ADDIS ABABA UNIVERSITY
ADDIS ABABA INSTITUTE OF TECHNOLOGY
School of Information Technology and Engineering

Investigating Malicious Capabilities of Android Malwares that Utilize Accessibility Services

by

Tekeste Fekadu

Supervised by

Dr. Fitsum Assamnew

A Thesis submitted in partial fulfillment of the requirements of
Master of Science in Cyber Security
(School of Information Technology and Engineering)

February 2025

Investigating Malicious Capabilities of Android Malwares that Utilize Accessibility Services

by

Tekeste Fekadu

Examiners' Committee

Name	Signature	Date
______________________ (**Proposal Advisor**)	____________	____________
______________________ (**Chairman of Department**)	____________	____________
______________________ (**Internal Examiner**)	____________	____________
______________________ (**External Examiner**)	____________	____________

A Thesis Submitted in Partial Fulfillment of the Requirements
For Masters of Science in Cyber Security
Addis Ababa Institute of Technology
School of Information Technology and Engineering
February 2025

Declaration

I, Tekeste Fekadu, declare that this thesis is my original work. All sources of information in this study have been appropriately acknowledged. I further confirm that this thesis has not been submitted either in part or in full for any other requirements to any other learning institution.

Declared By:

. . . .

Student's Name and Signature

Approved By:

. . . .

Advisor's Name and Signature

February 2025

Acknowledgment

First and foremost, I would like to thank the Almighty God for giving me the strength, wisdom, and perseverance to complete this work. Without His guidance and grace, none of this would have been possible.

I want to express my deepest gratitude to my advisor, Dr. Fitsum Assamnew, whose invaluable support, encouragement, and direction were crucial throughout the process.

I would also like to sincerely thank my friend Terefe Feyisa for his valuable assistance and contributions, especially with the machine learning tasks.

I extend my heartfelt thanks to my father, who has been a constant source of inspiration and motivation. His belief in the importance of education and his unwavering support made me determined to embark on this journey.

I am grateful to my partner, Mebrhit Gebru, for her comforting presence, understanding, and motivation during difficult times.

Finally, I would like to thank my friends for their guidance, encouragement, and support. Their advice and willingness to help have been invaluable to the success of this work.

Abstract

The Android accessibility service provides a range of powerful capabilities. These include observing user actions, reading on-screen content, and executing actions on behalf of the user. Although these features are designed to enhance the user experience for individuals with disabilities, they introduce design vulnerabilities that make the accessibility service susceptible to malicious exploitation. This research investigates how Android malware leverages accessibility services for malicious purposes. By analyzing a dataset of malicious applications, we identified common patterns of accessibility service abuse and developed a machine learning-based detection approach using TinyBERT and XGBoost models. We first manually compiled a base dataset of 134 accessibility service event patterns comprising source and sink API calls. These patterns were labeled according to specific malicious functionalities: *BlockAccess*, *ManipulateUI*, and *ContentEavesdrop*. To address data limitations, we generated callgraph from 121 malware samples using FlowDroid taint analysis and applied agglomerative clustering and fuzzy matching, ultimately expanding the dataset size to 1,497 patterns. Our classification experiments compared the performance of TinyBERT, a transformer-based model, and XGBoost, a gradient-boosted decision tree model, in classifying malicious functionalities. Results show TinyBERT's outstanding performance, achieving an accuracy of 97.7% and an F1 score of 97.6% over ten-fold cross-validation, compared to XGBoost's 90.4% accuracy and 90.0% F1 score. This study demonstrates the potential of transformer-based models in capturing sequential dependencies and contextual characteristics in API call patterns, enabling robust detection of accessibility service misuse. Our findings contribute a novel approach to detecting malicious behavior in Android malware and a valuable dataset that may aid similar research.

Keywords: Android malware, Accessibility Services, Accessibility Abuse, Machine Learning, BERT, TinyBERT, XGBoost, content eavesdropping, access blocking, cybersecurity, mobile security.

Table of Contents

Acknowledgment iv

Abstract v

Table of Contents vi

List of Figures ix

List of Tables xi

List of Abbreviations xii

1 Introduction 1
- 1.1 Background Information . . . 2
  - 1.1.1 Features Provided by Accessibility Service . . . 2
  - 1.1.2 Developing an Accessibility App . . . 2
  - 1.1.3 Abuse of Accessibility Service . . . 5
- 1.2 Machine Learning Methods and Tools . . . 7
- 1.3 Motivation of the Study . . . 8
- 1.4 Statement of the Problem . . . 9
- 1.5 Research Questions . . . 10
- 1.6 Objectives . . . 10
  - 1.6.1 General Objective . . . 10
  - 1.6.2 Specific Objectives . . . 11
- 1.7 Expected Contribution of the Study . . . 11
- 1.8 Scope/Delimitation . . . 12
- 1.9 Structure of the Document . . . 12

2 Literature Review 14
- 2.1 Literature Review . . . 14
- 2.2 Related Work . . . 16

3 Methodology 20
- 3.1 Research Design . . . 20
  - 3.1.1 Environment Setup . . . 22
- 3.2 Population and Sampling . . . 23
  - 3.2.1 Population . . . 23

3.2.2 Sampling Design and Sample Size . . . 24
3.3 Data Collection Methods . . . 25
3.4 Research Procedures . . . 25
3.4.1 Data Filtering . . . 25
3.4.2 Capability Analysis . . . 26
3.4.3 Functionality Analysis . . . 28
3.5 Dataset Preparation for ML Model . . . 31
3.6 Data Augmentation . . . 32
3.6.1 Automated Pattern Collection with Clustering . . . 33
3.7 Machine Learning Model Selection . . . 34
3.8 Machine Learning Training Process . . . 35
3.9 Validation and Evaluation . . . 36

4 Proposed Solution **41**
4.1 Overview of the Solution . . . 41
4.2 Architecture and Algorithm Description . . . 42

5 Experiment and Analysis **45**
5.1 Dataset Description . . . 45
5.2 Experimental Setup and Parameters . . . 47
5.3 Experiment 1: Parameter Tuning of the Base Dataset . . . 49
5.4 Experiment 2: Data Augmentation with Clustering . . . 50
5.5 Experiment 3: Classification Model Validation Using Cross-Validation . . . 52

6 Results and Discussion **55**
6.1 Introduction . . . 55
6.1.1 Capability Analysis . . . 55
6.1.2 Functionality Detection . . . 58
6.1.3 Limitations . . . 61

7 Summary and Future Work **63**

References **65**

A Data filtering Python script **70**

B Capability Analysis Python script **74**

C VirusTotal Batch Scanner Python script **80**

D Base Dataset Validation Script **84**

E Base Dataset Validation Results **86**

F Data Augmentation Script **87**

G Model Validation Script **90**

H Model Validation Results **95**

List of Figures

1.1 Accessibility service warning message 3

1.2 Declaring accessibility service 5

1.3 Sample accessibility service configuration 5

3.1 Sample filtering process. 26

3.2 Capability analysis process. 27

3.3 ML process 37

4.1 System architecture 44

5.1 Valid vs invalid samples 46

5.2 Invalid samples by family 46

5.3 Initial dataset. 48

5.4 Expanded dataset distribution 52

5.5 Model validation experiment. 54

6.1 Percentage of capability usage. 56

6.2 Event types capabilities. 57

6.3 Accessibility flag capabilities. 57

6.4 Capability to retrieve windows content. 58

6.5 TinyBERT Confusion Matrix. 59

6.6 XGBoost Confusion Matrix. 60

E.1 Base dataset validation result 86

H.1 Model validation result 95

List of Tables

1.1 Features provided via accessibility services 4

1.2 A summary of accessibility events 6

2.1 A summary of related works. 19

3.1 Software specifications 22

3.2 Hardware specifications 23

3.3 Discovered functionalities 38

3.4 Functionalities of the samples 39

3.5 Source and sinks for the model 40

5.1 Model Training Parameters 48

6.1 Classification Report (Performance Metrics) 61

List of Abbreviations

2FA	**2** (two) **F**actor **A**uthentication
ADB	**A**ndrid **D**ebug **B**ridge
API	**A**pplication **P**rogramming **I**nterface
BERT	**B**idirectional **E**ncoder **R**epresentations from **T**ransformers
ML	**M**achine **L**earning
NLP	**N**atural **L**anguage **P**rocessing
OS	**O**perating **S**ystem
POC	**P**roof **Of** **C**oncept
RQ	**R**esearch **Q**uestion
TTS	**T**ext **T**o **S**peech
UI	**U**ser **I**nterface
XGBoost	(e**X**treme **G**radient **B**oosting)
XML	e**X**tensible **M**arkup **L**anguage

Chapter 1

Introduction

Integrating accessibility features into operating systems is essential for ensuring technology is usable by all individuals, including those with disabilities, as mandated by federal laws [1]. Android, as one of the most widely used mobile operating systems [2], has made significant progress in this area by offering a variety of assistive features through its accessibility framework. These features include the Talkback screen reader, magnification options, and high-contrast text, which aid individuals who are blind or visually impaired [3]. Additionally, the platform provides live captioning and sound amplifiers to support those with hearing impairments [3]. Together, these built-in features empower users with disabilities to operate their devices more effectively and independently.

Beyond these default features, Android supports creating third-party accessibility applications through its comprehensive set of accessibility APIs [4]. This framework enables developers to design custom applications tailored to the specific needs of users with disabilities, further enhancing their experience. However, while these APIs play a crucial role in providing accessibility solutions, they also introduce potential risks, as malicious actors can misuse these capabilities to exploit vulnerable users and compromise security.

1.1 Background Information

Android provides accessibility service to provide user interface enhancements for users with disabilities or who may temporarily be unable to interact with a device fully [4]. For example, people with visual impairment, hearing disabilities, driving a car, or attending a loud party can be assisted with an accessibility service. Android OS provides accessibility apps such as a talkback screen reader and live captions by default. Additionally, it provides APIs for developers to build accessibility apps starting from Android 1.6 (API Level 4) [4]. Access to accessibility service is protected with a special BIND_ACCESSIBILITY_SERVICE permission. As this is a special permission, users must manually enable this permission by going to the device's settings screen. A warning dialog is presented for the user describing the dangers of enabling the permission as shown in Figure 1.1.

The following sections describe the features provided by the accessibility service, the steps required to build an accessibility app, and the abuse of this service by apps not designed to help people with disabilities. It also introduces machine learning methods and tools employed in the research.

1.1.1 Features Provided by Accessibility Service

Accessibility services can provide different features for accessibility apps. The accessibility service documentation groups these features into three categories: accessibility events, taking action for users, and gathering information [4]. Table 1.1 summarizes the features provided under each category.

1.1.2 Developing an Accessibility App

The following steps are required when developing an accessibility app [4].

Step 1. Create your custom accessibility service that extends from the AccessibilityService class.

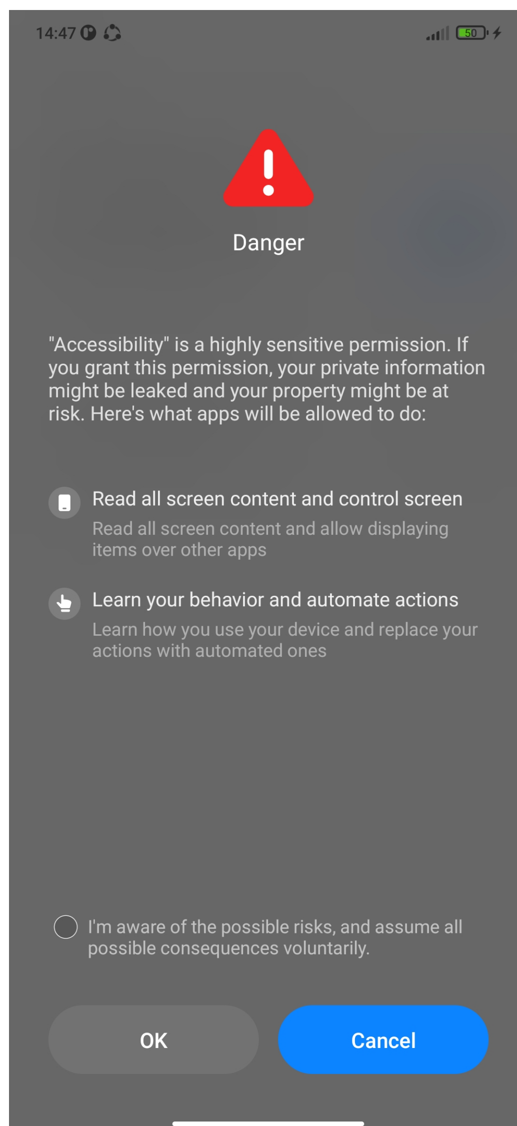

FIGURE 1.1: An accessibility service warning dialog on Xiaomi Redmi Note 8 device.

Step 2. Declare the service in Manifest. As shown in Figure 1.2, this step requires declaring and protecting the service with BIND_ACCESSIBILITY_SERVICE permission to ensure only the system can bind to it. Additionally, provide a configuration XML file that specifies the types of accessibility events that the service handles.

Step 3. Configure the accessibility service. The accessibility service is configured with an XML file. The configuration contains the type of events the accessibility app will register to monitor, capabilities to register, such as retrieving window content, the list of apps to target, and additional information about the service.

Table 1.1: Features provided via accessibility services (* indicates the feature is available only on devices starting from Android 8.0 (API 26)).

Category	**Feature**	**Description**
Accessibility events	Accessibility volume*	It allows the volume of its audio output to be controlled without affecting other sounds on the device.
	Fingerprint gestures*	It allows the fingerprint sensor to respond to directional swipes (up, down, left, and right).
	Multilingual text to speech*	Enables to support automatic language switching on a given single block of text using Android's text-to-speech (TTS) service.
Take action for users	Listen for gestures*	It enables users to listen for gestures and respond to them on behalf of the user, including continued gestures.
	Use accessibility actions	Allows to take actions on apps or the device, such as clicking buttons, toggling checkbox states, scrolling lists, and navigating to the Home screen.
	Use focus types	It allows users to select any visible user interface element and act on it starting from Android 4.1 (API 16).
Gather information	Get event details	Enables obtaining information about user interface interaction through accessibility event objects based on the view hierarchy.
	Get window change details	It lets users keep track of window updates to recognize when the app's user interface changes starting from Android 9 (API 28).
	Gather accessibility node details	Allows retrieval of window content by querying the view hierarchy.
	Process text*	Enables to identify and operate on specific units of text that appear on the screen.

Table 1.2 summarizes some accessibility events for which accessibility apps can configure to receive them from the system. A sample accessibility configuration is shown in Figure 1.3.

Step 4. Listen for accessibility events. On your custom accessibility service, override the methods of the AccessibilityService class to listen to accessibility events. The system will notify accessibility apps through the overridden onAccessibilityEevent() method when an AccessibilityEvent that matches the event filtering parameters specified in Step 3 has occurred.

Step 5. Respond to accessibility events. After receiving an accessibility event

```xml
<service
    android:name=".MyAccessibilityService"
    android:permission="android.permission.BIND_ACCESSIBILITY_SERVICE"
    android:label="Test Accessibility"
    android:exported="true">

    <intent-filter>
        <action android:name="android.accessibilityservice.AccessibilityService" />
    </intent-filter>

    <meta-data
        android:name="android.accessibilityservice"
        android:resource="@xml/accessibility_service_config" />
</service>
```

FIGURE 1.2: Declaration of custom MyAccessibilityService class in Manifest protected with BIND_ACCESSIBILITY_SERVICE permission and configuration file provided by accessibility_service_config XML file.

```xml
<?xml version="1.0" encoding="utf-8"?>
<accessibility-service
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:settingsActivity=""
    android:accessibilityEventTypes="typeViewClicked|typeViewTextChanged
     |typeWindowStateChanged"
    android:accessibilityFeedbackType="feedbackGeneric"
    android:notificationTimeout="1"
    android:accessibilityFlags="flagReportViewIds|flagRequestFilterKeyEvents"
    android:canRetrieveWindowContent="true"
    android:canRequestFilterKeyEvents="true" />
```

FIGURE 1.3: A sample accessibility configuration file that listens for click actions, text change, and state change events and registers a capability to retrieve window content.

in Step 4, it's time to take action. From the received AccessibilityEvent object, it's possible to determine the type of event using the getEventType() method and extract any label text associated with the view that fired the event using the getContentDescription() method.

1.1.3 Abuse of Accessibility Service

The intention of accessibility API is to help developers build accessibility apps that could help people with disabilities [4]. However, research by J. Huang et. al [6]

Table 1.2: A summary of some accessibility events [5].

Accessibility Event Type	**Purpose**
VIEW_CLICKED	Used to track user interactions by notifying when a view, like a button, is clicked.
VIEW_FOCUSED	Used to track focus changes in the UI components by indicating when a view gains focus.
VIEW_SELECTED	Notifies when selection occurs in a view, like selecting text in an Edittext view.
VIEW_TEXT_CHANGED	Monitors the occurrence of changes in text content within views.
WINDOW_STATE_ CHANGED	Notifies when the window state change occurs, like switching an app or a dialog is opened or closed.
NOTIFICATION_STATE_ CHANGED	Monitors when changes occur in the notification state, like a new notification.
VIEW_SCROLLED	It notifies when a view is scrolled, like scrolling in a Listview.
VIEW_TEXT_SELECTION_ CHANGED	Monitors when changes occur in text selection within views.
VIEW_HOVER_ENTER	Notifies when a hover state occurs on a view.
VIEW_HOVER_EXIT	Notifies the hover state has ended.

shows that benign and malicious apps can abuse this powerful capability. Benign apps could exploit accessibility APIs for their own purposes, such as automating user actions or auto-filling credentials rather than helping people with disabilities. On the other hand, malicious apps have also been observed to exploit accessibility features to steal information on user screens including the chat apps [7], transfer money without the knowledge of the user [8], ransomware[1] attacks [9], and automatically download other applications and posting reviews [10]. Additionally, a range of possible attacks that could occur by abusing accessibility services is demonstrated by previous academic research [11], [12], [13], [14], [15]. While most of the research has focused on capabilities gained by abusing accessibility services by demonstrating with a proof of concept (POC) code, the research by J. Huang et. al [6] investigated real-world attack functionalities achieved on malicious apps. However, a detailed analysis of the mechanism employed by malicious apps and

[1]Ransomware is any type of malware that demands a sum of money from the infected user while promising to “release” a hijacked resource in exchange [9].

the range of possibilities that could be achieved by abusing accessibility is not researched.

1.2 Machine Learning Methods and Tools

In this study, we employed a machine learning approach using BERT (Bidirectional Encoder Representations from Transformers), a pre-trained language model recognized for its effectiveness in natural language processing (NLP) tasks [16]. BERT was fine-tuned to classify malicious functionalities within Android malware, with each sample categorized into functionalities such as *BlockAccess*, *ManipulateUI*, or *ContentEavesdrop*. To handle data preprocessing, the BERT tokenizer was applied to ensure consistency with the model's input format. Model training utilized the AdamW optimizer [17] with a learning rate of 5e-5 and a batch size of 8. Performance was evaluated based on accuracy, precision, recall, and F1-score.

Additionally, XGBoost (Extreme Gradient Boosting), a highly efficient, scalable machine learning model based on gradient-boosting algorithms, was used as a comparison to evaluate performance across models. XGBoost is particularly valued in data science for its robustness, scalability, and feature-handling capabilities, especially with structured data, making it suitable for classifying API call patterns in this study [18].

Static analysis tools like FlowDroid [19] were also employed to perform taint analysis, enabling the identification of source-sink relationships in API calls. These tools enhanced the machine learning model by providing insights into data flows within the malware samples, enriching the context for effectively detecting and categorizing malicious behaviors.

1.3 Motivation of the Study

The Android accessibility service provides a range of powerful capabilities, including observing user actions, reading on-screen content, and executing actions like clicks on behalf of the user. While these features are intended to enhance the user experience, particularly for individuals with disabilities, they introduce design vulnerabilities that make the accessibility service susceptible to malicious exploitation. Some studies [11], [12], [13], [14], [15] have identified these design flaws, demonstrating their potential to be abused by malware in various attack scenarios.

In addition to academic research, real-world malware attacks [7], [8], [9], [10], [20] have leveraged the accessibility service to conduct powerful and invasive actions, such as automating user input, escalating permissions, disabling security features, and collecting sensitive data without user consent. These findings collectively underline that the accessibility service, while intended to be a beneficial tool, is a frequent target for misuse by malicious actors.

Despite the significant attention this issue has received in practice and academia, the abuse of accessibility services by malicious applications has not been extensively studied in terms of its full scope and technical implications. Most existing research focuses on individual case studies or specific vulnerabilities, leaving a gap in the comprehensive understanding of how malware systematically exploits these services to achieve malicious goals.

This research aims to fill that gap by conducting a thorough and systematic investigation of how Android malware abuses accessibility services. It will focus on understanding the range of malicious functionalities achieved through such abuse, the techniques used to manipulate accessibility services, and how machine learning can detect these actions. By leveraging state-of-the-art tools such as FlowDroid for static taint analysis and machine learning models like BERT and XGBoost for functionality classification, this study seeks to provide a more comprehensive understanding of this pervasive threat.

1.4 Statement of the Problem

The ideal purpose of the Android accessibility service is to assist individuals with disabilities by providing tools that enable them to interact with their devices more effectively [4]. Through a robust set of APIs, developers can create applications that enhance accessibility for users with various impairments, such as visual or motor disabilities. In this ideal scenario, the accessibility service enables actions like screen reading, magnification, and gesture-based navigation, fostering inclusivity and making technology accessible to all users.

However, these APIs have become targets for exploitation by malicious actors. Malicious developers can repurpose the capabilities of the accessibility service to perform invasive actions, such as stealing sensitive information, automating unauthorized actions, escalating privileges, and disabling security features. Malware leveraging these capabilities can operate with elevated privileges, often bypassing user consent and posing severe threats to privacy and data security [7], [8], [9], [10], [20].

The gap between the ideal scenario and the reality is that current academic research, while identifying certain vulnerabilities in the accessibility service [11], [13], [12], has focused primarily on specific attack scenarios. These studies tend to be limited in scope and do not address the full spectrum of malicious functionalities malware can achieve through accessibility service abuse. Moreover, the mechanisms used by malware to exploit these services have yet to be systematically analyzed.

This thesis aims to bridge this gap by conducting a comprehensive analysis of how Android malware exploits accessibility services to achieve a wide range of malicious functionalities. The research will explore these functionalities, identify the underlying mechanisms, and propose automated methods for detecting such abuses using machine learning models and static analysis techniques.

1.5 Research Questions

The aim of this research is to analyze how Android malware exploits accessibility services systematically. This study explores the malicious functionalities enabled by these services' abuse and identifies the underlying patterns. To achieve this, the research addresses the following key questions:

RQ1 What are the most commonly registered accessibility service capabilities in Android malware, and what makes them attractive for malicious purposes?

RQ2 What specific functionalities do malicious apps achieve by exploiting accessibility services?

RQ3 What is the effectiveness of combining machine learning models and static code analysis for detecting malicious accessibility service functionalities in Android apps?

Addressing these research questions will fill critical gaps in our current understanding of how malware misuses accessibility services. Furthermore, the study aims to develop automated detection mechanisms using machine learning models and static analysis tools.

1.6 Objectives

1.6.1 General Objective

This research investigates how malicious applications misuse Android accessibility services, addressing the gap in understanding their full malicious capabilities through static analysis and machine learning detection.

1.6.2 Specific Objectives

The specific objectives of this study are as follows:

- To review existing research, methodologies, and findings related to the security implications and abuse of Android accessibility services, identifying gaps in the current literature.
- To collect a comprehensive dataset of Android malware samples from reliable sources, filtering and identifying those that explicitly misuse accessibility services for malicious purposes.
- To perform a systematic analysis of the malware samples with a focus on examining the accessibility service capabilities registered in these applications.
- To investigate the malicious functionalities enabled by the abuse of accessibility services and automate the detection of these functionalities using a machine learning model.
- To evaluate the performance of the proposed automated detection model and analyze the results.

1.7 Expected Contribution of the Study

1. This study provides valuable insights for academic researchers and security analysts by offering a deeper understanding of the malicious use of Android accessibility services.
2. It establishes a comprehensive technical and methodological framework for investigating how malware exploits accessibility services using manual and automated analysis techniques.
3. It introduces an automated detection model designed to identify accessibility misuse effectively across functionalities such as content eavesdropping, UI manipulation, and access blocking.

4. The study contributes a labeled dataset of source-sink patterns specific to accessibility misuse by Android malware.

5. It establishes a structured and reproducible workflow for examining malware samples that use accessibility services, offering a reliable approach for processing, decompiling, and analyzing similar samples.

1.8 Scope/Delimitation

This study focuses on malicious applications' misuse of Android accessibility services, specifically targeting the Android operating system. The research will examine only those malicious Android apps that explicitly leverage accessibility services to achieve malicious functionalities such as content eavesdropping, manipulating UI, and access blocking. The malware samples for analysis are collected from public malware datasets, and the study will involve both automated static analysis and manual investigation to understand how the malware exploits these services.

The scope of this research is limited to Android apps that misuse accessibility services, excluding any benign or non-malicious apps that use these services for their intended purpose. Furthermore, the study does not cover other types of attacks or vulnerabilities outside the scope of accessibility service abuse, such as exploits targeting other Android system components. The focus is specifically on malware-driven abuse, and while tools like FlowDroid will be used for static analysis, dynamic runtime analysis is beyond the scope of this research.

1.9 Structure of the Document

The research is structured as follows. Chapter 2 provides a comprehensive review of accessibility services abuse, synthesizing findings from previous studies and identifying gaps in the literature. Chapter 3 describes the methodology used to conduct the research, including data collection, analysis techniques, and tools employed.

Chapter 4 introduces the architecture of the proposed automated detection system, outlining its components and functionality. In Chapter 5, the experiments are detailed, accompanied by an analysis of the results. Chapter 6 interprets the findings and explores their broader implications. Finally, Chapter 7 concludes the thesis by summarizing the key outcomes and offering recommendations for future research directions.

Chapter 2

Literature Review

2.1 Literature Review

Jang et al. [21] was the first academic research to discover vulnerabilities in accessibility service design on major OSs. They targeted four of the most popular OSs: Microsoft Windows, Ubuntu Linux, iOS, and Android. In Android particularly, their evaluation led to the discovery of three design flaws in Android accessibility services that could allow an attacker to bypass voice authentication, bypass the Android sandbox, and read the UI state of other applications.

Kraunelis et al. [12] showed the possibility of abusing accessibility services to conduct successful phishing attacks. Their attack mechanism works by monitoring an accessibility event to listen for app launch from the home or launcher screen and hijacking the execution of the legitimate app. Instead of the legitimate app, a malicious app with a similar user interface is launched to harvest user credentials. However, the researchers only targeted one event and showed how accessibility can be abused with a single functionality. This research will target more functionalities malicious apps practically abuse by leveraging accessibility services.

Fratantonio et al. [14] integrated a system overlay attack with abuse of accessibility service to uncover design flaws of the Android OS that could allow an attacker to achieve full control of the UI feedback loop and managing to execute powerful

attacks such as stealing user credentials, manipulating security PIN, and installation of additional malicious apps. The malicious app only needs system overlay permission (SYSTEM_ALERT_WINDOW) and accessibility service permission (BIND_ACCESSIBILITY_SERVICE). Some accessibility actions, such as auto-clicking to enable permissions, are visible to the user and raise suspicion. The integration of system overlay helped to cover those actions with benign displays. While Fratantonio et al. focused on assessing the Android OS to uncover design flaws resulting from insecurities of system overlay and accessibility service, in this research, we focus on accessibility service and investigate its abuse by malicious apps.

Research by Kalysch et al. [15] also demonstrated that the design of accessibility in the Android OS is flawed and exposes third-party apps to information leakage and denial of service attacks. The researchers presented their attack with POC malware that harvests sensitive information such as login credentials from third-party apps. The research by Kalysch et al. focuses on how accessibility services can be abused to steal sensitive information from third-party apps. In this research, we investigate what functionalities malicious apps could achieve by abusing accessibility services.

Lu et al. [22] demonstrated a successful approach to automatically enable USB debugging on Android devices by abusing the functionality of accessibility service, which otherwise required a complex task of manually enabling it. Android Debug Bridge (ADB) provides powerful capabilities for application developers, including file transfer, shell access, application installation and debugging, and port forwarding, which makes it an attractive target for attackers [22]. The researchers obtained ADB connection authorization without the user's manual involvement bypassing the USB debugging mode's security mechanism. Additionally, their attack bypasses alerts that the ADB Action Monitor displays to the user when sensitive behavior is detected.

Naseri et al. [13] investigated the Android accessibility service and found design flaws that expose apps to eavesdropping attacks, leaks of passwords, and other sensitive information. They demonstrated their attack by listening for standard

accessibility events and eavesdropping on them. The research by Naseri et al. recommends detection tools to avoid leaks on user input fields at the development level, detection of vulnerable apps at the Play Store level, and privacy monitoring tools for the end-user.

Leguesse et al. [23] is the first to study the impact of abusing accessibility services to reduce the forensic footprint of malware attacks significantly. They demonstrated the practicality of their attack with a cryptocurrency wallet stealing malware capable of hijacking the entire withdrawal process, including two-factor authentication (2FA). They compared the forensic footprint left by accessibility-enabled malware with a non-accessibility attack and found that it's significantly reduced. Through stealthy exploitation of Android accessibility service, the research by Leguesse et al. confirms that malware can hinder both the live and post-incident forensic investigation conducted by current malware detection and incident response tools.

2.2 Related Work

Diego et al. [24] performed static analysis using reverse engineering techniques on leading Android spyware [1] apps and documented how the Android APIs are abused to monitor user activities. They discovered that spyware apps abuse accessibility services to collect protected data of other apps, record keystrokes, and survive device reboots. The study by Diego et al. aims to discover the mechanism used by spyware apps that abuse Android APIs. Similarly, this research is directed at understanding the mechanism used by malicious apps. However, it takes a broader approach by analyzing spyware and other types of malware. While the focus is wider, the study specifically investigates the misuse of Accessibility Services APIs.

Diao et al. [11] systematically evaluated the usage of accessibility services on large-scale apps crawled from the Google Play Store using natural language processing, code reviews, and app scanning techniques. Consequently, they discovered that

[1] Spyware apps covertly monitor user's activities such as text messages, phone calls, e-mail communication, and location tracking.

accessibility service is misused for purposes other than helping people with disabilities. Moreover, they demonstrated that accessibility services can be abused for stealthy attacks such as installation hijacking and notification phishing. The research by Diao et al. focuses on evaluating the usage of the accessibility APIs and the design of their supporting architecture. Similarly, our research will also focus on assessing accessibility service usage. However, the approach they followed to document the functionalities of accessibility services was using the configuration description provided by the developers, which can be misleading or incomplete. Our work will be based on reverse engineering the source code of the malware samples. Moreover, while the study by Diao et al. is based on benign applications collected from Google Play, our study will be conducted on malicious apps that abuse accessibility services.

Ichioka et al. [25] used permission analysis and app scanning techniques to study the usage of accessibility services in apps distributed outside of the Google Play Store. Their research is partly motivated by the announcement of Google to remove apps that use accessibility services other than helping disabled people from Google Play Store [26]. As a result, they managed to include social networking services data sources by collecting a dataset of apps distributed through Twitter. Their dataset included the collection and analysis of 32,068 apps. The analysis result indicated that 960 apps utilized accessibility services, of which 79 were suspicious apps that were labeled as malware. While the research included the study of benign and malicious apps, their research is limited to investigating the usage rate of accessibility services. Our research will consist of the functionalities achieved with the abuse of accessibility services and a technical demonstration of how those functionalities were achieved.

More related to our work is the study by Huang et al. [6], who investigated the usage of accessibility services by conducting reverse engineering on a set of malicious apps whose functionality is recorded in analysis reports. In addition to malicious apps, their research included the usage of accessibility services in assistive apps designed for disabled people and benign non-assistive apps. However, their approach

is limited in discovering functionalities not documented in analysis reports. Furthermore, the results obtained are also limited to the number of analysis reports covered by the researchers. The major contribution of their research is a proposed privacy-enhanced extension of the default Android accessibility framework that enables more fine-grained control over accessibility features. The proposed solution will reduce the attack surface while maintaining the functionality of accessibility apps by controlling the data flow within accessibility apps. Our research aims to extensively study different functionalities achieved by malicious apps with the abuse of accessibility services by conducting reverse engineering and manual analysis on collected malware samples. This approach allows us to discover more functionalities that were not discovered before while also creating a better understanding of how they actually work.

From related works that focused on the improper usage of accessibility services in real-world applications, Diao et al. [11] targeted abuse by benign apps collected from the Google Play Store, while Ichioka et al. [25] investigated the rate of usage of accessibility service on malicious and benign apps distributed through Twitter. Similarly, Huang et al. [6] investigated functionalities abused by benign and malicious apps. Our work will build on these studies with a comprehensive analysis of what functionalities are achieved and a technical analysis of the underlying technical methods the malicious apps employ when abusing accessibility services. A summary of the related works is presented in Table 2.1 based on the research methodology used, contribution, and limitations.

Table 2.1: A summary of related works.

Year & Author	**Methodology**	**Contribution**	**Limitation**
2023, Diego et al. [24]	Static analysis on collected samples	Identified two new techniques consumer spyware apps use to bypass system-level isolation: invisible camera access and hiding app icons. They also documented in detail a range of privacy deficiencies in consumer spyware apps.	The study targets discovering the mechanism used only by spyware apps. Our study will include all categories of malicious apps.
2022, Ichioka et al. [25]	Permission analysis and app scanning	Included a dataset that focuses on apps distributed via URLs on Twitter and analyzed utilization rates of accessibility services on the collected samples.	The research is limited to investigating the rate of usage of accessibility services. Our research will include functionalities achieved by abusing accessibility services.
2021, Huang et al. [6]	Static analysis on samples identified by malware report	Studied actual usage of accessibility features in real-world apps and provided a privacy-enhanced accessibility framework.	Discovered malicious functionalities were limited to those reported by malware researchers, which missed some functionalities that were not documented. Our work will depend on reverse engineering and manual analysis of collected samples which allow us to discover more functionalities and better understand how they work.
2019, Diao et al. [11]	Natural language processing, code reviews, and app scanning	Conducted a large-scale study to measure how accessibility APIs are used in the real world and found some fundamental design flaws in the architecture that supports accessibility.	Documented functionalities of accessibility services using the configuration description, which can provide misleading or incomplete results. Our study will use reverse engineering and manual analysis to discover unique functionalities achieved by abusing accessibility services.

Chapter 3

Methodology

This chapter outlines the research design, methods, and tools to address the research problem. The study leverages qualitative and quantitative approaches to detect malicious functionalities in Android applications that exploit accessibility services. The following sections describe the environment setup, data collection methods, population and sampling design, and the machine learning training process.

3.1 Research Design

This research systematically investigates how malicious Android applications exploit accessibility services, aiming to uncover the specific methods and functionalities enabled by such misuse. By analyzing the registered accessibility service capabilities and identifying malicious behaviors, this study provides insights into the broader patterns of accessibility service abuse in Android malware.

The research design is structured to filter malware samples that leverage accessibility services, analyze the range of registered accessibility capabilities, and evaluate the malicious functionalities behind these usages. The design also incorporates machine learning techniques to automate the detection of these functionalities. The following steps outline the design in alignment with the research questions:

1. **Data Collection:** The study begins by gathering a dataset of Android malware samples from public sources. These samples are filtered to identify those explicitly leveraging accessibility services.

2. **Capability Analysis:** In this phase, the frequency of accessibility service capabilities registered across malware samples is quantitatively measured. This analysis provides statistical insights into the most common accessibility features abused by malware, addressing **RQ1**.

3. **Functionality Analysis:** The malware samples are randomly selected and decompiled for manual analysis. This enables the detection of the technical mechanisms through which malware manipulates accessibility services. By analyzing the code and tracing how these services are leveraged for malicious purposes, this step answers **RQ2** by identifying the specific functionalities enabled by the abuse and helps to prepare the dataset required to address **RQ3** by systematically classifying the technical mechanisms behind these manipulations.

4. **Model Training and Classification:** A machine learning models are trained to automatically detect and classify the malicious functionalities enabled by accessibility service abuse. This step addresses **RQ3** by leveraging machine learning models to automate detection.

5. **Evaluation:** The machine learning model's performance is evaluated using metrics such as precision, recall, and F1-score, ensuring the effectiveness of the automated detection system. This evaluation validates the model's accuracy in detecting and classifying the technical mechanisms of accessibility service abuse.

This research integrates quantitative and qualitative methods with machine learning-based automation to provide a comprehensive understanding of accessibility service abuse in Android malware.

Table 3.1: A summary of the software tools employed in the research.

Software	**Purpose**
Apktool	Decompiles APK files into Smali code and decodes resources, allowing for inspection of Android application structure.
JADX	Converts APK files into Java source code.
Jeb Decompiler by PNF Software	A disassembler and decompiler software for Android applications and native machine code.
Android Studio	Provides an integrated development environment for more detailed code review and analysis.
Kavanoz	Unpacks common packer-protected Android malware statically.
FlowDroid	Static taint analysis tool used to trace event-driven data flows from accessibility service events.
Python (NumPy, pandas, scikit-learn)	Automates sample filtering, data preprocessing, model evaluation, and overall analysis workflows, ensuring efficient and reproducible processes.
PyTorch (BERT)	Provides the machine learning framework for training and fine-tuning the BERT-based model to detect and classify malicious accessibility service functionalities.
XGBoost	Implements a gradient-boosted decision tree model to serve as a comparative approach in classifying malicious functionalities.
Matplotlib and Seaborn	Used for data visualization and generating graphical summaries of results, including performance metrics and confusion matrices.

3.1.1 Environment Setup

An appropriate software and hardware environment was set up to analyze the malicious Android applications in this research. This environment is designed to handle both the static analysis and manual investigation phases while ensuring the security and isolation of the malware samples.

The research environment includes software and hardware requirements for analyzing malicious Android applications and developing machine-learning models for automated detection. Table 3.1 summarizes the software tools employed in the research and their specific purposes. These tools are critical for decompiling the Android APK files, conducting static taint analysis, training the machine learning models, and ensuring efficient and reproducible analysis workflows.

TABLE 3.2: A summary of hardware specifications used for analysis.

Component	**Specification**
Processor	13th Gen Intel(R) Core(TM) i7-13620H, 2400 Mhz, 10 Core(s), 16 Logical Processor(s).
RAM	16 GB DDR4.
Storage	1 TB SSD.
Operating System	Windows 11 Pro (64-bit).
GPU	NVIDIA GeForce RTX 4050 Laptop GPU with CUDA version 12.2.

The malware samples under investigation are designed to exploit vulnerabilities in the Android OS. To safely handle these samples without posing a direct threat to the Android ecosystem or personal devices, the analysis was conducted on a Windows workstation. This secure and isolated platform enables reverse engineering tasks without directly exposing an Android device to malware. The Windows environment also supports high-performance workloads, such as decompilation and static analysis, ensuring efficient processing. Table 3.2 summarizes the hardware specifications of the Windows workstation used for the analysis.

3.2 Population and Sampling

3.2.1 Population

This research aims to systematically investigate the misuse of Android accessibility services by malicious applications. Therefore, the research population is defined as a subset of Android malware that exhibits both malicious intent and a dependency on accessibility services to carry out its malicious activities.

The following criteria characterize the research population:

1. **Malicious Intent:** The applications within this population demonstrate verifiable or suspected malicious intent, meaning they perform actions that compromise user privacy, data security, or device integrity.

2. **Accessibility Service Usage:** A defining characteristic of the population is the explicit use of Android's accessibility services for malicious purposes, such as content eavesdropping or blocking user access to device elements.

It is important to note that not all malware samples collected in the initial dataset necessarily exploit accessibility services. As such, the research population specifically excludes any malicious applications that do not show a clear dependency on accessibility services. This ensures the focus remains on understanding and analyzing the exploitation of these services in Android malware.

3.2.2 Sampling Design and Sample Size

The sampling design follows a stratified approach to ensure that the research population includes only Android malware samples that explicitly utilize accessibility services. Given the specific nature of the study, the sampling process was structured into multiple filtering stages to refine the dataset and ensure its relevance to the research objectives.

The initial dataset consisted of 1,800 Android malware samples collected from public malware repositories. To ensure the structural integrity and usability of the APK files, an initial filtering process was applied, which resulted in 738 valid APK samples.

Following this, a criterion-based selection process was used to extract only those samples that demonstrated malicious behavior and explicitly utilized accessibility services for malicious purposes. This process led to identifying 207 Android malware samples that serve as the final research population. The stratified selection approach ensures that the dataset effectively represents malware leveraging accessibility services, allowing for a more precise analysis of malicious functionalities.

3.3 Data Collection Methods

The first step in this research involves collecting a representative set of Android malware samples. The primary source of malware samples for this study is the vx-underground[1] website, a publicly accessible platform that aggregates malware samples from various sources. The platform categorizes malware based on family, following a naming convention derived from Kaspersky [27]. According to this convention, a malware family refers to a group of samples that share common characteristics such as their source code, operating principles, or the nature of their payload [28].

As a secondary source, the study also uses malware samples categorized by year from a GitHub repository maintained by a user known as sk3ptre[2]. This repository contains a collection of Android malware samples from the years 2018 to 2022.

3.4 Research Procedures

This section outlines the research procedures, including filtering the dataset, analyzing accessibility service capabilities, investigating functionalities, and classifying specific malicious functionalities. These procedures are integral to data preparation for machine learning analysis and functionality detection.

3.4.1 Data Filtering

The first step in the process involves filtering the malware dataset to isolate only those samples that exploit Android accessibility services. Since the research focuses on malware that leverages these services, a filtering mechanism is applied to identify apps requesting the required permissions. Any application using accessibility services must declare the `BIND_ACCESSIBILITY_SERVICE` permission in its

[1]https://www.vx-underground.org
[2]https://github.com/sk3ptre

Android Manifest file. This permission acts as a reliable indicator of apps using accessibility services.

The following steps outline the filtering process:

1. The malware samples are decompiled using `Apktool`, which extracts the Android Manifest file and other essential resources.
2. The Manifest file of each app is searched for the `BIND_ACCESSIBILITY_SERVICE` permission. This permission signifies that the app utilizes accessibility services.
3. Descriptive metadata, including sample information and hash values, is extracted and stored for reference.
4. The filtered dataset consists of malware samples that leverage accessibility services and is prepared for further analysis. This filtering process is automated using a Python script included in Appendix A.

This filtering process ensures that the dataset is refined to include only those malware samples that meet the research criteria. The final dataset forms the basis for subsequent capability and functionality analysis. The filtering process is illustrated in Figure 3.1.

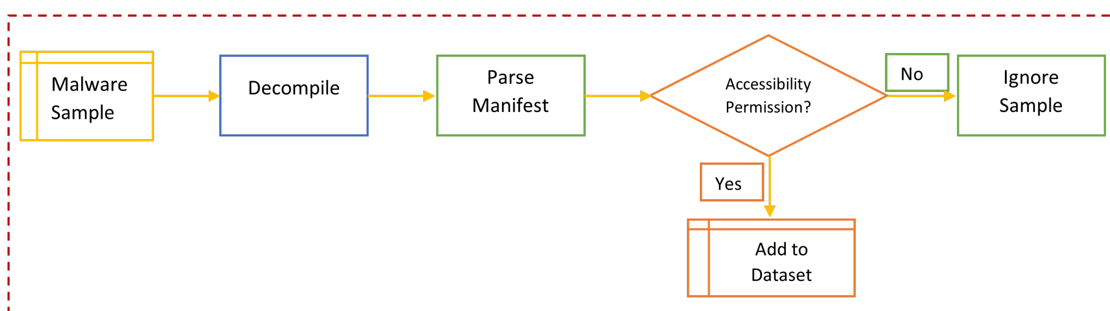

FIGURE 3.1: Sample filtering process.

3.4.2 Capability Analysis

Once the dataset is filtered, the next step involves identifying the accessibility service capabilities registered by the malware. Android accessibility services provide

various capabilities, such as observing user actions, reading on-screen content, and simulating user actions like clicks [4]. To leverage these capabilities, apps must explicitly configure them.

The capability analysis is performed using the following procedure:

1. `Apktool` decompiles the malware samples, extracting both the AndroidManifest and XML configuration files.

2. XML files are analyzed to locate any `<accessibility-service>` tags, which define the declared events, target applications, and registered capabilities.

3. For apps that configure their capabilities dynamically, the smali code is analyzed to extract capabilities set through methods such as `setServiceInfo(AccessibilityServiceInfo)`.

4. The identified capabilities are stored in a dataset. This dataset categorizes each malware sample based on its configured accessibility service capabilities. This capability analysis process is automated using a Python script included in Appendix B.

This process enables a comprehensive analysis of the capabilities used by malware to exploit Android's accessibility services, which answers **RQ1** regarding the most frequently registered capabilities. Figure 3.2 illustrates the capability analysis process.

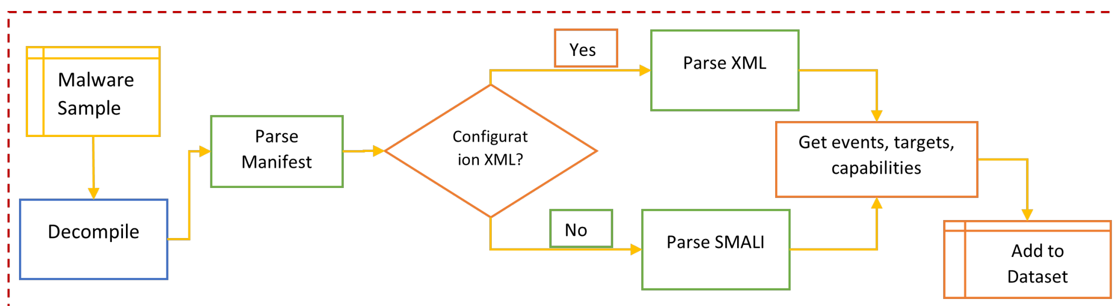

FIGURE 3.2: Capability analysis process.

3.4.3 Functionality Analysis

The functionality analysis aims to determine how malicious apps use their registered capabilities to achieve specific malicious functionalities. This phase addresses **RQ2** by analyzing how malware misuses accessibility services to perform actions like content eavesdropping, UI manipulation, or access blocking.

To investigate the malicious functionalities leveraged through Android accessibility services, we manually inspected 10% of the samples' population. The manual inspection involved decompiling malware samples and inspecting their code to identify key points where data flows originated (sources) and where they were used (sinks). These sources and sinks were mapped to specific malicious functionalities.

During the manual investigation, we followed a structured approach to identify the purpose of specific functionalities within the code. The key steps we followed in this process are:

- **Locating the target class and method:** Identify the classes and methods associated with the accessibility service, focusing on the events and actions initiated through these methods. As a result, we targeted a class that extends from AccessibilityService and looked for the overridden method, onAccessibilityEvent(AccessibilityEvent event).
- **Identifying event types:** Looking for the call to event.getEventType() to identify which event types the malware is interested in listening to. This method returns constants that specify the type of event, such as events to indicate the text input fields have changed, user interactions with UI elements, changes in the app's UI, and monitoring when specific elements gain focus.
- **Filtering events:** Analyze how events are filtered and processed to trigger certain malicious behaviors. For example, looking for conditional checks that filter events based on specific attributes (e.g., package names, class names, view IDs) can reveal targeted applications or specific actions. Looking for

code that contains a hardcoded whitelist or blacklist of package names of popular apps (e.g., banking apps, social media, instant messaging, antiviruses) to indicate a focus on them. Looking for conditional checks on view types (e.g., EditText, Button) which might indicate an intention to capture input.

- **Extracting data:** Examining how data is extracted from the system. For example, looking for calls that retrieve text or content from UI elements using method calls such as getText() or getContentDescription(), which might indicate an intention to obtain sensitive data. Checking for logging methods that store or send extracted data.
- **Action taken:** Identifying actions taken in response to specific events or extracted data. For example, if a TYPE_VIEW_TEXT_CHANGED event triggers a log statement or a network request, it indicates the intention to capture keystrokes or other sensitive input. An event filter that checks the presence of certain app and proceeding with a global action to lock the screen or navigate to the home screen might indicate an intention to block access.
- **Manipulation of other apps:** Determine if the malware interacts with or manipulates other apps. For example, looking for attempts to get information about other windows or activities, suggesting potential UI manipulation or phishing attempts. Looking for methods like performAction(AccessibilityNodeInfo.ACTION_CLICK), which can simulate user clicks potentially for covertly interacting with other apps.
- **Permission Checks:** Analyzing any permission checks or actions on a permission dialog that might indicate attempts to bypass security features.
- **Security bypass techniques:** Examine methods used by the malware to bypass Android's security mechanisms, such as disabling notifications, disabling protection, or bypassing app restrictions.
- **Log information:** Check for calls to Log.d() or similar methods to identify any code that logs information about the purpose of the code. Debug logs can provide insight into the intended functionality or clarify complex logic.

Through this manual analysis, we uncovered several important insights that guided the creation of the dataset for the ML model:

- Code obfuscation: Nearly all analyzed samples were obfuscated, making manual inspection challenging. However, the Android system APIs, including those related to accessibility services, were not obfuscated. This allowed us to focus our investigation on calls to system APIs (e.g., AccessibilityService methods), significantly improving our ability to identify the purpose of the code.
- Uniformity of malicious behavior: Despite variations in coding style among different developers, we found that targeting the system APIs alone was sufficient to deduce the purpose of the functionality. This uniformity allowed us to generalize specific behaviors and translate them into recognizable patterns.

To systematically analyze how malware exploits Android accessibility services, a manual investigation was conducted on a randomly selected subset of 12 samples, representing 10% of the total research population. Each sample was decompiled and examined to identify malicious behaviors enabled by accessibility services. The analysis focused on tracing API call sequences to determine how malware uses these services for malicious purposes.

Table 3.3 summarizes the specific functionalities identified across the analyzed samples. Each discovered functionality was assigned a label for simplified reference in subsequent discussions. Additionally, each functionality was mapped to one of the three main categories of accessibility service abuse:

1. Content Eavesdropping: Capturing sensitive user data, such as keystrokes, on-screen content, or notifications.
2. UI Manipulation: Automating actions such as granting permissions, interacting with security settings, and performing clicks.

3. Access Blocking: Preventing access to specific applications, settings, or security-related features.

Table 3.4 presents detailed information about the analyzed samples, including the malware family, package ID, MD5 hash, and the functionalities discovered in each sample. The functionalities are listed using the corresponding labels from Table 3.3 for clarity.

The functionality analysis provides a structured understanding of how malware misuses accessibility services for different malicious objectives. Based on the functionality analysis findings, we developed a dataset consisting of a categorization label for each functionality and a list of source system API calls invoked to reach a certain sink API call. Table 3.5 shows an example source and sinks, which outlines how specific actions are identified as either content eavesdropping, UI manipulation, or access blocking.

3.5 Dataset Preparation for ML Model

The base dataset for training the machine learning model was constructed based on insights gained from a manual investigation of a selected subset of Android malware samples. This analysis focused on identifying patterns in how malicious apps misuse Android accessibility services to perform various malicious actions, such as logging keystrokes, capturing sensitive information from notifications, and blocking user access to system settings.

Accordingly, a labeled dataset was created where each sample was categorized based on its specific functionalities. The data points in the dataset consisted of source-sink pairs, which were used to represent the flow of sensitive information or malicious activity. Each entry in the dataset contains:

- **Categorization Label:** The label indicates the type of malicious functionality (e.g., content eavesdropping, UI manipulation, access blocking).

- **Source-Sink Pairs:** A list of key sources (data origin points) and sinks (data usage or leak points) that were identified in the samples. These pairs map the flow of sensitive information or actions the malicious app initiates.

This labeled dataset can be used as a rule-based approach to detect predefined patterns of accessibility service abuse, but it has significant limitations. Malware authors frequently modify their code to evade detection, making static rules ineffective. The rationale for using ML is to generalize beyond predefined rules, allowing the system to recognize new variations of malicious functionality without requiring constant manual updates. Additionally, ML models are more resilient to obfuscation techniques such as renaming methods, inserting redundant API calls, or altering execution order when compared to a rule-based approach.

The base dataset is the foundation for training the machine learning model to detect malicious functionalities in Android malware. By including labeled examples of malicious behavior, the dataset allows the model to learn patterns and associations between source-sink flows and specific types of malicious actions.

3.6 Data Augmentation

The initial dataset for this study was a limited set of manually analyzed patterns across three functional categories: *BlockAccess*, *ManipulateUI*, and *ContentEavesdrop*. Although this dataset provided essential insights, its scale was constrained due to the intensive manual analysis required to trace API calls from sources to sinks and accurately categorize them. Expanding the dataset manually for hundreds of malware samples would be impractical and resource-intensive.

To overcome these limitations, we employed FlowDroid, a taint analysis tool that performs static information flow analysis. FlowDroid identifies data flows by tracing sensitive information from specified source API calls (entry points) to sink API calls (endpoints where data may be misused or leaked) [19]. However, FlowDroid

alone does not categorize the intent or functionality behind these flows, such as determining if the behavior is intended for *UI manipulation*, *content eavesdropping*, or *access blocking*.

Our approach leverages FlowDroid's detailed taint analysis as a foundation but enhances it with clustering to identify and group similar patterns. Since FlowDroid outputs raw call graphs without functional categorization, clustering enables us to detect patterns that resemble those in the base dataset. By grouping API call sequences with similar structures, we can automatically assign labels based on their proximity to known malicious patterns. This significantly expands the dataset while reducing the need for manual intervention, ensuring that the classification model generalizes well to new and evolving malware samples.

3.6.1 Automated Pattern Collection with Clustering

The automated pattern collection process involves taint analysis, feature extraction, and clustering-based labeling phases. The steps are as follows:

1. Load the base dataset, which includes patterns with source and sink API calls manually classified into functional categories.
2. Perform taint analysis on each malware sample using FlowDroid. Extract the call graph from FlowDroid's XML output, which contains pairs of source-sink API calls representing possible data flows.
3. Convert each extracted source-sink pair into feature embeddings using a language model to capture semantic similarities.
4. Apply a clustering algorithm to the feature embeddings, where each cluster groups patterns similar to those in the base dataset.
5. For each cluster, compute similarity scores between new patterns and known labeled patterns in the base dataset.
6. Assign labels based on a similarity threshold to classify patterns as *BlockAccess*, *ManipulateUI*, or *ContentEavesdrop*.

Clustering enables automated pattern identification, reducing reliance on manual analysis while significantly expanding the dataset. By grouping similar API call sequences based on their structural and functional similarities, clustering ensures that newly discovered patterns are consistently labeled according to the manually classified base dataset. This process enhances dataset diversity, improves model generalization, and provides a more robust foundation for training ML models to classify Android malware functionalities accurately.

3.7 Machine Learning Model Selection

In this study, the unique structure and characteristics of the dataset drove the selection of a suitable machine-learning model. Moreover, the size of the dataset was taken into consideration. The dataset contains patterns of source and sink API calls, each labeled according to specific functionality categories. This data exhibits characteristics of both structured and sequence-based information.

1. Structured Aspect. The dataset contains a defined structure with source-sink API calls and labels corresponding to specific functionalities (BlockAccess, ManipulateUI, or ContentEavesdrop). This can be interpreted as a form of structured data in that each record consists of a predefined format of elements (API call pairs). Based on this tabular nature of the data, we selected XGBoost. XGBoost is a gradient boosting framework that performs well on tabular data, especially on structured or engineered features [18]. Given the small dataset, XGBoost performs efficiently without the need for extensive pre-training [18].

2. Sequence-Based Aspect. In addition to its structured nature, each pattern in the dataset is also a sequence of API calls. If certain source API calls frequently lead to specific sink API calls, this can reveal a functional pattern (e.g., data leakage or UI manipulation). Because of this sequence-based nature, we selected BERT, which is designed to capture the order and contextual relationships [16]. These models can process the text-like sequences

of API calls and recognize patterns within the sequence that relate to each functionality. However, due to the small dataset, we selected a particular type of BERT called TinyBERT. TinyBERT is a compact transformer-based model that captures sequence dependencies well, with significantly fewer parameters than the original BERT model.

3.8 Machine Learning Training Process

The machine learning training process was designed to detect and classify malicious functionalities in Android malware samples using two models: TinyBERT and XGBoost. TinyBERT was chosen for its strength in sequence classification, while XGBoost was selected for its high performance on structured data.

The process involved the following steps:

1. **Data Preprocessing:** The dataset was filtered, labeled, and preprocessed to ensure consistency. For TinyBERT, the text data was tokenized using the BERT tokenizer, mapping labels for malicious functionalities to integer values. For XGBoost, TF-IDF vectorization was applied to the dataset, converting text data into structured numerical representations compatible with the model.

2. **Model Selection:** Two models were selected for training: TinyBERT and XGBoost. TinyBERT was fine-tuned for sequence classification tasks, where each input sequence (source-sink patterns) was classified into one of the predefined categories of malicious functionality. XGBoost was applied using structured feature vectors generated from TF-IDF to categorize patterns.

3. **Training:** For TinyBERT, the model was fine-tuned using the labeled dataset, with the AdamW optimizer set at a learning rate of 5e-5 and a batch size of 10. The training spanned 5 epochs. For XGBoost, model parameters like learning rate, maximum depth, and a number of estimators were tuned to optimize performance on the structured dataset.

4. **Evaluation:** Both models were evaluated on precision, recall, and F1-score metrics. Additionally, a classification report and accuracy were generated to compare performance. This allowed a comprehensive assessment of each model's ability to detect malicious functionalities effectively.

5. **Prediction and Comparison:** Once trained, each model was used to predict malicious functionalities in new patterns and their results are compared.

Figure 3.3 illustrates the machine learning workflow, covering data preparation, model-specific preprocessing, training, and evaluation, ultimately supporting **RQ3** by establishing a reliable method to detect malicious functionalities in Android malware.

3.9 Validation and Evaluation

The proposed machine learning model was validated using 10-fold cross-validation to ensure its robustness and generalization to unseen data. Cross-validation is an essential technique in machine learning that helps assess the model's performance by partitioning the dataset into multiple folds and training the model iteratively on different folds while keeping one fold aside for validation [29].

This study divided the entire dataset of Android malware samples into 10 equal folds. The model was trained on 9 folds during each iteration and validated on the remaining fold. This process was repeated 10 times, allowing each fold to serve as a validation set once. The final evaluation metrics were averaged across all folds to provide an overall assessment of the model's performance.

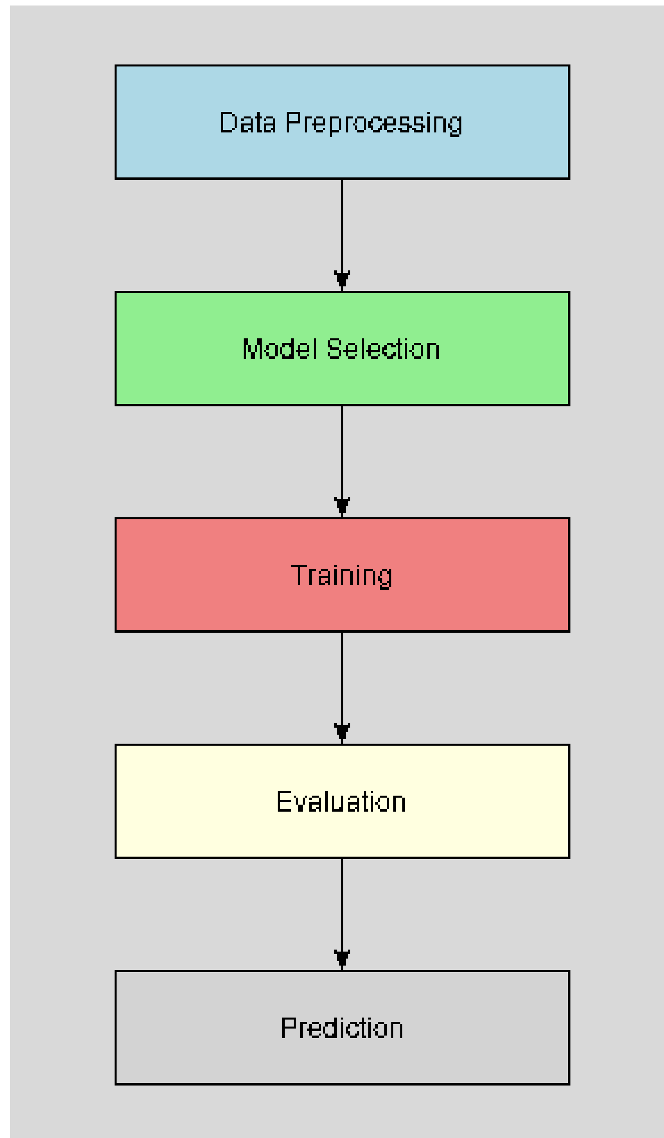

Figure 3.3: The machine learning process.

Table 3.3: List of identified functionalities and their categories

Label	Functionality Description	Category
F1	Log input, focus, click, push notification, and/or other key stroke events	ContentEavesdrop
F2	Block access to setting screen	BlockAccess
F3	Perform click actions to activate, include, and enable setting	ManipulateUI
F4	Set malware as the default SMS app	ManipulateUI
F5	Block access when attempting to reset the system	BlockAccess
F6	Block access when attempting to remove or uninstall the malware	BlockAccess
F7	Block access when attempting to enable Google Play Protect	BlockAccess
F8	Perform cancel button action on uninstall setting screen	ManipulateUI
F9	Block access to accessibility permission setting	BlockAccess
F10	Block access to app menu	BlockAccess
F11	Disable Google Play Protect	ManipulateUI
F12	Block access to enable Google Play Protect setting	BlockAccess
F13	Block access to antivirus app install via Google Play and browser	BlockAccess
F14	Clear recent apps list	ManipulateUI
F15	Perform clicks on activate, enable, etc buttons on the setting page	ManipulateUI
F16	Block access to system update	BlockAccess
F17	Auto grant permissions via permission controller and package installer	ManipulateUI
F18	Lock the screen based on command request from the server	BlockAccess
F19	Clear all notifications	ManipulateUI
F20	Dismiss notification or other dialog box	ManipulateUI
F21	Pin app	ManipulateUI
F22	Block access to notification access setting screen	BlockAccess
F23	Enable Ignore battery optimization	ManipulateUI

TABLE 3.4: Manually analyzed malware samples and their identified functionalities

No.	Sample Family	Package ID	MD5 Hash	Identified Functionalities
1.	Anubis	wocwvy.czyxoxmbauu.slsa	2c522f3527def8ac97958cd2c89a7c29	F1, F2, F3, F4, F5, F6, F7
2.	Banking	kijxlnbftwdhbbet.eaafsym.fziuffcjyjetmqxsmcd	d9f7b42cace711da6fecd6e015e1492e	F1, F4, F5, F6, F8, F9, F10, F11, F12, F13, F14, F15
3.	BitterAPT	display.Launcher	ea3b4cde5ef86acfe2971345a2d57cc0	F1
4.	BlackRock	cmbmpqod.bfrtuduawoyhr.mlmrncmjbdecuc	aef6feeb7fd1af99b5539e8d5a2b712a	F1, F16, F17, F18, F19, F20
5.	DoNot	com.tencent.mobileqq	51b9d09d57365fa4e09251b0072eff1d	F1
6.	EventBot	com.example.eventbot	7107ac3bccd8db274b21f0e494e3eccc	F6, F21
7.	FakeContact	anubis.bot.myapplication	c448ae9ad80f088e9296f08a114605e2	F1, F3, F6, F22
8.	SpyMax	com.covidtz.suffix	f48cb8a945f94c76450065725df069cf	F1
9.	FluBot	com.tencent.mm	1a2a4044cf18eed59e66c413db766145	F1, F4, F6, F11, F17, F23
10.	JobRat	dev.example. trendbanternew	5e176f25144811137618db5592fd84d13	F1
11.	TangleBot	com.ltjkqj.erfycvar	5e176f25144811137618db5592fd84d13	F1, F2, F3, F17
12.	TargetIsrael	cmf0.c3b5bm90zq.patch	e8c58ef7416592ff41624e8308cd6288	F1

Table 3.5: List of sources and sinks for the functionality detection model.

Category	**Sources**	**Sinks**
Content Eavesdrop	int getEventType() android...AccessibilityNodeInfo getSource() java.util.List getText() java.util.Date getTime() java.io.FileInputStream openFileInput(...) java.lang.String readLine()	java.io.FileOutputStream openFileOutput(...) void write(java.lang.String) java.net.URLConnection openConnection() void setRequestMethod(...)("...") java.io.OutputStream getOutputStream() void write(...) void connect() void write(...)
Block Access	java.lang.CharSequence getPackageName() boolean contains(...)("...") java.lang.CharSequence getClassName() boolean contains(...)("android.app.alertdialog") java.util.List getText() java.lang.StringBuilder append(...) android...ApplicationInfo getApplicationInfo(...) java.lang.CharSequence getApplicationLabel(...) boolean contains(...)	android.content.Intent addCategory(...)("...") void startActivity(...)
Manipulate UI	java.util.List getText() java.lang.StringBuilder append(...) boolean contains(...)("...") android...ApplicationInfo getApplicationInfo(...) java.lang.CharSequence getApplicationLabel(...) boolean contains(...) android...AccessibilityNodeInfo getSource() java.util.List findAccessibilityNodeInfosByText(...) boolean contains(...)("...") boolean contains(...)("...")	boolean performAction(int)(...)

Chapter 4

Proposed Solution

This chapter presents the solution to detect malicious functionalities in Android apps that exploit accessibility services. The solution combines static analysis with machine learning techniques. By leveraging static taint analysis and machine learning models, the proposed solution aims to provide a scalable, automated method for detecting functionalities like content eavesdropping, UI manipulation, and access blocking.

4.1 Overview of the Solution

The proposed solution involves an approach that effectively detects malicious functionalities by combining manual investigation, automated taint analysis, and machine learning. Initially, manual investigation identifies key malicious patterns that help inform the automated detection process.

FlowDroid, a static taint analysis tool, traces information flows within the app from sources to sink [19]. The results of this static analysis are used to identify patterns and behaviors aligned with malicious intent. These patterns are then processed by selected machine learning models. The selected models are TinyBERT and XGBoost. TinyBERT was chosen for its strength in sequence classification, while XGBoost was selected for its high performance on structured data. The

models are then trained to classify source-sink pairs into predefined categories of malicious behavior.

The solution also employs a 10-fold cross-validation method to evaluate the model, ensuring its generalizability and effectiveness across different malware samples. Once trained, the model can make accurate predictions on new samples by identifying source-sink patterns related to malicious functionalities.

4.2 Architecture and Algorithm Description

The architecture of the proposed solution involves several key steps, from decompilation to machine learning-based detection. Figure 4.1 illustrates the system architecture:

1. **Decompilation:** APK files are decompiled using tools like Apktool, Jeb, and JADX. The manifest file and code are decompiled to identify whether the app requests the `BIND_ACCESSIBILITY_SERVICE` permission.

2. **Source, Sink, and Purpose Identification:**The taint analysis helps reveal the app's true purpose and behavior.

3. **Data Preprocessing:** The identified source-sink pairs are tokenized and preprocessed to fit the input format required by the models. Labels are assigned to different functionalities, such as content eavesdropping, UI manipulation, and access blocking.

4. **Model Training:** For TinyBERT, the model was fine-tuned using the labeled dataset, with the AdamW optimizer set at a learning rate of 5e-5 and a batch size of 10. For XGBoost, model parameters like learning rate, maximum depth, and a number of estimators were tuned to optimize performance on the structured dataset. During training, the model learns to classify the malicious functionalities based on source-sink patterns.

5. **Evaluation:** The models are evaluated using 10-fold cross-validation, where precision, recall, and F1-score are calculated for each fold. This ensures that the models are generalizable and perform well across unseen samples.

6. **Prediction:** Once the model is trained and validated, it can be deployed to predict malicious functionalities in new malware samples. The source-sink pairs identified by FlowDroid are fed into the trained model, which classifies them into categories such as content eavesdropping, UI manipulation, or access blocking. The results of these predictions help us understand the malicious behaviors embedded in the app.

The proposed solution relies on ML models instead of a rule-based approach. Rule-based detection methods rely on predefined patterns making them effective for identifying known threats but inadequate against evolving malware techniques. As noted in [30], rule-based systems struggle to adapt to sophisticated evasion techniques employed by persistent attackers. Moreover, traditional rule-based approaches do not generalize well to novel malware due to their dependence on static signatures and explicit rules [31]. Given these limitations, this research leverages ML techniques to enhance the detection and classification of Android malware functionalities, providing a more robust and scalable approach to security.

In summary, the proposed solution integrates static analysis with automated machine learning techniques to efficiently detect malicious functionalities in Android apps that exploit accessibility services.

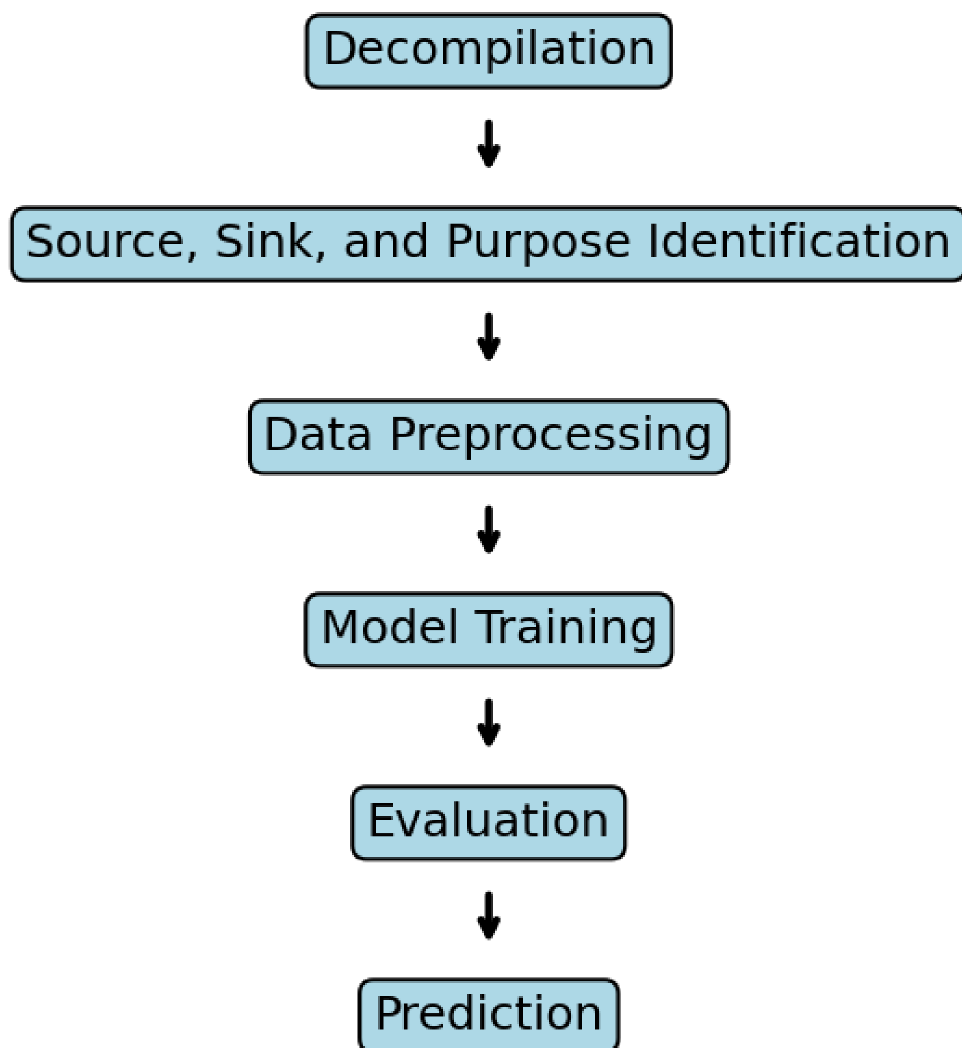

Figure 4.1: System architecture for malicious functionality detection

Chapter 5

Experiment and Analysis

This chapter presents the experiments to achieve the research objectives outlined in the methodology section. Each research phase underwent continuous testing and optimization to ensure accurate results. The next sections will discuss details about the dataset, experiments, results, and any modifications made to improve outcomes.

5.1 Dataset Description

In total, 1,800 samples were collected, representing various malware families. Valid metadata was successfully collected from 738 samples. In comparison, the remaining 1,062 samples were categorized as invalid (see Figure 5.1 for a breakdown of valid and invalid samples). The invalid samples either have corrupted binaries, missed mandatory components, or could not be decompiled. Figure 5.1 shows the percentage of valid versus invalid samples.

Most invalid APK samples (98.4%) belonged to two malware families, Android.Hummingbad and Android.Rummus, as shown in Figure 5.2. We suspect these families employ novel techniques to render analysis tools unusable.

Of the 738 valid samples, 207 were found to use accessibility services. However, a notable observation was made where 208 accessibility services were registered

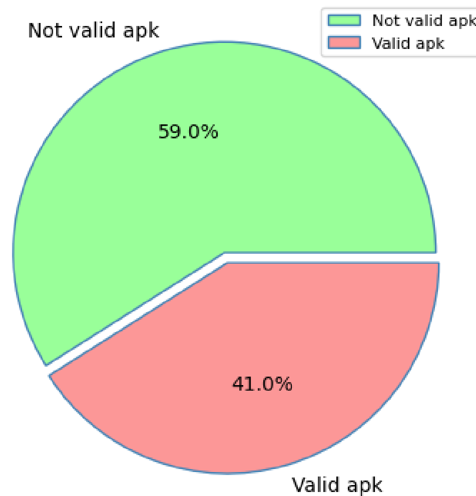

FIGURE 5.1: The number of valid and invalid APKs in collected samples.

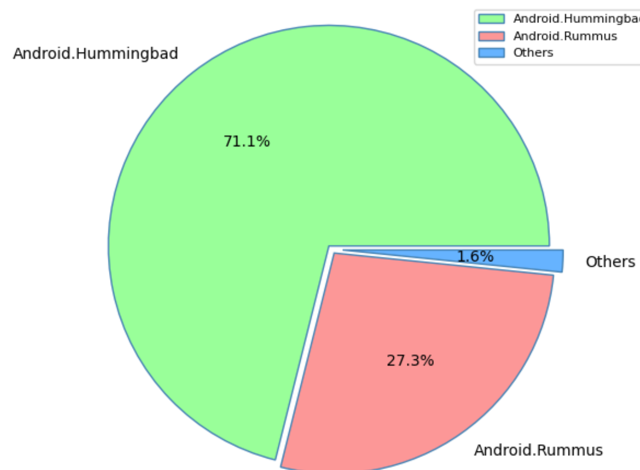

FIGURE 5.2: Number of invalid APK samples filtered by family.

across these samples, indicating that one sample registered two accessibility services. The analysis of unique samples based on various criteria also revealed that there are 153 unique samples by family name, 149 unique samples by accessibility service name, and 78 unique samples by package name. The fact that only 78 unique package names were found indicates that many malware samples are variants of the same app, even though they may differ in their family names or accessibility service configurations.

To ensure that all 207 Android samples utilizing accessibility services were indeed malicious, each sample was scanned using the VirusTotal API. A custom script

was developed to automate this scanning process by submitting the samples in batches and retrieving the scan results from VirusTotal's database. The script extracted relevant details, including file name, hash value, antivirus engine used, and the corresponding scan results, and stored them in a CSV dataset for further analysis. The results confirmed that all 207 samples were flagged as malicious by more than 21 antivirus engines. This validation step ensured that the dataset exclusively contained malicious applications abusing accessibility services. Source code of the batch scanner script is included in Appendix C

By randomly selecting 21 malware samples (10% of the population) that use accessibility services, the initial data set was constructed through manual investigation. Consequently, a total of 134 patterns were collected from the samples, each describing the behavior of Android applications that exploit accessibility services. The patterns consist of Android API functionalities expressed as text sequences to provide context. The dataset was split into three classes: *ManipulateUI* (47 records), *BlockAccess* (41 records), and *ContentEavesdrop* (46 records), as shown in Figure 5.3.

The dataset contains a sequence of API calls in a format of *[package name]: [return type] [function name]([parameter type])([arguments])* and a label describing the functionality of the sequence.

5.2 Experimental Setup and Parameters

The training and validation setup was designed with the parameters shown in Table 5.1:

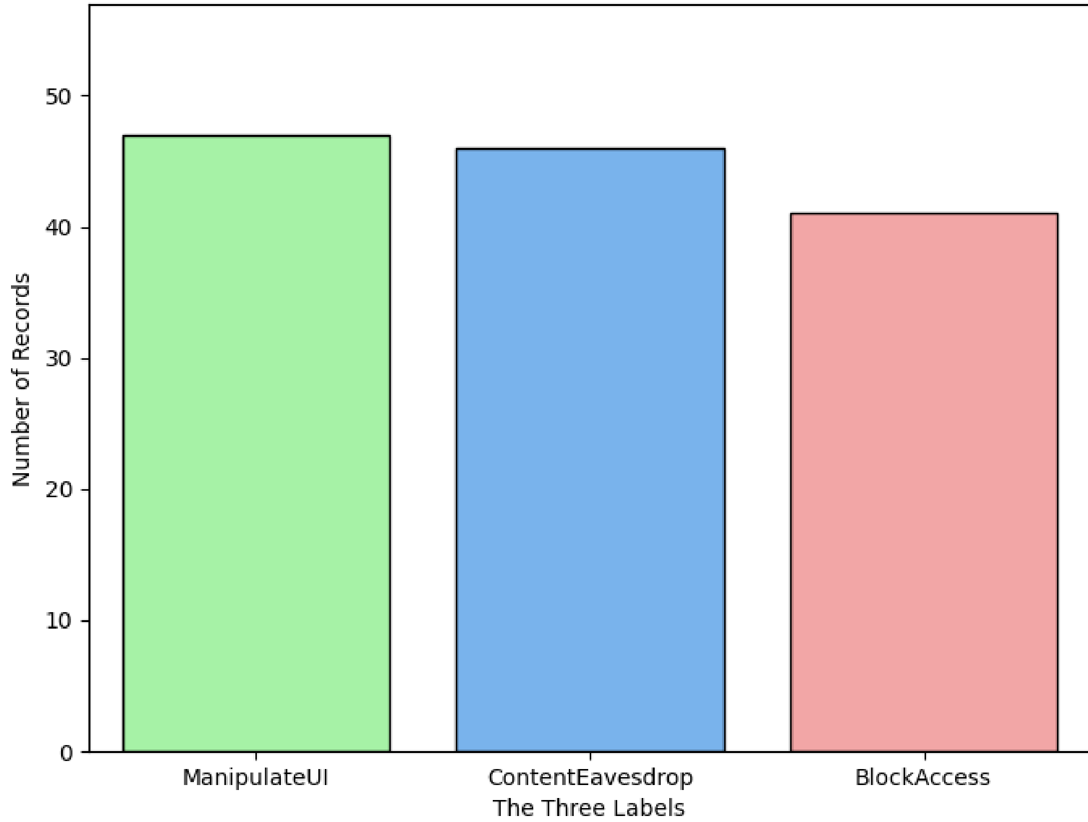

FIGURE 5.3: Initial dataset.

TABLE 5.1: Model Training Parameters

Parameter	**BERT-tiny**	**XGBoost**
Learning Rate	5e-5	0.1
Batch Size	10	Not applicable (tree-based)
Number of Epochs	5	Not applicable (tree-based)
Optimizer	AdamW	Gradient Boosting
Maximum Length (Before Chunking)	1400 tokens	Not applicable
Maximum Length (After Chunking)	512 tokens	Not applicable
Max Depth	Not applicable	6
Number of Trees	Not applicable	100
Evaluation Metric	Accuracy, F1-Score	mlogloss, Accuracy, F1-Score

5.3 Experiment 1: Parameter Tuning of the Base Dataset

The base dataset contains three functionalities categorized as *BlockAccess*, *ManipulateUI*, and *ContentEavesdrop*. The initial experiment aimed to assess the distinctiveness of these categories by measuring the Euclidean distance between their embeddings. Euclidean distance is a widely used metric in high-dimensional vector spaces to measure the similarity between points, where a higher distance indicates greater dissimilarity and a lower distance suggests high similarity or overlap between the clusters [32], [33].

The initial results indicated a Euclidean distance of 0.38 between BlockAccess and ContentEavesdrop, 0.42 between ContentEavesdrop and ManipulateUI, and 0.13 between ManipulateUI and BlockAccess. These values demonstrate a significant overlap between the clusters, particularly between ManipulateUI and BlockAccess, which had the lowest distance of 0.13. This suggests that the embeddings of these categories were closely aligned, limiting their ability to represent distinct functionalities.

The overlapping nature of these clusters can be attributed to the structural and functional similarities in the API call sequences defining each category. ManipulateUI and BlockAccess frequently use the same API sequences, such as capturing an accessibility event, retrieving event descriptions, checking for package names, and executing actions that either modify the UI or block access to a target application. The presence of these shared sequences results in embeddings that are highly similar, making it difficult to achieve clear separability between the categories.

To improve the distinctiveness of the dataset, parameter tuning techniques were applied. First, redundant components such as return types and parameter types were removed from API call sequences. For example, the format:

[package name]: [return type] [function name]([parameter type])([arguments])

was simplified by removing return types and parameter types, as these elements are generic and commonly appear across all clusters. This step helped to reduce redundancy and focus on more meaningful components, such as function names and arguments. Next, special characters such as dots, colons, semicolons, quotes, and braces were replaced with spaces to prevent these symbols from influencing the embedding representations while maintaining the semantic relationships between tokens. Finally, text normalization was applied by converting all text to lowercase, ensuring consistency in representation.

After parameter tuning, new Euclidean distance measurements showed 0.59 between BlockAccess and ContentEavesdrop, 0.69 between ContentEavesdrop and ManipulateUI, and 0.32 between ManipulateUI and BlockAccess. These values indicate a significant improvement in cluster separability compared to the original dataset. However, achieving complete separability remains challenging due to the inherent similarities in accessibility-based malicious behaviors. Some overlap is expected, as malware may use similar API sequences to perform both UI manipulation and access blocking, making absolute differentiation difficult.

Despite these challenges, the improvements in Euclidean distance demonstrate that the dataset became more distinct after parameter tuning, which is crucial for enhancing the performance of the classification model. The refined dataset ensures that machine learning models can better differentiate between malicious functionalities based on accessibility service abuse.

The source code for testing this experiment is included in Appendix D. Figure E.1 in Appendix E illustrates a detailed result of the experiment.

5.4 Experiment 2: Data Augmentation with Clustering

To enhance the diversity and representation of the dataset, a data augmentation experiment was conducted using FlowDroid for taint analysis and clustering-based

pattern identification. The manually created base dataset was limited in scale due to the intensive effort required for manual analysis. Expanding the dataset was necessary to improve the model's generalization and ensure it could effectively classify new and unseen malicious behaviors exploiting Android accessibility services.

To expand the dataset, FlowDroid was used to perform static taint analysis on 121 Android malware samples. The remaining 86 samples from the total 207 malware samples were excluded because FlowDroid was unable to successfully analyze them. FlowDroid traces data flows from predefined entry points where sensitive information is accessed (source) to endpoints where data is used or leaked (sink) [19]. Each malware sample was processed through FlowDroid, which generated an XML-based call graph, mapping source-to-sink relationships. From these call graphs, a total of 42,910 API call patterns were extracted. However, FlowDroid's output does not automatically indicate whether a pattern corresponds to UI Manipulation, Content Eavesdropping, or Access Blocking. Thus, additional processing was required to classify these patterns into the appropriate functional categories.

Since manually labeling each extracted pattern was impractical, agglomerative clustering was applied to group similar patterns based on their structure and semantic similarity to the base dataset. This clustering approach allowed patterns with similar API call sequences and intent to be grouped. After clustering, fuzzy matching using cosine similarity was applied within each group. This step ensured that new patterns were assigned to the closest base dataset label by evaluating their semantic similarity to previously labeled patterns.

After the fuzzy matching process the total dataset size was expanded to 1,497 patterns in which 878 belong to ManipulateUI label, 384 to ContentEavesdrop label, and 235 to BlockAccess label. Figure 5.4 shows the distribution of the expanded dataset.

To ensure the reliability of the augmented dataset, manual validation was conducted on a randomly selected 5% of the newly identified patterns from each

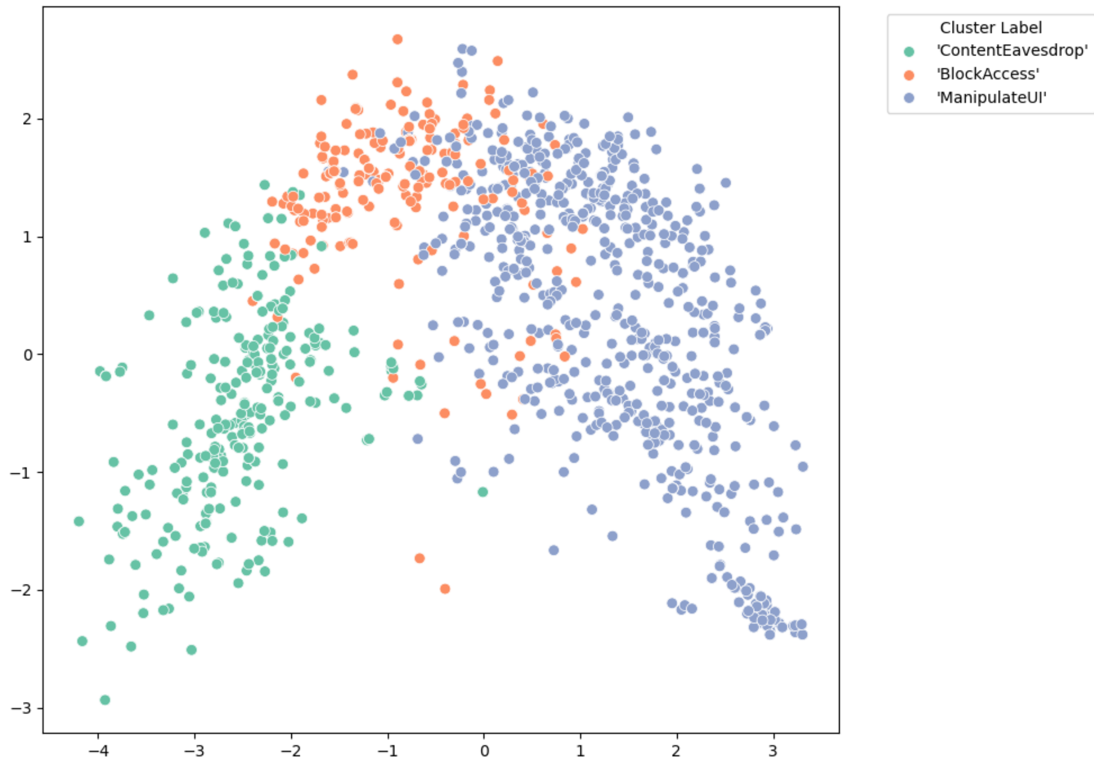

FIGURE 5.4: Pattern distribution per cluster.

category. This validation process involved manually reviewing the API call sequences to verify their correctness in the context of their assigned category and writing equivalent Java code to simulate the extracted API sequences. The simulation confirms that the augmented sequences correctly reproduced the expected accessibility service misuse behavior.

The source code for testing this experiment is included in Appendix F.

5.5 Experiment 3: Classification Model Validation Using Cross-Validation

The final experiment evaluates the model's generalizability and robustness across different portions of the dataset. By using k-fold cross-validation, we assess how well the model performs on unseen data from multiple random splits, validating that its high performance is not due to overfitting on specific segments of the dataset.

The methodology for this experiment involves applying 10-fold cross-validation to assess model performance and stability. In each fold, one subset serves as the validation set, while the model is trained on the remaining nine. This rotation allows every data point to be used in both training and validation. Both TinyBERT and XGBoost are trained using their optimal parameters from prior experiments, with metrics such as accuracy, precision, recall, and F1 score recorded for each fold. The average and standard deviation of these scores are then calculated across all folds, allowing for a comparison of model stability. A lower standard deviation indicates a more stable model performance across the dataset.

The comparison between TinyBERT and XGBoost shows that TinyBERT performs better, achieving 97.66% accuracy and F1-score compared to XGBoost's 90% (see Figure 5.5). TinyBERT's stronger performance comes from its ability to recognize the order and context of API call patterns. As a transformer-based model, TinyBERT is particularly effective at identifying patterns in the sequence of API calls, which helps it generalize more reliably across different classes.

XGBoost, on the other hand, builds its understanding using TF-IDF features. While this approach is competitive, it doesn't capture the detailed relationships within API sequences as TinyBERT did. XGBoost is more efficient in terms of computation, but TinyBERT's slightly higher computational needs are justified by its improved accuracy. Therefore, TinyBERT is preferred for detecting function call patterns in this study.

In summary, TinyBERT's outstanding performance and stability make it better suited for distinguishing malware functionalities from API sequence patterns.

The source code for testing this experiment is included in Appendix G, and Figure H.1 in Appendix H shows its corresponding result.

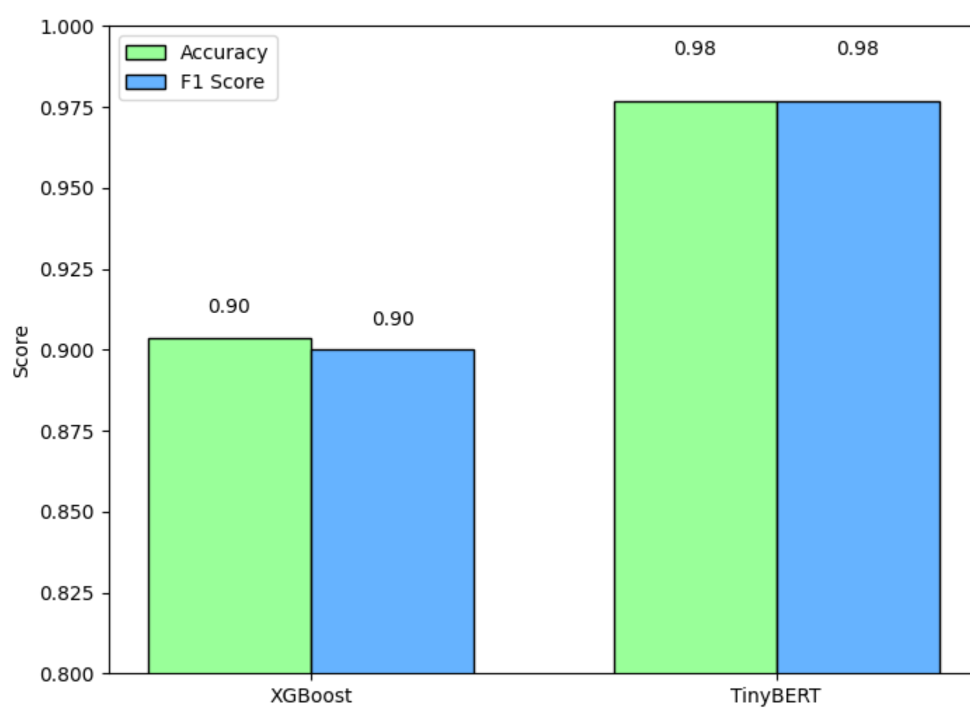

FIGURE 5.5: Model validation experiment.

Chapter 6

Results and Discussion

6.1 Introduction

This chapter presents the results obtained from the experiments conducted in this research and discusses these findings in the context of the research objectives. Its purpose is to demonstrate how the proposed solution addressed the problem of understanding the purposes of malicious Android applications to leverage accessibility services. The results are structured to align with the specific objectives of the research.

6.1.1 Capability Analysis

The scope of this research is specifically limited to detecting the functionalities of accessibility service abuse in malicious applications. To ensure that the dataset consists only of malicious samples, all analyzed applications were scanned using the VirusTotal API. The scan results confirm that each sample was flagged as malware by at least 21 antivirus engines. By focusing only on malicious apps, this study provides a targeted approach to understanding and classifying different types of accessibility service abuse.

The capability analysis result revealed that malware samples extensively used accessibility service capabilities. As shown in Figure 6.1, most samples relied heavily on EventTypes capability to monitor system and user interactions. Additionally, many samples used RetrieveWindowsContent to access data from other applications.

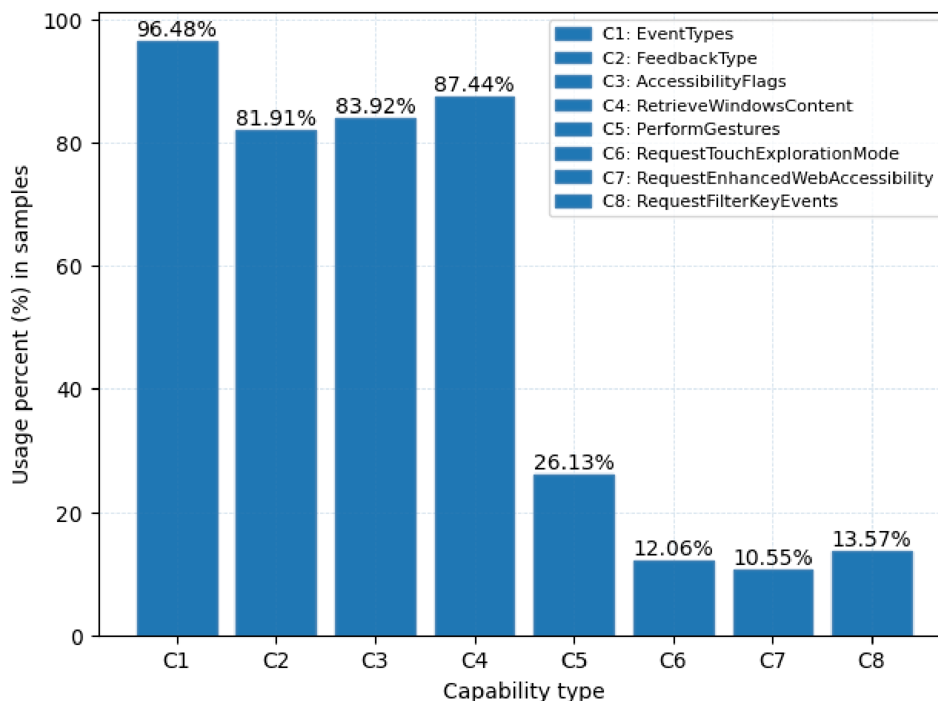

Figure 6.1: Percentage of capability usage.

The heavy registration of EventTypes (see Figure 6.2) highlights the malware developers' intent to track various user behaviors and interactions. These malware samples could dynamically observe and respond to user actions by monitoring various system and window changes. This behavior points to the malware's objective to manipulate the user interface and monitor critical actions, creating opportunities for malicious interference.

The registration of AccessibilityFlags (see Figure 6.3) shows that many malware samples sought to gain enhanced control over the device's UI elements. These flags allow the malware to gather detailed information about app windows and manipulate components that are typically not accessible. This enhanced control underscores the malware's ability to interfere with or modify the device's user interface to further its malicious objectives.

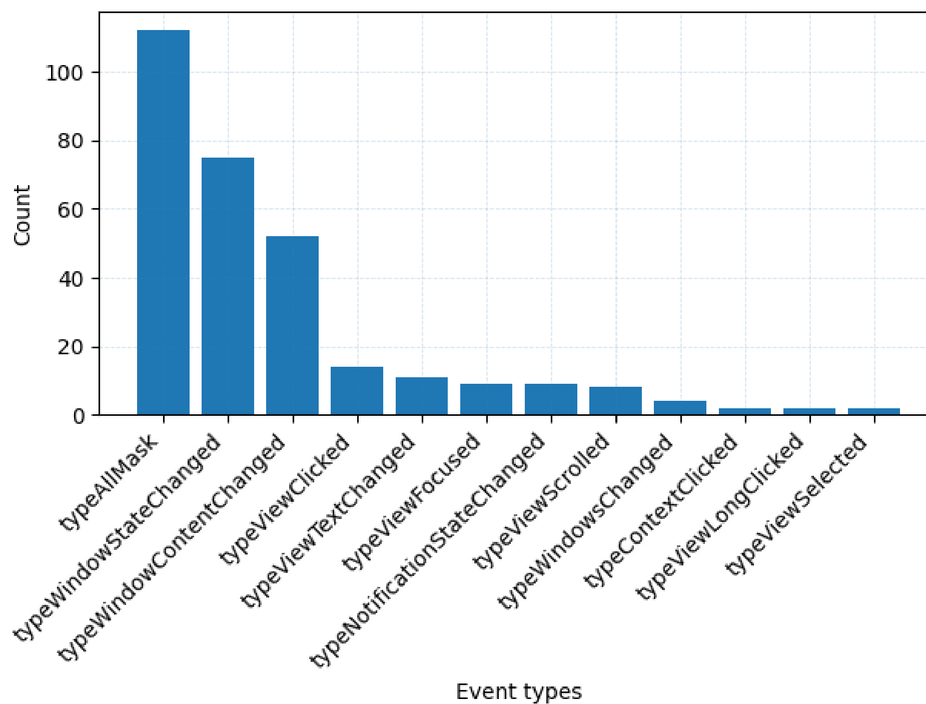

FIGURE 6.2: Event types capabilities.

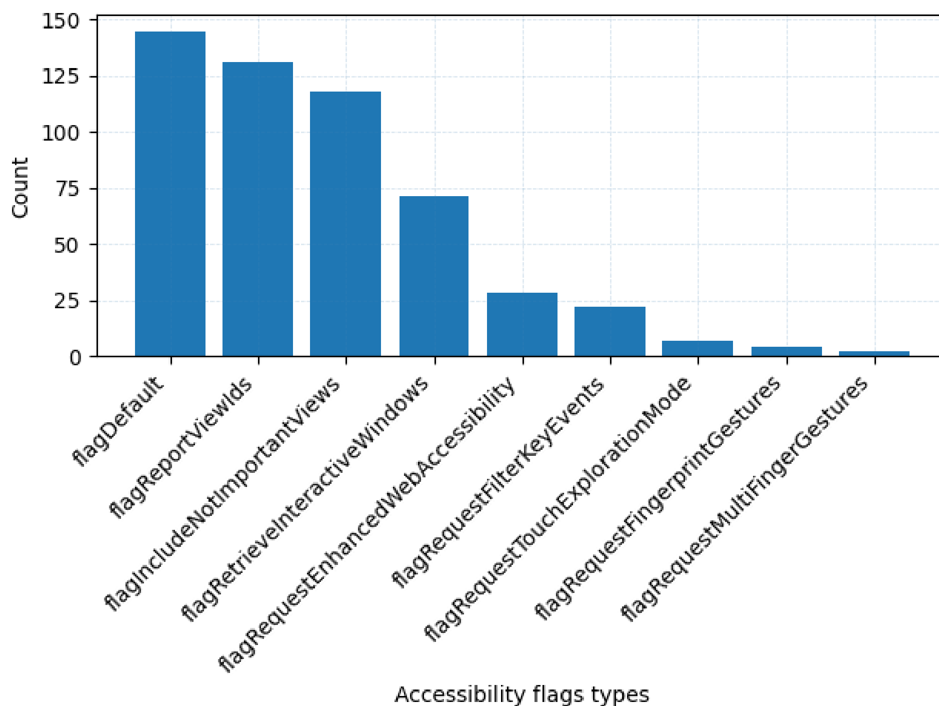

FIGURE 6.3: Accessibility flag capabilities.

The analysis of RetrieveWindowsContent (see Figure 6.4) revealed its critical role in malware functionality. Out of the 199 samples analyzed, 174 enabled this flag. This widespread use implies that accessing content from other apps' windows is a key strategy for malware. It allows the malware to extract sensitive information, such as user inputs or confidential data from other applications, making it a highly effective tool for data theft or surveillance.

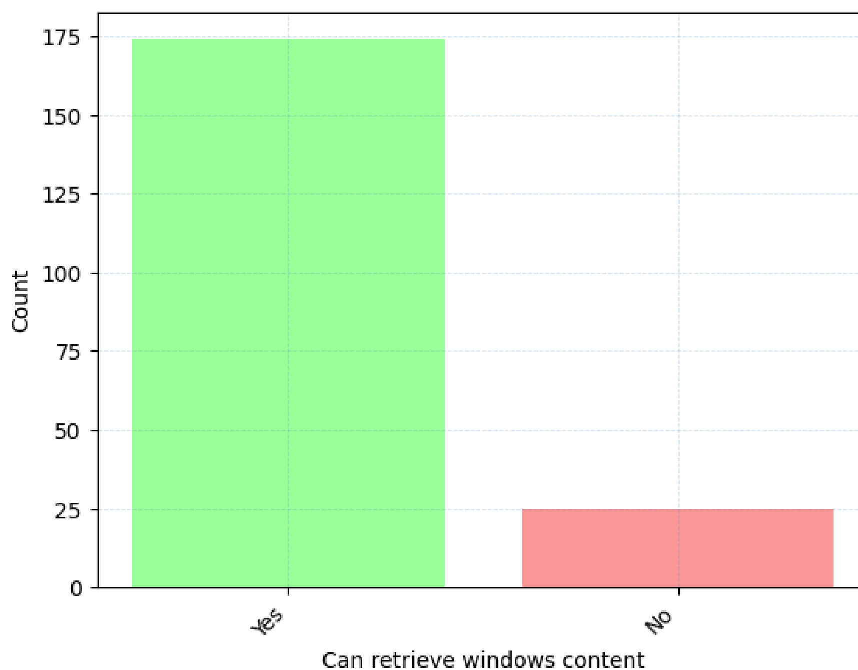

Figure 6.4: Capability to retrieve windows content.

6.1.2 Functionality Detection

Both TinyBERT and XGBoost models were evaluated using a confusion matrix and classification report to assess their performance across the three target classes: BlockAccess, ManipulateUI, and ContentEavesdrop. These metrics provided a comprehensive view of the model's classification capabilities regarding precision, recall, F1-score, and overall accuracy.

TinyBERT achieved an average accuracy of 97.66% with an F1-score of 97.67%, demonstrating high predictive performance and consistency, as indicated by its low standard deviations (0.0086 for both accuracy and F1-score). As the confusion matrix in Figure 6.5 shows, this model effectively minimizes classification errors

across all three classes. For instance, it has low misclassification rates, with only 17 misclassifications between ManipulateUI and other categories and very few false positives and negatives for both BlockAccess and ContentEavesdrop. These results indicate that TinyBERT's ability to capture semantic characteristics in API call patterns translates to more robust performance and stable generalization across folds.

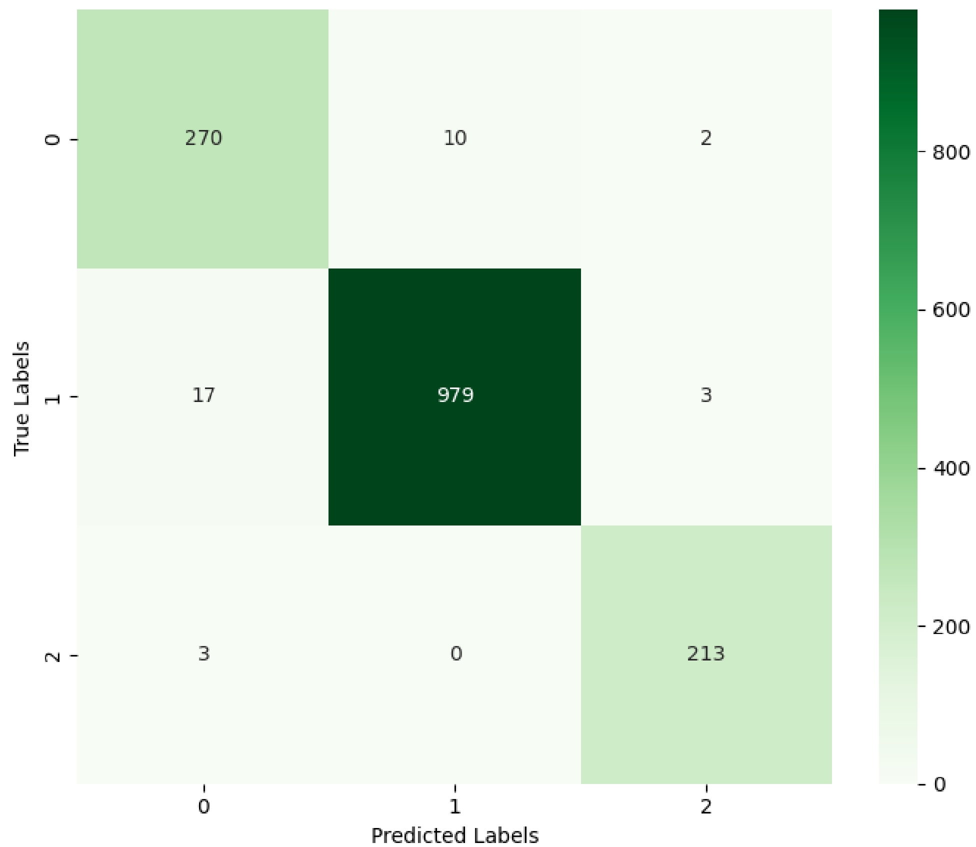

Figure 6.5: TinyBERT Confusion Matrix.

On the other hand, XGBoost, while performing well with an average accuracy of 90.38% and an F1-score of 90.00%, has higher standard deviations (0.0200 and 0.0208) and shows a broader range of misclassifications. The confusion matrix shown in Figure 6.6 reveals that XGBoost has higher false positives and negatives, particularly with the ManipulateUI class, where it misclassifies 89 cases as BlockAccess and 33 as ContentEavesdrop. This suggests that while XGBoost effectively captures certain API pattern features, its reliance on TF-IDF feature representation may limit its ability to fully capture the contextual relationships present in API sequences.

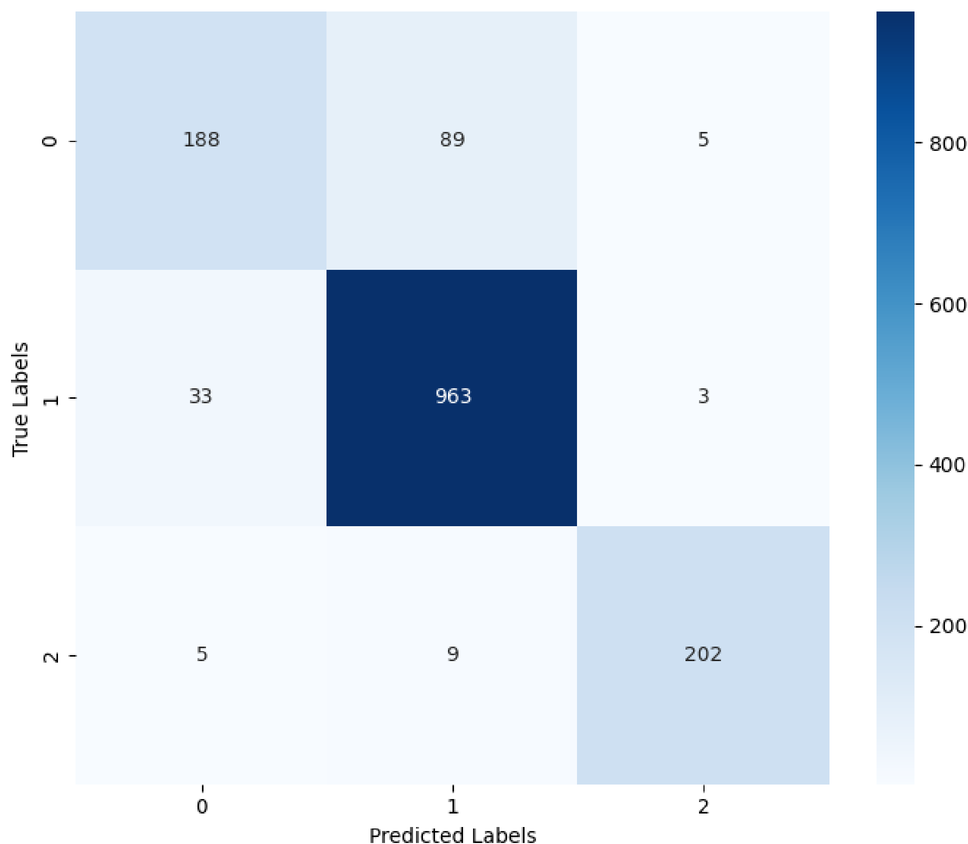

FIGURE 6.6: XGBoost Confusion Matrix.

Table 6.1 provides the detailed classification report, summarizing precision, recall, F1-score, and support for each class.

The results demonstrate that the TinyBERT model effectively detects malicious functionalities based on Android accessibility API patterns. The high F1-scores across all classes indicate that the model balances precision and recall well. The lower standard deviation demonstrates its high predictive performance and consistency.

Classifying the functionalities of malware that abuse accessibility services has several practical applications in cybersecurity, malware detection, and forensic analysis. Identifying whether malware engages in content eavesdropping, UI manipulation, or access blocking helps end users, security researchers, and analysts assess the potential risks posed by different malware families.

For end users, functionality classification provides forensic insights into the threats posed by malicious apps on their devices. Security researchers, on the other hand,

Table 6.1: Classification Report (Performance Metrics)

TinyBERT				
Class	**Precision**	**Recall**	**F1-Score**	**Support**
BlockAccess	0.93	0.96	0.94	282
ManipulateUI	0.99	0.98	0.98	999
ContentEavesdrop	0.98	0.99	0.98	216
Accuracy	0.98 (Overall)			
Macro Avg	0.97	0.97	0.97	1497
Weighted Avg	0.98	0.98	0.98	1497
XGBoost				
Class	**Precision**	**Recall**	**F1-Score**	**Support**
BlockAccess	0.83	0.67	0.74	282
ManipulateUI	0.91	0.96	0.93	999
ContentEavesdrop	0.96	0.94	0.95	216
Accuracy	0.90 (Overall)			
Macro Avg	0.90	0.86	0.87	1497
Weighted Avg	0.90	0.90	0.90	1497

can use this classification to develop targeted defense mechanisms against such malware. For example, detecting malware that blocks access to security settings helps in designing prevention strategies against privilege escalation. Similarly, identifying UI manipulation threats can lead to improved user consent mechanisms to mitigate unauthorized actions. By offering a detailed understanding of specific malicious behaviors enabled through accessibility services, this research contributes to improving malware detection techniques and response strategies.

6.1.3 Limitations

While the TinyBERT model demonstrated strong performance in detecting malicious functionalities, there are several limitations to this study:

- **Dataset Size and Diversity:** The dataset, though augmented, may not fully represent the breadth of real-world malware samples. A larger, more diverse dataset that includes more malware families and different types of attacks could provide deeper insights and improve the model's robustness.
- **Limited Generalization:** The model was trained on a dataset of specific malware functionalities based on Android accessibility service misuse. This narrows the generalizability of the model. Malware outside the studied functionalities or with different attack strategies may not be effectively detected. Expanding the dataset to include a broader range of malware behaviors could enhance the model's versatility.
- **Callgraph Extraction.** The use of packers and implementation of accessibility event processing in native languages like C or C++ prevents the extraction of call graphs using taint analysis tools like Flowdroid. Additionally, FlowDroid encountered issues with some samples, likely due to complex code structures or larger file sizes, resulting in incomplete or failed analyses. The model will fail to detect functionality in such samples.

Chapter 7

Summary and Future Work

This thesis explored the detection and analysis of malicious functionalities that exploit Android accessibility services using a lightweight machine learning approach with TinyBERT and XGBoost. While accessibility services are designed to assist users with disabilities, they have also been misused by malware to monitor user interactions, manipulate UI elements, and steal sensitive data. This study aimed to develop an effective classification method to identify these malicious functionalities based on API call patterns and method signatures.

Through careful data collection and filtering, a dataset was created to reflect how malware abuses accessibility services. The capability analysis revealed that malware frequently exploits features such as EventTypes and RetrieveWindowsContent, confirming its intent to track user behavior and extract sensitive data. Classifying these functionalities provides practical cybersecurity benefits, as it helps security analysts and researchers develop targeted defense mechanisms. For example, identifying malware that performs UI manipulation can guide the development of stronger user consent mechanisms, while detecting access-blocking malware informs strategies to prevent privilege escalation attacks.

Using TinyBERT for classification yielded high accuracy and F1 scores of 97.66%, demonstrating that lightweight models can efficiently detect malware functionalities even with limited computational resources. This makes them suitable for deployment in mobile and embedded security solutions. However, limitations such

as dataset size, classification diversity, and malware obfuscation indicate the need for further refinements. Future work will focus on expanding the dataset, improving generalization, increasing classification diversity, and incorporating more advanced transformer models to better capture evolving malware patterns.

In conclusion, this research contributes to the growing field of malware classification by demonstrating the effectiveness of TinyBERT in detecting and categorizing malicious accessibility service abuse. By providing a structured approach to analyzing accessibility service misuse, this study enhances malware detection and response strategies. The findings and methodologies presented offer a foundation for future research and practical cybersecurity applications, reinforcing the importance of machine learning in proactive malware defense.

References

[1] IT Accessibility Laws and Policies Section508.gov, 2020. URL https://www.section508.gov/manage/laws-and-policies/#508-policyhttps://www.section508.gov/manage/laws-and-policies.

[2] Statista. Global market share of mobile operating systems, 2023. Available at https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems/.

[3] Android accessibility overview - Android Accessibility Help, . URL https://support.google.com/accessibility/android/answer/6006564.

[4] Create your own accessibility service — Android Developers, . URL https://developer.android.com/guide/topics/ui/accessibility/service.

[5] AccessibilityService Reference. URL https://developer.android.com/reference/android/accessibilityservice/AccessibilityService.

[6] Jie Huang, Michael Backes, and Sven Bugiel. A11y and privacy don't have to be mutually exclusive: Constraining accessibility service misuse on Android. *Proceedings of the 30th USENIX Security Symposium*, pages 3631–3648, 2021.

[7] Skygofree — a sophisticated spyware Trojan for Android — Kaspersky official blog. URL https://www.kaspersky.com/blog/skygofree-smart-trojan/20717/.

[8] Gustuff: weapon of mass infection, 2019. URL https://blog.group-ib.com/gustuff.

[9] Robert Lipovský and Lukas Stefanko. Android Ransomware: From Android Defender to DoubleLocker. *Eset*, pages 1–15, 2018. URL https://www.welivesecurity.com/wp-content/uploads/2018/02/Android$_{R}$ansomware$_{F}$rom$_{A}$ndroid$_{D}$efender$_{t}$o$_{D}$oublelocker.pdf.

[10] McAfee Mobile Threat Report: Mobile Malware Is Playing Hide and Steal. URL https://www.mcafee.com/content/dam/consumer/en-us/docs/2020-Mobile-Threat-Report.pdf.

[11] Wenrui Diao, Yue Zhang, Li Zhang, Zhou Li, Fenghao Xu, Xiaorui Pan, Xiangyu Liu, Jian Weng, Kehuan Zhang, and Xiao Feng Wang. Kindness is a risky business: On the usage of the accessibility APIs in Android. *RAID 2019 Proceedings - 22nd International Symposium on Research in Attacks, Intrusions and Defenses*, pages 261–275, 2019.

[12] Joshua Kraunelis, Yinjie Chen, Zhen Ling, Xinwen Fu, and Wei Zhao. On Malware Leveraging the Android Accessibility Framework. *ICST Transactions on Ubiquitous Environments*, 1(4):e1, 2015. doi: 10.4108/ue.1.4.e1.

[13] Mohammad Naseri, Nataniel P. Borges, Andreas Zeller, and Romain Rouvoy. AccessiLeaks: Investigating Privacy Leaks Exposed by the Android Accessibility Service. *Proceedings on Privacy Enhancing Technologies*, 2019(2): 291–305, 2019. doi: 10.2478/popets-2019-0031.

[14] Yanick Fratantonio, Chenxiong Qian, Simon P. Chung, and Wenke Lee. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. *Proceedings - IEEE Symposium on Security and Privacy*, pages 1041–1057, 2017. ISSN 10816011. doi: 10.1109/SP.2017.39.

[15] Anatoli Kalysch, Davide Bove, and Tilo Müller. How android's UI security is undermined by accessibility. *ACM International Conference Proceeding Series*, 2018. doi: 10.1145/3289595.3289597.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[17] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2019.

[18] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794. ACM, 2016.

[19] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FLOWDROID: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *ACM SIGPLAN Notices*, 49(6):259–269, 2014. ISSN 15232867. doi: 10.1145/2594291.2594299.

[20] Insidious Android malware gives up all malicious features but one to gain stealth — WeLiveSecurity. URL https://www.welivesecurity.com/2020/05/22/insidious-android-malware-gives-up-all-malicious-features-but-one-gain-stealth.

[21] Yeongjin Jang, Chengyu Song, Simon P. Chung, Tielei Wang, and Wenke Lee. A11y attacks: Exploiting accessibility in operating systems. *Proceedings of the ACM Conference on Computer and Communications Security*, pages 103–115, 2014. ISSN 15437221. doi: 10.1145/2660267.2660295.

[22] Hui Lu, Xiaohan Helu, Chengjie Jin, Yanbin Sun, Man Zhang, and Zhihong Tian. Salaxy: Enabling USB Debugging Mode Automatically to Control Android Devices. *IEEE Access*, 7:178321–178330, 2019. ISSN 21693536. doi: 10.1109/ACCESS.2019.2958837.

[23] Yonas Leguesse, Mark Vella, Christian Colombo, and Julio Hernandez-Castro. Reducing the Forensic Footprint with Android Accessibility Attacks. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12386 LNCS:22–38, 2020. ISSN 16113349. doi: 10.1007/978-3-030-59817-4_2.

[24] U C San Diego, Sam Havron, Stefan Savage, Geoffrey M Voelker, and Damon Mccoy. No Privacy Among Spies : Assessing the Functionality and Insecurity of Consumer Android Spyware Apps.

[25] Shuichi Ichioka, Estelle Pouget, Takao Mimura, Jun Nakajima, and Toshihiro Yamauchi. Analysis of Android Applications Shared on Twitter Focusing on Accessibility Services. *Journal of Information Processing*, 30(Wisa 2020): 601–612, 2022. ISSN 18826652. doi: 10.2197/IPSJJIP.30.601.

[26] Google will remove Play Store apps that use Accessibility Services for anything except helping disabled users. URL https://www.androidpolice.com/2017/11/12/google-will-remove-play-store-apps-use-accessibility-services-anything-except-helping-disabled-users/.

[27] Directory: samples/Families. URL https://samples.vx-underground.org/samples/Families.

[28] Rules for naming. URL https://encyclopedia.kaspersky.com/knowledge/rules-for-naming.

[29] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2, pages 1137–1143, 1995.

[30] M. Mahboubi, J. Smith, and R. Lee. Enhancing adaptive malware detection using machine learning techniques. *Journal of Information Security Research*, 36(2):117–132, 2025. URL https://www.sciencedirect.com/science/article/pii/S1084804525000244.

[31] M. Haidur, A. Brown, and C. Wilson. A comparative study of svm-based and rule-based malware detection methods. *International Journal of Computer Science & Security*, 18(4):45–59, 2024. URL https://ph.pollub.pl/index.php/iapgos/article/download/6366/4697.

[32] C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Springer, 2018.

[33] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2022.

Appendix A

Data filtering Python script

```
def filter_accessible_sample(apkfile: str, dataset: str):
    '''
    Parse the samples AndroidManifest file to extract metadata information about
    the sample and check the mandatory accessibility servic permission to detect
    if the sample is an accessibility sample. All decoded information is stored
    to a CSV dataset file.

    Params:
        - apkfile (str): the sample apk file
        - dataset (str): CSV file to store filtered dataset

    Returns:
        - None
    '''

    global accessibility_samples, non_accessibility_samples, invalid_samples
    service_names = []

    # The last folder name before the file is the family name
    family = os.path.basename(os.path.dirname(apkfile))
    hash = calculate_hash(apkfile)

    try:
        apk = APK.from_file(apkfile).parse_resource()
        manifest = apk.get_manifest()
        apk.close()
    except Exception as e:
        invalid_samples += 1

        with open (ERROR_DIR + "exception.txt", "a") as f:
            f.write("\n" + str(e))
```

```
        print(f"{bcolors.FAIL}Failed to decode <{apkfile}> sample.\nError: {e} {bcolors.ENDC}")
        info_data = [
            family,
            'None',
            'None',
            'None',
            'None',
            'None',
            'None',
            'None',
            apkfile,
            hash
        ]
        write_csv_file(dataset, info_data)
        return False

    root = ET.fromstring(manifest)
    services = root.findall(".//service")

    # Mandatory permission to use accessibility services
    PERM = "android.permission.BIND_ACCESSIBILITY_SERVICE"

    # Iterate through service elements
    for service in services:
        try:
            if service.get('{http://schemas.android.com/apk/res/android}permission') == PERM:
                name = service.get('{http://schemas.android.com/apk/res/android}name')
                if name:
                    service_names.append(name)
        except:
            pass

    uses_accessibility = len(service_names) > 0

    # if the service starts with '.' add the package name before it
    for idx, s in enumerate(service_names):
        if s and s.startswith('.'):
            service_names[idx] = apk.package_name + s

    # Write decode information to dataset
    info_data = [
        family,
        str(apk.package_name),
        str(apk._version_code),
        str(apk._version_name),
        str(apk._min_sdk_version),
        str(apk._target_sdk_version),
        int(uses_accessibility),
```

```
        '|'.join(service_names),
        apkfile,
        hash
    ]
    write_csv_file(dataset, info_data)

    if uses_accessibility:
        accessibility_samples += 1
        print(f"{bcolors.HEADER}Found Accessibility Service {bcolors.ENDC}")
        print(f"{bcolors.OKGREEN} Family: {info_data[0]} {bcolors.ENDC}")
        print(f"{bcolors.OKGREEN} Package: {info_data[1]} {bcolors.ENDC}")
        print(f"{bcolors.OKGREEN} Version Code: {info_data[2]} {bcolors.ENDC}")
        print(f"{bcolors.OKGREEN} Version Name: {info_data[3]} {bcolors.ENDC}")
        print(f"{bcolors.OKGREEN} Min SDK: {info_data[4]} {bcolors.ENDC}")
        print(f"{bcolors.OKGREEN} Target SDK: {info_data[5]} {bcolors.ENDC}")
        print(f"{bcolors.OKCYAN}  Accessibility Service: {info_data[7]} {bcolors.ENDC}")
        print(f"{bcolors.OKGREEN} File Name: {info_data[8]} {bcolors.ENDC}")
        print(f"{bcolors.OKGREEN} SHA256 Hash: {info_data[9]} {bcolors.ENDC}")
    else:
        non_accessibility_samples += 1
        print(f"{bcolors.FAIL}Not Accessibility Sample {bcolors.ENDC}")

    return uses_accessibility


def filter_accessibility_samples(sample_dir:str, dataset: str):
    '''
    Prepare a dataset from collected malware samples. The dataset will filter
    samples that use accessibility services.

    Params:
        - sample_dir (str): directory that contains the sample malwares
        - dataset (str): CSV file to store filtered dataset

    Returns:
        - None
    '''

    print(f"{bcolors.HEADER}Filtering samples started...{bcolors.ENDC}")

    # Create a dataset CSV file and write the header
    create_accessibility_csv_file(dataset)

    # Load all files in the given malware sample folder
    files = glob.glob(sample_dir + '/**/*', recursive=True)
    for f in files:
        if not os.path.isdir(f):
            filter_accessible_sample(f, dataset)
```

```
all_samples = accessibility_samples + non_accessibility_samples + invalid_samples
print(f"{bcolors.HEADER}Total samples: {all_samples}{bcolors.ENDC}")
print(f"{bcolors.HEADER} Accessibility: {accessibility_samples}{bcolors.ENDC}")
print(f"{bcolors.HEADER} Non-Accessibility: {non_accessibility_samples}{bcolors.ENDC}")
print(f"{bcolors.HEADER} Invalid: {invalid_samples} {bcolors.ENDC}")
print(f"{bcolors.HEADER}Filtering samples completed!{bcolors.ENDC}")
```

Appendix B

Capability Analysis Python script

```
def get_flag_value(pattern: str, smali_code: str) -> int:
    # Find matches in the code using the regular expression pattern
    match = re.search(pattern, smali_code)

    # If a match is found, extract the register and print it
    if match:
        # Get the matched line
        matched_line = match.group(0)
        register = re.search(r'iput.*?(\w+),', matched_line).group(1)
        value_pattern = r'const.*?{0}, (.*?0x[0-9a-fA-F]+)'.format(register)

        # Split the code into lines
        lines = smali_code.split('\n')
        lines = [line.strip() for line in lines]

        # Find the index of the matched line
        start_index = lines.index(matched_line)

        # Iterate through the lines of code in reverse order from the matched line index
        for line in reversed(lines[:start_index]):
            asgmt = re.search(value_pattern, line)
            if asgmt:
                # If the value is found, convert it to integer and return
                return int(asgmt.group(1), 16)  # Convert hexadecimal to integer

    return -2   # Return -2 if no value is found


def get_capability_from_smali(smali_file: str, capability_dataset: str, info_data):
    '''
    Function to check if capability is registered dynamically in the accessibility
```

```
    services source code. Only eventTypes, feedbackTypes, and accessibilityFlags
    can be registered dynamically.

    Params:
        - smali_file (str): the smali file to parse for capability
        - capability_dataset (str): capability dataset CSV file
        - info_data (List): row information of packer dataset

    Returns:
        - True if capability found
    '''

    try:
        with open(smali_file, "r") as f:
            smali_code = f.read()

        event_types_value = get_flag_value(r'iput.*?eventTypes:I', smali_code)
        feedback_type_value = get_flag_value(r'iput.*?feedbackType:I', smali_code)
        flag_type_value = get_flag_value(r'iput.*?flags:I', smali_code)

        # Reverse the combined event types back to individual flags
        if event_types_value == -2:
            event_types = None
        elif event_types_value == -1:
            event_types = "typeAllMask"
        else:
            event_types = [e for e, f in event_types_flags.items() if event_types_value & f]
            event_types = '|'.join(event_types)

        if feedback_type_value == -2:
            feedback_type = None
        elif feedback_type_value == -1:
            feedback_type = "feedbackAllMask"
        else:
            feedback_type = [e for e, f in feedback_type_flags.items() if feedback_type_value &
            feedback_type = '|'.join(feedback_type)

        if flag_type_value == -2:
            flag_type = None
        else:
            flag_type = [e for e, f in flag_type_flags.items() if flag_type_value & f]
            flag_type = '|'.join(flag_type)

        data = [
            info_data[0],   # family
            info_data[1],   # pkg
            info_data[2],   # srvc
            event_types,
```

```
            feedback_type,
            flag_type,
            0,0,0,0,0,
            info_data[5],   # path
            info_data[6]    # hash
        ]
        write_csv_file(capability_dataset, data)
        return True
    except Exception as e:
        print(f"{bcolors.WARNING}Error reading smali file: {e}{bcolors.ENDC}")
        return False


def get_capability_from_xml(xml_file: str, capability_dataset: str, info_data):
    '''
    Function to parse the XML file containing accessibility capabilities.

    Params:
        - xml_file (str): the XML file to parse for capability
        - capability_dataset (str): capability dataset CSV file
        - info_data (List): row information of packer dataset

    Returns:
        - True if capability found
    '''

    try:
        tree = ET.parse(xml_file)
        root = tree.getroot()

        # Find accessibility-service element
        if root.tag == 'accessibility-service':
            service = root
        else:
            return False

        # service = root.get("accessibility-service")
        # Extract values from the accessibility-service element
        event_types = service.get('{http://schemas.android.com/apk/res/android}accessibilityEven
        feedback_type = service.get('{http://schemas.android.com/apk/res/android}accessibilityFe
        accessibility_flags = service.get('{http://schemas.android.com/apk/res/android}accessibi
        can_retrieve_window_content = service.get('{http://schemas.android.com/apk/res/android}c
        can_perform_gestures = service.get('{http://schemas.android.com/apk/res/android}canPerfo
        can_request_touch_exploration_mode = service.get('{http://schemas.android.com/apk/res/an
        can_request_enhanced_webAccessibility = service.get('{http://schemas.android.com/apk/res
        can_request_filter_key_events = service.get('{http://schemas.android.com/apk/res/android

        if not event_types and not feedback_type and not accessibility_flags and not can_retriev
```

```
            return False

        crwc = 1 if can_retrieve_window_content == "true" else 0
        cpg = 1 if can_perform_gestures == "true" else 0
        crtem = 1 if can_request_touch_exploration_mode == "true" else 0
        crew = 1 if can_request_enhanced_webAccessibility == "true" else 0
        crfke = 1 if can_request_filter_key_events == "true" else 0

        data = [
            info_data[0],   #family
            info_data[1],   #pkg
            info_data[2],   #srvc
            event_types,
            feedback_type,
            accessibility_flags,
            crwc,
            cpg,
            crtem,
            crew,
            crfke,
            info_data[5],   # path
            info_data[6]    # hash
        ]
        write_csv_file(capability_dataset, data)
        return True
    except Exception as e:
        print(f"{bcolors.WARNING}Error parsing XML file: {e}{bcolors.ENDC}")

    return False


def parse_capabilities(packer_dataset: str, capability_dataset: str):
    '''
    Parse capabilities registered by samples that use accessibility services.

    Inputs:
        - packer_dataset (str): packer dataset CSV file
        - capability_dataset (str): capability dataset CSV file

    Returns:
        - None
    '''

    global total_no_capability
    print(f"{bcolors.HEADER}Capability parsing started...{bcolors.ENDC}")

    # Create capability dataset CSV file
    create_capability_csv_file(capability_dataset)
```

```
# Read packer dataset CSV file
csv_data = read_csv_file(packer_dataset)
for row in csv_data:
    got_accessibility = True
    apkfile = row[5]
    if apkfile != '' and apkfile != 'apk_file_name':
        got_accessibility = False

        print(f"{bcolors.OKBLUE}Processing {apkfile} ...")

        # Decompile only resource files with '-s' flag since we need only the xml file
        if decompile_apk(apkfile, "d -s -f", "output") == 0:
            # Check all versions of xml directories. Apps can separately store different xml
            # files depending on the API version they target.
            all_dirs = [d for d in os.listdir(RES_OUTPUT) if os.path.isdir(os.path.join(RES_
            xml_paths = [d for d in all_dirs if d.startswith("xml")]

            # The higher version holds more supported values of capabilities
            xml_paths = reversed(xml_paths)
            for p in xml_paths:
                if got_accessibility:
                        break

                full_p = RES_OUTPUT + p + '/'
                xml_files = glob.glob(full_p + '/*.xml')
                for xml in xml_files:
                    got_accessibility = get_capability_from_xml(xml, capability_dataset, row
                    if got_accessibility:
                        break

    if not got_accessibility:
        # Didn't get capability configuration from XML file
        # Check if the capability is registered dynamically inside the acessibility
        # service decompiled smali source code
        if row[3] == '0' or (row[3] == '1' and row[4] == '1'):
            # Not packed sample or packed but unpacked successfully
            # Decompile the source code since we need the smali code
            if decompile_apk(row[5], "d -f", "output") == 0:
                smali_path = f"output/"
                services = str(row[2]).split('|')

                # Get all smali directories for decompiled sample
                all_dirs = [d for d in os.listdir(smali_path) if os.path.isdir(os.path.join(
                smali_paths = [d for d in all_dirs if d.startswith("smali")]
                for p in smali_paths:
                    if got_accessibility:
                        break
```

```
            for s in services:
                # Build a path by converting the class code to path
                smali_file = smali_path + p + '/' + s.replace(".", "/") + ".smali"
                if os.path.isfile(smali_file):
                    got_accessibility = get_capability_from_smali(smali_file, capabi
                    if got_accessibility:
                        break
                else:
                    print(f"{bcolors.HEADER} Ignoring non existing smali file: {smal
    else:
        total_no_capability += 1
        print(f"{bcolors.HEADER} No capability for {apkfile}.{bcolors.ENDC}")
else:
    total_no_capability += 1
    print(f"{bcolors.HEADER} No capability for {apkfile}.{bcolors.ENDC}")
```

Appendix C

VirusTotal Batch Scanner Python script

```
import requests
import hashlib
import os
import csv
import time
import pandas as pd

# VirusTotal API URL
VT_API_URL = "https://www.virustotal.com/api/v3/files/"

# VirusTotal API key
API_KEY = '8396bb4fb6c98e03...'

headers = {
    "x-apikey": API_KEY
}

not_detected = 0

#
def get_file_hash(file_path):
    '''
    Function to get the file SHA256 hash
    '''

    sha256_hash = hashlib.sha256()
    with open(file_path, "rb") as f:
```

```
        # Read the file in chunks of 4K to avoid memory issues with large files
        for byte_block in iter(lambda: f.read(4096), b""):
            sha256_hash.update(byte_block)
    return sha256_hash.hexdigest()

def scan_file(file_path):
    '''
    Function to scan a file and get the results from VirusTotal
    '''
    file_hash = get_file_hash(file_path)
    url = VT_API_URL + file_hash

    response = requests.get(url, headers=headers)
    if response.status_code == 200:
        json_data = response.json()

        if json_data.get('data'):
            detections = json_data['data']['attributes']['last_analysis_results']
            detection_info = {}

            # Collect detections only
            for engine, result in detections.items():
                if result['category'] != 'undetected' and result['category'] != 'type-unsupporte
                    detection_info[engine] = result['category']

            # Only return results if there were detections
            if detection_info:
                return file_hash, detection_info
            else:
                # No detections, don't store this file
                not_detected += 1
                return None
        else:
            print(f"File not detected in VirusTotal: {file_path}")
            return None
    else:
        print(f"Error fetching results for {file_path}. Status code: {response.status_code}")
        return None

def save_to_csv(filename, file_hash, detections, output_file):
    '''
    Function to store a detection result in CSV after each successful scan
    '''

    fieldnames = ['File Name', 'File Hash', 'Engine', 'Detection']

    # Write the results to CSV after each file is processed
    with open(output_file, mode='a', newline='') as file:
```

```
        writer = csv.writer(file)

        # Write header if the file is empty
        if file.tell() == 0:
            writer.writerow(fieldnames)

        # Write the detection info for each engine
        for engine, detection in detections.items():
            writer.writerow([filename, file_hash, engine, detection])

def process_samples(sample_dir, output_file):
    '''
    Function to process multiple malware samples with rate limit control
    '''
    request_counter = 0

    # Time of the last request
    last_request_time = time.time()

    for filename in os.listdir(sample_dir):
        file_path = os.path.join(sample_dir, filename)

        if os.path.isfile(file_path):
            # Check if we need to wait for rate limit (4 requests per minute)
            current_time = time.time()
            elapsed_time = current_time - last_request_time

            # If 60 seconds have passed and we've made 4 requests, wait for the next minute
            if request_counter >= 4 and elapsed_time < 60:
                wait_time = 60 - elapsed_time
                print(f"Rate limit reached. Waiting for {wait_time} seconds.")
                time.sleep(wait_time)
                request_counter = 0
                last_request_time = time.time()

            print(f"Processing {filename}...")
            result = scan_file(file_path)
            if result:
                file_hash, detection_info = result
                # Save the result immediately
                save_to_csv(filename, file_hash, detection_info, output_file)
            request_counter += 1
            last_request_time = time.time()

            # Wait for a short time between requests
            time.sleep(1)

if __name__ == "__main__":
```

```
sample_dir = "accessibility_only_samples"
process_samples(sample_dir, "malware_results.csv")

print(f"{not_detected} samples are not detected as malicious")

# Read the CSV file
df = pd.read_csv("malware_results.csv")

# Filter out only malicious detections
df_malicious = df[df['Detection'] == 'malicious']

# Count how many engines flagged each sample as malicious, based on File Hash
result = df_malicious.groupby('File Hash')['Engine'].nunique()

print(result)
```

Appendix D

Base Dataset Validation Script

```
import numpy as np
import pandas as pd
from tqdm import tqdm
from sklearn.metrics.pairwise import euclidean_distances
from sentence_transformers import SentenceTransformer

def calculate_adjacent_distance(base_df, subtitle):
    # Load SentenceTransformer model
    model = SentenceTransformer('all-MiniLM-L6-v2')

    # Generate embeddings for each text
    tqdm.pandas(desc="Generating Embeddings")
    base_df['embedding'] = base_df['text'].progress_apply(lambda x: model.encode(x))

    # Convert texts into sets of API calls
    base_df['api_set'] = base_df['text'].apply(lambda x: set(x.split()))

    # Group embeddings and API sets by label
    label_groups = base_df.groupby('label')
    label_to_embeddings = {label: np.stack(group['embedding'].tolist()) for label, group in labe

    # Calculate adjacent cluster Euclidean distance
    def calculate_adjacent_euclidean_distance(label_to_embeddings):
        sorted_labels = sorted(label_to_embeddings.keys())  # Sort labels for adjacency
        adjacent_distances = []
        for i in range(len(sorted_labels) - 1):
            emb1 = np.mean(label_to_embeddings[sorted_labels[i]], axis=0).reshape(1, -1)  # Resh
            emb2 = np.mean(label_to_embeddings[sorted_labels[i + 1]], axis=0).reshape(1, -1)  #
            distance = euclidean_distances(emb1, emb2)[0][0]  # Compute distance between cluster
            adjacent_distances.append((sorted_labels[i], sorted_labels[i + 1], distance))
```

```
        # Add comparison between the last and the first label
        centroid_last = np.mean(label_to_embeddings[sorted_labels[-1]], axis=0).reshape(1, -1)
        centroid_first = np.mean(label_to_embeddings[sorted_labels[0]], axis=0).reshape(1, -1)
        similarity_last_first = euclidean_distances(centroid_last, centroid_first)[0, 0]
        adjacent_distances.append((sorted_labels[-1], sorted_labels[0], similarity_last_first))
        return adjacent_distances

    # Calculate adjacent-cluster Euclidean distance
    adjacent_euclidean = calculate_adjacent_euclidean_distance(label_to_embeddings)

    # Print adjacent-cluster Euclidean distances
    print(f"Adjacent-Cluster Euclidean Distance {subtitle}")
    for label1, label2, distance in adjacent_euclidean:
        print(f"Euclidean distance between {label1} and {label2}: {distance:.4f}")

base_df_orig = pd.read_csv("../data/base_patterns_orig.csv", quotechar="'")
calculate_adjacent_distance(base_df_orig, "before parameter tunning ")

base_df_opt = pd.read_csv("../data/base_patterns_opt.csv", quotechar="'")
calculate_adjacent_distance(base_df_opt, "after parameter tunning")
```

Appendix E

Base Dataset Validation Results

```
Generating Embeddings: 100%|██████████| 134/134 [00:03<00:00, 34.36it/s]
Adjacent-Cluster Euclidean Distance before parameter tunning
Euclidean distance between BlockAccess and ContentEavesdrop: 0.4298
Euclidean distance between ContentEavesdrop and ManipulateUI: 0.4767
Euclidean distance between ManipulateUI and BlockAccess: 0.2066
Generating Embeddings: 100%|██████████| 134/134 [00:01<00:00, 82.10it/s]
Adjacent-Cluster Euclidean Distance after parameter tunning
Euclidean distance between BlockAccess and ContentEavesdrop: 0.5874
Euclidean distance between ContentEavesdrop and ManipulateUI: 0.6812
Euclidean distance between ManipulateUI and BlockAccess: 0.3178
```

Figure E.1: Base dataset validation experiment result.

Appendix F

Data Augmentation Script

```
warnings.filterwarnings('ignore')

edge_color = 'black'
colors = ['#99ff99', '#66b3ff', '#ff9999', '#ffcc99', '#2ecc71', '#3498db', '#f1c40f', '#f39c12'

base_pattern_file = '../data/base_patterns_opt.csv'
flowdroid_file = '../data/callgraph_flowdroid_opt.csv'

# Load base patterns with known labeled patterns
base_df = pd.read_csv(base_pattern_file, quotechar="'")

# Load FlowDroid patterns
flowdroid_df = pd.read_csv(flowdroid_file, quotechar="'")

# Combine datasets and add labels for FlowDroid patterns
combined_df = pd.concat([base_df, flowdroid_df], ignore_index=True)
combined_df['label'] = combined_df['label'].fillna('Unlabeled')

# Count total records
total_flowdroid_records = flowdroid_df.shape[0]
print("Total Flowdroid Records:", total_flowdroid_records)

from sentence_transformers import SentenceTransformer
model = SentenceTransformer('paraphrase-MiniLM-L6-v2')
bert_embeddings = model.encode(combined_df['text'].tolist())

# Dimensionality reduction to enhance differentiation
pca = PCA(n_components=50)
reduced_embeddings = pca.fit_transform(bert_embeddings)
```

```
# Generate mean embeddings for each label in the base dataset
labels_map = {"BlockAccess": 0, "ManipulateUI": 1, "ContentEavesdrop": 2}
mean_embeddings = {}

for label in labels_map:
    label_texts = base_df[base_df['label'] == label]['text']
    label_embeddings = np.array([model.encode(text) for text in label_texts])
    # PCA on mean embedding
    mean_embeddings[label] = pca.transform([label_embeddings.mean(axis=0)])[0]

# Calculate similarities and assign labels with a more precise threshold
assigned_labels = []
for embedding in reduced_embeddings:
    similarities = {label: cosine_similarity([embedding],
        [mean_embeddings[label]])[0][0] for label in labels_map}
    best_label, best_similarity = max(similarities.items(), key=lambda x: x[1])

    # Only assign a label if similarity meets the threshold
    if best_similarity >= 0.7:
        assigned_labels.append(best_label)
    else:
        assigned_labels.append("No Similar Pattern")

# Add the assigned labels to the dataframe and save
combined_df['label'] = assigned_labels

# Save only the labeled patterns (excluding those marked 'No Similar Pattern')
filtered_df_bert = combined_df[combined_df['label'] != "No Similar Pattern"]

# Apply single quotes to each value in 'text' and 'label'
filtered_df_bert['text'] = filtered_df_bert['text'].apply(lambda x: f"'{x}'")
filtered_df_bert['label'] = filtered_df_bert['label'].apply(lambda x: f"'{x}'")

# Save to CSV with fields enclosed in single quotes
filtered_df_bert.to_csv('../data/patterns_all.csv', index=False, quotechar="'", escapechar='\\',

# Fuzzy Matching with Cosine Similarity Thresholds
def compute_similarity_scores(base_df, flowdroid_df):
    matched_labels = []

    # Iterate through FlowDroid patterns
    for idx, flow_text in enumerate(flowdroid_df['text']):
        max_similarity = 0
        best_label = 'No Similar Pattern'

        # Compare each FlowDroid pattern with all base patterns using fuzzy ratio
        for base_text, label in zip(base_df['text'], base_df['label']):
            similarity = fuzz.ratio(flow_text, base_text) / 100
```

```
            if similarity > max_similarity and similarity >= similarity_threshold:
                max_similarity = similarity
                best_label = label

        matched_labels.append(best_label)

    return matched_labels

# Apply similarity scoring to FlowDroid patterns
flowdroid_df['fuzzy_label'] = compute_similarity_scores(base_df, flowdroid_df)

# Merge fuzzy labels into combined_df
combined_df.loc[combined_df.index >= len(base_df), 'fuzzy_label'] = flowdroid_df['fuzzy_label'].

# Map DBSCAN clusters to base pattern labels using the fuzzy matching results
combined_df['cluster_label'] = np.where(dbscan_labels == -1, 'No Similar Pattern', combined_df['

cluster_distribution_bert = filtered_df_bert['label'].value_counts()
print("BERT - Pattern Distribution per Cluster Label:\n", cluster_distribution_bert)

labels = ['BlockAccess', 'ManipulateUI', 'ContentEavesdrop']

# Define bar width and x-axis positions
bar_width = 0.35
x = np.arange(len(labels))

# Plot the bar charts
fig, ax = plt.subplots(figsize=(10, 6))
bars1 = ax.bar(x - bar_width / 2, cluster_distribution_bert, bar_width,
    label='BERT Clustering', color=colors[0], edgecolor=edge_color)

# Add labels, title, and ticks
ax.set_ylabel('Number of Patterns')
# ax.set_title('Pattern Distribution per Cluster Label (BERT vs DBSCAN)')
ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.legend()

# Add value labels above bars
def add_labels(bars):
    for bar in bars:
        yval = bar.get_height()
        ax.text(bar.get_x() + bar.get_width() / 2, yval + 10, f'{int(yval)}', ha='center', va='b

add_labels(bars1)

plt.tight_layout()
plt.show()
```

Appendix G

Model Validation Script

```
labels_map = {"BlockAccess": 0, "ManipulateUI": 1, "ContentEavesdrop": 2}
class_weights = {0: 2.16, 1: 1.0, 2: 1.0}  # example weights (adjust based on imbalance ratio)

# Define vectorization function for XGBoost
def tfidf_vectorization(texts):
    vectorizer = TfidfVectorizer(max_features=5000, ngram_range=(1, 2))
    return vectorizer.fit_transform(texts).toarray()

# Calculate class weight for XGBoost class inbalance correction
def calculate_class_weight():
     # Label distribution of the dataset
    total_samples = 999 + 282 + 216
    class_distribution = {
        "BlockAccess": 282,
        "ManipulateUI": 999,
        "ContentEavesdrop": 216
    }

    # Calculate the weights inversely proportional to the class frequencies
    class_weights = {
        0: total_samples / class_distribution["BlockAccess"],
        1: total_samples / class_distribution["ManipulateUI"],
        2: total_samples / class_distribution["ContentEavesdrop"]
    }

    return class_weights

# Define XGBoost model with tuned parameters
xg_model = XGBClassifier(
    objective='multi:softmax',
    num_class=3,
```

```
    eval_metric='mlogloss',
    use_label_encoder=False,
    learning_rate=0.1,
    max_depth=6,
    n_estimators=100,
    scale_pos_weight=calculate_class_weight()  # Adjusted weights
)

# Cross-Validation Setup
kf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
xg_accuracy_scores, xg_f1_scores, xg_all_preds, xg_all_labels = [], [], [], []
tb_accuracy_scores, tb_f1_scores, tb_all_preds, tb_all_labels = [], [], [], []

# Prepare dataset for XGBoost
xg_X = tfidf_vectorization(df['text'])
tb_X = df['text']
y = df['label']

fold = 1

# XGBoost Cross-Validation
print("\nStarting 10-fold Cross-Validation for XGBoost...")
for train_index, test_index in kf.split(xg_X, y):
    X_train, X_val = xg_X[train_index], xg_X[test_index]
    y_train, y_val = y[train_index], y[test_index]

    # Train and evaluate on each fold
    xg_model.fit(X_train, y_train)
    y_pred = xg_model.predict(X_val)

    # Store results for later summary
    accuracy = accuracy_score(y_val, y_pred)
    f1 = f1_score(y_val, y_pred, average='weighted')
    xg_accuracy_scores.append(accuracy)
    xg_f1_scores.append(f1)
    xg_all_preds.extend(y_pred)
    xg_all_labels.extend(y_val)

    print(f"Fold {fold} - Accuracy: {accuracy:.4f}, F1 Score: {f1:.4f}")
    fold += 1

# TinyBERT Cross-Validation
print("\nStarting 10-fold Cross-Validation for TinyBERT...")
for train_index, val_index in kf.split(tb_X, y):
    train_texts, val_texts = tb_X[train_index], tb_X[val_index]
    train_labels, val_labels = y[train_index], y[val_index]

    # Use custom word-level tokenizer
```

```
    train_encodings = {’input_ids’: [], ’attention_mask’: []}
    val_encodings = {’input_ids’: [], ’attention_mask’: []}

    for text in train_texts.tolist():
        enc = tokenizer_word.encode_plus(text, truncation=True, padding=True, max_length=512)
        train_encodings[’input_ids’].append(enc[’input_ids’])
        train_encodings[’attention_mask’].append(enc[’attention_mask’])

    for text in val_texts.tolist():
        enc = tokenizer_word.encode_plus(text, truncation=True, padding=True, max_length=512)
        val_encodings[’input_ids’].append(enc[’input_ids’])
        val_encodings[’attention_mask’].append(enc[’attention_mask’])

    train_dataset = TextDataset(train_encodings, train_labels.tolist())
    val_dataset = TextDataset(val_encodings, val_labels.tolist())
    train_loader = DataLoader(train_dataset, batch_size=10, shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=10, shuffle=False)

    # Initialize model
    model_tinybert, optimizer, device = initialize_tiny_bert_model()
    train_tiny_bert_model(model_tinybert, optimizer, device, train_loader)

    # Evaluate TinyBERT
    preds, labels_list = evaluate_tiny_bert_model(model_tinybert, device, val_loader)
    tb_accuracy_scores.append(accuracy_score(labels_list, preds))
    tb_f1_scores.append(f1_score(labels_list, preds, average=’weighted’))
    tb_all_preds.extend(preds)
    tb_all_labels.extend(labels_list)

# Summary of Cross-Validation Results
xg_mean_accuracy = np.mean(xg_accuracy_scores)
xg_mean_f1_score = np.mean(xg_f1_scores)
xg_std_accuracy = np.std(xg_accuracy_scores)
xg_std_f1_score = np.std(xg_f1_scores)

tb_mean_accuracy = np.mean(tb_accuracy_scores)
tb_mean_f1_score = np.mean(tb_f1_scores)
tb_std_accuracy = np.std(tb_accuracy_scores)
tb_std_f1_score = np.std(tb_f1_scores)


print("\n10-Fold Cross-Validation Results of XGBoost:")
print(f"Average Accuracy: {xg_mean_accuracy:.4f}    {xg_std_accuracy:.4f}")
print(f"Average F1 Score: {xg_mean_f1_score:.4f}    {xg_std_f1_score:.4f}")

print("\n10-Fold Cross-Validation Results of TinyBERT:")
print(f"Average Accuracy: {tb_mean_accuracy:.4f}    {tb_std_accuracy:.4f}")
print(f"Average F1 Score: {tb_mean_f1_score:.4f}    {tb_std_f1_score:.4f}")
```

```
# Overall Classification Report and Confusion Matrix
print("\nOverall Classification Report of XGBoost:")
print(classification_report(xg_all_labels, xg_all_preds, target_names=labels_map.keys()))

# Overall Classification Report and Confusion Matrix
print("\nOverall Classification Report of TinyBERT:")
print(classification_report(tb_all_labels, tb_all_preds, target_names=labels_map.keys()))

xg_conf_matrix = confusion_matrix(xg_all_labels, xg_all_preds)
tb_conf_matrix = confusion_matrix(tb_all_labels, tb_all_preds)
# Plot confusion matrices for both models
fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# XGBoost Confusion Matrix
sns.heatmap(xg_conf_matrix, annot=True, fmt='.0f', cmap='Blues', ax=axes[0])
# axes[0].set_title('XGBoost - Average Confusion Matrix')
axes[0].set_xlabel('Predicted Labels')
axes[0].set_ylabel('True Labels')

# TinyBERT Confusion Matrix
sns.heatmap(tb_conf_matrix, annot=True, fmt='.0f', cmap='Greens', ax=axes[1])
# axes[1].set_title('TinyBERT - Average Confusion Matrix')
axes[1].set_xlabel('Predicted Labels')
axes[1].set_ylabel('True Labels')

plt.tight_layout()
plt.show()

# Plotting results
models = ['XGBoost', 'TinyBERT']
accuracy_scores = [xg_mean_accuracy, tb_mean_accuracy]
f1_scores = [xg_mean_f1_score, tb_mean_f1_score]

fig, ax = plt.subplots(figsize=(8, 6))
x = np.arange(len(models))
width = 0.35

# Plot bars with error bars
bar1 = ax.bar(x - width/2, accuracy_scores, width, label='Accuracy', color=colors[0], edgecolor=
bar2 = ax.bar(x + width/2, f1_scores, width, label='F1 Score', color=colors[1], edgecolor=edge_c

# Set labels, title, and display values
ax.set_xticks(x)
ax.set_xticklabels(models)
ax.set_ylim(0.8, 1.0)
ax.set_ylabel('Score')
ax.legend()
```

```
# Add data labels on top of bars
def add_labels(bars):
    for bar in bars:
        yval = bar.get_height()
        ax.text(bar.get_x() + bar.get_width()/2, yval + 0.01, f'{yval:.2f}', ha='center', va='bo

add_labels(bars1)
add_labels(bars2)

# Show plot
plt.show()
```

Appendix H

Model Validation Results

```
thesis_experiment.ipynb   thesis_experiment.ipynb (output)
 6
 7   10-Fold Cross-Validation Results of XGBoost:
 8   Average Accuracy: 0.9038 ± 0.0200
 9   Average F1 Score: 0.9000 ± 0.0208
10
11   10-Fold Cross-Validation Results of TinyBERT:
12   Average Accuracy: 0.9766 ± 0.0086
13   Average F1 Score: 0.9767 ± 0.0086
14
15   Overall Classification Report of XGBoost:
16                     precision    recall  f1-score   support
17
18        BlockAccess       0.83      0.67      0.74       282
19       ManipulateUI       0.91      0.96      0.93       999
20   ContentEavesdrop       0.96      0.94      0.95       216
21
22           accuracy                           0.90      1497
23          macro avg       0.90      0.86      0.87      1497
24       weighted avg       0.90      0.90      0.90      1497
25
26
27   Overall Classification Report of TinyBERT:
28                     precision    recall  f1-score   support
29
30        BlockAccess       0.93      0.96      0.94       282
31       ManipulateUI       0.99      0.98      0.98       999
32   ContentEavesdrop       0.98      0.99      0.98       216
33
34           accuracy                           0.98      1497
35          macro avg       0.97      0.97      0.97      1497
36       weighted avg       0.98      0.98      0.98      1497
37
38
```

FIGURE H.1: Model validation experiment result.