

Received 12 April 2025, accepted 25 April 2025, date of publication 29 April 2025, date of current version 15 May 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3565526

RESEARCH ARTICLE

LLM-MalDetect: A Large Language Model-Based Method for Android Malware Detection

RUIRUI FENG^{1,3}, HUI CHEN², SHUO WANG¹, MD MONJURUL KARIM³,
AND QINGSHAN JIANG³

¹College of Mathematics and Information Science, Hebei University, Baoding 071002, China

²School of Artificial Intelligence, Shenzhen Polytechnic University, Shenzhen 518055, China

³Shenzhen Key Laboratory for High Performance Data Mining, Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China

Corresponding authors: Hui Chen (huichen@szpu.edu.cn) and Shuo Wang (shuowang@hbu.edu.cn)

This work was supported in part by Shenzhen Polytechnic University Research Fund under Grant 6023310010K, Grant 6023310009K, and Grant 6023312038K; in part by the National Key Research and Development Program of China under Grant 2021YFF1200100 and Grant 2021YFF1200104; in part by the Special Projects fund in Key Areas of Guangdong Provincial Department of Education under Grant 2023ZDZX2085; in part by the Natural Science Foundation of Hebei Province under Grant F2021201055; and in part by the Innovation Capacity Enhancement Program-Science and Technology Platform Project, Hebei, under Grant 22567623H.

ABSTRACT Android malware poses a significant cybersecurity threat, enabling unauthorized data access, financial fraud, and device compromise. Although deep learning methods are widely used for malware detection, they often struggle with stability and adaptability in the face of evolving threats. While large language models (LLMs) have shown promise in this area, their application to Android malware detection remains underexplored, particularly with regard to optimizing the semantic relationships within Android application packages (APKs). To address this gap, we introduce LLM-MalDetect, a novel framework that improves LLM-based APK analysis by explicitly modeling semantic dependencies and leveraging structured prompt engineering for optimized detection. Our approach formalizes LLM adaptation through a robust string-based feature extraction method and a tailored fine-tuning strategy to enhance precision. Evaluations on benchmark datasets demonstrate that LLM-MalDetect achieves up to 98.97% accuracy, outperforms existing methods in terms of robustness, and enables real-time analysis.

INDEX TERMS Android malware detection, data privacy, information security, large language model, prompt engineering.

I. INTRODUCTION

Mobile devices, particularly smartphones and tablets, have become essential tools for millions worldwide, supporting a broad spectrum of activities such as web browsing, social media, gaming, banking, and navigation. According to Counterpoint [1], Android accounts for approximately 74% of mobile devices, outpacing both iOS and Harmony OS. Android's open-source nature fosters a diverse development ecosystem, resulting in a vast range of applications for communication, productivity, entertainment, and utilities. However, this openness, while facilitating innovation and customization, exposes the Android platform to significant security risks, including malware, phishing, data breaches,

and unauthorized access. Android malware poses a severe threat as it often targets sensitive user data and financial information. In 2023, Kaspersky reported blocking over 33.8 million malware, adware, and riskware attacks on mobile devices, marking a 50% increase from the previous year [2]. The most prevalent threat was adware, constituting 40.8% of all threats detected. Additionally, over 1.3 million malicious installation packages were identified with 154,000 containing mobile banking Trojans. These statistics underscore the escalating cybersecurity threats faced by Android users.

Given the increasing number and sophistication of malware threats targeting Android devices, numerous approaches have been developed for malware detection [3], [4]. Traditional techniques such as signature-based detection and heuristic analysis [5] are early methods used to identify malicious apps. However, these methods struggle with

The associate editor coordinating the review of this manuscript and approving it for publication was Sedat Akleylek³.

modern Android malware due to their reliance on predefined signatures or simple heuristics, which makes them less effective against polymorphic and obfuscated threats. Static analysis, which involves analyzing the code or structure of the application without executing it, has shown some success, but is also vulnerable to evasion techniques. On the other hand, dynamic analysis analyzes the behavior of apps during execution but can be resource-intensive and limited by sandboxing techniques that prevent accurate detection of real-world behaviors [6]. More recently, machine learning (ML) based methods [7], [8] have been employed, which train models to classify apps based on features such as permissions, API calls, and other static or dynamic attributes [9]. Although these methods have shown promise, they are often limited by the feature representations they use, which do not capture the full complexity of malware behavior. As threats continue to evolve, there is a growing need for more adaptive and effective solutions that can handle increasingly sophisticated attack techniques.

Machine learning (ML) models, including decision trees, random forests, support vector machines (SVM), naive Bayes, and logistic regression, are commonly used to address these challenges. Additionally, deep learning (DL) models such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), and transformers are employed to train classifiers that distinguish between malicious and benign software. However, while DL methods have shown promise, they still face challenges in accurately detecting Android malware, especially when relying on simplistic feature representations such as the inclusion of specific fields within Android application packages (APKs) [10]. These approaches often struggle to capture the complex and dynamic nature of modern malware, which limits their performance in real-world scenarios. As a result, more innovative and adaptive solutions are needed to address the evolving nature of threats. Addressing these specific limitations also requires innovative approaches capable of analyzing extensive data, identifying subtle patterns, and adapting to emerging threats with minimal human intervention.

Large language models (LLMs) have emerged as promising solutions to the challenges faced by traditional malware detection methods. Among these, generative pre-trained transformer (GPT) models [11] have shown significant potential in overcoming these challenges by leveraging their advanced capabilities in data processing and pattern recognition. LLMs excel at analyzing vast amounts of textual and coded data, allowing them to recognize intricate patterns and anomalies that traditional methods may overlook [12]. Through fine-tuning and adaptation, LLMs can be leveraged for applications such as automatic threat detection, malware analysis, vulnerability identification, and security policy generation. By processing extensive datasets, LLMs can detect subtle anomalies, improving accuracy in identifying complex and dynamic malware. For example, the BERT [13] model, comprising an encoder and decoder, excels in masked language modeling and next-sentence prediction. In contrast,

LLMs, primarily decoder-based, are optimized for natural language generation. Although BERT effectively represents natural language through vector embeddings, LLMs exhibit reasoning capabilities closer to human cognition. By integrating LLMs, malware detection systems can be more adaptive, accurately identifying new and unknown threats, improving response time, and ensuring proactive system maintenance.

Despite these promising applications, research on LLMs in Android malware detection is still limited and faces significant challenges, including constrained feature scopes, inadequate feature diversity, insufficient dataset quality, and issues with scalability and adaptation to evolving threats. For example, ChatPhishDetector [14] effectively utilizes LLMs for phishing detection, but is limited to analyzing web-based structures, making it insufficient for detecting more sophisticated Android malware. Consequently, AppPoet [15] incorporates features such as permissions, application programming interface (API) calls, and uniform resource locators (URLs), but lacks comprehensive feature diversity, restricting its adaptability to complex malware scenarios. Besides, Maltracker [16] focuses on enhancing dataset augmentation to improve detection but struggles with the need for multilanguage compatibility, which hinders its practicality in Android-specific contexts. Moreover, Al-Karaki et al. [17] provide an extensive overview of LLMs for malware detection but highlight the lack of a unified approach to systematically address the varied challenges in Android malware detection, including the handling of polymorphic and obfuscated malware. Furthermore, Maltracker [16] also demonstrates limitations in leveraging LLMs to enhance datasets for improved detection performance, facing challenges with fine-grained feature extraction and dynamic adaptation to new threats. These challenges underscore the need for a more versatile, feature-rich, and scalable LLM-based solution for Android malware detection. Given these limitations, the following research questions arise:

- RQ1:** Which APK file features effectively represent malware while being suitable for large-scale model processing?
- RQ2:** How can LLMs be integrated into the Android malware detection framework to improve detection performance?
- RQ3:** Can an LLM-based framework enhance threat detection, reduce response times, and mitigate risks in the detection of modern Android malware?

To address these research questions, we propose LLM-MalDetect, a comprehensive framework that involves three key stages: data preprocessing, feature engineering, and model fine-tuning. In data preprocessing, we decode APKs and analyze their component files. Feature engineering includes feature extraction and feature selection. Feature extraction focuses on identifying crucial attributes such as permissions, API calls, and strings. Although permissions and function calls are commonly used features [4], strings representing descriptions and other text-based details are

less frequently used in previous studies. During the feature selection phase, we apply mutual information to quantify the correlations between features and labels, which allows us to exclude features with low relevance and thereby minimize computational demands without sacrificing accuracy. For model fine-tuning, we employ LLMs for Android malware detection using the Mistral model [18] as the backbone network and the LoRA technique [19] for fine-tuning. Additionally, prompt engineering is introduced at the beginning of training to enhance model performance. This approach addresses **RQ1** by analyzing features such as names, icons, and strings; **RQ2** by integrating LLMs through reasoning and fine-tuning; and **RQ3** by enhancing threat detection through systematic experimental validation, reducing response times through learning from limited samples, and validating the framework's relevance to modern malware using recently collected Android datasets. After training, we conduct experiments with the model, followed by an in-depth analysis of the results.

The summarized contributions of this paper are as follows:

- We propose a feature extraction method capturing string-based features from APKs to enable a comprehensive representation of Android malware.
- We introduce a prompt engineering framework processing strings within APKs, enabling the model to infer application functions and improve detection accuracy.
- We leverage LLMs to analyze feature relationships to enhance detection performance while maintaining high reliability across varying sample sizes.

The remaining part of the article is organized as follows: Section II provides a summary of background and related work. Section III describes the methods and principles employed in our study. Section IV details the experimental settings and results. Section V analyzes the performance in ablation experiments. The conclusion is presented in Section VI.

II. RELATED WORK

This section reviews related work on Android malware detection with a focus on feature extraction, classification methods, and the integration of LLMs.

A. FEATURE EXTRACTION IN ANDROID MALWARE DETECTION

Feature extraction is the process of deriving features from original files to obtain data for modeling, including static or dynamic methods. Static methods analyze the APK file directly without executing the malware, whereas dynamic methods require installing and running the APK to observe runtime behaviors. For example, Nataraj et al. [27] process malware as binary files without any additional treatment before being converted into grayscale images for detection. Based on the above method, Freitas et al. [28] extract DEX files from APKs, convert the title and data into RGB images, and construct a large dataset named MalNet. Karimi

and Moattar [29] analyze opcode distributions in Android ransomware detection. Li et al. [7] use permissions to detect Android malware and identify 22 of the most important permissions for classification. Arp et al. [20] conduct a detailed analysis of the collected APK files from eight aspects, including hardware features, requested permissions, software components, populated intents, restricted API calls, usage permissions, suspicious API calls and network addresses. SWORD [30] is a semantic-aware dynamic malware detection tool that captures app semantics using system-call sequences and applies information theory concepts to detect malware. Wang et al. [31] propose an automatic malware detection method that leverages text semantics of network traffic by treating network flows as text documents. Zhang et al. [32] take optical character recognition to extract textual content from pages. DroidRL [25] employs a reinforcement learning-based wrapper feature selection approach with RNN and word embedding to efficiently select relevant Android malware features, achieving high accuracy with only 24 features. Ou and Xu [33] propose a novel static-sensitive subgraph-based method that leverages sensitive function call graphs to identify malicious behaviors. However, existing feature processing methods [23], [34] face challenges in integrating multi-source features. Some works [7], [29] focus on a single feature, while others combine disparate features using early fusion techniques [23]. Conducting comprehensive detection using a single feature is challenging. Moreover, concatenating outputs from different models fails to account for the correlations among various features, which limits model performance [35]. The utilization of LLMs facilitates alignment with multimodal data [36], enabling simultaneous processing of diverse feature types.

B. CLASSIFICATION METHODS FOR ANDROID MALWARE DETECTION

Based on the feature extraction, researchers apply classification algorithms to predict whether a file is benign or malicious. The most common algorithms are the ML and DL algorithms [3], [22], [24]. For instance, some works [5], [37] apply simple independent classifiers, such as support vector machines and decision trees, for malware classification. Droid-Sec [38] is the earliest method to apply deep learning in the detection of Android malware, utilizing a combination of convolutional and residual neural networks for classification. Zhang et al. [39] apply extreme learning machines to malware detection, employing a stacking method to integrate results from different extreme learning machine classifiers. CNNXGB [23] extracts and analyzes authorization-sensitive features, converts basic blocks into images for CNN-based learning, and employs XGBoost for classification. Besides, Wang et al. [31] transform malware detection task into a text classification problem by capturing network traffic headers and using n-gram representation to classify malware. Yu et al. [40] propose a red-blue confrontation framework that simulates immunological processes and lets the detection

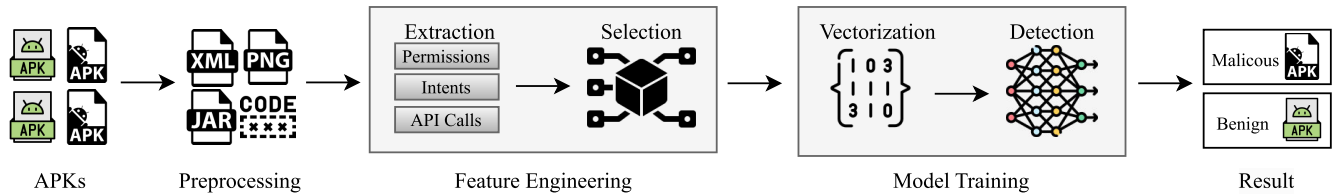


FIGURE 1. Generic framework of android malware detection.

TABLE 1. Used features and detection method of existing frameworks.

Framework	Used features	Detection methods
Drebin [20]	Permissions, components and API calls	SVM
DroidFusion [21]	Permissions, API calls, and commands	Model fusion
MLDroid [22]	permission and API calls	Multiple machine learning techniques
CNNXGB [23]	Permissions, API calls, and binary code	CNN and XGB
DroidMalwareDetector [24]	Permissions, API calls, and intents	CNN
DroidRL [25]	Permissions, intent actions, and op codes	Reinforcement learning and random forest
TransMalDE [10]	Permissions and API calls	Transformer encoder and decoder
AMDDLmodel [26]	Permissions and API calls	CNN
AppPoet [15]	Permissions, API calls, and URL	Pretrained model representation

model and generative model engage in mutual attacks, facilitating the automatic optimization of detection techniques. MsDroid [41] uses graph neural networks to analyze local code snippets around sensitive APIs, enhancing robustness and interpretability. Deng et al. [10] leverage edge computing to enhance real-time detection and use a Transformer-based model to improve accuracy. Although some works [25], [31] incorporate NLP techniques, they often overlook the correlations between features. On the other hand, studies [10], [25] further refine classification methods by incorporating ensemble learning, hybrid DL models, and innovative classification strategies. For example, DroidFusion [21] proposes a multilevel classifier fusion framework that leverages ranking-based aggregation techniques (e.g., AAB, CDB, RAPC, RACD) to combine the outputs of multiple classifiers (such as decision trees and naive bayes), thereby achieving improved accuracy over independent models. Similarly, Al Zaabi and Mouheb [8] employ a broad spectrum of static features including permissions, API calls, services, activities, and opcodes. They evaluate classifiers such as Random Forests, Decision Trees, and Gaussian Processes. Besides, DroidMDetection [24] introduces a CNN-LSTM hybrid approach that leverages DL to capture sequential patterns in static features, which leads to enhanced malware detection. Finally, Aldini and Petrelli [42] propose an image-based approach where APK components are converted into grayscale images. Their CNN-based models (including variants integrated with LSTM and SVM) demonstrate very high detection and classification accuracy. These contributions complement earlier works that relied on simpler classifiers to illustrate an evolving trend toward integrating diverse features and sophisticated classification architectures to achieve state-of-the-art performance in Android malware detection.

C. ANDROID MALWARE DETECTION USING LLMs

LLMs have various applications in the field of malware detection, including malware honeypots, identification of text-based threats, code analysis for detecting malicious intent, trend analysis of malware, and detection of non-standard disguised malware [17]. However, as an emerging technology, the use of LLMs in malware detection research is still in an exploratory stage. For example, Koide et al. [14] is a novel system that uses LLMs to detect phishing websites effectively. By leveraging prompt engineering and HTML simplification, the system achieves detection without the need to train additional models, attaining a precision of 98.7% and a recall of 99.6%. Yu et al. [16] expand datasets by leveraging LLMs to translate targeted malicious functions across different programming languages. Zhao et al. [15] is an LLM-assisted system designed for detecting Android malware. It selects typical features (permissions, APIs, URLs, and uses-feature). A pre-trained embedding model is used to convert features into machine-readable representation vectors, facilitating data augmentation and enhancing the performance of Android malware detection.

D. LIMITATIONS OF EXISTING WORKS AND OUR CONTRIBUTIONS

Current Android malware detection methods still suffer from notable shortcomings. Firstly, feature extraction-based approaches [7], [23], [29] rely predominantly on a narrow set of features, such as permissions and API calls, or they employ rudimentary early fusion techniques (e.g., simple concatenation) to combine features. Such methods fail to capture the intricate relationships and contextual information present in an APK's diverse components (e.g., opcodes, services, activities, and resource files). This limited represen-

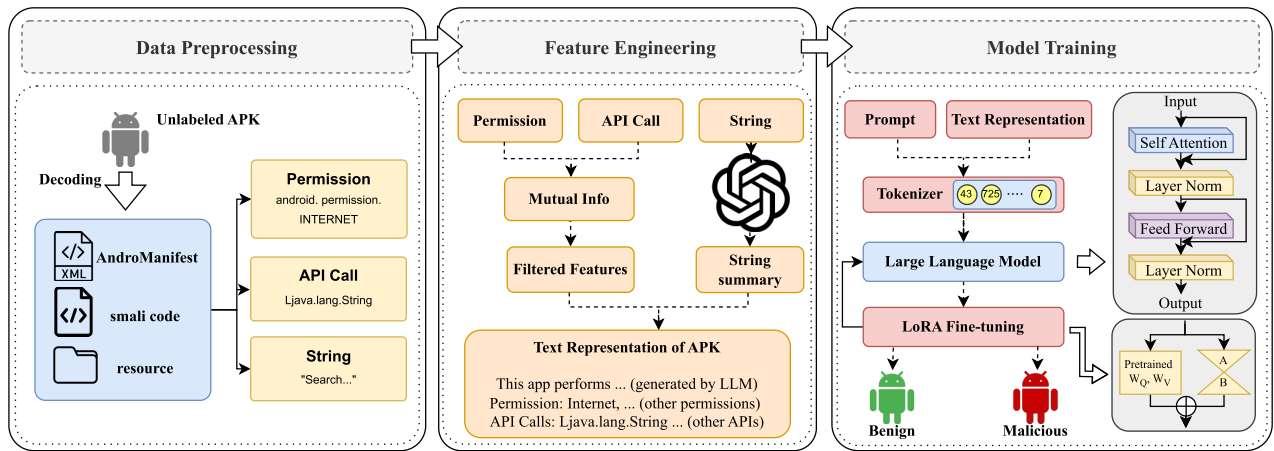


FIGURE 2. Proposed architecture of LLM-MalDetect.

tation often leads to inadequate modeling of malware behavior, particularly against sophisticated, obfuscated threats. Secondly, classification-based methods [42], [43] are often applied in isolation or rely on simplistic aggregation strategies. Although ensemble and fusion-based approaches (e.g., DroidFusion) have advanced the field by combining multiple classifiers, they often struggle with model generalization and task-specific adaptation. In many cases, the classifiers are decoupled from the nuanced inter-feature correlations, which results in suboptimal performance when handling the complexity of modern Android malware [5], [38], [39]. Collectively, these methods tend to overlook the deep semantic relationships among features, leading to models that are not fully optimized for the dynamic nature of malware. Furthermore, LLM-based detection approaches [14], [15], [16] typically use pre-trained embeddings for data augmentation or fine-tuning but often suffer from limited feature diversity and tokenization constraints, which restrict their ability to fully capture the semantic richness of APK components. Additionally, these approaches usually focus on text-based threats and overlook the potential of string-based features that could significantly enhance malware representation.

DroidFusion [21] employs heterogeneous ensemble learning to fuse multi-machine learning methodologies, reducing overfitting risks through variance minimization across base learners. While this improves robustness, its reliance on bagging-style aggregation limits nuanced semantic analysis of APK feature interdependencies. Similarly, Random Forest [8] optimizes decision tree ensembles via randomized feature subspaces, achieving efficiency but inheriting tree-based models' inability to model non-axis-parallel decision boundaries in high-dimensional malware feature spaces. In contrast to these static ensemble approaches, AMD-DLmodel [26] and image-based methods [42] employ CNNs to capture local sequential patterns through translation-invariant filters. However, their inductive bias toward spatial locality constrains recognition of globally distributed feature interactions critical for advanced obfuscation detection. The

DroidMDetection [24] framework implements dual architectural analysis but suffers from attention diffusion when correlating structural (e.g., control flow) and behavioral (e.g., API call) feature modalities. Our LLM-MalDetect addresses these limitations through transformer-based self-attention mechanisms that dynamically quantify cross-feature affinities, enabling explicit modeling of both local and global semantic relationships while maintaining interpretability through gradient-based attribution.

To address these limitations, our proposed framework improves Android malware detection in several key ways. First, in the realm of feature extraction, our approach introduces a novel method for capturing string-based features from APKs, thereby enriching the feature set beyond traditional permissions and API calls. Second, for classification, we integrate fine-tuned LLMs into the detection pipeline using prompt engineering and the LoRA fine-tuning method to better interpret the complex relationships among features. This integration bridges the gap between static feature extraction and sophisticated classification, enabling more adaptive and context-aware decision-making. LLM-MalDetect overcomes the limitations of traditional classifiers and the shallow fusion of multi-source features. This leads to improved detection accuracy, better generalization to evolving malware, and reduced computational overhead during inference. Fig. 1 illustrates the generic Android malware detection framework which includes three periods: preprocessing, feature engineering, and model training. In contrast, our approach processes string-based features and leverages fine-tuned large models for improved interpretation and classification. We outline the features and methods used in some of the representative researches in Table 1.

III. PROPOSED METHODOLOGY

In this section, we introduce LLM-MalDetect, an LLM-based framework developed to improve Android malware detection by leveraging the feature identification and analysis capabilities of LLMs. As shown in Fig. 2, our approach

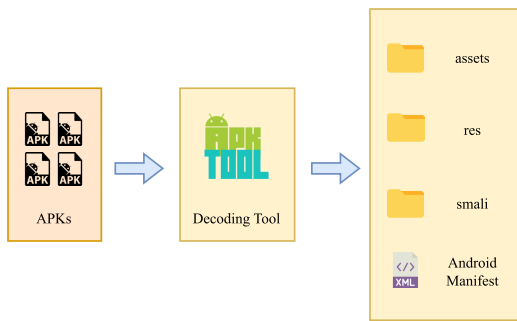


FIGURE 3. The procession of decoding APKs.

begins with APKs and proceeds through three main stages: data preprocessing, feature engineering, and model training. This framework is designed to significantly improve both the accuracy and efficiency of malware detection on Android platforms.

- **Stage-1 Data Preprocessing:** APKTool [44] is used to decompile APKs, extracting the Android Manifest, Smali code, and resource files. While the manifest and code are commonly analyzed [20], [22], our work uniquely focuses on resource files for feature extraction.
- **Stage-2 Feature Engineering:** Features including permissions, API calls, and strings are extracted from decoding files. Mutual information [34] filters permissions and API calls, while the LLM glm-4-flash is used to process strings. These are combined into a unified textual representation of the APK.
- **Stage-3 Model Training:** We employ prompt engineering to elicit outputs from LLM. The prompt and textual representation is tokenized for LLM input. The model is fine-tuned on labeled data using LoRA [19], enabling efficient and accurate malware classification.

In the proposed framework, the LLM is used to process strings and functions as the fundamental model for Android malware detection, which is accomplished through LoRA fine-tuning. Our innovations focus on leveraging strings for Android malware detection and utilizing the LLM to generate results. The following sections detail each component of the LLM-MalDetect framework.

A. DATA PREPROCESSING

1) DECODING

To ensure file integrity, we validate APK files using checksum verification and remove corrupted ones. The process of decoding APKs is shown in Fig. 3. Using APKTool [44], we decompile the unlabeled APKs, analyze the resulting files, and categorize them into distinct groups based on their characteristics and functionalities.

2) EXTRACTED FILES

We analyzed the decompiled files, among which the AndroidManifest is widely utilized in most research efforts. The Smali code, which represents the functions invoked

during software execution, is also of significant importance. We observed that resource files house critical operational assets during Android runtime, motivating our systematic extraction of these understudied features.

- **AndroidManifest:** Located in the root directory, the AndroidManifest.xml file provides crucial information about the application, including its components (e.g., activities, services, required permissions, and essential hardware features) [45].
- **Smali code:** Smali code represents the bytecode format for Android applications. Although Android software is primarily written in Java, it is compiled into a DEX file for execution on the Android Virtual Machine. Smali code is produced by decompiling these DEX files, providing an accessible format for analysis [28].
- **Resource:** This directory stores resources such as text files and images, which can be accessed by the application at runtime. Files in this folder remain in their original format rather than being converted to binary and are included in the APK along with the source code.

Among these files, AndroManifest and Smali code are predominantly utilized in researches [15], [34], while the resources are frequently disregarded. We concentrate on analyzing the features of resources. After decoding APKs, we need to read these files line by line and apply regular expressions to identify contents. For example, we extract API name “Ljava/lang/String;” from “.method private Method(Ljava/lang/String;)Z”.

B. FEATURE ENGINEERING

Feature engineering encompasses both feature extraction and feature selection. During the feature extraction phase, necessary features are obtained from the decoded files. In the feature selection phase, these features are filtered to optimize computational efficiency. The process of feature extraction and selection is illustrated in Fig. 4. The main innovation in this process lies in the extraction and selection of strings.

1) FEATURE EXTRACTION

The purpose of this step is to extract features from the preprocessed files. The extracted features include permissions, API calls, and strings, with strings being our innovation point. The meanings of these features are as follows.

Permissions are specified in the AndroidManifest.xml file and are required to access protected areas of the system or other applications, such as contacts and Short Message Service (SMS). According to Google’s developer guides [45], there are 318 different permissions in the Android operating system, including those necessary for an application to install APKs. Permissions in the Android system can be categorized into three types: regular permissions, dangerous permissions, and restricted permissions. Their distribution is illustrated in Fig. 5. Among these, dangerous permissions require user approval before use. Permissions effectively reflect the

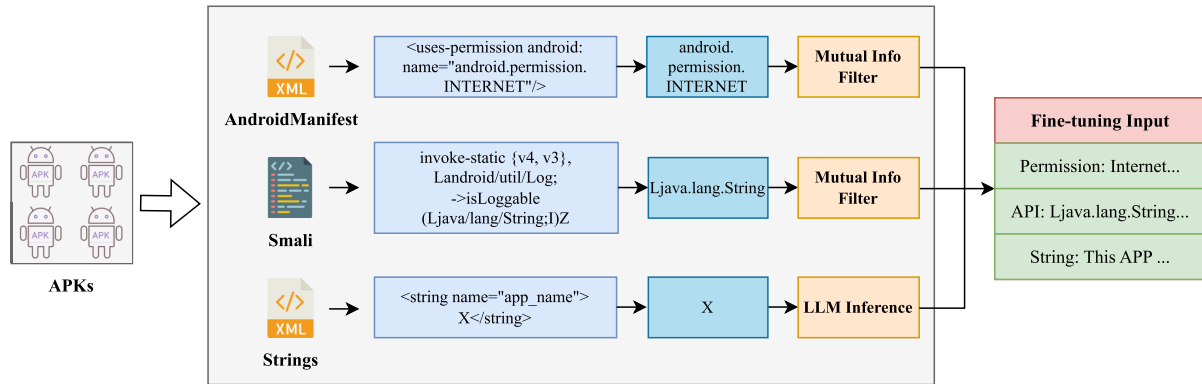


FIGURE 4. The process of feature extraction and selection.

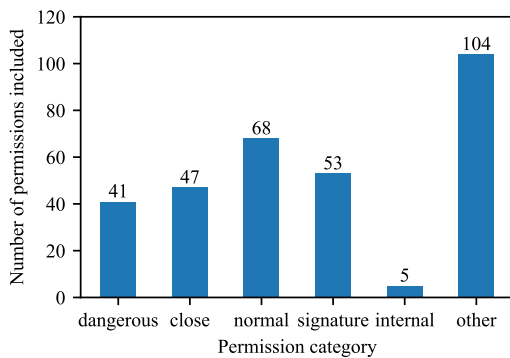


FIGURE 5. The number of permissions in different categories.

software’s intent and thus serve as a valuable reference for malware detection.

The extraction of permission information is commonly used to identify Android malware [20], [23]. For instance, it is typical for photography applications to request permissions related to the camera and photo album. However, if such an application requests access to contacts, it is likely to be malware. Besides, traditional malware detection methods often allow models to recognize malware through a combination of features, but they overlook the underlying information that these features may conceal.

API calls are found within smali code, which is written in assembly language. API calls refer to functions in external libraries that developers use when developing [28]. Such libraries facilitate the addition of functionality and features to applications. We read the smali code and filter lines to identify function declarations. For example, to determine whether an APK utilizes the “android.telephony.SmsManager” function, we extract all lines containing “invoke” instructions and search for the string “android/telephony/SmsManager”.

Similarly to permission features, API calls can be represented as vectors. A value of zero indicates that the corresponding feature is absent, while a non-zero value signifies that this feature is present in the APK file.

Strings used in our method are derived from two main sources: the “strings.xml” file and the smali code. The strings.xml file contains the application’s name, a description of itself, and other interface-related information. Strings in the smali code define additional strings utilized during the program’s runtime.

These string features improve interpret ability in two key ways:

- 1) **Behavioral Transparency:** Malware often exhibits discrepancies between declared functionality (e.g., “weather forecast app” in descriptions) and embedded technical indicators (e.g., “sms_premium_access” in smali strings). Such mismatches provide auditors with direct linguistic evidence of deceptive practices.
- 2) **Permission-Context Validation:** String semantics enable cross-checking of requested permissions against stated purposes (e.g., a calculator app containing “location_tracking_enabled” strings). This creates explainable decision pathways where model predictions align with human-interpretable feature relationships.

By surfacing linguistically anomalous patterns and functional contradictions, string features bridge the gap between statistical detection and forensic analysis.

2) MUTUAL INFORMATION TO FILTER PERMISSIONS AND APIs

To enhance feature engineering, extracted features are systematically selected. Permissions and APIs, represented through one-hot encoding, are filtered by feature importance, while textual features, such as strings, are condensed using an LLM’s summarization function. This approach ensures the retention of the most informative features, optimizing model performance [21].

There are at least 318 permission features and 5,000 common API calls, each serving distinct functions within the Android system. Due to their large quantity, it is challenging to include all features in the classification. To reduce computational load during model fine-tuning, mutual information is employed to evaluate the correlation

between features and labels. We then sort the features by relevance to select those that have a greater impact on the classification results.

Mutual information is used to measure the dependency between two variables. Based on the dependency, we can remove less important features. The mutual information of event X and event Y is defined as [34]:

$$I(X; Y) = H(X, Y) - H(X|Y) - H(Y|X), \quad (1)$$

where X is the feature matrix, Y is the label matrix, $H(X, Y)$ is the joint entropy of X and Y , $H(X|Y)$ and $H(Y|X)$ are conditional entropies. Joint entropy and conditional entropy can be calculated by [34]:

$$H(X, Y) = - \sum p(x, y) \log p(x, y), \quad (2)$$

$$H(X|Y) = - \sum_{x \in X} p(x) \sum_{y \in Y} p(y|x) \log p(y|x), \quad (3)$$

$$H(Y|X) = - \sum_{y \in Y} p(y) \sum_{x \in X} p(x|y) \log p(x|y), \quad (4)$$

where $p(x, y)$ is a different combination probability of x and y , $p(x|y)$ and $p(y|x)$ are conditional probability. The mutual information between a feature and a label indicates the extent of the feature's impact on the judgment result. A higher mutual information value signifies a greater influence.

Algorithm 1 Mutual Information-Based Feature Selection

Input: Features $x[m, n]$, Labels $y[n]$

Result: Filtered Features $x[m', n]$

```

for  $i = 1$  to  $m$  do
     $nx = 0, ny = 0, nxy = 0;$ 
    for  $j = 1$  to  $n$  do
        if  $x[i, j] == 1$  then
             $nx++;$ 
        end
        if  $y[j] == 1$  then
             $ny++;$ 
        end
        if  $x[i, j] == 1$  and  $y[j] == 1$  then
             $nxy++;$ 
        end
    end
     $p(x, y) = nxy / n;$ 
     $p(x|y) = nx / ny;$ 
     $HXY = - p(x, y) \log p(x, y);$ 
     $HX = - nx * HXY / n;$ 
     $HY = - ny * HXY / n;$ 
     $I = HXY - HX - HY;$ 
end
return  $x[m', n];$ 

```

Based on mutual information, we propose the feature selection algorithm described in Algorithm 1 to filter permissions and API calls. For each feature, we calculate the mutual information relative to the label and sort the features

TABLE 2. Mutual information between features and label.

Name	Mutual Information
transact	0.1892
onServiceConnected	0.1784
bindService	0.1774
attachInterface	0.1774
ServiceConnection	0.1761
Binder	0.1675
SEND_SMS	0.0700
getCanonicalName	0.1317
getMethods	0.1196
cast	0.1169
URLDecoder	0.1086
Signature	0.1083
SmsManager	0.0938
READ_PHONE_STATE	0.0928
getBinder	0.0883

based on their mutual information values. Subsequently, features with higher mutual information are retained.

We organize the permissions and function calls, calculate the mutual information between each feature and the label, and rank them by their values. Table 2 presents a portion of the results. Function calls exhibit higher mutual information than permission features, highlighting their more substantial role in the detection process. This finding suggests that malware generation often involves invoking specific APIs to gather information. We selected 300 features for Android malware detection based on two primary considerations:

- **Different importance of features:** Various malware features exhibit varying degrees of influence on detection outcomes. To optimize computational efficiency without substantially compromising detection performance, we intentionally adopted a reduced feature set.
- **Large language model constraints:** LLMs typically enforce a maximum token limit of 4096. Given that each permission declaration or API name comprises multiple tokens, selecting 300 features ensures compliance with this architectural constraint while maintaining sufficient discriminative power.

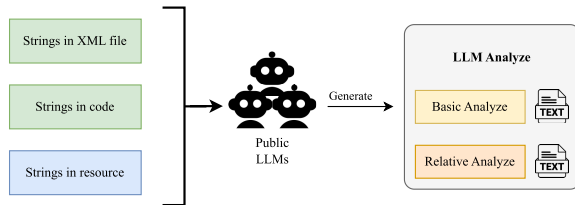
Permissions within the same group share similar access privileges. For example, the CALL_LOG group (e.g., READ_CALL_LOG) exposes communication metadata, enabling social engineering or surveillance; the LOCATION group (e.g., ACCESS_FINE_LOCATION) risks geolocation tracking for invasive monitoring; and the STORAGE group (e.g., READ_EXTERNAL_STORAGE) could exfiltrate sensitive files or deploy ransomware. The permission groups, along with their respective permissions, are organized and presented in Table 3. The first 8 permission groups are classified by Google [45], and the other is grouped by us. This grouping approach reduces the total number of features while retaining all necessary information.

3) LLM TO PROCESS STRINGS

The string processing workflow with LLM [46] is illustrated in Fig. 6, which consists of three key phases: multi-source

TABLE 3. The group of selected permissions and their corresponding behaviors.

Group Name	Related
CALL LOG	Telephony features
CONTACTS	Contacts and their profiles
PHONE	Telephony features
CALENDAR	User's calendar
LOCATION	Access the device location
STORAGE	Shared external storage
CAMERA	Access camera or capture images/video
MICROPHONE	Access microphone's audio
ACCOUNT	Manage accounts
PACKAGE	Manage packages
SETTINGS	Read or change system settings
CONNECTION	Access to network or other devices
DEVICE STATE	Get physical states of device
OTHER APP	Change other applications' setting
DICTIONARY	Change users' dictionary
BIND	Bind to system control
DISPLAY	Show content and notifications
Locking	Decide when to lock

**FIGURE 6.** Utilization of public LLM services to process strings.

input integration, hierarchical analysis, and result synthesis. The input strings are collected from three distinct sources in Android applications:

- Strings in XML file containing UI labels and metadata, which often reveal application components and permissions requirements.
- Strings in code reflecting functional logic through API calls and system interactions.
- Strings in resource including localized text and asset descriptions that complement functional understanding.

We employ the glm-4-flash model [46] through its public API to process these heterogeneous strings, considering its cost-effectiveness and adequate performance for text comprehension tasks. The analysis adopts a two-tiered approach:

- **Tier-1:** Basic analysis identifies key entities and categorizes functional domains, creating a baseline profile of the application's capabilities.
- **Tier-2:** Associative analysis evaluates contextual relationships, assessing combinatorial risks and behavioral inconsistencies by analyzing interactions between identified functions.

The workflow culminates in merging both analytical perspectives through attention-based weighting in the model's final layers, where basic features provide structural anchors

while associative insights contribute risk-aware interpretations. This dual-layer processing enables comprehensive feature representation while maintaining human-interpretable traces through the analysis chain.

4) TEXT REPRESENTATION OF APK

After filtering permissions and APIs using mutual information and analyzing strings with an LLM, the results are integrated to form a structured textual representation of the APK. This representation begins with the LLM-generated summary of the application's core functionality and metadata, followed by a sequential list of critical permissions and API calls. The unified text format ensures compatibility with LLM input requirements, serving as the foundation for subsequent model training and classification tasks.

C. MODEL TRAINING

Model training consists of two primary steps: constructing prompts and fine-tuning LLM. The first step involves representing features as text, and the second step employs the LoRA method to fine-tune the language model. Through model training, we obtained a classification model for Android malware detection. The classification method we use is LLM fine-tuning, which is a new Android malware detection method.

1) PROMPT CONSTRUCTIONS

The purpose of this step is to construct the input data for fine-tuning LLM, which requires constructing a prompt and then combining it with the text representation. The detailed operations are as follows.

The theory of prompt engineering is employed to generate prompts that are appropriate for our task. The purpose of this step is to allow the LLM to analyze the outcome before fine-tuning. If the LLM is directly required to determine whether an APK is malicious software, it will refuse to adopt a positive or negative answer due to the application of the mechanism of hallucination elimination. To generate good prompts, we analyze the principles of text generation. Based on sequence to sequence model [47], we design the target of this step as:

$$P(Y|X, \mathcal{P}) = \prod_{t=1}^M P(y_t|Y_{<t}, X, \mathcal{P}) \in \mathcal{L}, \quad (5)$$

where X is training data, Y is content generated by LLM, $\mathcal{P}$ is the prompt words that we design, and $\mathcal{L}$ is the classification result. Our prompt is to suppose a situation of a game that lets LLM guess when restricting outputs. As illustrated in Fig. 7, the initial response to the prompt is that it is impossible to make a decision. However, once the appropriate prompt is constructed, the model begins to attempt making a guess.

We propose Algorithm 2 based on (5), which is used to generate a good prompt, and combines the prompt with the text representation to form the input for the LLM. This algorithm consists of three steps. Firstly, prompts are

Prompt: Determine whether Android Package is malicious based on permissions and functions:
android.os.Binder, Ljava.lang.Class.cast, ..., ACCESS_WIFI_STATE

Response: This list of items appears to be a combination of Android API calls and permissions. It is **not possible to determine** if this is malware or benign based on this information alone. Some of these items are required for normal Android system functionality, while others may be used by malware to gain unauthorized access or perform malicious actions. Without additional context, it is impossible to make an accurate determination.

Prompt: Let's play a game. I have sorted out some texts and their corresponding labels. I show you the text sequence and you guess the corresponding label. The only answers are "malicious" and "benign":
android.os.Binder, Ljava.lang.Class.cast, ..., ACCESS_WIFI_STATE

Response: Based on the given text sequence, I would guess the label to be **"malicious"**. The sequence contains several dangerous permissions and actions, such as "READ_PHONE_STATE", ..., "CHANGE_COMPONENT_ENABLED_STATE". These actions and permissions are often used by malware to gain unauthorized access to sensitive information or to persistently run on a device even after a reboot. The sequence also includes "INTERNET" and "WRITE_GSERVICES", which are also potentially malicious.

FIGURE 7. Difference between the responses of direct prompt and optimized prompt.

generated to provide LLM with a starting point. Secondly, the vectors and texts are transferred to the natural language. Finally, these data are added to the input, thus completing the training data. Here, two significant challenges must be addressed when transferring features to texts. Firstly, the names of permissions or API calls are typically presented in a specific coding style (camel-case or snake-case) that is challenging to interpret. Secondly, the strings are often too lengthy to align with the context of the model. To address these issues, we implement a two-step process. Firstly, we replace special symbols with spaces, and then we segment the resulting continuous string into individual words. Additionally, we employ the LLM to evaluate software functionality and compliance based on these string features, using them as input for fine-tuning.

Permissions and API call labels in an APK file are binary, marked as "1" if used and "0" if not. The value "1" indicates that the APK utilizes the specified permission or API call, whereas the value "0" signifies that it does not. Three strategies can be selected when transferring features to the LLM's inputs.

- **Positive:** convert permissions and API calls with label "1" to texts, ignore other features;
- **Negative:** include features whose label is "0", and exclude labeled "1" features;
- **Full:** use all features to attach their labels whether it is "1" or "0".

We use the above three different settings for comparison purposes and execute related experiments.

2) TOKENIZER OF INPUT TEXT

To transform input text into a vector format interpretable by the LLM, we employ a tokenizer-based process shown in Fig. 8. It comprises three steps:

Algorithm 2 Feature Transfer Method for LLM Inputs

Input: vector features $x[m', n]$, text features $t[n]$,
Prompt $\mathcal{P}$, number of samples n , size of
embedding m'

Result: LLM input $s[n]$

Initial $s[n]$, prompt;

for $i = 1$ to n **do**

$s[i] = \text{prompt};$

for $j = 1$ to m' **do**

if $x[i, j] == 1$ **then**

$s[i] += \text{feature name};$

end

end

if $\text{len}(t[i]) > 1024$ **then**

$t[i] = \text{Summary}(t[i]);$

end

$s[i] += t[i];$

end

return $s[n];$

- **Vocabulary-Based Tokenization:** The text representation of the APK is tokenized using the pre-trained vocabulary, splitting the input into subword units aligned with the model's pretraining schema.
- **Token-to-ID Mapping:** Each token is mapped to a unique numerical identifier (ID), converting the text into a sequence of discrete integers for numerical processing.
- **Embedding Conversion:** The token IDs are passed through the LLM's embedding layer, transforming them into dense vector representations. This produces a continuous, high-dimensional tensor compatible with the architecture of the model.

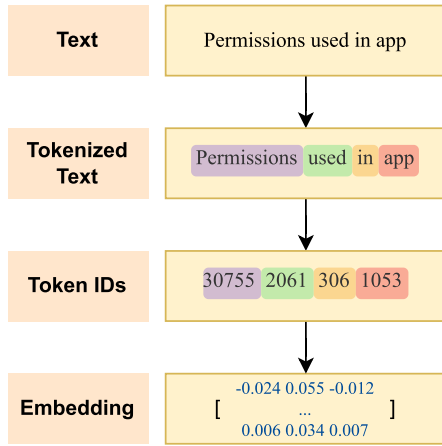


FIGURE 8. The flow of tokenizer.

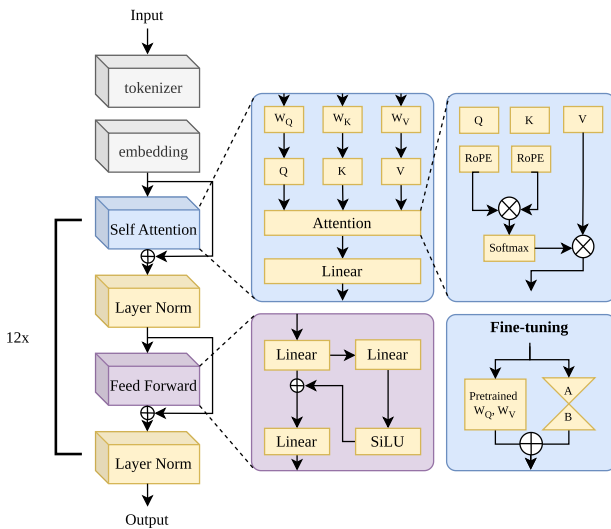


FIGURE 9. Algorithm flow of fine-tuning model.

3) ARCHITECTURE OF LARGE LANGUAGE MODEL

After completing feature extraction and selection, we need to utilize these features to construct fine-tuning prompts and subsequently fine-tune the LLM for Android malware detection. At this stage, we have acquired the dialogue data for the LLM. The next phase is to fine-tune the model and achieve the goal of Android malware detection.

We use the Mistral model [18] with 7B parameters for Android malware detection. It was released in Oct. 2023 and achieved the best performance at that time. Furthermore, a 7B model does not require too many GPU resources to run, making it a popular choice in the industry. This section provides a detailed overview of the model architecture and training methods, as illustrated in Fig. 9.

LLM is the model based on the decoder of the transformer architecture. In our framework, we implement Mistral 7B [18], a representative LLM that predicts the next

token based on the former tokens. The training target of our prediction task is to use features to predict whether the APK is malicious. Inside the decoder module, there are two sub-modules, the self-attention module and the feed-forward module. Self-attention and feed-forward are calculated by [48]:

$$\text{Attention}(Q, K, V) = V \text{softmax} \frac{K^T Q + M}{\sqrt{d}}, \quad (6)$$

$$\text{FeedForward}(x) = W_2 \text{GeLU}(W_1 x + b_1) + b_2, \quad (7)$$

where Q, K, V are trainable parameters which are calculated by input embeddings, M is the mask matrix, d is dimension of word embeddings. W_1, W_2 are the weight matrices of the science department, b_1, b_2 are learnable weight vectors, and $\text{SiLU}(\cdot)$ is the nonlinear activation function. LayerNorm applies after self-attention and feed-forward layers stabilize gradients by maintaining variance within a controlled range across network depths. It is defined as [49]:

$$\text{LayerNorm}(x) = \frac{g}{\sigma} \otimes (x - \mu) + b, \quad (8)$$

where g and b are learnable weights, $\otimes$ is element-wise multiplication, and μ and σ are the mean and variance of x . This ensures stable activations, improving training efficiency.

4) LORA FINE-TUNING FOR MODEL TRAINING

We use LoRA as the method to make LLM suitable for Android malware detection. We construct the input of LLM by translating features into natural language. Our key viewpoint is to use LLMs to enhance data processing and detect Android malware. LoRA is a method of selecting fine-tuning parameters, which only affects specific parameters during the fine-tuning process. LoRA is based on the idea that the change of fine-tuned LLMs can be expressed by the multiple of two matrices [19]:

$$W_0 + \Delta W = W_0 + BA, \quad (9)$$

where A and B are trainable parameters, W_0 is model's initial parameters, ΔW is the change of model's parameters.

Algorithm 3 is proposed based on the architecture of LLM [48] and LoRA fine-tuning [19], and it is used to fine-tune the LLM to obtain classification results. We divide the training set into different batches and then use the training data to fine-tune the model. The LoRA fine-tuning method is applied to reduce the resources required for model fine-tuning and the time consumed during the fine-tuning process. We optimize the process of hyperparameter selection by comparing the results of multiple rounds of fine-tuning.

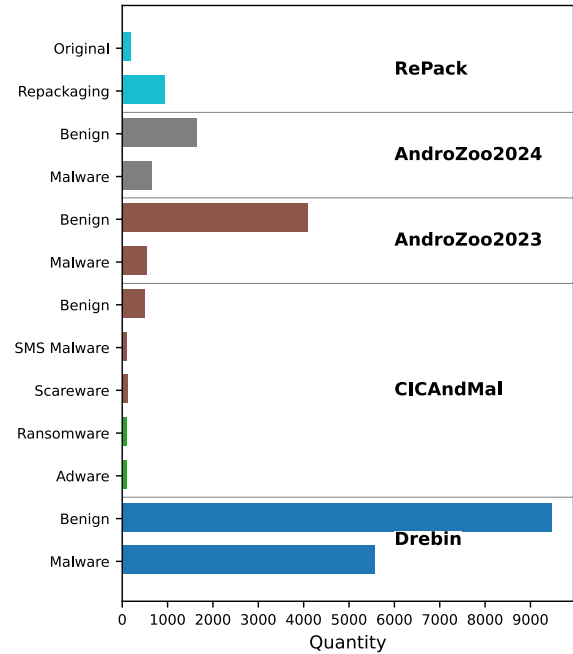
Based on the final output of the LLM, we obtain the detection results, which are classified as either Malicious or Benign. Supposing the number of samples to be processed is n , and the number of parameters in the large model is P . The time complexity of using the large model for prediction is $O(nP)$.

TABLE 4. Datasets used in our experiment.

Dataset	Drebin	CICAndMal	AndroZoo-2023	AndroZoo-2024	RePack
Number of Samples	15036	926	4627	2277	18073
Number of Malware	5560	426	545	645	12603
Collected Time	2014	2019	2023.01-2023.06	2024.01-2024.06	2019
Labeled by	Arp et al.	MahdaviFar et al.	VirusTotal	VirusTotal	Li Li et al.

Algorithm 3 Fine-Tuning LLM on Obtained Features**Input:** training data $\mathcal{D}=(X, Y)$, tokenizer model $\mathcal{T}$ **Result:** fine-tuned LLM's parameters W $\mathcal{D} = \mathcal{T}(\mathcal{D})$;divide $\mathcal{D}$ into different batches;**for** $batch$ **in** $batches$ **do****for** x, y **in** $batch$ **do** $x = \text{embedding}(x), y = \text{embedding}(y)$;**for** $block$ **in** $blocks$ **do** $\hat{y} = x / (\sqrt{(1/len) \sum_{i=1}^{len} x_i^2})$; $Q, K, V = \text{RoPE}(W\hat{y})$; $\hat{y} = \text{Attention}(Q, K, V) + x$; $\hat{y} = \text{FeedForward}(\hat{y})$;**end** $loss = y - \hat{y}$;Backforward loss to W_q and W_v ;Change B and A by ΔW ;**end****end**Save LoRA params B and A ; W update using eq. (8);

Evaluate model on test set;

return W ;**FIGURE 10.** Dataset category distribution.**IV. EXPERIMENTS**

In this section, we validate the proposed framework by comparing the results of Android malware detection with those obtained using alternative methods. This section begins by outlining the datasets, environment settings, and evaluation metrics followed by the execution of the experiments.

A. DATASET

The Android malware detection dataset is derived from Drebin [20], CICAndMal [9], AndroZoo [50], and RePack [51]. The distributions of datasets are shown in Fig. 10. Drebin is a well-established Android malware dataset comprising 15,036 Android applications, of which 5,560 are classified as malware and 9,476 as benign software. CICAndMal consists of 926 Android application samples, including 545 malware instances and 400 benign applications. AndroZoo is a continuously expanding repository of Android applications collected from multiple sources, currently containing 24,995,217 applications. A portion of these applications has been labeled as either malware or benign, whereas others remain in the dataset without

explicit annotation. Repack is a dataset of repackaged software, extracted from AndroZoo based on similarity calculations.

Building on the three initial datasets, we derive four datasets for comparative experiments, as detailed in Table 4. The processing steps are as follows:

- The Drebin dataset is fully utilized for Android malware detection, with permissions and API calls serving as the basis for LLM analysis due to the lack of original files.
- The CICAndMal dataset is used for Android malware detection.
- Given the extensive size of the AndroZoo dataset, a subset is selected. We collected 4,627 samples from third-party sources (e.g., F-Droid, AppChina) between January and June 2023, comprising 4,082 benign applications and 545 malware samples. The latter half of the year's data is not included due to increased volume and prolonged feature extraction time.
- From AndroZoo, we also extract 2,277 samples from January to June 2024, including 1,632 benign applications and 645 malware samples. Data from the latter half of 2024 remains unannotated and is excluded.
- We selected a subset of RePack apps for experiments based on their sha256 values, which include 188 benign and 938 repackaged apps.

TABLE 5. Environment settings.

Environment	Setting
Processor	Intel Xeon Gold 6240
RAM	256 GB
GPU	Nvidia GeForce RTX 2080 Ti
Python	3.12
Operation System	Ubuntu 20.04

TABLE 6. Configuration of supervised fine-tuning.

Items	Configuration Values
Epochs	3
Batch size	8
Learning Rate	5e-5
Learning Rate Scheduler	cos
Compute Type	fp16
Training Method	Supervised Fine-tuning

B. ENVIRONMENT SETTING

We implement LLM-MalDetect in Python, using Apk-tool [44] to extract features from APKs. The Mistral 7B model is fine-tuned with the LLaMA Factory [52] framework. Configurations of experiments and fine-tuning are detailed in Table 5. We divide 80% of the data for training and 20% for testing. Table 6 summarizes the fine-tuning parameters used in our experiments, including the number of epochs, batch size, learning rate, scheduler, compute type, and training method, ensuring consistent and reproducible experimental settings.

The primary concern for real-world deployment lies in computational cost. On the client side, ML remains the predominant detection method due to its proven performance and rapid computational capabilities. On the server side, Android malware detection leveraging LLMs presents a viable alternative. Crucially, our approach eliminates the need for deploying massive LLMs (e.g. SecGPT [53]), instead utilizing compact large models with reduced parameter sizes. This enables deployment with GPU memory consumption limited to approximately 8 GB, thereby satisfying practical computational budgets. To ensure dataset diversity, our experiments employ Android application data collected over the past five years, effectively capturing evolving malware trends. Furthermore, we conduct comparative analyses between malicious software, equipping our system to combat emerging fraud patterns that exploit malware.

C. EVALUATION METRICS

In our experiment, we employ the most commonly utilized metrics in classification problems to evaluate our results, including accuracy, precision, recall, and F1 score. Among these, accuracy (Acc) is the primary indicator, precision (P) and recall (R) represent the reduction of false positives and vulnerability reduction, respectively, and F1 is a comprehensive

indicator. These metrics are calculated as follows:

$$\text{Acc} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}, \quad (10)$$

$$P = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad (11)$$

$$R = \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad (12)$$

$$F_1 = \frac{2PR}{P + R}, \quad (13)$$

where false negative (FN) is the number of incorrectly predicted benign software, true positive (TP) is the number of correctly predicted malware, true negative (TN) is the number of correctly predicted benign software, and false positive (FP) is the number of incorrectly predicted malware. Among these metrics, accuracy and F1-score can validate the overall performance of detection results. Notably, the F1-score is less affected by class imbalance, offering more robust reliability than accuracy. In addition to conventional metrics, TPR (equivalent to Recall) and FPR (defined as $\text{FP}/(\text{FP} + \text{TN})$) are critical indicators. Specifically, TPR reflects the capability to comprehensively identify malware samples, while FPR represents the false alarm rate. In purpose of “rather over-reporting than missing threats”, TPR acts as a more crucial role than FPR in malware detection tasks, as excessive false negatives could lead to catastrophic security breaches.

D. DETECTION RESULTS AND ANALYSIS

To evaluate the detection capabilities of LLM-MalDetect, we compare it with several Android malware detection methods: DroidFusion [21], Random Forest [8], Image-based [42], DroidMDetection [34], and AMDDLmodel [26]. DroidFusion combines multiple machine learning classifiers to improve performance. Random Forest uses random forests to aggregate predictions from decision trees. The Image-based method transforms binary code into images for malware detection using CNNs. DroidMDetection integrates LSTM for sequential and CNN for local features, whereas AMDDLmodel employs deep learning to learn high-level features. Random Forest and AMDDLmodel are implemented using publicly available source code, whereas the others are re-implemented.

As shown in Table 7, LLM-MalDetect demonstrates significant superiority on the Drebin dataset, achieving an accuracy of 98.97% and an F1 score of 97.79%, indicating robust detection capabilities for classic malware patterns. Its recall of 98.33% suggests minimal false negatives, making it suitable for high-security scenarios. In comparison, DroidFusion (F1 96.50%) and DroidMDetection (F1 96.84%) show competitive performance but fall short in recall and accuracy. Random Forest (F1 96.67%) achieves high recall but lags in precision, potentially leading to higher false positives.

The results on CICAndMal dataset are presented in Table 8. LLM-MalDetect maintains its lead with an accuracy of 96.61% and an F1 score of 96.51%, showcasing its ability to parse unstructured data like API call sequences.

TABLE 7. Android malware detection results on Drebin.

Model	Accuracy	Precision	Recall	F1 Score
DroidFusion [21]	96.76%	97.55%	95.48%	96.50%
Random Forest [8]	95.30%	97.12%	96.24%	96.67%
DroidMDetection [34]	96.13%	96.51%	97.17%	96.84%
Image-based [42]	93.24%	94.16%	93.22%	93.68%
AMDDLmodel [26]	93.56%	94.23%	95.94%	95.08%
LLM-MalDetect	98.97%	97.26%	98.33%	97.79%

TABLE 8. Android malware detection results on CICAndMal.

Model	Accuracy	Precision	Recall	F1 Score
DroidFusion [21]	93.68%	94.88%	93.59%	94.23%
Random Forest [8]	91.67%	92.93%	88.60%	90.72%
DroidMDetection [34]	95.79%	94.77%	96.16%	95.45%
Image-based [42]	94.57%	95.87%	94.96%	95.41%
AMDDLmodel [26]	91.67%	92.93%	90.80%	91.85%
LLM-MalDetect	96.61%	97.00%	96.03%	96.51%

TABLE 9. Android malware detection results on AndroZoo-2023.

Model	Accuracy	Precision	Recall	F1 Score
DroidFusion [21]	90.76%	89.22%	92.40%	90.78%
Random Forest [8]	91.19%	93.02%	86.94%	91.25%
DroidMDetection [34]	94.57%	92.34%	93.86%	93.09%
Image-based [42]	88.76%	86.97%	90.71%	88.80%
AMDDLmodel [26]	90.21%	91.80%	90.04%	90.91%
LLM-MalDetect	93.68%	94.45%	93.24%	93.84%

TABLE 10. Android malware detection results on AndroZoo-2024.

Model	Accuracy	Precision	Recall	F1 Score
DroidFusion [21]	89.71%	88.05%	89.97%	89.00%
Random Forest [8]	92.39%	93.74%	91.18%	92.44%
DroidMDetection [34]	91.79%	93.77%	90.16%	91.93%
Image-based [42]	93.57%	94.87%	93.96%	94.41%
AMDDLmodel [26]	88.47%	87.63%	89.08%	88.34%
LLM-MalDetect	94.16%	94.00%	95.03%	94.51%

Its precision of 97.00% highlights superior control over false positives in complex scenarios. DroidMDetection (F1 95.45%) performs well but struggles with precision, likely due to limitations in modeling long sequential dependencies. The Image-based method (F1 95.41%) excels in feature extraction but shows lower recall, indicating challenges in capturing dynamic behavioral patterns.

Table 9 and Table 10 show the detection results on AndroZoo dataset in 2023 and 2024. LLM-MalDetect exhibits strong performance on both AndroZoo datasets, achieving an F1 score of 93.84% on AndroZoo-2023 and 94.51% on AndroZoo-2024, though its lead narrows compared to other datasets. The Image-based method (F1 94.41% on AndroZoo-2024) demonstrates competitive performance, suggesting its effectiveness in detecting novel code variations. In contrast, DroidFusion (F1 89.00% on AndroZoo-2024) and AMDDLmodel (F1 90.01% on AndroZoo-2023) show inconsistent performance, highlighting their sensitivity

to data distribution and limited adaptability to emerging threats.

Results show that LLM-MalDetect outperforms the baseline methods. The image-based method performs the worst due to its limited information capture. Although Random Forest offers moderate performance, its speed and low computational demands make it suitable for practical use. In-depth analysis with large models marginally improves performance compared to complex network approaches.

To verify the capability of detecting unseen samples, we trained the model on the AndroZoo-2023 dataset and tested it on the AndroZoo-2024 dataset. Experimental results show an accuracy of 85.35%, precision of 88.54%, recall of 91.42%, and F1-score of 89.96%. Overall, the performance hovers around 90%, demonstrating effective detection even with unseen samples.

We conducted an estimation of the time required for the LLM-MalDetect approach. Lightweight Android malware

TABLE 11. Average detection time for per sample.

Phase	Samples per Batch	Average time (s)
Data preprocessing	18	0.40
Feature selection	1	0.58
Model fine-tuning	2	3.25
Model inference	2	1.26

TABLE 12. Android malware detection results on RePack.

Model	Accuracy	Precision	Recall	F1 Score
Droid Fusion [21]	76.47%	91.87%	81.29%	82.26%
Random Tree [8]	79.08%	95.16%	81.94%	88.06%
DroidMDeteion [34]	84.95%	99.93%	84.99%	91.86%
Image-based [42]	84.56%	99.87%	84.64%	91.63%
AMDDLmodel [26]	82.35%	98.41%	83.22%	90.18%
LLM-MalDetect	90.85%	94.59%	92.92%	93.75%

detection techniques based on ML [54] can complete detections within milliseconds, whereas the LLM-MalDetect method requires several seconds. As shown in Table 11, each sample consumes 3.25 seconds during the model fine-tuning stage and 1.26 seconds during inference. Furthermore, we believe that the trend toward lightweight large models [55] and the shift of computing resources into dedicated assets may define future developments. Consequently, exploring the application of LLMs in practical tasks remains a meaningful pursuit.

Repackaging means cracking the software, making modifications, and then re-releasing it [56]. Repackaging threatens the Android ecosystem by depriving developers of benefits, spreading malware, and increasing market maintainers' workload. Because it is similar to the original software, it is difficult to detect using conventional methods.

To verify the effectiveness of our method in real-world detection, we evaluate repackaging detection using a subset of the RePack dataset containing 188 benign and 938 repackaged apps (filtered by filenames starting with "0"). DroidMDetection and Image-Based methods exhibit near-random guessing due to feature overlap between original and repackaged apps. LLM-MalDetect outperforms all baselines (Table 12), demonstrating its robustness to this evasion tactic.

V. ABLATION-BASED PERFORMANCE ANALYSIS

We conducted an ablation study to assess the contribution of different feature types to the overall performance of our model. By systematically removing specific feature sets, we aim to understand their individual importance and how they impact detection accuracy. This helps in identifying the most crucial features for our system, optimizing its efficiency, and reducing computational overhead.

A. COMPARE OF DIFFERENT LLMs

To compare the results of different models, we expand our evaluation to include additional LLMs (e.g., Llama 8B,

TABLE 13. Results of different LLMs.

LLM	Accuracy	Precision	Recall	F1 Score
Mistral	98.97%	97.26%	98.33%	97.79%
Llama 8b	98.34%	96.66%	98.54%	97.59%
Llama 3b	97.67%	95.61%	97.66%	96.63%
Llama 1b	97.93%	96.47%	97.59%	97.03%

TABLE 14. Results under different features.

Used features	Accuracy
Permissions + API Calls	95.21%
Permissions + Strings	93.78%
API Calls + Strings	95.88%
All features	96.61%

Llama 3B, and Llama 1B) [57] alongside the original models on the Drebin dataset. Table 13 demonstrates consistent trends: Mistral and Llama 8B have comparable parameter sizes and similar performance outcomes. Llama 8B achieves a higher recall rate, while its other metrics are slightly lower than those of Mistral. The smaller models, Llama 3B and Llama 1B, also show comparable results, with a performance decrease of approximately 1.04% compared to the preceding models.

B. INFLUENCE OF DIFFERENT FEATURE TYPES

To verify the effectiveness of the components, we conduct an ablation experiment. Three separate experiments are performed, each removing a different set of features: permissions, API calls, and string features. The results are presented in Table 14. The removal of distinct features impacts the experimental outcomes. Additionally, the following observations can be made:

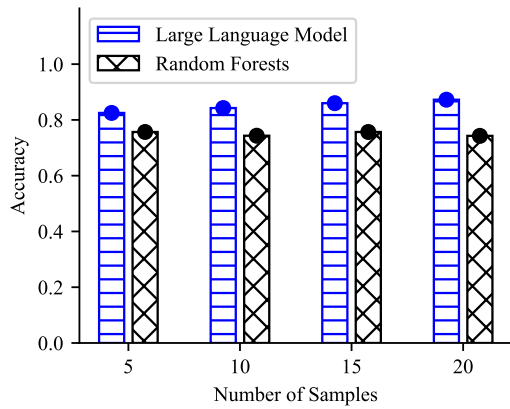
- The influence of API calls on the results is particularly significant, indicating that invoking specific APIs can effectively identify Android malware.
- The removal of string features led to a 1.4% decline in detection accuracy, suggesting that the inclusion of novel string features contributes to improving the accuracy of Android malware detection.
- The findings also reveal that incorporating strings processed by LLM enhances the extraction of information within Android software, thereby enriching the representation of Android malware.

C. IMPACT OF INCLUDED FEATURES

In this experiment, we randomly select 2,000 samples as training data and 1,000 samples as testing data. As described in Section III-C1, we apply three different settings in this experiment: positive, negative, and full. These settings determine which subset of features is used. The various indicators of the experimental results are presented in Table 15.

TABLE 15. Performance of our method under different parts of features.

Setting	Accuracy	Precision	Recall	F1 Score
Positive	88.7%	82.81%	87.12%	84.91%
Negative	87.3%	84.64%	82.7%	83.66%
Full	90.3%	87.24%	87.47%	88.03%

**FIGURE 11.** Performance comparison between LLM-based method and random forests under the data scarcity.

The performance of the model using only positive features surpasses that of the model using only negative features, while utilizing all features yields even greater accuracy. The experiment implemented on malicious features achieves higher accuracy than benign features.

The recall rate for positive samples is equivalent to that for all samples, indicating a strong correlation between malware and malicious features. The accuracy of negative samples is higher than that of positive samples but lower than that of the comprehensive feature set. This discrepancy may arise from the diverse feature categories of benign software, making it challenging to capture all relevant information.

The recall is more critical than precision in Android malware identification because missed detections can lead to economic losses and privacy breaches. Furthermore, the features labeled “1” constitute a small portion of all permissions and API calls. In positive settings, the resources consumed are minimized.

D. DATA SCARCITY IN TRAINING SETS

LLMs have achieved significant success in natural language processing, particularly in few-shot learning tasks [11]. They can also demonstrate exceptional performance even with minimal training data. This capability is particularly critical in the context of emerging cybersecurity threats. When new variants of Android malware emerge, security analysts often face the challenge of detecting malicious patterns from a limited number of samples.

We randomly select APKs as samples from the dataset, with 300 samples designated as the test set and the training sets consisting of 5, 10, 15, and 20 samples, respectively.

Then we employ the LoRA [19] method to fine-tune the model using this data. Our model is compared with the random forest, which is the most commonly used model in Android malware detection literature [21].

The results shown in Fig. 11 indicate that the detection capability of LLMs can be significantly enhanced through fine-tuning with a limited quantity of data. When five samples are randomly selected for training, LLM-MalDetect achieves an accuracy of 82.50%, while the random forest model attains an accuracy of 75%. As the number of fine-tuning samples increases, the performance of LLM-MalDetect exhibits a gradual upward trend, reaching 87.25% with 20 samples. In contrast, the performance of the random forest model remains largely unchanged during this process. This may be attributed to the relative discreteness of the samples, which prevents traditional methods from effectively capturing the potential relationships between features.

VI. CONCLUSION

This work introduces LLM-MalDetect, a robust framework for Android malware detection that uses LLMs to improve accuracy. Our method enhances malware detection by extracting APK features (permissions, API calls, strings) to better differentiate malicious behaviors. The LLM-MalDetect framework utilizes a Mistral backbone fine-tuned with LoRA optimization, augmented by prompt-engineered training data to boost detection accuracy and task adaptability. Experimental results confirm that LLM-MalDetect achieves or even exceeds the accuracy of traditional methods and shows strong potential for addressing this critical challenge. Experimental results demonstrate that LLM-MalDetect achieves an improved performance with an accuracy of 98.97%, precision of 97.26%, recall of 98.33% recall, and F1 score of 97.79%, outperforming established models such as random forest, DroidMDetection, image-based method and AMDDLmodel in Android malware detection. The framework maintains reliable performance even with limited sample sizes, which highlights its effectiveness across diverse scenarios. While LLM-MalDetect achieves high accuracy, its inference time of 1.26 seconds per sample exceeds lightweight ML methods operating in milliseconds, which is a challenge for real-time deployment. Future work may address this through lightweight LLMs or edge hardware acceleration, balancing trade-offs between precision and latency for practical adoption.

Future efforts for advancing LLM-MalDetect will focus on refining the model architecture to capture logical structures more effectively and optimize the model for complex reasoning tasks. Expanding the framework to cover a broader range of sophisticated cybersecurity applications, such as automated vulnerability discovery, real-time threat mitigation, and advanced incident response, is a priority. Additionally, it is meaningful to explore the detection of obfuscated malware (e.g., repackaged or polymorphic Android malware) to enhance the robustness of the method.

against evasion techniques. Finally, collaboration between LLMs and human experts is crucial to establish hybrid defense mechanisms that integrate code-level detection capabilities, thereby enhancing security through the synthesis of human expertise and machine-generated insights. These directions aim to establish LLM-MalDetect as a versatile and reliable tool for addressing modern malware detection and broader cybersecurity challenges.

REFERENCES

- [1] (2024). *Counterpoint Quarterly Smartphones*. [Online]. Available: https://www.counterpointresearch.com/research_portal/counterpoint-quarterly-smartphones-q1-2024
- [2] (2023). *The Mobile Malware Threat Landscape in 2023*. [Online]. Available: <https://securelist.com/mobile-malware-report-2023/11964/>
- [3] P. Yadav, N. Menon, V. Ravi, S. Vishvanathan, and T. D. Pham, "EfficientNet convolutional neural networks-based Android malware detection," *Comput. Secur.*, vol. 115, Apr. 2022, Art. no. 102622.
- [4] L. Meijin, F. Zhiyang, W. Junfeng, C. Luyu, Z. Qi, Y. Tao, W. Yinwei, and G. Jiaxuan, "A systematic overview of Android malware detection," *Appl. Artif. Intell.*, vol. 36, no. 1, Dec. 2022, Art. no. 2007327.
- [5] Z.-U. Rehman, S. N. Khan, K. Muhammad, J. W. Lee, Z. Lv, S. W. Baik, P. A. Shah, K. Awan, and I. Mehmood, "Machine learning-assisted signature and heuristic-based detection of malwares in Android devices," *Comput. Electr. Eng.*, vol. 69, pp. 828–841, Jul. 2018.
- [6] T. Lu and J. Wang, "DOMR: Toward deep open-world malware recognition," *IEEE Trans. Inf. Forensics Security*, vol. 19, pp. 1455–1468, 2024.
- [7] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an, and H. Ye, "Significant permission identification for machine-learning-based Android malware detection," *IEEE Trans. Ind. Informat.*, vol. 14, no. 7, pp. 3216–3225, Jul. 2018.
- [8] A. A. Zaabi and D. Mouheb, "Android malware detection using static features and machine learning," in *Proc. Int. Conf. Commun., Comput., Cybersecurity, Informat. (CCCCI)*, Nov. 2020, pp. 1–5.
- [9] D. S. Keyes, B. Li, G. Kaur, A. H. Lashkari, F. Gagnon, and F. Massicotte, "Entropylyzer: Android malware classification and characterization using entropy analysis of dynamic characteristics," in *Proc. Reconciling Data Anal., Autom., Privacy, Secur., Big Data Challenge*, 2021, pp. 1–12.
- [10] X. Deng, Z. Wang, X. Pei, and K. Xue, "TransMalDE: An effective transformer based hierarchical framework for IoT malware detection," *IEEE Trans. Netw. Sci. Eng.*, vol. 11, no. 1, pp. 140–151, Jan. 2023.
- [11] T. B. Brown et al., "Language models are few-shot learners," in *Proc. 34th Int. Conf. Neural Inf. Process. Syst.*, vol. 33, Jan. 2020, pp. 1–25.
- [12] M. M. Karim, D. H. Van, S. Khan, Q. Qu, and Y. Kholodov, "AI agents meet blockchain: A survey on secure and scalable collaboration for multi-agents," *Future Internet*, vol. 17, no. 2, p. 57, Feb. 2025.
- [13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. North Amer. Chapter Assoc. Comput. Linguistics*, 2019, pp. 4171–4186.
- [14] T. Koide, H. Nakano, and D. Chiba, "ChatPhishDetector: Detecting phishing sites using large language models," *IEEE Access*, vol. 12, pp. 154381–154400, 2024.
- [15] W. Zhao, J. Wu, and Z. Meng, "AppPoet: Large language model based Android malware detection via multi-view prompt engineering," *Expert Syst. Appl.*, vol. 262, Mar. 2025, Art. no. 125546.
- [16] Z. Yu, M. Wen, X. Guo, and H. Jin, "Maltracker: A fine-grained NPM malware tracker copiloted by LLM-enhanced dataset," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Sep. 2024, pp. 1759–1771.
- [17] J. Al-Karaki, M. Al-Zafar Khan, and M. Omar, "Exploring LLMs for malware detection: Review, framework design, and countermeasure approaches," 2024, *arXiv:2409.07587*.
- [18] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. Singh Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. Le Scao, T. Lavril, T. Wang, T. Lacroix, and W. El Sayed, "Mistral 7B," 2023, *arXiv:2310.06825*.
- [19] J. E. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, and W. Chen, "LoRA: Low-rank adaptation of large language models," in *Proc. 10th Int. Conf. Learn. Represent.*, Jan. 2021, pp. 1–18.
- [20] D. Arp, M. Spreitzerbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 23–26.
- [21] S. Y. Yerima and S. Sezer, "DroidFusion: A novel multilevel classifier fusion approach for Android malware detection," *IEEE Trans. Cybern.*, vol. 49, no. 2, pp. 453–466, Feb. 2019.
- [22] A. Mahindru and A. L. Sangal, "Mldroid—Framework for Android malware detection using machine learning techniques," *Neural Comput. Appl.*, vol. 33, no. 10, pp. 5183–5240, 2021.
- [23] H. Chen, Z. Li, Q. Jiang, A. Rasool, and L. Chen, "A hierarchical approach for Android malware detection using authorization-sensitive features," *Electronics*, vol. 10, no. 4, p. 432, Feb. 2021.
- [24] A. T. Kabakus, "DroidMalwareDetector: A novel Android malware detection framework based on convolutional neural network," *Expert Syst. Appl.*, vol. 206, Nov. 2022, Art. no. 117833.
- [25] Y. Wu, M. Li, Q. Zeng, T. Yang, J. Wang, Z. Fang, and L. Cheng, "DroidRL: Feature selection for Android malware detection with reinforcement learning," *Comput. Secur.*, vol. 128, May 2023, Art. no. 103126.
- [26] M. Amir, M. W. Iqbal, M. Nosheen, M. U. Ashraf, A. Shaf, K. A. Almarhabi, A. M. Alghamdi, and A. A. Bahaddad, "AMDDLmodel: Android smartphones malware detection using deep learning model," *PLoS ONE*, vol. 19, no. 1, Jan. 2024, Art. no. e0296722.
- [27] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: Visualization and automatic classification," in *Proc. 8th Int. Symp. Visualizat. Cyber Secur.*, Jul. 2011, pp. 1–7.
- [28] S. Freitas, R. Duggal, and D. H. Chau, "MalNet: A large-scale image database of malicious software," in *Proc. 31st ACM Int. Conf. Inf. Knowl. Manage.*, Oct. 2022, pp. 3948–3952.
- [29] A. Karimi and M. H. Moattar, "Android ransomware detection using reduced opcode sequence and image similarity," in *Proc. 7th Int. Conf. Comput. Knowl. Eng. (ICCKE)*, Oct. 2017, pp. 229–234.
- [30] S. Bhandari, R. Panihar, S. Naval, V. Laxmi, A. Zemmari, and M. S. Gaur, "SWORD: Semantic aWare andOid malwaRe detector," *J. Inf. Secur. Appl.*, vol. 42, pp. 46–56, Oct. 2018.
- [31] S. Wang, Q. Yan, Z. Chen, B. Yang, C. Zhao, and M. Conti, "Detecting Android malware leveraging text semantics of network flows," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 5, pp. 1096–1109, May 2018.
- [32] W. Zhang, Q. Jiang, L. Chen, and C. Li, "Two-stage ELM for phishing Web pages detection using hybrid features," *World Wide Web*, vol. 20, no. 4, pp. 797–813, Jul. 2017.
- [33] F. Ou and J. Xu, "S3Feature: A static sensitive subgraph-based feature for Android malware detection," *Comput. Secur.*, vol. 112, Jan. 2022, Art. no. 102513.
- [34] F. Taher, O. Al Fandi, M. Al Kfairi, H. Al Hamadi, and S. Alrabae, "A proposed artificial intelligence model for android-malware detection," *Informatics*, vol. 10, no. 3, p. 67, Aug. 2023.
- [35] Y. Wang, Q. Mao, H. Zhu, J. Deng, Y. Zhang, J. Ji, H. Li, and Y. Zhang, "Multi-modal 3D object detection in autonomous driving: A survey," *Int. J. Comput. Vis.*, vol. 131, no. 8, pp. 2122–2152, Aug. 2023.
- [36] J.-B. Alayrac et al., "Flamingo: A visual language model for few-shot learning," in *Proc. Adv. Neural Inf. Process. Syst.*, Jan. 2022, pp. 23716–23736.
- [37] Y. Ye, D. Wang, T. Li, D. Ye, and Q. Jiang, "An intelligent PE-malware detection system based on association mining," *J. Comput. Virology*, vol. 4, no. 4, pp. 323–334, Nov. 2008.
- [38] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-sec: Deep learning in Android malware detection," in *Proc. ACM Conf. SIGCOMM*, Aug. 2014, pp. 371–372.
- [39] W. Zhang, H. Ren, Q. Jiang, and K. Zhang, "Exploring feature extraction and ELM in malware detection for Android devices," in *Proc. Adv. Neural Netw. Cham, Switzerland: Springer*, Jan. 2015, pp. 489–498.
- [40] Q. Yu, J. Ren, J. Zhang, S. Liu, Y. Fu, Y. Li, L. Ma, J. Jing, and W. Zhang, "An immunology-inspired network security architecture," *IEEE Wireless Commun.*, vol. 27, no. 5, pp. 168–173, Oct. 2020.
- [41] Y. He, Y. Liu, L. Wu, Z. Yang, K. Ren, and Z. Qin, "MsDroid: Identifying malicious snippets for Android malware detection," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 3, pp. 2025–2039, Mar. 2022.
- [42] A. Aldini and T. Petrelli, "Image-based detection and classification of Android malware through CNN models," in *Proc. 19th Int. Conf. Availability, Rel. Secur.*, Jul. 2024, pp. 1–11.
- [43] P. He, Y. Xia, X. Zhang, and S. Ji, "Efficient query-based attack against ML-based Android malware detection under zero knowledge setting," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2023, pp. 90–104.

- [44] (2024). *Apktool: A Tool for Reverse Engineering Android Apk Files*. [Online]. Available: <https://github.com/iBotPeaches/Apktool>
- [45] (2024). *Android Developer Guides*. [Online]. Available: <https://developer.android.com/guide>
- [46] T. Glm et al., “ChatGLM: A family of large language models from GLM-130B to GLM-4 all tools,” 2024, *arXiv:2406.12793*.
- [47] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” in *Proc. Adv. Neural Inf. Process. Syst.*, Jan. 2014, p. 3104.
- [48] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, vol. 30, Jun. 2017, pp. 5998–6008.
- [49] J. Lei Ba, J. Ryan Kiros, and G. E. Hinton, “Layer normalization,” 2016, *arXiv:1607.06450*.
- [50] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, “AndroZoo: Collecting millions of Android apps for the research community,” in *Proc. IEEE/ACM 13th Work. Conf. Mining Softw. Repositories (MSR)*, May 2016, pp. 468–471.
- [51] L. Li, T. F. Bissyandé, and J. Klein, “Rebooting research on detecting repackaged Android apps: Literature review and benchmark,” *IEEE Trans. Softw. Eng.*, vol. 47, no. 4, pp. 676–693, Apr. 2021.
- [52] Y. Zheng, R. Zhang, J. Zhang, Y. YeYanhan, and Z. Luo, “LlamaFactory: Unified efficient fine-tuning of 100+ language models,” in *Proc. 62nd Annu. Meeting Assoc. Comput. Linguistics*, vol. 3, 2024, pp. 400–410.
- [53] clauditer. (2023). *Secgpt*. [Online]. Available: <https://github.com/Clouditera/secgpt>
- [54] A. Wajahat, J. He, N. Zhu, T. Mahmood, A. Nazir, F. Ullah, S. Qureshi, and S. Dev, “Securing Android IoT devices with GuardDroid transparent and lightweight malware detection,” *Ain Shams Eng. J.*, vol. 15, no. 5, May 2024, Art. no. 102642.
- [55] C. Liu, H. Zhang, K. Zhao, X. Ju, and L. Yang, “LLMEmbed: Rethinking lightweight LLM’s genuine function in text classification,” in *Proc. 62nd Annu. Meeting Assoc. Comput. Linguistics*, vol. 1, 2024, pp. 7994–8004.
- [56] H. Rafiq, N. Aslam, M. Aleem, B. Issac, and R. H. Randhawa, “AndroMalPack: Enhancing the ML-based malware classification by detection and removal of repacked apps for Android systems,” *Sci. Rep.*, vol. 12, no. 1, p. 19534, Nov. 2022.
- [57] A. Grattafiori et al., “The llama 3 herd of models,” 2024, *arXiv:2407.21783*.



RUIRUI FENG received the B.S. degree in optoelectronic information science and engineering from Nankai University, in 2021. He is currently pursuing the M.S. degree in software engineering with Hebei University. His research interests include mobile security, with an emphasis on the application of large language models in cybersecurity, web, data mining, and AI applications.



HUI CHEN received the Master of Science (M.S.) degree in information and communication engineering from Nanjing University of Posts and Telecommunications, in 2014, and the Ph.D. degree from Shenzhen Institutes of Advanced Technology (SIAT), Chinese Academy of Sciences, in 2021. She is currently a Lecturer with the School of Artificial Intelligence, Shenzhen Polytechnic University. Her research interests include data security, machine learning, data mining, and pattern recognition, with a focus on developing innovative solutions to complex computational challenges. She has published over 20 SCI-indexed articles and filed more than ten patents, demonstrating her strong contributions to advancing research and technology in her field.



SHUO WANG received the Ph.D. degree in computer software and theory from Renmin University of China. He is currently an Associate Professor and a Master’s Supervisor with the Key Laboratory of Machine Learning and Computational Intelligence, Hebei University. His research interests include natural language processing, data fusion, and knowledge fusion.



MD MONJURUL KARIM received the B.E. and M.E. degrees in computer science and technology from Northwestern Polytechnical University, Xi’an, China, in 2012 and 2015, respectively, and the Ph.D. degree in computer science and technology from Beijing Institute of Technology, in 2023. He was a Research and Teaching Assistant with the Southern University of Science and Technology, Shenzhen. He is currently a Postdoctoral Research Fellow with Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences. His research interests include generative AI, large language models, blockchain, Web3, multi-tier computing, software-defined networking, and information-centric networking.



QINGSHAN JIANG received the Ph.D. degree in mathematics from Chiba Institute of Technology, Japan, and the Ph.D. degree in computer science from the University of Sherbrooke, Canada. He was a Postdoctoral Fellow with The Fields Institute for Research in Mathematical Sciences, University of Toronto, Canada. He is currently a Professor with Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China. He has published articles more than 200 and his research work has been supported by the National Nature Science Foundation Grants, the National Research Plans, Guangdong Science and Technology Program Funds, Shenzhen Technology Development Foundation Grants, and the National Key Research and Development Program of China. His research interests include data mining, big data analysis, data security, pattern recognition, and AI.

...