, the model exhibited severe overfitting and failed to generalize to unseen data. Due to these limitations, a selection of prominent pretrained transformer-based language models was evaluated, including: BERT-base, BERT-base-uncased, BERT-large-uncased, RoBERTa-base, and RoBERTa-large. By including a single output layer [47], these transformer models—which had been pretrained on large text corpora—could be efficiently adjusted for malware classification. **BERT-large-uncased** was chosen for the final system following extensive comparative testing because it struck the ideal balance between classification(90% accuracy on held out data) performance and manageable computational training needs.

Then, as the final stage, **Inference Deployment via REST API** comes to the pipeline. To facilitate practical usage, the trained detection models were embedded in a FastAPI-based RESTful API for seamless integration and real-time classification of APKs uploaded by users. To improve the classification accuracy and dependability, the backend uses a strong majority voting method across predictions from three specialized transformer models (BERT-large-uncased) that have been trained separately on API calls, permissions, and their combination.

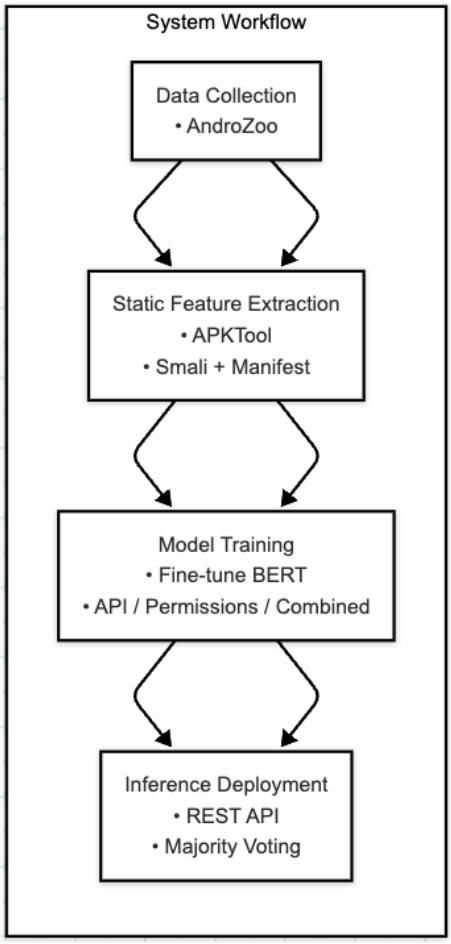


Figure 3.1: Initial Workflow Diagram

On this proposed system, every APK is subjected to static analysis, a reverse engineering method that examines the configuration files and code of the APK without executing it. With this type of approach it ensures a safe and scalable analysis environment, avoiding the potential risks associated with the dynamic analysis.

3.1.2 How does the proposed system work

As mentioned earlier, the APK is initially decompiled using APKTool, which unpacks the application into its component files. This decompilation process produces two key artifacts essential for feature extraction: the AndroidManifest.xml file, which contains the application's permissions and metadata; and the .smali files,

which represent the compiled Dalvik bytecode and provide detailed insights into the application's internal behavior and API usage. A visual representation of this workflow is provided in Figure 3.2 to enhance understanding of the overall process.

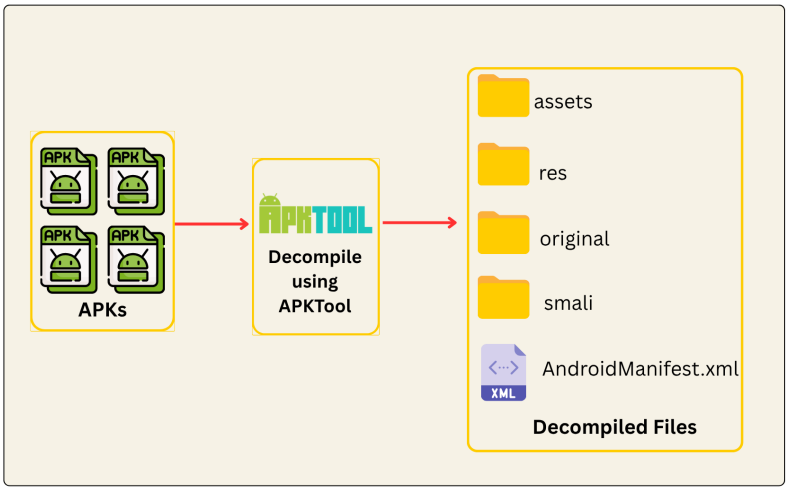


Figure 3.2: Static Feature Extraction from APKs

After decompilation and feature extraction, each feature set is treated as textual input and tokenized using the BERT tokenizer. Following this approach, three separate instances of the BERT-large-uncased model were trained. The first model was trained exclusively on API call tokens, enabling the system to learn behavioral patterns from the application's code structure. The second model was trained solely on permission tokens, allowing analysis based on the application's declared resource access. The third, a combined model, was trained on the concatenation of both API and permission tokens, leveraging both declarative and behavioral data to potentially achieve higher accuracy. A detailed overview of this model training workflow is illustrated in Figure 3.3 for better understanding.

This feature set split was intentionally made to enable ensemble-based inference strategies (such as majority voting) and to evaluate the individual and combined predictive potential of permission declarations and API call patterns.

The system will function as a FastAPI-built RESTful service during inference.

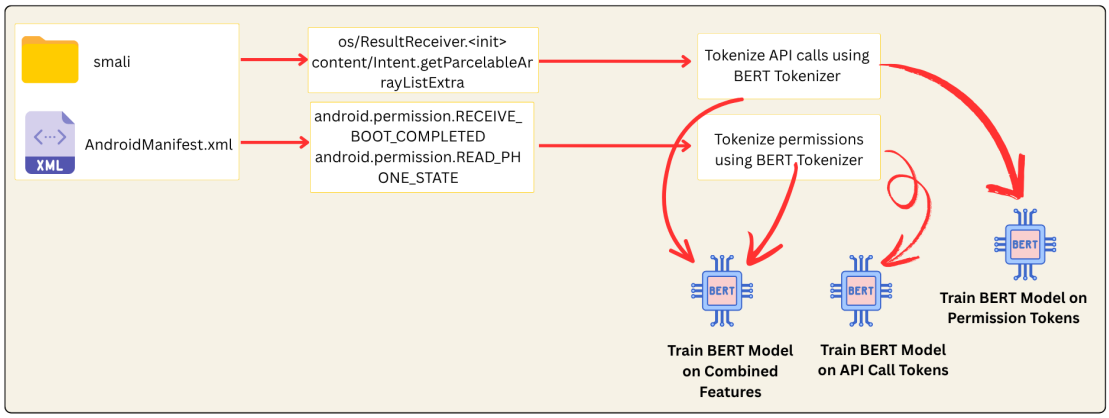


Figure 3.3: Training three distinct models

When an APK is uploaded, it undergoes the same static analysis pipeline used during training. To generate distinct predictions, the extracted features are fed into each of the three trained models. The algorithm then uses a majority vote procedure, flagging the APK in accordance with the classification of at least two models as malicious. This ensemble approach improves reliability by mitigating the weaknesses of any single model.

A dynamic React-based front-end will serve the user with the final analyzed results. It uses API calls to communicate with the backend and presents the malware classification decision, which includes whether the app is benign or malicious.

This design is optimized for speed, safety, and modularity, which are key strengths of this method. Also, it showcases the capabilities of LLMs in accurately labeling Android apps without requiring runtime execution or sandboxing.

A full system diagram can be found below, see Figure 3.4

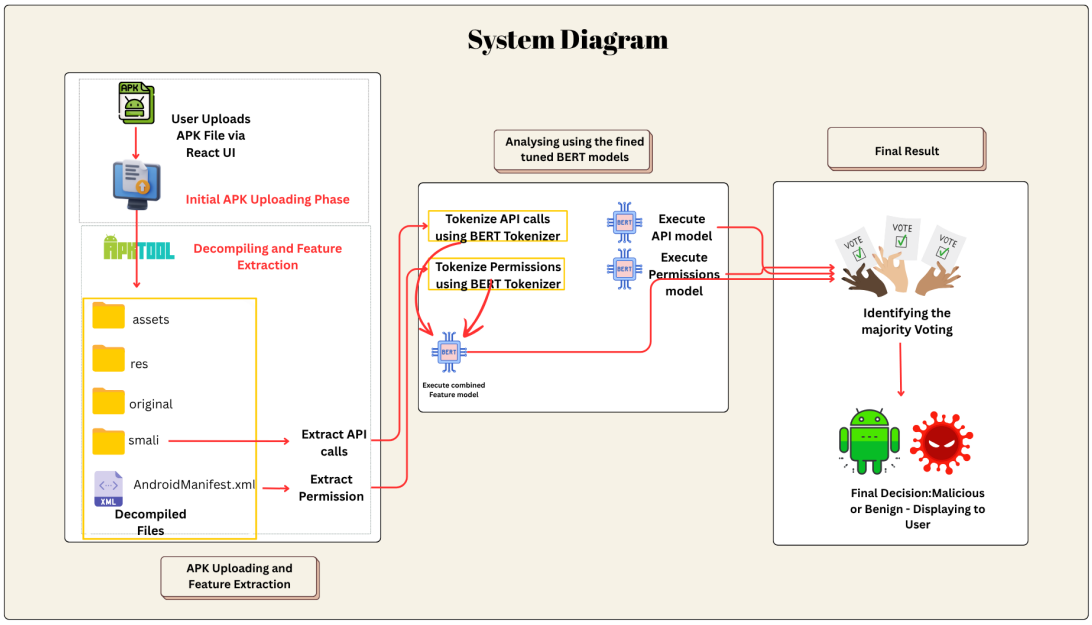


Figure 3.4: System Diagram

3.2 Data Collection and Pre-processing

The quality and structure of the training data are crucial to the success of any machine learning system. This subsection outlines the datasets utilized in this work and the techniques used to prepare them for model training. It begins by demonstrating how to identify benign and malicious Android APKs in publicly available repositories, then extracts static elements such as permissions and API calls. Preprocessing procedures, including tokenization, cleaning, and data formatting, are also covered to guarantee that the generated dataset is representative and acceptable for use in transformer-based models.

3.2.1 Data Collection

Publicly available, accessible, and generally recognized datasets have been utilized to develop and evaluate the Android malware detection system, ensuring repeatability, legality, and a broad range of application types. Initially, two datasets were planned

for use: CIC-AndMal2017 [12] and Drebin [2], both of which are well-known corpora in the study of Android malware. However, due to certain difficulties, the Drebin dataset was excluded. The reasons for this exclusion are explained later in this section.

CIC-AndMal2017

The CIC-AndMal2017 dataset [12], curated by the Canadian Institute for Cybersecurity, comprises a total of 10,854 Android APK files. Out of that, 4,354 malicious applications were categorized in various malware classifications, and 6,500 were benign applications. This malicious sample was categorized into several real-world families and behavioral characteristics, including Adware, Ransomware, Scareware, and SMS Malware. Since this dataset was given priority for training and evaluation, it provides a fair representation of both benign and malicious applications.

We downloaded the APK files directly from the official distribution source and verified their integrity through checksum validation where available. This helps to keep the integrity and consistency of the data used on the tests. No manual alterations were made to these APKs, and their labeling was also preserved as originally defined by the dataset creators.

Drebin

The Drebin dataset consists of 5560 samples from 179 malware families, which was gathered between 2010-2012 [2]. While Drebin is a valuable and resourceful dataset, it only contains malicious samples. Due to this limitation, Drebin was excluded from the testing.

All the datasets used in this study were obtained from publicly available academic sources. They were utilized entirely in accordance with the guidelines set out in each licensing and usage agreement.

3.2.2 Data Preparation

To prepare the data for input into the transformer-based malware classifiers, a systematic static analysis procedure was applied to each APK file. This process involved decompiling the APKs, extracting relevant static features from the decompiled code, cleaning and normalizing the extracted data, and organizing it into three distinct training sets to support targeted experimentation.

The first stage of the process was APK decompilation. For this, APKTool—a widely used reverse engineering tool—was utilized. APKTool unpacks the binary APK into a human-readable directory structure. This includes a collection of .smali files, which represent the Dalvik bytecode and contain method-level source logic, as well as the AndroidManifest.xml file, which defines the app's permissions, components, and configurations. This decompilation step is essential for static analysis, as it provides the raw data needed for feature extraction.

The second stage was static feature extraction, focusing on two key categories: **API calls** and **permission sets**. These features were selected because they are known to provide strong predictive signals for malware detection and reflect both behavioral and configuration-level aspects of an Android application.

For **API call extraction**, which is the third stage, to identify Android API invocations, each .smali file in the application was iterated through, and a regular expression was used for extraction (see Figure 3.5). Method calls in Dalvik bytecode are indicated by lines that comprise invoke instructions and a reference, such as Landroid/ for Android framework classes. These referenced method signatures were captured as tokens after extraction (e.g., Landroid/net/Uri;->parse(...)). This generates a list of each app's API call tokens. Such API usage patterns can be strong indicators of malicious behavior (e.g., using telephony or network APIs).

```python
# Define API call patterns to exclude (UI-related, utility, v4/v7, etc.)
EXCLUDED_API_PATTERNS = [      Chat ⌘+L Edit ⌘+I
    "support/v4/", "support/v7/","widget/", "view/", "util/", "drawable/", "graphics/", "transition/"
]

def should_exclude_api_call(api_call):
    """Check if the API call matches any pattern in EXCLUDED_API_PATTERNS."""
    return any(pattern in api_call for pattern in EXCLUDED_API_PATTERNS)

def extract_api_calls(smali_folder):
    """Extracts API calls from smali files while filtering unwanted API calls."""
    api_calls = set()  # Use set to store unique API calls

    if not os.path.exists(smali_folder):
        print(f"Warning: Smali folder not found: {smali_folder}")
        return list(api_calls)

    smali_pattern = re.compile(r'invoke-.*? Landroid/([^;]+);->([^(\s]+)\(')

    try:
        for root, _, files in os.walk(smali_folder):
            for file in files:
                if file.endswith(".smali"):
                    smali_path = os.path.join(root, file)
                    with open(smali_path, 'r', encoding="utf-8", errors="ignore") as f:
                        for line in f:
                            match = smali_pattern.search(line)
                            if match:
                                api_call = f"{match.group(1)}.{match.group(2)}"

                                # Skip API calls matching any excluded pattern
                                if should_exclude_api_call(api_call):
                                    continue

                                api_calls.add(api_call)  # Add to set to ensure uniqueness
```

Figure 3.5: API calls extraction for the creation of API calls Dataset

For **Permission Extraction**, Python's ElementTree was used to parse AndroidManifest.xml (Refer Figure 3.6). The stated permissions of the application (ex: android.permission.INTERNET) were obtained by gathering all "permission" tags. Tokens were used to hold these permission strings. It is well recognized that important signals for malware detection may be found in manifest information, which includes permissions, intent filters, and other features [48].

```python
# List of permissions to exclude
EXCLUDED_PERMISSIONS = {
    "android.permission.ACCESS_NETWORK_STATE",
    "android.permission.INTERNET",
    "android.permission.RECEIVE_BOOT_COMPLETED",
    "android.permission.ACCESS_WIFI_STATE",
    "android.permission.READ_PHONE_STATE",
    "android.permission.WAKE_LOCK",
    "android.permission.WRITE_EXTERNAL_STORAGE" #Added on 04.12.2025
}

def extract_permissions(manifest_path):
    """Extracts permissions from AndroidManifest.xml while excluding specific permissions."""
    print(f"Extracting permissions from: {manifest_path}")
    permissions = set()  # Use set to prevent duplicates

    if not os.path.exists(manifest_path):
        print(f"Warning: Manifest file not found: {manifest_path}")
        return list(permissions)

    try:
        tree = ET.parse(manifest_path)
        root = tree.getroot()
        for elem in root.findall(".//uses-permission"):
            permission = elem.get('{http://schemas.android.com/apk/res/android}name')
            if permission and permission not in EXCLUDED_PERMISSIONS:
                permissions.add(permission)  # Add to set to ensure uniqueness
    except ET.ParseError as e:
        print(f"Error parsing {manifest_path}: {e}")

    return list(permissions)  # Convert back to list before returning
```

Figure 3.6: Permission extraction for the creation of Permission only Dataset

The final stage in the Data preparation process is feature cleaning and Normalization. To avoid biasing the model toward frequently occurring method calls or permissions, duplicate tokens were removed. Additionally, to ensure compatibility with the pre-trained BERT tokenizer and to mitigate data sparsity caused by case mismatches, all tokens were normalized—primarily by converting them to lowercase.

To evaluate the predictive power of different static feature types, three distinct input datasets were created. Each dataset represents an application as a space-separated sequence of tokens: one using only API calls, another using only permissions, and a third combining both into a single token sequence. These sequences are treated as text documents, enabling independent and combined evaluation of each feature set's contribution to prediction accuracy. By maintaining separate datasets

for APIs and permissions, we ensure that the model can learn from each component individually. Prior research, such as MalBERT [49], has shown that BERT-based models can distinguish between benign and malicious applications with approximately 97% accuracy using only manifest contents. This highlights the predictive strength of permissions and configuration settings. Similarly, frequently used API calls serve as reliable indicators of malicious behavior.

3.3 Model Architecture

This section delves further into the architecture of the models used for malware detection. It begins with an explanation of the motivation for using transformer-based language models, such as BERT, highlighting their contextual learning advantages over traditional methods. The following subsection illustrates the fine-tuning approach used to adjust pre-trained models for the binary classification task utilizing extracted static characteristics.

3.3.1 Rationale for LLM Selection

To design a proper and accurate malware detection system that balances accuracy, generalizability, interpretability, and computational practicality, selecting the exemplary model architecture is an essential first step. In the testing phase, several approaches were systematically evaluated, starting with large code-specific language models, traditional machine-learning techniques, and transformer-based text encoders. Ultimately, the most successful model was identified as the transformer-based text encoders.

StarCoder [49] was the first model used in the experiment. It is a large language model with 15 billion parameters, designed to understand and generate source code. Since Starcoder is capable of identifying patterns in Android .smali code and

manifest declarations, as it has been trained on a massive collection of programming languages and can comprehend semantically significant code sequences [49].

However, even the StarCoder [49] is conceptually appealing, this model is not feasible to deploy in the environment. It was unable to fine-tune and used in a real-time pipeline without access to specialized infrastructure since the model's size surpassed the GPU memory limitations. Furthermore unacceptable latency and instability were experienced by even simple inference. This result highlights a significant drawback of using code-specific LLMs in environments with restricted resources, despite their capabilities expanding [49].

As a baseline comparison, the Random Forest classifier [50] was implemented using the extracted static features (API calls and permissions). Random Forest is a popular model due to its inexpensive training costs, interpretability, and resilience. Although this achieved perfect accuracy on the poor training dataset, performance decreased dramatically, revealing signs of overfitting. Accordingly, we turned to transformer-based language models, which have demonstrated state-of-the-art results in many text classification tasks.

Given the limitations of both LLMs and classical ML methods, hence moved to transformer-based language models, which are intended for natural language understanding. Transformer-based models have proven to be highly effective in a vast range of classification and sequence modeling applications. These models are well suited to handle structured but ungrammatical textual inputs such as API calls and permissions where treating them as word-level tokens in a bag of words or sentence format [49].

For ease of testing and repeatability, implementations from the Hugging Face Transformers library were utilized. The two main architectures that were assessed were:

- **BERT** [51] **(Bidirectional Encoder Representations from Transform-**

ers): BERT is a popular transformer model that learns contextual representations and uses deep bidirectional attention to encode text. With a minimum classification head, its pre-trained weights can be adjusted to accommodate activities that follow [49].

- **RoBERTa** [52] **(A Robustly Optimized BERT Pretraining Approach)**: A variant of BERT that trains on bigger mini-batches, extends training time, and eliminates the next sentence prediction target. Because of these enhancements, RoBERTa typically outperforms BERT in a number of benchmark tests [52].

For the selection process, four pre-trained variants were tested: BERT-Base-uncased, BERT-Large-uncased, Roberta-Base, and Roberta-Large. The identical input format (space-separated token sequences) and the uncased WordPiece tokenizer were used to test all models for compatibility. The prepared datasets were used to fine-tune for binary classification (malicious vs. benign), and the validation accuracy served as the basis for the performance comparison.

Even though Roberta-large and Bert-large-uncased both achieved high classification accuracy, Bert-large-uncased was ultimately chosen for the proposed system because all feature sets (API, permissions, and combined) consistently showed good accuracy and required less memory and GPU footprint than RoBERTa-large.

3.3.2 Fine Tuning Process

To enable the transformer-based models to classify Android applications as malicious or benign accurately, we adopted a standard fine-tuning approach widely used in natural language processing tasks. Every model was initialized with a pre-trained transformer (BERT) and refined on a custom-labeled dataset, regardless of whether it was trained on permissions, API requests, or a combination of features. The overall

objective was to learn task-specific representations relevant to Android malware detection while utilizing the semantic capabilities of pre-trained embeddings.

Model Structure and Architecture: In addition to the transformer's final pooled [CLS] token representation, a fully connected linear classification layer was added to create each model. This [CLS] vector, the output of the BERT architecture, captures the entire input sequence and is passed as input to the classifier. Then the vector is mapped to a binary label (malware or benign) by the final classification head, which computes cross-entropy loss during training using a softmax activation.

This configuration facilitates transfer learning which is a methodology that has shown great efficacy in both NLP and security contexts—from general language comprehension to the particular job of malware classification.

Tokenization of Input Features: At this stage, the raw features, such as permission lists, API call sequences, or both, were transformed into plain text strings using tokens separated by spaces. Then, Hugging Face BertTokenizer was utilized to tokenize these strings, splitting them into subword tokens that fit the model's pre-trained vocabulary. In order to ensure consistent input dimensions, the Maximum token length was set to 128 tokens, the Tail of the longer sequences was truncated, and shorter sequences were padded. This limit does mean that very long lists of APIs or permissions may have some terms omitted, which is a limitation that will be discussed later in this section.

The tokenizer automatically generated the attention mask to indicate which tokens required focus during training. PyTorch tensors were used to store and feed the token IDs and attention masks to the model in batches.

Training Configuration and Hyperparameters: The AdamW optimizer, which decouples weight decay from the learning rate update, is a reliable option for fine-tuning transformers. It was used to train all three model variants: API-only, permission-only, and combined. Following experimentation and adjustment, the

final hyperparameters were selected as illustrated in Table 3.1.:

Parameter	**Value**
Optimizer	AdamW
Learning Rate	2e-5
Batch Size	8 or 16 (depending on GPU memory)
Epochs	3–5
Sequence Length	128 tokens (truncation and padding applied)
Loss Function	CrossEntropyLoss
Early Stopping	Enabled (based on validation loss plateau)

Table 3.1: Selected hyperparameters used for BERT model fine-tuning

PyTorch, NumPy, and other relevant random seeds were fixed to ensure predictable splits and initialization. The training data was separated from a validation split, usually 20%, which was used to track overfitting and generalization performance. Early stopping was applied if the validation performance plateaued or degraded, and training and validation losses were recorded over epochs.

Hardware Constraints: A dedicated server with two NVIDIA GeForce RTX 4090 GPUs, each featuring approximately 24 GB of VRAM and an 8.9 compute capacity, was used for all model training. However, to maintain simplicity and consistency in training, only one GPU (GPU 0) was used for fine-tuning.

The large memory capacity of the RTX 4090 allowed a batch size of up to 16 and a sequence length of 128 tokens without encountering memory-related issues. Despite the enhanced hardware capabilities, a conservative configuration was maintained to ensure generalization and prevent overfitting.

Checkpoint Selection and Final Results: After the fine-tuning process, the optimal checkpoint for each model was selected based on validation accuracy. The final BERT-large-uncased (combined) model achieved a test accuracy of approximately 92% and a training loss of around 0.2. Individually, the permission-only and API-only versions performed slightly worse, particularly the permission-only model. Below table 3.2 illustrates the test accuracies of the three separate models individually.

Model	**Train Loss Trend**	**Test Accuracy**	**Observations**
API-only	Decreasing consistently ($0.44 \rightarrow 0.20$)	**86.93%**	Solid performance, likely due to the informative nature of API calls.
Permissions-only	Plateaued around 0.67–0.68	**81.94%**	Nearly random guessing, the model failed to learn meaningful patterns from permission tokens alone.
Combined	Decreasing consistently ($0.48 \rightarrow 0.19$)	**91.50%**	Solid performance, slightly below API-only. Indicates that most predictive power came from APIs.

Table 3.2: Model performance metrics for different feature sets

3.4 Training and Evaluation

After preparing the dataset and selecting the model architecture, the next steps involve in training and evaluating the models. This section covers the methods used during the training phase and it also outlines the evaluation metrics used to assess performance, including accuracy, precision, recall, and F1-score, which provide a quantitative foundation for model comparison.

3.4.1 Training Protocol

To maintain a balance between malware and benign ratios, the combined dataset was split into 80% training and 20% testing, separated by class. To ensure repeatability, we fixed all random seeds across libraries before shuffling the dataset and initializing

the model weights. This guarantees that each iteration of the training process begins with the same conditions. The training loop involved batching and tokenizing inputs for each epoch, calculating loss, and computing gradients. After each epoch, an evaluation was performed using a held-out test set. A dropout (default 0.1 in BERT) and a weight decay have been utilized to regularize the data. Typically, convergence was achieved within three epochs. Due to low data quantity, we did not perform cross-validation and instead reported findings for the single-held-out test split. Furthermore, the model training codes are documented in appendix A.

3.4.2 Evaluation Metrics

To evaluate the effectiveness of the proposed malware detection models, standard classification metrics were used, including **accuracy, precision, recall, and F1-score**.

- **Accuracy**: Accuracy measures the overall percentage of correctly predicted instances across all samples.
- **Precision**: This highlights the models false positive rate by projecting the malicious applications that were truly malicious.
- **Recall**: Recall measures the proportion of actual malicious apps that were correctly identified while reflecting the model's ability to avoid false negatives.
- **F1-score**: This reflects the harmonic mean of precision and recall, which provides a single balanced statistic that is especially effective in circumstances with class imbalance or where precision and recall are equally significant.

The final model, based on BERT-large-uncased and trained on the combined characteristics (API calls + permissions), achieved approximately 90% test accuracy (See Figure 3.7). More crucially, it achieved accuracy and recall in the low 0.90s,

yielding an F1-score of around 0.92 (See Figure 3.8). This suggests that the model is not only accurate but also trustworthy in distinguishing between dangerous and benign applications, with no noticeable bias toward either class.

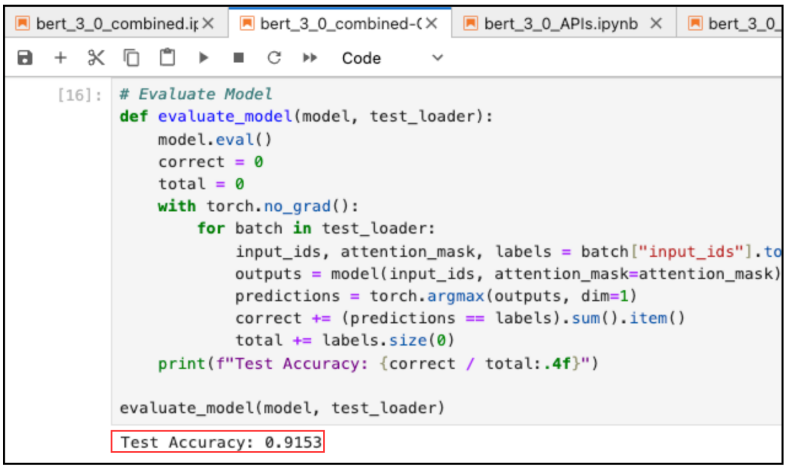


Figure 3.7: Combined Model Test Accuracy

```
print("Combined Model Metrics:")
evaluate_model(model, test_loader)
```

```
Combined Model Metrics:
Sample predictions: [np.int64(1), np.int64(0), np.int64(0), np.int64(0),
int64(1), np.int64(1)]
              precision    recall  f1-score   support

      benign       0.94      0.89      0.92        94
   malicious       0.89      0.94      0.91        83

    accuracy                           0.92       177
   macro avg       0.92      0.92      0.92       177
weighted avg       0.92      0.92      0.92       177
```

Figure 3.8: Combined Model Metrics

By contrast, the API-only and permission-only models exhibit somewhat poorer performance, with test accuracies ranging from 80-86%. While still reasonable, this indicates that focusing on a single feature type overlooks critical signals that the combined model can better leverage. Also, the combined model benefited from

feature complementarity, in which permissions contextualized API activities and vice versa.

All evaluation measures were computed solely on the held-out test set, guaranteeing that no data leaking occurred between training and assessment. These findings provide support for the notion that the transformer-based architecture, particularly the combined input model, is effective for classifying static Android malware. Figure reffig:finetuningprocess illustrates the full fine-tuning process.

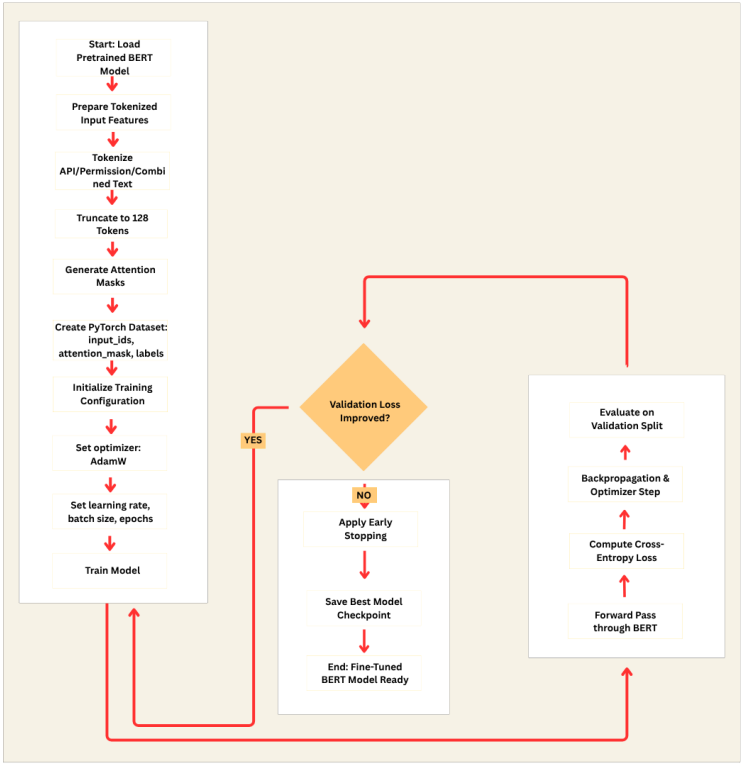


Figure 3.9: Fine Tuning Process

3.5 Ethical and Legal Considerations

Throughout the study, ethical rules and guidelines were followed when handling all the malware and benign datasets. CICAndMal2017 [12] dataset is freely accessible

for academic studies. No APKs were shared publicly with any third parties. Since the study was based solely on extracted static features, specifically API call patterns and stated permissions, no proprietary or possibly malicious code was ever distributed. Moreover, since the study primarily employed a static analysis approach, it also provided a safe research environment, minimizing any legal and ethical risks associated with executing malware.

Furthermore, to ensure the safety of the environment, analysis of the APKs was conducted in a controlled environment, which helps prevent unintentional malware execution or system compromise. From a privacy aspect, the datasets contained no user-specific information, merely metadata and application-level parameters.

Finally, we emphasize that the resulting detection system is an experimental prototype. It is designed as a proof of concept for academic purposes, not as a production-ready anti-malware solution. Any real-world deployment would need comprehensive validation, regular upgrades, and compliance with regulatory and operational standards.

3.6 Limitations

Although the suggested system yields promising results, there are some minor limitations worth acknowledging.

The BERT-large-uncased model is a powerful but resource-intensive model with around 340 million parameters. Due to GPU memory limitations, the input sequence length was limited to 128 tokens, and batch sizes were restricted to 8-16, which delayed the training process. If an app has a long list of permissions or API requests, any items exceeding the 128-token limit are removed. This indicates the loss of some key indicators of malicious behavior in sophisticated APKs. Moreover, since this approach focuses only on static analysis, it indicates that this approach cannot identify malware that activates during runtime, such as dynamic code load-

ing, obfuscated logic, or network-triggered payloads.

Although random seeds were used to ensure consistent training, GPU-based operations can still introduce some degree of unpredictability. Additionally, since the training dataset consisted solely of CIC-AndMal2017 [12], the testing dataset should be expanded to improve the accuracy and generalizability of the results.

4 Implementation

This chapter illustrates the practical implementation of the Android Malware detection system using BERT. The main goal was to implement a scalable, safe, and accurate pipeline capable of analyzing Android APK files utilizing static characteristics and classifying them with pre-trained transformer models. The implementation consists of a backend inference engine, a front-end for users, and a transformer-based ensemble model framework.

4.1 System Architecture

This section illustrates the practical realization of the proposed Android malware detection system. It focuses on the integration of system components, including the backend inference engine built using FastAPI, the Frontend built with React.js, and the ensemble framework, which coordinates predictions from multiple fine-tuned BERT models.

4.1.1 Backend Inference Engine (FastAPI)

The backend system is fully implemented using FastAPI, functioning as the core analysis engine responsible for receiving APK files and executing the entire malware detection workflow. Upon receiving an APK upload, the system initiates a decompilation process using APKTool, which extracts key internal files such as the

AndroidManifest.xml and the .smali bytecode representations of the application's logic.

Following decompilation, static feature extraction is performed. Application permissions are parsed from the manifest file using Python's xml.etree.ElementTree, while API calls are extracted from the smali files using regular expressions. These extracted features are then independently fed into three pre-trained transformer models (based on BERT architecture), each of which classifies the app as either benign or malicious. Finally, the system applies a majority voting strategy: if at least two models classify the app as benign, the final decision is benign; otherwise, the app is labeled as malicious.

This modular backend architecture isolates the inference logic, making the system easier to maintain, update, and deploy in isolated contexts.

4.1.2 Frontend Interface (React.js)

The front-end has been built using React.js, and this enables users to interact with the system in a seamless way. This will allow users to upload an APK and trigger the analytical workflow using the back-end API (see Figure 4.1). After the features have been extracted, the user will be able to get clear visual feedback of the APK, whether it is malicious or benign.

4.1.3 Model Ensemble Framework

As previously outlined, the core of the system is an ensemble of three transformer-based models, each built upon the bert-large-uncased architecture and fine-tuned on distinct feature subsets. The first model focuses exclusively on API calls extracted from .smali bytecode, capturing the application's runtime behavior. The second model is trained on permissions defined in the AndroidManifest.xml, reflecting the developer's declared intentions. The third model leverages a combined input of both

API call sequences and permission tokens, enabling it to learn from the interplay between these two feature types. This ensemble strategy enhances classification robustness by capturing complementary perspectives, where API calls represent execution flow and permissions offer insights into potential capabilities.

The majority voting mechanism employed by the system strikes a balance between sensitivity and specificity, producing more reliable predictions than any single model alone. The system's modular architecture enables flexibility and future extensibility: models can be independently retrained or replaced, new static features can be integrated without requiring significant architectural changes, and the frontend and backend are decoupled to support scalable deployments. This design also lays the groundwork for future enhancements, including integration with dynamic analysis techniques, interaction with mobile emulators, and real-time threat monitoring systems.

4.2 APK Processing and Feature Extraction

The initial process of the malware detection pipeline is the static analysis of the APK files. After the user uploads an APK via the interface, decompilation will be done by the backend using APKTool and produce human-readable Dalvik bytecode files (.smali) and the AndroidManifest.xml. From these outputs, two primary types of static features are extracted: permissions and API calls. Permissions are extracted by parsing the <uses-permission> tags from the manifest using Python's ElementTree library. Meanwhile, API calls are identified by scanning the smali files with regular expressions that detect method invocation patterns. These invocations, such as Landroid/net/Uri;->parse(), are normalized into structured tokens like android.net.Uri.parse.

To ensure consistency and reduce noise, the backend automatically removes duplicate tokens. It filters out UI-related and boilerplate APIs using a custom blacklist

(for example, patterns matching support/v4, widget, drawable, etc.). These tokens are represented as strings, resulting in structured inputs for transformer models. Feature extraction is completely static, which ensures safety and reproducibility.

4.3 Real-Time Inference with Pretrained Models

Extracted features are then assessed by three independently fine tuned instances of the bert-large-uncased model and those are **API call classifier**, **permission classifie**r and **combined feature classifier** models.

Each model has been fine-tuned on its respective feature set using supervised training. At inference time, the extracted token sequences are routed through the appropriate tokenizer, padded/truncated to 128 tokens, and fed into the model. The result is a logit vector [53] reflecting class probabilities, from which the prediction (malicious or benign) is calculated using argmax.

The final classification decision employs an ensemble majority voting strategy. Out of three models, at least two categorize the APK as an malicious, the APK will be marked as malicious if not benign. This method increases the robustness by leveraging the complementary signals from both permissions and API calls.

4.4 Web-Based User Interface

The system features a lightweight and user-friendly frontend built with React.js, designed to provide an intuitive interface for interacting with the malware detection backend. Users can upload APK files either through a drag-and-drop mechanism or by selecting files manually via a widget. Once an APK is submitted, a RESTful API request is sent to the FastAPI backend, which initiates real-time analysis of the file. After processing, the interface displays the final classification result—indicating whether the application is benign or malicious—along with a summary of the ex-

tracted static features, such as API calls and permissions. This frontend design enhances usability while maintaining full integration with the core analysis system.

Visual feedback (e.g., clean versus malicious ads, spinner while processing) improves the user experience. The frontend communicates securely with the backend using CORS-enabled HTTP requests. The interface's simplicity allows non-technical people to interact with the malware detection system. Refer to the below figures for all the User Interfaces (See Figures 4.2, Figure 4.3, Figure 4.4, Figure 4.5, and Figure 4.6).

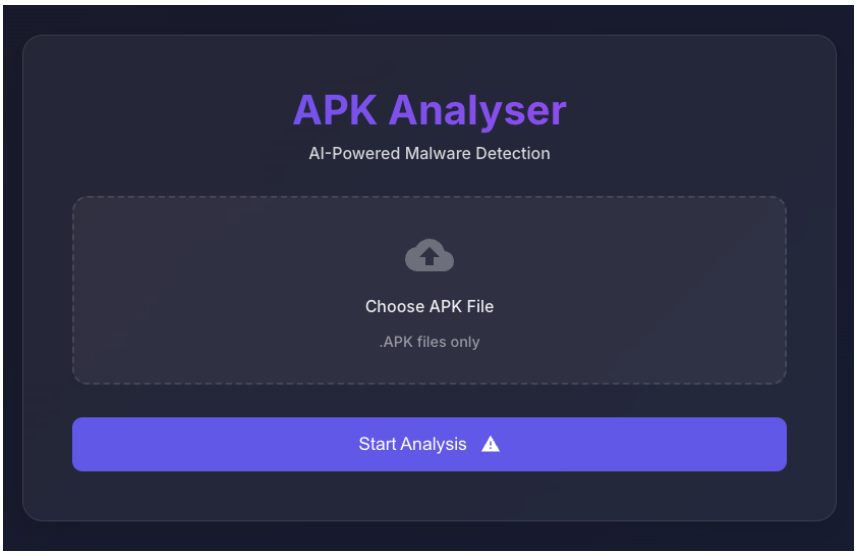


Figure 4.1: Front End of the System

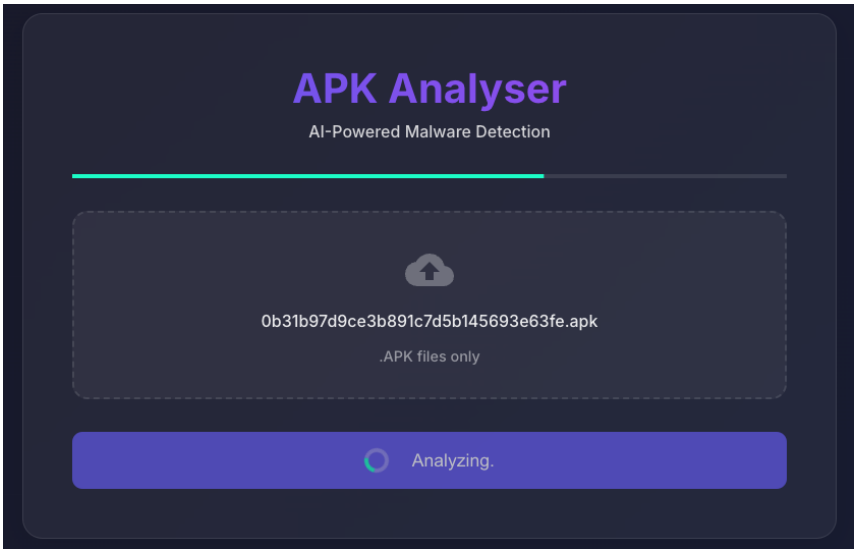


Figure 4.2: Middle of File Analysis Process

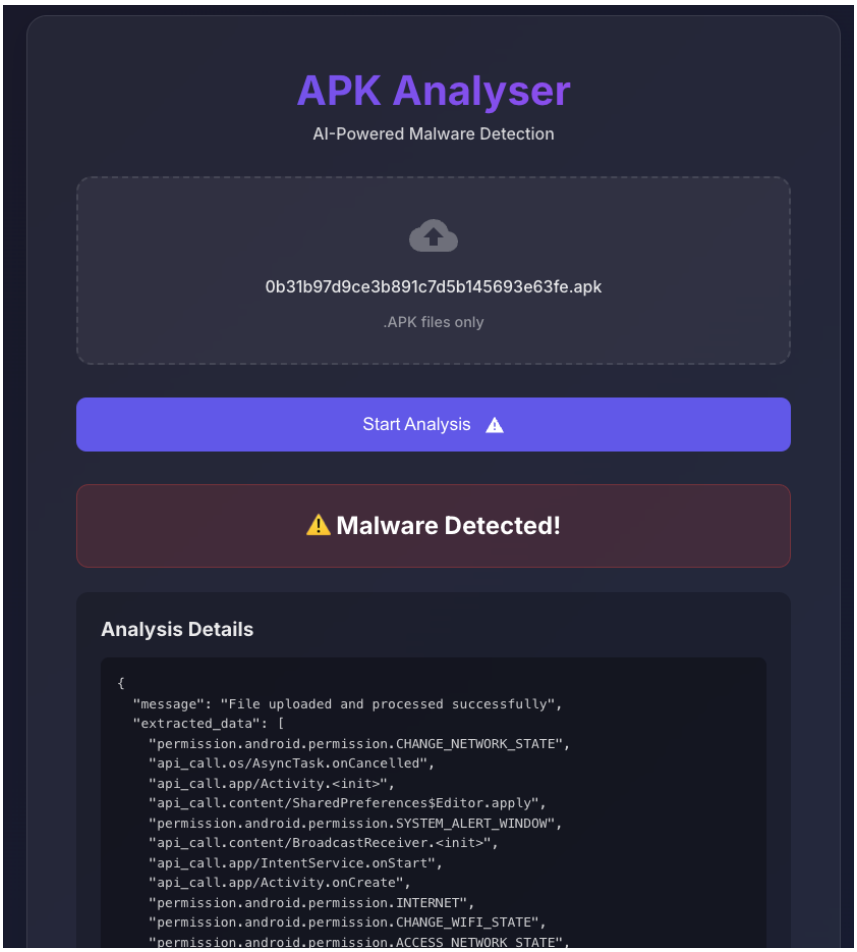


Figure 4.3: Malicious APK

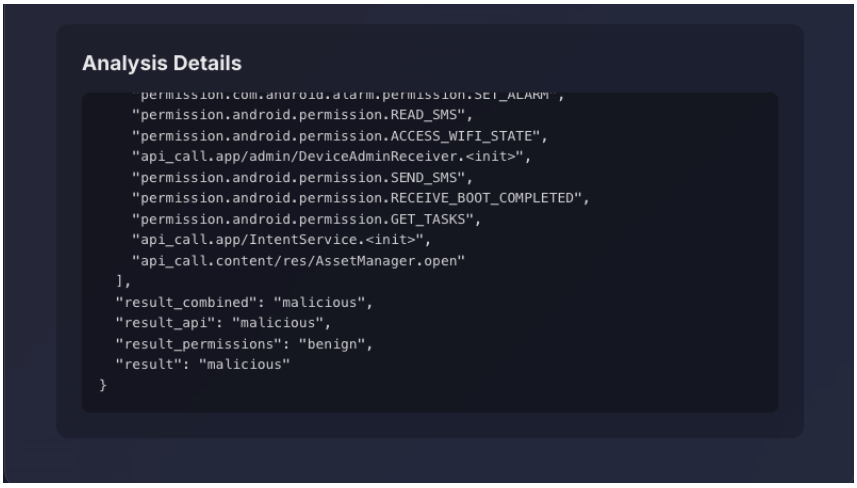


Figure 4.4: Extracted Features and Voting Process for Malicious APK

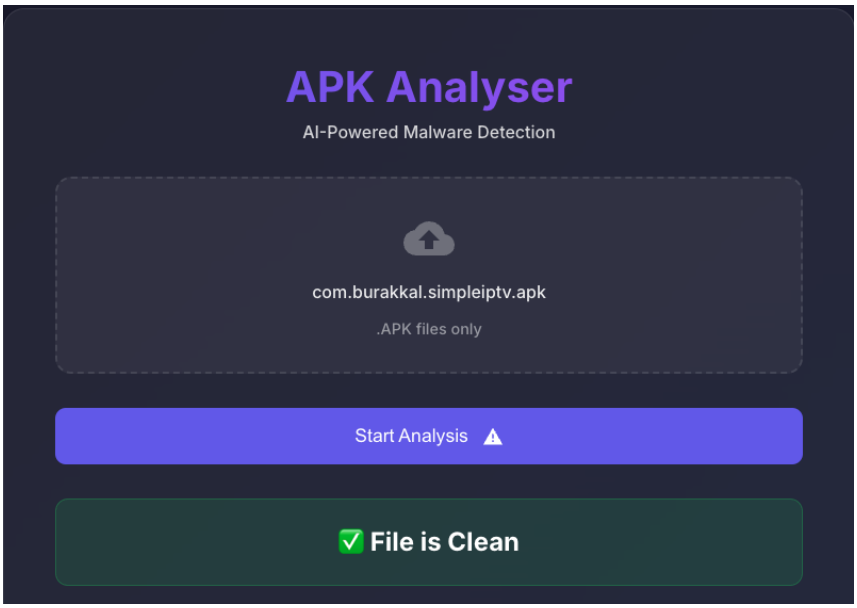


Figure 4.5: Benign APK

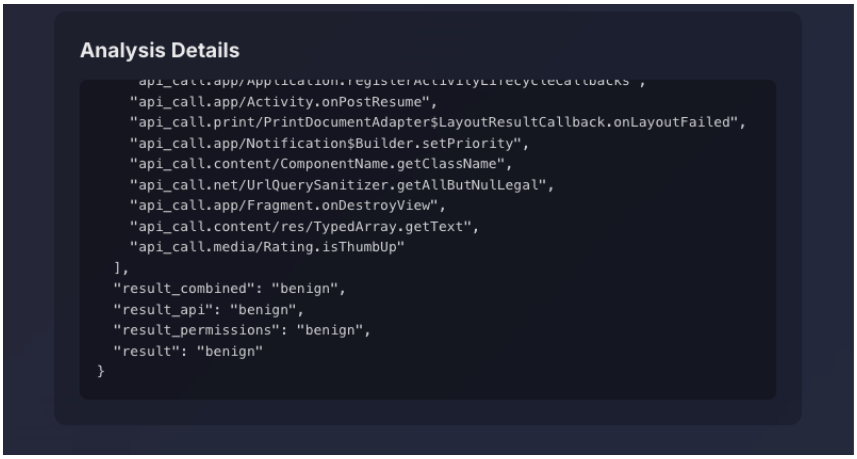


Figure 4.6: Extracted Features and Voting Process for Benign APK

4.5 Deployment Considerations

The backend was developed using FastAPI, which is a high-performance Python framework ideal for scenarios like this. The system includes the following deployment infrastructure;

- API Server: Runs as a Uvicorn instance alongside Gunicorn in production.
- Model Files: Pretrained model files have been loaded into memory at startup (See Figure 4.7).
- Security: CORS is enabled with recommended constraints for the production
- Directories: To provide isolation and traceability, file processing is divided into separate directories (uploads, decompiled, extracted_data) (See Figure 4.8).

```
# Load BERT Model and Tokenizer
MODEL_PATH_COMBINED = "./bert/malware_bert_model_combined.pth"
MODEL_PATH_API = "./bert/malware_bert_model_api.pth"
MODEL_PATH_PERMISIIONS = "./bert/malware_bert_model_permission.pth"
BERT_MODEL_NAME = "bert-large-uncased"
```

Figure 4.7: Loading pre-trained model files

```
# File storage paths
UPLOAD_FOLDER = "uploads"
DECOMPILE_FOLDER = "decompiled"
EXTRACTED_DATA_FOLDER = "extracted_data"
```

Figure 4.8: File Storage Paths

4.6 Monitoring and Updates

Monitoring is critical for maintaining operational performance and model accuracy over time. The following strategies have been included, with some intended for future implementation.

- Model Drift Identification: The system can be configured to log feature distributions and prediction trends on a regular basis, allowing for the identification of input drift or performance degradation.
- Dataset Expansion: Labeling new APKs and adding those to the training dataset allows for fine-tuning the models.
- Version Control: Version control ensures that model weights, tokenizer settings, and API modifications behave consistently across deployments.

5 Results Analysis and Discussion

5.1 Quantitative Analysis

The proposed LLM-based Android was statistically assessed using a set of benign and malicious applications. Fortunately, the proposed model outperformed expectations in all metrics. The detection accuracy was around 92% with an F1 score of roughly 91-92%. These results are competitive with to prior methods. Drebin, a traditional static-analysis technique, identified around 83.72% of malware in its study [2]. MaMaDroid, which abstracts API requests into behavioral Markov chains, reported a 91-99% F1 measure on a large dataset [40]. Similarly, MalDozer, a CNN-based framework, averaged 96-99% F1 across different datasets [42]. The LLM method proposed in this thesis produces accuracy and F1 values of 91-92%, which meet or slightly surpass the baselines. High precision and recall indicate accurate malware detection with few false alarms. A summary of this comparison is presented in Table 5.1.

Method	**Accuracy**	**F1 Score**
Proposed LLM Approach	**92%**	**91.5%**
Drebin	**94%**	**83.72%**
MaMaDroid	-	**95%**
MalDozer	**97%**	**97%**
LAMD	-	**85.71%**

Table 5.1: Accuracy and F1 score of some comparative models

The Table 5.1 clearly illustrates that the F1 score and accuracy are slightly lower compared to other methods; however, these can be improved with further enhancements. Several technical factors contribute to the lower performance. The first factor is the Dataset and Evaluation Differences. Traditional detectors often employ big, curated datasets with clear train/test divisions. For example, Drebin [2] examined around 123k benign and 5.5k malware applications (random 66/33 train/test), but when compared to the proposed solution, this has only been evaluated on a smaller dataset (around 900), which can have an impact on accuracy. This performance can be significantly improved by increasing the size of the dataset. Notably, achieving the current accuracy and F1 score using only 900 samples is a considerable accomplishment, especially when compared to other solutions that rely on thousands of samples.Differences in assessment criteria (e.g., cross-validation versus fixed split) can also cause minor metric adjustments. Notably, MalDozer [42] published findings for tens of thousands of programs (37k benign versus 20k malware), which may not be exactly comparable to the current dataset makeup of the proposed solution.

Furthermore Drebin [2] utilizes a linear SVM with constructed static features, whereas MalDozer uses convolutional neural networks (CNNs) to evaluate opcode or API sequences [42]. To take advantage of semantic context, the proposed system uses a transformer-based large language model (LLM), such as a fine-tuned BERT. However, LLMs may struggle with long and structurally complicated inputs, potentially resulting in the truncation of essential data. In contrast, Drebin and MalDozer use compact or structured input formats to better retain essential signals. Furthermore, whereas LLMs examine code sequentially, Android applications frequently have complex call networks and layered components that are not adequately represented in flat text input, reducing the model's capacity to identify subtle malicious activity.

Moreover, Drebin [2] uses manually developed features such as permissions and

API calls, whereas MalDozer [42]learns patterns from raw opcodes. In contrast, the LLM-based method prioritizes semantic interpretation of code and plain language, which may overlook simple but useful statistical signals employed in Drebin. This tradeoff may impair sensitivity to tiny, harmful cues. Furthermore, fine-tuning large language models need a wide and extensive data set. When fine-tuned on a small malware dataset, LLMs might not generalize, suffer from pretraining biases, or produce incorrect interpretations (hallucinations). These issues cause the performance gap with specialized models such as CNNs or SVMs.

Even though the proposed system faces these kinds of technical limitations, Large Language Models (LLMs) improve explainability by producing human-readable explanations for malware categorization, providing greater insight than classic feature-based models such as Drebin [2]. The semantic understanding of LLMs allows them to recognize suspicious functions, permissions, and behavior in readable language, which helps analysts and builds user trust. LLMs also display robustness against obfuscated and novel malware due to their large-scale pretraining, enabling the detection of malicious intent even from unfamiliar code. While LLMs' raw accuracy is significantly lower than that of traditional models, their capacity to explain and generalize enhances overall usability and usefulness in real-world circumstances.

So, to enhance the effectiveness of the proposed LLM-based malware detection system, several improvements can be considered. First and foremost, expansion of training data is a must, so for that approaches like as GANs or VAEs can be used to enrich training data, introducing different and synthetic malware patterns that improve the model's generalization and F1 score [54]. Pretraining the language model on domain-specific corpora, such as Android codebases or malware family archives, can also aid in its understanding of malware semantics. These solutions address the issues of low data variety and domain adaptability. In addition, integrating LLMs with classical models using knowledge distillation or ensemble approaches can pro-

vide a balance of accuracy and computing economy [55]. Smaller distilled models, such as DistilBERT, preserve the majority of the LLM's performance while being easier to implement [55].

An evaluation was conducted using three BERT-based classifiers: one trained on requested permissions, one on API call tokens, and another using a combination of both feature sets. Their performance, averaged over cross-validation, is reported in terms of accuracy, precision, recall, and F1 score. The bar chart below provides a visual summary of the model performance (see Figure 5.1).

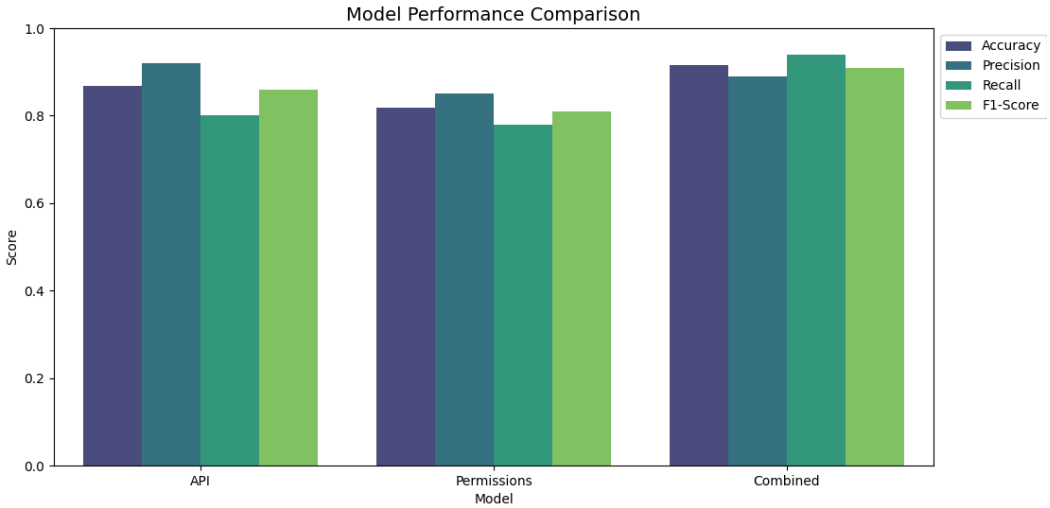


Figure 5.1: Model Performance Comparison

The combined model outperforms single-feature models, with an overall accuracy of 92% and an F1-score of approximately 92%. For, instance, the permission only model classifier achieved 82% accuracy 81-83% F1 scores (See Figure 5.2 and 5.3), whereas the API-only model achieved around 87% accuracy and 86-88% F1 scores (Refer Figure 5.4 and 5.5). The combined model has the highest accuracy of 92% and recall 89-94%, suggesting low false-positive and false-negative errors (See Figure 5.6 and 5.7). The permissions-only model exhibited accuracy around 82% and

reduced recall around 78-86.0%, indicating a cautious approach that misclassified a few benign applications while missing more malware instances. A heatmap was generated to visualize the performance results of the models, providing additional insight into the classification patterns and supporting the findings discussed above (See Figure 5.8).

```
[13]: print("Permissions Model Metrics:")
      evaluate_model(model, test_loader)

      Permissions Model Metrics:
      Sample predictions: [np.int64(1), np.int64(0), np.int64(0), np.int64(1), np.int64(1), np.int64(0),
      int64(0), np.int64(1)]
                    precision    recall  f1-score   support

            benign       0.80      0.86      0.83        78
         malicious       0.85      0.78      0.81        77

          accuracy                           0.82       155
         macro avg       0.82      0.82      0.82       155
      weighted avg       0.82      0.82      0.82       155
```

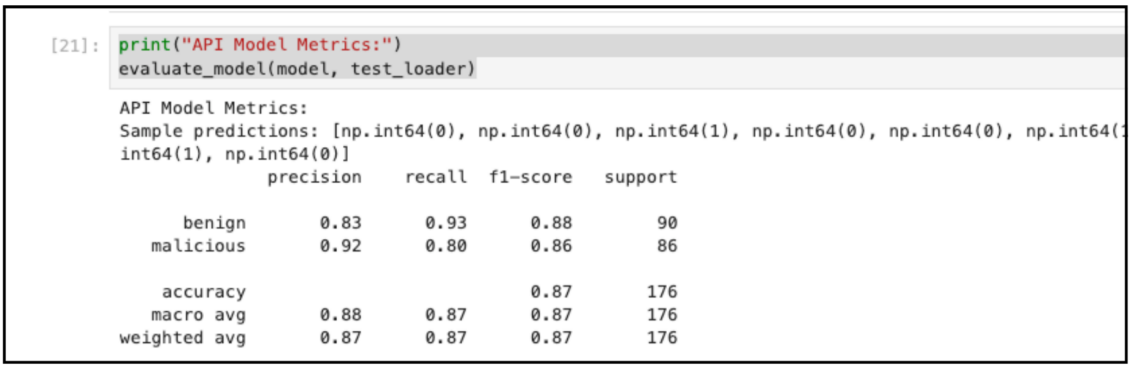
Figure 5.2: Permission Only Model Metrics

```
[11]: # Evaluate Model
      def evaluate_model(model, test_loader):
          model.eval()
          correct = 0
          total = 0
          with torch.no_grad():
              for batch in test_loader:
                  input_ids, attention_mask,
                  outputs = model(input_ids,
                  predictions = torch.argmax(
                  correct += (predictions ==
                  total += labels.size(0)
          print(f"Test Accuracy: {correct / t

      evaluate_model(model, test_loader)

      Test Accuracy: 0.8194
```

Figure 5.3: Permission Only Model Test Accuracy



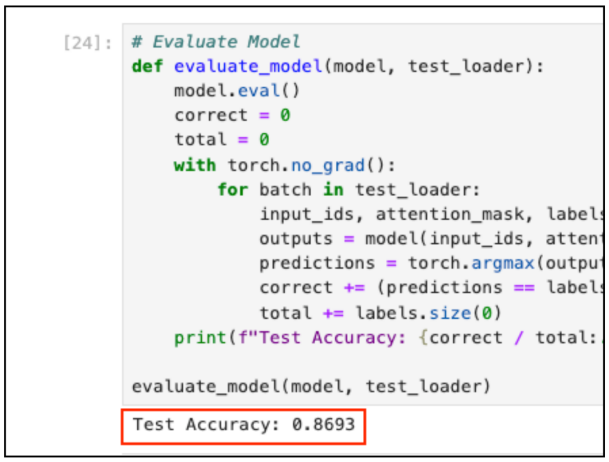
```
[21]: print("API Model Metrics:")
      evaluate_model(model, test_loader)

API Model Metrics:
Sample predictions: [np.int64(0), np.int64(0), np.int64(1), np.int64(0), np.int64(0), np.int64(1
int64(1), np.int64(0)]
              precision    recall  f1-score   support

      benign       0.83      0.93      0.88        90
   malicious       0.92      0.80      0.86        86

    accuracy                           0.87       176
   macro avg       0.88      0.87      0.87       176
weighted avg       0.87      0.87      0.87       176
```

Figure 5.4: API Only Model Metrics



```
[24]: # Evaluate Model
      def evaluate_model(model, test_loader):
          model.eval()
          correct = 0
          total = 0
          with torch.no_grad():
              for batch in test_loader:
                  input_ids, attention_mask, labels
                  outputs = model(input_ids, attent
                  predictions = torch.argmax(output
                  correct += (predictions == labels
                  total += labels.size(0)
          print(f"Test Accuracy: {correct / total:.

      evaluate_model(model, test_loader)

      Test Accuracy: 0.8693
```

Figure 5.5: API only Model Test Accuracy

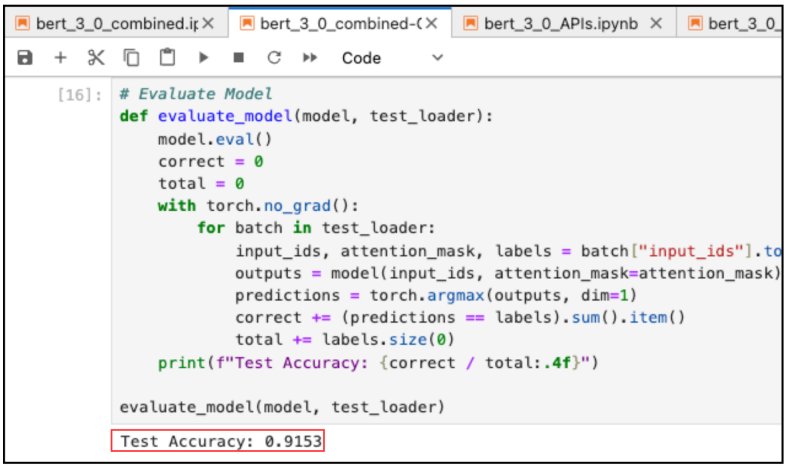


Figure 5.6: Combined Model Test Accuracy

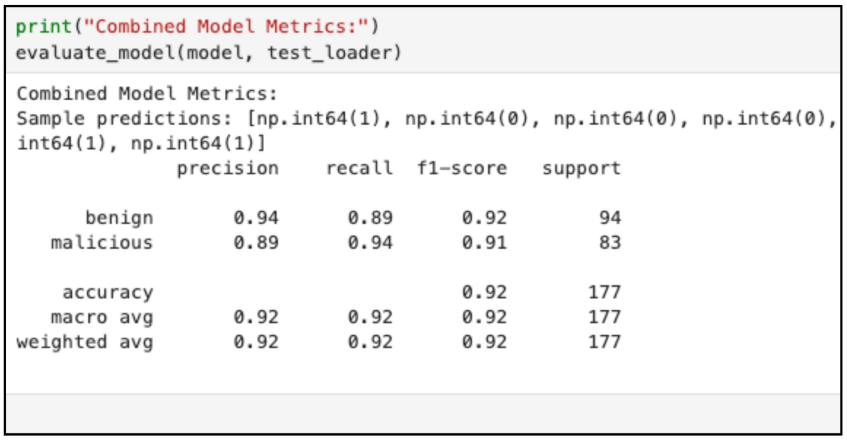


Figure 5.7: Combined Model Metrics

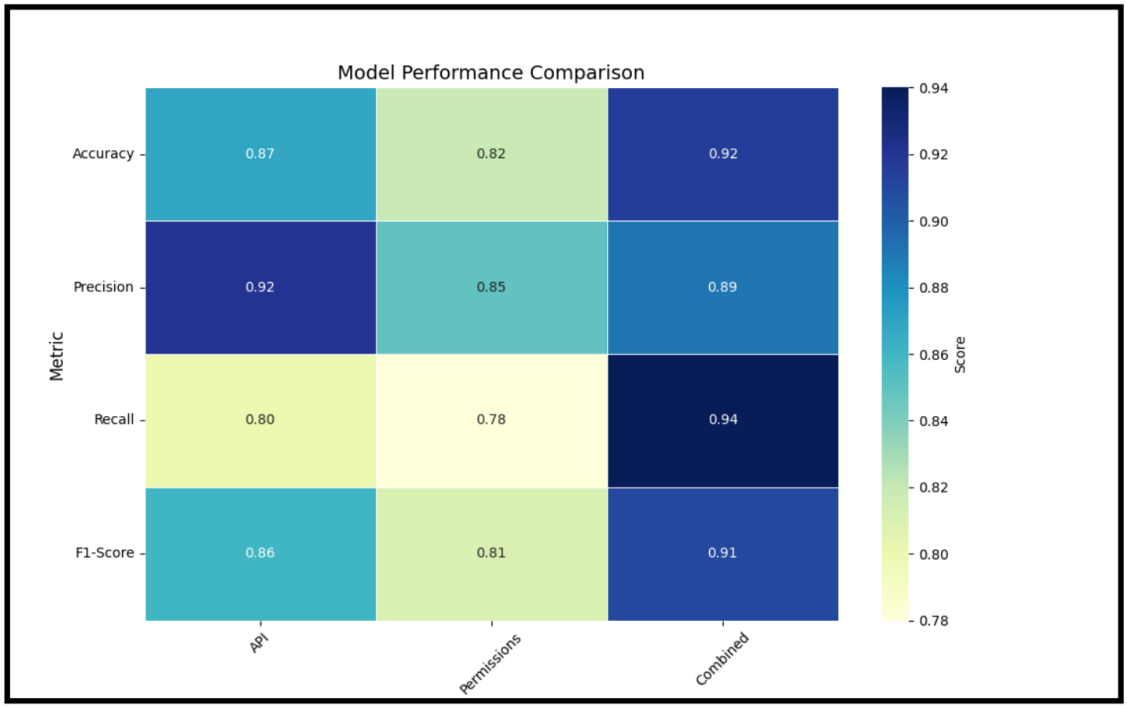


Figure 5.8: Model HeatMap

The bar chart in Figure 5.9 illustrates the class balance between the final dataset used for model training and evaluation. The whole collection comprises around 480 benign apps and 420 malicious applications. The balanced distribution is required to guarantee that the model does not develop a classification bias for either class. The near equal representation of both classes allows for fair training and accurate evaluation of accuracy, recall, and F1-score across malware detection tasks.

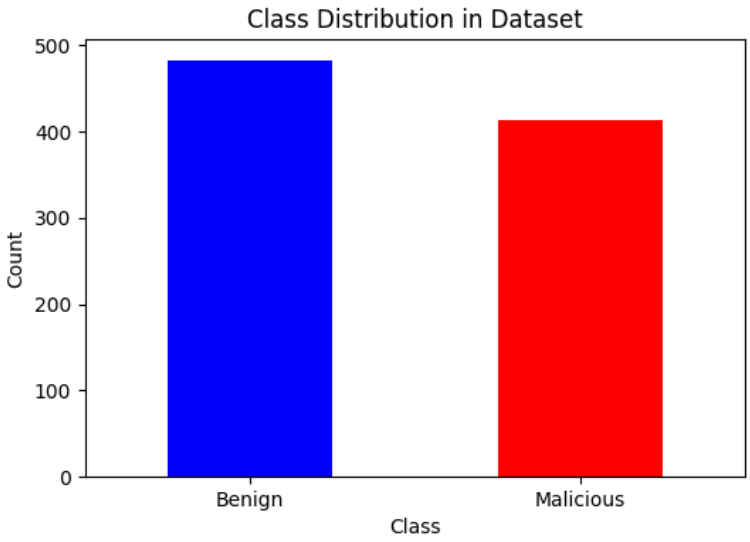


Figure 5.9: Data Distribution in the training Data Set

These results illustrate that combining both permissions with API calls offers the highest detection accuracy. The API only model is relatively accurate and has a higher recall for benign and a lower recall for malicious. The combined model's better recall (and thus high accuracy) implies it detects practically all malware with minimal false alarms. Overall, the quantitative results show that richer features lead to stronger, more balanced performance in malware classification.

5.2 Qualitative Analysis

Proposed solution's feature extraction is completely based on static. The model takes an app's manifest permissions and deconstructed API call sequences as "token" text. BERT recognizes tiny patterns in tokens, such as typical API call sequences, which indicate spying activities. Because the features (permission and API function names) are human-readable, the model's judgments may be explained by referring to specific tokens. Attention weights in the combined model emphasize crucial permissions (e.g., SEND_SMS) and sensitive APIs (e.g., phone or camera

calls), suggesting harmful intent.

Combining permissions and API characteristics strengthens the classifier. Prior work has proven that relying just on permissions reduces speed since other manifest features include malware signals [11]. Proposed model's "combine" setup is similar to the complete manifest in that it captures more contextual indicators, such as when an app does not seek a harmful permission but nevertheless calls unsafe APIs. In real world scenarios, interpretability matters: An analyst can see that the proposed model flagged an app due to it has requested RECORD_AUDIO and used network-sending APIs, for example. This makes it easy to believe and validate the detection. So, in summary, the LLM-based method offers extensive characteristics and interpretability through attention. Leveraging permission and API information results in the most trustworthy malware indicators.

5.3 Insights and Observations

The combined model consistently beats the individual feature models. Using permissions and APIs gives complementary information, allowing the classifier to detect malware patterns that one feature set alone may overlook. The combined model performs best due to synergy. The permission-only model is simpler and often easier to read (it shows which permission requests are questionable), but it fails to detect stealthy malware that requires few overt rights. It has good accuracy but reduced recall, leading to more false negatives. The API-only approach captures dynamic behaviors more effectively and with great accuracy. However, because it lacks information about what the app is permitted to do (permissions), it occasionally generates false positives on benign apps that use popular libraries.

Overall, each model has advantages: permission features are user-friendly and lightweight, API features are rich in behavioral signals, and their combination produces strong performance. The key problem is keeping up with emerging malware

methods that use elements other than static manifests.

5.4 Interpretation of Findings

The findings can be evaluated from both methodological and practical perspectives. Methodologically, the high accuracy and balanced precision/recall show that big language models can extract malware related semantics from static characteristics. This confirms that code and security signals may be properly expressed using language-like inputs. Consistent detection of malicious content suggests that the pretrained language knowledge is effectively leveraged (for example, understanding that certain API sequences often imply harmful intent).

Compared to simpler models, the LLM's performance suggests that semantic embeddings increase detection without losing reliability. For example, the model's capacity to include correlated feature contexts (such as coupled permission sets) most certainly contributes to its performance. This approach verifies the notion that including semantic reasoning with feature analysis improves accuracy and explainability.

5.5 Limitations

Despite the positive results, there are some limitations. The proposed solution is exclusively based on **static features** extracted from APK. This cannot identify and detect malware whose behavior is determined by runtime conditions or external commands. According to Drebin's authors, static analysis cannot often detect attacks relying on dynamic code loading or reflection [2]. Again, **obfuscation resilience** or else heavy code obfuscation can remove the textual clues the model uses. Since LLMs rely on extracted API calls and texts, complete obfuscation would nullify such signals. According to prior work [56], it indicates that decompiled code com-

plexity considerably inhibits existing models. In practice, sophisticated malware could exploit this by obscuring its behavior. The next identified limitation was the **model unpredictability**. Pre trained model that have been used is not specifically trained for Android malware. The algorithm's findings can be unpredictable, with minor prompt alterations leading to various interpretations and the possibility of false links. There is limited control over its internal reasoning beyond careful prompt design.

Another limitation identified was the **computational cost**. BERT-based models are resource-heavy. Running a complete transformer on-device for each app is presently impractical; hence, the assessment assumes analysis on a powerful server or offline batch. Real-time scanning requires model compression or a two-stage pipeline. Used a classifier in this research project, like other ML models, may be deceived by adversarial approaches. Malware writers may create inputs that confound the language model, such as adding irrelevant tokens, without using adversarial training or robust pre-processing.

5.6 Implications for Research and Practice

Finally, all of these findings point in numerous directions. Large language models have proved effective and explainable for Android malware detection. Future studies might involve pretraining LLMs on large code corpora or integrating multi-model data (e.g., code tokens and dynamic logs). Incorporating dynamic analysis is an important next step; for example, feeding sequences of system call logs or network events into a transformer may detect behavior that static characteristics miss. In practice, LLM-based detectors might be employed in app store vetting or as a secondary check following simpler heuristics.

In order to solve performance limits, real-time deployments could employ smaller distilled models or run analysis in the cloud. To respond to changing threats, it's

crucial to fine-tune the model on newly found malware on a regular basis.

Overall, leveraging LLMs in malware detection creates new opportunities for semantic analysis. By treating app code and manifests as "language," we can capture nuanced patterns of malicious behavior. Future research will likely focus on more complex prompt-based analysis, multilingual models (for regional app marketplaces), and interaction with standard signature approaches. The proposed BERT-based model is very accurate and interpretable, highlighting the potential of merging language models with security features for Android protection research and practice.

6 Conclusion and Future Work

6.1 Summary of Contributions

This thesis presented a static-analysis-based Android malware detection system that uses transformer-based language models to examine program permissions and API call behavior. The system was created, constructed, and assessed using publicly accessible datasets (CIC-AndMal2017) [12], with a focus on feature interpretability and model performance. Key contributions include:

- Quantitative analysis indicates that combined static features yield higher accuracy and F1-scores compared to single-feature models. This directly addresses Research Question 1 (Refer to Section 1.3).

- A newly developed ensemble inference mechanism combines three fine-tuned BERT-based classifiers (API-only, permissions-only, and combined), enhancing both robustness and accuracy. This directly addresses Research Question 2 (Refer to Section 1.3), demonstrating that combining API calls and permissions leads to improved malware identification compared to using any single model alone.

- An end-to-end solution was developed, comprising an automated APK feature extraction pipeline, a model inference back-end using FastAPI, and an interactive React-based web front-end for real-time analysis. This implementation

effectively addresses Research Question 3 (Refer to Section 1.3) by demonstrating the feasibility of a practical, real-time malware detection system.

- To avoid execution of potentially harmful code, a safety-aware architecture that relies on static reverse engineering rather than dynamic execution.

6.2 Recommendations

Based on the insights obtained from the development and evaluation, several practical recommendations can be made:

- Higher accuracy results on combined feature sets over individual feature set: These results consistently suggest that merging permissions and API calls into a single feature vector improves efficiency.
- Majority voting technique increases generalization by exploiting the capabilities of individual models.
- Avoiding huge models unless you have enough GPU memory, as those are powerful, but those models are impractical for real-world deployments.
- In order to increase model clarity and decrease training noise, the usage of pattern matching is used in this work.

6.3 Future Research Directions

Even though the system achieved excellent performance in the benchmark data sets, there may be several opportunities to advance the work.

- **Integration of dynamic analysis**: Combining runtime behaviors such as system calls, network traffic, and execution traces with static features may improve detection coverage, particularly for obfuscated malware.

- **Adversarial robustness**: Identifying how well the models resist evasion via API concealment, permission misuse, or adversarial token insertion will improve the system's reliability in hostile environments.

- **Fine-grained classification**: Future models could predict malware families or categories rather than binary labels, allowing for forensic examination and targeted action.

- **Lightweight transformer alternatives**: Models such as DistilBERT [57] or TinyBERT [58] can be used for on-device inference if full-scale BERT is computationally expensive.

- **Dataset extension and diversity**: Including newer apps from real-world markets (such as Google Play and third-party shops) may result in greater generalization to unknown threats.

- **Model explainability frameworks**: Incorporating tools such as SHAP or LIME can assist analysts in visualizing feature importance, increasing trust in automated judgments.

References

[1] M. A. Ashawa and S. Morris, "Analysis of mobile malware: A systematic review of evolution and infection strategies", *Journal of Information Security and Cybercrimes Research*, vol. 4, no. 2, pp. 1–29, 2021. DOI: 10.26735/krvi8434.

[2] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket", in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, ISBN 1-891562-35-5, San Diego, CA, USA: Internet Society, 2014. DOI: 10.14722/ndss.2014.23247.

[3] We Are Social and Meltwater, *Digital 2024 Global Overview Report*, [Accessed: 15-Feb-2025], 2024. [Online]. Available: https://datareportal.com/reports/digital-2024-global-overview-report.

[4] International Data Corporation (IDC), *Global market share held by mobile operating systems from 2009 to 2025*, [Accessed: 15-Feb-2025], 2025. [Online]. Available: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/.

[5] Kaspersky, *Attacks on mobile devices significantly increase in 2023*, Accessed: 2025-02-27, 2023. [Online]. Available: https://www.kaspersky.com/about/press-releases/attacks-on-mobile-devices-significantly-increase-in-2023.

[6] Kaspersky, *Android vs. iOS Security Comparison 2023*, [Accessed: 16-Feb-2025], 2023. [Online]. Available: https://www.kaspersky.com/resource-center/threats/android-vs-iphone-mobile-security.

[7] I. Bouchrika, *Latest statistics on mobile and desktop usage*, Accessed: 2025-04-15, 2025. [Online]. Available: https://research.com/software/mobile-vs-desktop-usage.

[8] M. A. Ashawa and S. Morris, "Analysis of android malware detection techniques: A systematic review", *International Journal of Cyber-Security and Digital Forensics (IJCSDF)*, vol. 8, no. 3, pp. 177–187, 2019. DOI: 10.17781/P002605.

[9] U. e Hani Tayyab, F. B. Khan, M. H. Durad, A. Khan, and Y. S. Lee, "A survey of the recent trends in deep learning based malware detection", *Journal of Cybersecurity and Privacy*, vol. 2, no. 4, p. 41, 2022. DOI: 10.3390/jcp2040041.

[10] Z. Xu, X. Fang, and G. Yang, "Malbert: A novel pre-training method for malware detection", *Computers & Security*, vol. 111, p. 102 458, 2021. DOI: 10.1016/j.cose.2021.102458.

[11] B. Souani, A. Khanfir, A. Bartel, K. Allix, and Y. L. Traon, "Android malware detection using bert", in *Applied Cryptography and Network Security Workshops (ACNS 2022)*, ser. Lecture Notes in Computer Science, vol. 13285, Rome, Italy: Springer, 2022, pp. 575–591. DOI: 10.1007/978-3-031-16815-4_31.

[12] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community", in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16, Austin, Texas: ACM, 2016, pp. 468–471, ISBN: 978-1-4503-4186-8. DOI: 10.1145/2901739.2903508.

[13] K. Tam, A. Feizollah, N. B. Anuar, R. B. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques", *ACM Computing Surveys*, vol. 49, no. 4, pp. 1–41, 2017. DOI: 10.1145/3017427.

[14] GeeksforGeeks, *History of android*, Accessed: 2025-04-25, 2025. [Online]. Available: https://www.geeksforgeeks.org/history-of-android/.

[15] R. Geuens, *What's android's market share? (updated jan 2025)*, Accessed: 2025-04-25, 2025. [Online]. Available: https://soax.com/research/android-market-share.

[16] ACT | The App Association, "Security and trust from an app maker's point of view", ACT | The App Association, Tech. Rep., Nov. 2021. [Online]. Available: https://actonline.org/wp-content/uploads/App-Association-Security-and-Trust-from-an-App-Makers-Point-of-View-November-2021.pdf.

[17] R. M. William J. Buchanan Simone Chiale, "A methodology for the security evaluation within third-party android marketplaces", *Computers & Security*, vol. 23, pp. 88–98, 2017. DOI: https://doi.org/10.1016/j.diin.2017.10.002.

[18] A. Shibly, "Android operating system: Architecture, security challenges and solutions", *International Journal of Computer Applications*, 2016. DOI: 10.13140/RG.2.1.4966.3126.

[19] A. Reijonen, "The evolution of mobile malware", Master's Thesis, Jamk University of Applied Sciences, Apr. 2024. [Online]. Available: https://www.theseus.fi/bitstream/handle/10024/851969/Reijonen_Arttu.pdf?sequence=2&isAllowed=y.

[20] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution", *IEEE Symposium on Security and Privacy*, vol. 45, no. 21, pp. 12–17, May 2012. DOI: 10.1109/SP.2012.16.

[21] F-Secure Labs, *Trojan:android/fakeplayer.a*, Accessed: 2025-04-27, 2010. [Online]. Available: https://www.f-secure.com/v-descs/trojan-android-fakeplayer.shtml.

[22] I. Lunden, *Android accounted for 79% of all mobile malware in 2012, 96% in q4 alone, says f-secure*, Accessed: 2025-04-27, Mar. 2013. [Online]. Available: https://techcrunch.com/2013/03/07/f-secure-android-accounted-for-79-of-all-mobile-malware-in-2012-96-in-q4-alone/.

[23] Malpedia Contributors, *Cerberus (android)*, Accessed: 2025-04-27, 2019. [Online]. Available: https://malpedia.caad.fkie.fraunhofer.de/details/apk.cerberus.

[24] Malpedia Contributors, *Vultur (android)*, Accessed: 2025-04-27, 2021. [Online]. Available: https://malpedia.caad.fkie.fraunhofer.de/details/apk.vultur.

[25] G. Kaur and A. H. Lashkari, *Understanding android malware families (uamf): The foundations – article 1*, Accessed: 2025-04-27, Jan. 2021. [Online]. Available: https://www.itworldcanada.com/blog/understanding-android-malware-families-uamf-the-foundations-article-1/441562.

[26] H. Wang, Z. Liu, J. Liang, *et al.*, "Beyond google play: A large-scale comparative study of chinese android app markets", in *Proceedings of the 2018 Internet Measurement Conference*, Boston, MA, USA: Association for Computing Machinery, 2018. DOI: 10.1145/3278532.3278558.

[27] P. Kotzias, J. Caballero, and L. Bilge, *How did that get in my phone? unwanted app distribution on android devices*, Oct. 2021. [Online]. Available: 10.1109/sp40001.2021.00041.

[28] K. S. Adu-Manu, R. K. Ahiable, J. K. Appati, and E. E. Mensah, "Phishing attacks in social engineering: A review", *Journal of Cyber Security*, 2023. DOI: 10.32604/jcs.2023.041095.

[29] S. Dong, M. Li, W. Diao, *et al.*, *Understanding android obfuscation techniques: A large-scale investigation in the wild*, Jan. 2018. DOI: 10.1007/978-3-030-01701-9_10.

[30] W. F. Elsersy, A. Feizollah, and N. B. Anuar, "The rise of obfuscated android malware and impacts on detection methods", *PeerJ Computer Science*, vol. 8, 2022. DOI: 10.7717/peerj-cs.907.

[31] A. I. Aysan, F. Sakiz, and S. Sen, "Analysis of dynamic code updating in android with security perspective", *IET Information Security*, vol. 13, pp. 167–292, 3 2019. DOI: 10.1049/iet-ifs.2018.5316.

[32] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications", in *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS)*, The Internet Society, 2014. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2017/09/10_5_0.pdf.

[33] D. Tynan, *Hummingbad android malware: Who did it, why, and is your device infected?*, Accessed: 2025-04-27, Jul. 2016. [Online]. Available: https://www.theguardian.com/technology/2016/jul/06/what-is-hummingbad-malware-android-devices-checkpoint/.

[34] Z. Chen, "A comprehensive study of privacy leakage vulnerability in android app logs", in *Proceedings of the 39th IEEE/ACM International Conference on*

Automated Software Engineering (ASE 2024), New York, NY, United States: ACM, 2024, pp. 2510–2513. DOI: 10.1145/3691620.3695609.

[35] Ö. Aslan and R. Samet, "A comprehensive review on malware detection approaches", *IEEE Access*, vol. 8, pp. 6249–6271, 2020. DOI: 10.1109/ACCESS.2019.2963724.

[36] R. Tahir, "A study on malware and malware detection techniques", *International Journal of Education and Management Engineering (IJEME)*, pp. 20–30, 2018. DOI: 10.5815/ijeme.2018.02.03.

[37] D. Natsos and A. L. Symeonidis, "Transformer-based malware detection using process resource-utilization metrics", *Journal of Information Security and Applications*, vol. 75, p. 104 250, 2025. DOI: 10.1016/j.rineng.2025.104250.

[38] K. Shaukat, S. Luo, and V. Varadharajan, "A novel method for improving the robustness of deep learning-based malware detectors against adversarial attacks", *Engineering Applications of Artificial Intelligence*, vol. 116, p. 105 082, 2022. DOI: 10.1016/j.engappai.2022.105461.

[39] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android", in *Proceedings of the 9th International Conference on Security and Privacy in Communication Networks (SecureComm 2013)*, vol. 127, Cham, Switzerland: Springer International Publishing, 2013, pp. 86–103. DOI: 10.1007/978-3-319-04283-1_6.

[40] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models", in *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*, San Diego, California: Internet Society, 2017. DOI: 10.14722/ndss.2017.23353.

[41] A. H. E. Fiky, "Deep-droid: Deep learning for android malware detection", *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, vol. 9, no. 12, pp. 122–125, 2020. DOI: 10.35940/ijitee.L7889.1091220.

[42] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "Maldozer: Automatic framework for android malware detection using deep learning", *Digital Investigation*, vol. 24, S48–S59, 2018. DOI: 10.1016/j.diin.2018.01.007.

[43] H.-D. Huang and H.-Y. Kao, "R2-d2: Color-inspired convolutional neural network (cnn)-based android malware detections", in *2018 IEEE International Conference on Big Data (Big Data)*, Seattle, WA, USA: IEEE, 2018, pp. 2633–2642. DOI: 10.1109/BigData.2018.8622324.

[44] W. Zhao, J. Wu, and Z. Meng, "Apppoet: Large language model based android malware detection via multi-view prompt engineering", *Expert Systems with Applications*, vol. 262, 2025. DOI: 10.1016/j.eswa.2024.125546.

[45] X. Qian, X. Zheng, Y. He, S. Yang, and L. Cavallaro, "Lamd: Context-driven android malware detection and classification with llms", Apr. 2025. DOI: 10.48550/arXiv.2502.13055.

[46] R. Feng, H. Chen, S. Wang, M. M. Karim, and Q. Jiang, "LLM-MalDetect: A large language model-based method for android malware detection", *IEEE Access*, vol. 13, pp. 81 347–81 364, 2025. DOI: 10.1109/ACCESS.2025.3565526.

[47] K. Kumari, *Building language models: A step-by-step bert implementation guide*, Accessed: 2025-05-13, Apr. 2025. [Online]. Available: https://www.analyticsvidhya.com/blog/2023/06/step-by-step-bert-implementation-guide/.

[48] R. Sato, D. Chiba, and S. Goto, "Detecting android malware by analyzing manifest files", in *Proceedings of the Asia-Pacific Advanced Network*, vol. 36, Daejeon, Korea, 2013, pp. 23–31. DOI: 10.7125/APAN.36.4.

[49] A. Rahali and M. A. Akhloufi, "Malbert: Malware detection using bidirectional encoder representations from transformers", in *2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Melbourne, Australia, 2021, pp. 3226–3231. DOI: 10.1109/SMC52423.2021.9659287.

[50] M. S. Alam and S. 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*, Beijing, China, 2013, pp. 663–669. DOI: 10.1109/GreenCom-iThings-CPSCom.2013.122.

[51] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding", *CoRR*, vol. 1, 2018. DOI: 10.48550/arXiv.1810.04805.

[52] I. Solaiman, M. Brundage, J. Clark, *et al.*, "Release strategies and the social impacts of language models", vol. 1, 2019. DOI: 10.48550/arXiv.1908.09203.

[53] K. Grace-Martin, *What is a logit function and why use logistic regression?*, Accessed: 2025-05-13. [Online]. Available: https://www.theanalysisfactor.com/what-is-logit-function/.

[54] K. Stalin and M. B. Mekoya, *Improving android malware detection through data augmentation using wasserstein generative adversarial networks*, 2024. DOI: 10.48550/arXiv.2403.00890. arXiv: 2403.00890.

[55] C. Rondanini, B. Carminati, E. Ferrari, A. Gaudiano, and A. Kundu, *Malware detection at the edge with lightweight llms: A performance evaluation*, 2025.

DOI: https://doi.org/10.48550/arXiv.2403.00890. arXiv: 2503.04302 [cs.CR].

[56] Y. He, H. She, X. Qian, *et al.*, *On benchmarking code llms for android malware analysis*, License: CC BY-NC-SA 4.0, Apr. 2025. DOI: 10.1145/3713081.3731745.

[57] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: Smaller, faster, cheaper and lighter", in *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*, Vancouver, BC, Canada, 2019. DOI: 10.48550/arXiv.1910.01108.

[58] X. Jiao, Y. Yin, L. Shang, *et al.*, "Tinybert: Distilling bert for natural language understanding", *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2019. DOI: 10.18653/v1/2020.findings-emnlp.372.

Appendix A Implemented Code Snippets

API model Training Codes

[2]:
```
import torch
import transformers
import pandas as pd
from sklearn.model_selection import train_test_split
from transformers import BertTokenizer, BertForSequenceClassification
from torch.utils.data import DataLoader, Dataset
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
```

[3]:
```
# Load dataset
df = pd.read_csv("../datasets/api_calls_only.csv").dropna()
```

[4]:
```
# Load BERT tokenizer
tokenizer = BertTokenizer.from_pretrained("bert-large-uncased")
```

[5]:
```
# Tokenize permissions text (include attention_mask)
def tokenize_function(text):
    tokens = tokenizer(text, padding="max_length", truncation=True, max_length=128, return_tensors="pt")
    return tokens["input_ids"][0], tokens["attention_mask"][0]

df[["tokenized", "attention_mask"]] = df["api_calls"].apply(lambda x: pd.Series(tokenize_function(x)))
```

[6]:
```
in_test_split(df[["tokenized", "attention_mask"]].values.tolist(), df["label"].tolist(), test_size=0.2, random_state=42)



torch.as_tensor(d[0]).clone().detach() for d in data])
tack([torch.as_tensor(d[1]).clone().detach() for d in data])
els, dtype=torch.long)




t_ids[idx], "attention_mask": self.attention_masks[idx], "labels": self.labels[idx]}
```

[7]:
```
# Create Dataloaders
train_dataset = MalwareDataset(X_train, y_train)
test_dataset = MalwareDataset(X_test, y_test)
train_loader = DataLoader(train_dataset, batch_size=8, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=8, shuffle=False)
```

Figure A.1: API Code 1

[8]:
```
# Load BERT model
model = BertForSequenceClassification.from_pretrained("bert-large-uncased", num_labels=2)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

```
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 1024, padding_idx=0)
      (position_embeddings): Embedding(512, 1024)
      (token_type_embeddings): Embedding(2, 1024)
      (LayerNorm): LayerNorm((1024,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-23): 24 x BertLayer(
          (attention): BertAttention(
            (self): BertSdpaSelfAttention(
              (query): Linear(in_features=1024, out_features=1024, bias=True)
              (key): Linear(in_features=1024, out_features=1024, bias=True)
              (value): Linear(in_features=1024, out_features=1024, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
```

[9]:
```
# Define Optimizer and Loss Function
optimizer = torch.optim.AdamW(model.parameters(), lr=2e-5, weight_decay=0.01) # Reduce from 2e-5
loss_fn = torch.nn.CrossEntropyLoss()
```

[10]:
```
# Training Loop (Modified to include attention_mask)
def train_model(model, train_loader, epochs=7):
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        for batch in train_loader:
            input_ids, attention_mask, labels = batch["input_ids"].to(device), batch["attention_mask"].to(device), batc
            optimizer.zero_grad()
            outputs = model(input_ids, attention_mask=attention_mask).logits  # Include attention_mask
            loss = loss_fn(outputs, labels)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        print(f"Epoch {epoch+1} Loss: {total_loss / len(train_loader)}")
```

Figure A.2: API Code 2

```
[ ]: # Train Model
train_model(model, train_loader)
```

```
Epoch 1 Loss: 0.4352254495024681
Epoch 2 Loss: 0.37127459032291715
Epoch 3 Loss: 0.29360353282060137
Epoch 4 Loss: 0.27219482363117015
Epoch 5 Loss: 0.24413511652330105
Epoch 6 Loss: 0.19564919673245063
```

```
[20]: from sklearn.metrics import classification_report, f1_score, precision_score, recall_score

def evaluate_model(model, test_loader):
    model.eval()
    y_true = []
    y_pred = []
    with torch.no_grad():
        for batch in test_loader:
            input_ids = batch["input_ids"].to(device)
            attention_mask = batch["attention_mask"].to(device)
            labels = batch["labels"].to(device)

            outputs = model(input_ids, attention_mask=attention_mask).logits
            predictions = torch.argmax(outputs, dim=1)

            y_true.extend(labels.cpu().numpy())
            y_pred.extend(predictions.cpu().numpy())

    print("Sample predictions:", y_pred[:10])  # Debugging line
    print(classification_report(y_true, y_pred, target_names=['benign', 'malicious']))
```

```
[22]: # Save Model
torch.save(model.state_dict(), "malware_bert_model_api_2.pth")
```

Figure A.3: API Code 3

```
[21]: print("API Model Metrics:")
evaluate_model(model, test_loader)
```

```
API Model Metrics:
Sample predictions: [np.int64(0), np.int64(0), np.int64(1), np.int64(0), np.int64(0), np.int64(1), np.int64(1), np.int
64(0), np.int64(1), np.int64(0)]
              precision    recall  f1-score   support

      benign       0.83      0.93      0.88        90
   malicious       0.92      0.80      0.86        86

    accuracy                           0.87       176
   macro avg       0.88      0.87      0.87       176
weighted avg       0.87      0.87      0.87       176
```

Figure A.4: API Code 4

[24]:
```python
# Evaluate Model
def evaluate_model(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for batch in test_loader:
            input_ids, attention_mask, labels = batch["input_ids"].to(device), batch["attention_mask"].to(device), batc
            outputs = model(input_ids, attention_mask=attention_mask).logits
            predictions = torch.argmax(outputs, dim=1)
            correct += (predictions == labels).sum().item()
            total += labels.size(0)
    print(f"Test Accuracy: {correct / total:.4f}")

evaluate_model(model, test_loader)
```

```
Test Accuracy: 0.8693
```

Figure A.5: API Code 5

[1]:
```python
import torch
import transformers
import pandas as pd
from sklearn.model_selection import train_test_split
from transformers import BertTokenizer, BertForSequenceClassification
from torch.utils.data import DataLoader, Dataset
```

[2]:
```python
# Load dataset
df = pd.read_csv("../datasets/permissions_only_12_04_2025.csv").dropna()
```

[3]:
```python
# Load BERT tokenizer
tokenizer = BertTokenizer.from_pretrained("bert-large-uncased")
```

[4]:
```python
# Tokenize permissions text (include attention_mask)
def tokenize_function(text):
    tokens = tokenizer(text, padding="max_length", truncation=True, max_length=128, return_tensors="pt")
    return tokens["input_ids"][0], tokens["attention_mask"][0]

df[["tokenized", "attention_mask"]] = df["permissions"].apply(lambda x: pd.Series(tokenize_function(x)))
```

[5]:
```python
# Train-test split
X_train, X_test, y_train, y_test = train_test_split(df[["tokenized", "attention_mask"]].values.tolist(), df["label"].to

class MalwareDataset(Dataset):
    def __init__(self, data, labels):
        self.input_ids = torch.stack([torch.as_tensor(d[0]).clone().detach() for d in data])
        self.attention_masks = torch.stack([torch.as_tensor(d[1]).clone().detach() for d in data])
        self.labels = torch.tensor(labels, dtype=torch.long)

    def __len__(self):
        return len(self.input_ids)

    def __getitem__(self, idx):
        return {"input_ids": self.input_ids[idx], "attention_mask": self.attention_masks[idx], "labels": self.labels[id
```

[6]:
```python
# Create Dataloaders
train_dataset = MalwareDataset(X_train, y_train)
test_dataset = MalwareDataset(X_test, y_test)
train_loader = DataLoader(train_dataset, batch_size=8, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=8, shuffle=False)
```

Figure A.6: Permission Code 1

```
[7]: # Load BERT model
model = BertForSequenceClassification.from_pretrained("bert-large-uncased", num_labels=2)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

```
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-large-uncased and
are newly initialized: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

```
[7]: BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 1024, padding_idx=0)
      (position_embeddings): Embedding(512, 1024)
      (token_type_embeddings): Embedding(2, 1024)
      (LayerNorm): LayerNorm((1024,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-23): 24 x BertLayer(
          (attention): BertAttention(
            (self): BertSdpaSelfAttention(
              (query): Linear(in_features=1024, out_features=1024, bias=True)
              (key): Linear(in_features=1024, out_features=1024, bias=True)
              (value): Linear(in_features=1024, out_features=1024, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=1024, out_features=1024, bias=True)
              (LayerNorm): LayerNorm((1024,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=1024, out_features=4096, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=4096, out_features=1024, bias=True)
            (LayerNorm): LayerNorm((1024,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=1024, out_features=1024, bias=True)
      (activation): Tanh()
```

Figure A.7: Permission Code 2

[8]:
```python
# Define Optimizer and Loss Function
optimizer = torch.optim.AdamW(model.parameters(), lr=2e-5, weight_decay=0.01) # Reduce from 2e-5
loss_fn = torch.nn.CrossEntropyLoss()
```

[9]:
```python
# Training Loop (Modified to include attention_mask)
def train_model(model, train_loader, epochs=7):
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        for batch in train_loader:
            input_ids, attention_mask, labels = batch["input_ids"].to(device), batch["attention_mask"].to(device), batc
            optimizer.zero_grad()
            outputs = model(input_ids, attention_mask=attention_mask).logits  # Include attention_mask
            loss = loss_fn(outputs, labels)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        print(f"Epoch {epoch+1} Loss: {total_loss / len(train_loader)}")
```

10]:
```python
# Train Model
train_model(model, train_loader)
```

```
Epoch 1 Loss: 0.6988011136268958
Epoch 2 Loss: 0.7015403677255679
Epoch 3 Loss: 0.7079687179663242
Epoch 4 Loss: 0.7123036399865762
Epoch 5 Loss: 0.6948819489051135
Epoch 6 Loss: 0.6685657963538781
Epoch 7 Loss: 0.6574595834200199
```

11]:
```python
# Evaluate Model
def evaluate_model(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for batch in test_loader:
            input_ids, attention_mask, labels = batch["input_ids"].to(device), batch["attention_mask"].to(device), batc
            outputs = model(input_ids, attention_mask=attention_mask).logits
            predictions = torch.argmax(outputs, dim=1)
            correct += (predictions == labels).sum().item()
            total += labels.size(0)
    print(f"Test Accuracy: {correct / total:.4f}")

evaluate_model(model, test_loader)
```

```
Test Accuracy: 0.8194
```

Figure A.8: Permission Code 3

[14]:
```
# Save Model
torch.save(model.state_dict(), "malware_bert_model_permission_2.pth")
```

[12]:
```
from sklearn.metrics import classification_report, f1_score, precision_score, recall_score

def evaluate_model(model, test_loader):
    model.eval()
    y_true = []
    y_pred = []
    with torch.no_grad():
        for batch in test_loader:
            input_ids = batch["input_ids"].to(device)
            attention_mask = batch["attention_mask"].to(device)
            labels = batch["labels"].to(device)

            outputs = model(input_ids, attention_mask=attention_mask).logits
            predictions = torch.argmax(outputs, dim=1)

            y_true.extend(labels.cpu().numpy())
            y_pred.extend(predictions.cpu().numpy())

    print("Sample predictions:", y_pred[:10])  # Debugging line
    print(classification_report(y_true, y_pred, target_names=['benign', 'malicious']))
```

[13]:
```
print("Permissions Model Metrics:")
evaluate_model(model, test_loader)
```

```
Permissions Model Metrics:
Sample predictions: [np.int64(1), np.int64(0), np.int64(0), np.int64(1), np.int64(1), np.int64(0), np.int64(1), np.int
64(1), np.int64(0), np.int64(1)]
              precision    recall  f1-score   support

      benign       0.80      0.86      0.83        78
   malicious       0.85      0.78      0.81        77

    accuracy                           0.82       155
   macro avg       0.82      0.82      0.82       155
weighted avg       0.82      0.82      0.82       155
```

Figure A.9: Permission Code 4

[24]:
```python
import torch
import transformers
import pandas as pd
from sklearn.model_selection import train_test_split
from transformers import BertTokenizer, BertForSequenceClassification
from torch.utils.data import DataLoader, Dataset
```

[25]:
```python
# Load dataset
df = pd.read_csv("../datasets/combined_features_all_in_one.csv").dropna()
```

[26]:
```python
# Load BERT tokenizer
tokenizer = BertTokenizer.from_pretrained("bert-large-uncased")
```

[27]:
```python
# Tokenize permissions text (include attention_mask)
def tokenize_function(text):
    tokens = tokenizer(text, padding="max_length", truncation=True, max_length=128, return_tensors="pt")
    return tokens["input_ids"][0], tokens["attention_mask"][0]

df[["tokenized", "attention_mask"]] = df["combined_features"].apply(lambda x: pd.Series(tokenize_function(x)))
```

[28]:
```python
# Train-test split
X_train, X_test, y_train, y_test = train_test_split(df[["tokenized", "attention_mask"]].values.tolist(), df["label"].to

class MalwareDataset(Dataset):
    def __init__(self, data, labels):
        self.input_ids = torch.stack([torch.as_tensor(d[0]).clone().detach() for d in data])
        self.attention_masks = torch.stack([torch.as_tensor(d[1]).clone().detach() for d in data])
        self.labels = torch.tensor(labels, dtype=torch.long)

    def __len__(self):
        return len(self.input_ids)

    def __getitem__(self, idx):
        return {"input_ids": self.input_ids[idx], "attention_mask": self.attention_masks[idx], "labels": self.labels[id
```

[29]:
```python
# Create Dataloaders
train_dataset = MalwareDataset(X_train, y_train)
test_dataset = MalwareDataset(X_test, y_test)
train_loader = DataLoader(train_dataset, batch_size=8, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=8, shuffle=False)
```

Figure A.10: Combine Code 1

[30]:
```
# Load BERT model
model = BertForSequenceClassification.from_pretrained("bert-large-uncased", num_labels=2)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

```
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-large-uncased and
are newly initialized: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

[30]:
```
BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 1024, padding_idx=0)
      (position_embeddings): Embedding(512, 1024)
      (token_type_embeddings): Embedding(2, 1024)
      (LayerNorm): LayerNorm((1024,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-23): 24 x BertLayer(
          (attention): BertAttention(
            (self): BertSdpaSelfAttention(
              (query): Linear(in_features=1024, out_features=1024, bias=True)
              (key): Linear(in_features=1024, out_features=1024, bias=True)
              (value): Linear(in_features=1024, out_features=1024, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=1024, out_features=1024, bias=True)
              (LayerNorm): LayerNorm((1024,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=1024, out_features=4096, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=4096, out_features=1024, bias=True)
            (LayerNorm): LayerNorm((1024,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=1024, out_features=1024, bias=True)
      (activation): Tanh()
```

Figure A.11: Combine Code 2

[31]:
```
# Define Optimizer and Loss Function
optimizer = torch.optim.AdamW(model.parameters(), lr=2e-5, weight_decay=0.01) # Reduce from 2e-5
class_weights = torch.tensor([1.0, 2.0]).to(device)
loss_fn = torch.nn.CrossEntropyLoss(weight=class_weights)
```

[10]:
```
# Training Loop (Modified to include attention_mask)
def train_model(model, train_loader, epochs=7):
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        for batch in train_loader:
            input_ids, attention_mask, labels = batch["input_ids"].to(device), batch["attention_mask"].to(device), batc
            optimizer.zero_grad()
            outputs = model(input_ids, attention_mask=attention_mask).logits  # Include attention_mask
            loss = loss_fn(outputs, labels)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        print(f"Epoch {epoch+1} Loss: {total_loss / len(train_loader)}")
```

[15]:
```
# Train Model
train_model(model, train_loader)
```

```
Epoch 1 Loss: 0.4785419466455331
Epoch 2 Loss: 0.3553588304543093
Epoch 3 Loss: 0.34042967814073133
Epoch 4 Loss: 0.2996809407231513
Epoch 5 Loss: 0.26800727601466556
Epoch 6 Loss: 0.21387396304962342
Epoch 7 Loss: 0.19225218892097473
```

Figure A.12: Combine Code 3

[16]:
```
# Evaluate Model
def evaluate_model(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for batch in test_loader:
            input_ids, attention_mask, labels = batch["input_ids"].to(device), batch["attention_mask"].to(device), batc
            outputs = model(input_ids, attention_mask=attention_mask).logits
            predictions = torch.argmax(outputs, dim=1)
            correct += (predictions == labels).sum().item()
            total += labels.size(0)
    print(f"Test Accuracy: {correct / total:.4f}")

evaluate_model(model, test_loader)
```

```
Test Accuracy: 0.9153
```

[17]:
```
# Save Model
torch.save(model.state_dict(), "malware_bert_model_combined_2.pth")
```

Figure A.13: Combine Code 4

[17]:
```
# Save Model
torch.save(model.state_dict(), "malware_bert_model_combined_2.pth")
```

[18]:
```
from sklearn.metrics import classification_report, f1_score, precision_score, recall_score

def evaluate_model(model, test_loader):
    model.eval()
    y_true = []
    y_pred = []
    with torch.no_grad():
        for batch in test_loader:
            input_ids = batch["input_ids"].to(device)
            attention_mask = batch["attention_mask"].to(device)
            labels = batch["labels"].to(device)

            outputs = model(input_ids, attention_mask=attention_mask).logits
            predictions = torch.argmax(outputs, dim=1)

            y_true.extend(labels.cpu().numpy())
            y_pred.extend(predictions.cpu().numpy())

    print("Sample predictions:", y_pred[:10])
    print(classification_report(
        y_true, y_pred,
        target_names=['benign', 'malicious'],
        zero_division=0  # Suppress UndefinedMetricWarning
    ))
```

[19]:
```
print("Combined Model Metrics:")
evaluate_model(model, test_loader)
```

```
Combined Model Metrics:
Sample predictions: [np.int64(1), np.int64(0), np.int64(0), np.int64(0), np.int64(0), np.int64(0), np.int64(0), np.int
64(0), np.int64(1), np.int64(1)]
              precision    recall  f1-score   support

      benign       0.94      0.89      0.92        94
   malicious       0.89      0.94      0.91        83

    accuracy                           0.92       177
   macro avg       0.92      0.92      0.92       177
weighted avg       0.92      0.92      0.92       177
```

Figure A.14: Combine Code 5