PRESIDENCY UNIVERSITY
Private University Estd. in Karnataka State by Act No. 41 of 2013
Itgalpura, Rajankunte, Yelahanka, Bengaluru – 560064

ANALYSIS AND IDENTIFICATION OF MALICIOUS MOBILE APPLICATIONS USING MACHINE LEARNING

A PROJECT REPORT

Submitted by

BHOOMIKA S HORAPETI - 20221COM0165
ROSHAN RAJU KK - 20221COM0144
AMOGH ARUN MALAGE - 20221COM0171

Under the Supervision of,

Mr. SUNIL KUMAR SAHOO

Assistant Professor

School of Computer Science and Engineering

BACHELOR OF TECHNOLOGY

IN

COMPUTER ENGINEERING

PRESIDENCY UNIVERSITY

BENGALURU

DECEMBER 2025

PRESIDENCY UNIVERSITY

Private University Estd. in Karnataka State by Act No. 41 of 2013

Itgalpura, Rajankunte, Yelahanka, Bengaluru – 560064

PRESIDENCY SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

BONAFIDE CERTIFICATE

Certified that this report **"ANALYSIS AND IDENTIFICATION OF MALICIOUS MOBILE APPLICATIONS USING MACHINE LEARNING"** constitutes bonafide work completed by **BHOOMIKA S HORAPETI (20221COM0165), ROSHAN RAJU KK (20221COM0144), and AMOGH ARUN MALAGE (20221COM0171)**, who have successfully carried out the project work and submitted this report for partial fulfillment of the requirements for the award of the degree of **BACHELOR OF TECHNOLOGY** in **COMPUTER ENGINEERING** during 2025-26.

Mr. Sunil Kumar Sahoo K	**Ms.Benitha Christinal J**	**Dr. Sampath A** **Dr. Geetha A**	**Dr. Pallavi R**
Project Guide	Program Project Coordinator	School Project Coordinators	Head of the Department
PSCS	PSCS	PSCS	PSCS
Presidency University	Presidency University	Presidency University	Presidency University

Dr. Shakkeera L	**Dr. Duraipandian N**
Associate Dean	Dean
PSCS	PSCS & PSIS
Presidency University	Presidency University

Examiners

Sl. No	Name	Signature	Date
1	Mr. Midhun Lal V S		
2	Mr. Asif Ahmad Najar		

PRESIDENCY UNIVERSITY

PRESIDENCY SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

DECLARATION

We the students of final year B.Tech in **COMPUTER ENGINEERING** at Presidency University, Bengaluru, named **BHOOMIKA S HORAPETI (20221COM0165), ROSHAN RAJU KK (20221COM0144), and AMOGH ARUN MALAGE (20221COM0171)**, hereby declare that the project work titled "**ANALYSIS AND IDENTIFICATION OF MALICIOUS MOBILE APPLICATIONS USING MACHINE LEARNING**" has been independently carried out by us and submitted in partial fulfillment for the award of the degree of B.Tech in **COMPUTER ENGINEERING** during the academic year of 2025-26. Further, the matter embodied in the project has not been submitted previously by anybody for the award of any Degree or Diploma to any other institution.

BHOOMIKA S HORAPETI **USN: 20221COM0165**

ROSHAN RAJU KK **USN: 20221COM0144**

AMOGH ARUN MALAGE **USN: 20221COM0171**

PLACE: BENGALURU

DATE: November 2025

ACKNOWLEDGEMENT

We want to express our sincere thanks to all who helped us throughout this project work. Without the guidance and support we got from so many people this work would not have been possible at all honestly.

We extend our heartfelt thanks to our beloved **Chancellor, Pro-Vice Chancellor, and Registrar** for their continuous motivation and support during the project completion time.

We sincerely thank our internal guide **Mr. Sunil Kumar Sahoo, Assistant Professor**, Presidency School of Computer Science and Engineering, Presidency University, for his invaluable guidance and moral support and encouragement throughout the entire duration of our project work. His timely inputs and mentorship helped a lot in shaping this research and we are really grateful for that.

We remain deeply grateful to **Dr. Pallavi R, Professor and Head of the Department**, Computer Engineering, Presidency University, for her mentorship and constant encouragement throughout.

We express our sincere appreciation to **Dr. Duraipandian N**, Dean PSCS & PSIS, and **Dr. Shakkeera L**, Associate Dean, Presidency School of Computer Science and Engineering, along with the Management of Presidency University for providing the necessary facilities and intellectually stimulating environment that helped in the completion of our project work.

We are grateful to **Ms.Benitha Christinal J**, Program Project Coordinator, and **Dr. Sampath A K** and **Dr. Geetha A**, School Project Coordinators, Presidency School of Computer Science and Engineering, for facilitating problem statements, coordinating reviews, monitoring progress, and providing valuable support and guidance throughout.

We also thank the Teaching and Non-Teaching staff of Presidency School of Computer Science and Engineering and staff from other departments who extended their valuable help and cooperation.

BHOOMIKA S HORAPETI
ROSHAN RAJU KK
AMOGH ARUN MALAGE

Abstract

So basically when is come to Android apps and security right there is this huge problem that most people dont even realize. Like millions of apps are being uploaded to app stores every year and not all of them are safe. Some of them look completely normal but they are actually malware in disguise. Old antivirus methods that rely on signatures and known patterns they just cant keep up with new types of malware that keep changing and evolving.

That is exactly what we are trying to address in this project where we developed a machine learning based system to detect malicious mobile applications. We used the mh100klabels.csv dataset which contains metadata of more than 100000 Android applications with labels showing whether each app is benign or malicious. The dataset includes information like permissions requested and developer details and download counts and all that stuff.

We went with Random Forest classifier for this because honestly it works really well with high dimensional data and doesn't overfit too easily. The preprocessing involved cleaning up missing values and encoding categorical features and splitting the data into training and testing sets. We trained the model on 70 percent of the data and tested on the remaining 30 percent.

The results were actually quite good. We got an overall accuracy of 90.94 percent which shows the model can distinguish between safe and unsafe apps with decent reliability. For benign apps specifically the precision was 0.97 and recall was 0.92 which means very few safe apps were wrongly flagged as malicious. For malicious apps the recall was 0.81 meaning we caught most of the bad apps though precision was lower at 0.60 which indicates some false positives.

We also built a web application where users can either enter app metadata manually or upload a CSV file with multiple apps to get predictions. The interface shows confidence scores with visual indicators making it easy to understand whether an app is likely safe or not. Whether this completely solves the mobile security problem we cant say for sure but it definitely provides a useful tool for identifying potentially harmful applications before they cause damage.

Contents

List of Figures

List of Tables

Abbreviations

Abbreviation	Full Form
AI	Artificial Intelligence
API	Application Programming Interface
APK	Android Package Kit
AV	Antivirus
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSS	Cascading Style Sheets
CSV	Comma Separated Values
DBN	Deep Belief Network
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation

Chapter 1

Introduction

1.1 Background

So when is come to smartphones and mobile applications right the world has completely changed in the last decade or so. Like everyone has a smartphone now and we use apps for literally everything from banking to ordering food to communicating with friends to even controlling our home appliances. Android specifically dominates the market with around 72 percent global market share as of 2024 which is absolutely massive if you think about it. There are billions of Android devices out there in peoples pockets.

Now here's the thing that really concerns me about this whole situation. The Google Play Store has more than 3.5 million apps available and thousands of new apps are uploaded every single day. And while Google does have security measures in place like Play Protect and manual reviews, malicious apps still manage to slip through. I read somewhere that in 2023 alone Google removed over 2 million policy violating apps from the store. Thats a lot of bad apps that were available for download at some point.

The problem is that traditional antivirus solutions rely on something called signature based detection. What this means is they maintain a database of known malware signatures and when they scan an app they check if it matches any known signatures. This worked okay back in the day but modern malware is way more sophisticated now. Attackers use techniques like code obfuscation and polymorphism and encryption to change how their malware looks each time. So even if you catch one version the next version looks completely different and escapes detection.

Honestly I've seen family members and friends install apps that looked completely legitimate but turned out to be stealing their data or showing intrusive ads. One of my relatives actually had their banking credentials stolen through a fake calculator app can you believe that. The app

looked normal and did basic calculations but in the background it was recording keystrokes. These kinds of stories are becoming more common which is why we decided to work on this project.

Machine learning offers a really promising solution to this problem because it can learn patterns from data rather than relying on fixed rules. If you train a model on thousands of benign and malicious apps it can learn to identify subtle patterns that distinguish the two categories. Even if malware changes its appearance the underlying behavior patterns often remain similar and a good ML model can pick up on that.

1.2 Statistics of Project

Let me share some numbers that really highlight why this project matters.

According to a report by AV-TEST Institute there are more than 1.3 billion malware samples registered globally and over 450,000 new malware and potentially unwanted applications are detected every day [**?**]. When is come to mobile specifically the numbers are equally concerning. Kaspersky detected over 5.7 million mobile malware attacks in 2023 which is actually lower than previous years but still significant.

Here are some more stats that kind of shocked me when I was doing research. Around 24,000 malicious mobile apps are blocked from app stores every single day. Adware accounts for roughly 39 percent of mobile malware followed by trojans at 25 percent and riskware at 22 percent. Banking trojans specifically saw a 100 percent increase in 2023 compared to the previous year. India ranks among the top 5 countries most affected by mobile malware attacks.

For our project specifically we are using the mh100klabels.csv dataset which contains metadata from over 100,000 Android applications. The dataset has roughly 87 percent benign apps and 13 percent malicious apps which represents a realistic distribution you would find in the wild. Most apps are safe but that 13 percent of malicious apps can cause serious harm to users.

The Random Forest classifier we implemented achieved 90.94 percent overall accuracy on this dataset. For the malicious class specifically we got precision of 0.60 and recall of 0.81 which means we catch most of the bad apps even if we have some false positives. In a security context having high recall is often more important because you really dont want to miss actual malware.

1.3 Prior Existing Technologies

So what solutions already exist for detecting malicious Android apps? Let me break it down.

Google Play Protect is the most widely used since it comes built into Android devices. It scans apps before and after installation and also does periodic scans. Google claims it scans over 125 billion apps daily. But even with all that infrastructure malicious apps still get through regularly. Play Protect relies on a combination of signature matching and some machine learning but details of their implementation are proprietary.

Traditional antivirus apps like Norton Mobile Security and Avast Mobile Security and McAfee are also popular. These mostly use signature databases and heuristic analysis. They work well for known malware but struggle with new and zero-day threats. Plus they consume significant battery and system resources which users complain about a lot.

On the academic side there have been several research projects. Drebin developed by Arp et al. is a famous static analysis tool that extracts features from APK files like permissions and API calls and uses SVM for classification. It achieved high detection rates but was developed years ago and newer malware has evolved to evade such tools.

DroidSec by Yuan et al. used deep belief networks for malware detection which was one of the earlier applications of deep learning in this domain. More recent work has explored CNNs on opcode sequences and hybrid approaches combining static and dynamic analysis.

Table 1.1 shows a comparison of major existing solutions.

Table 1.1: Comparison of Existing Malware Detection Systems

System	Approach	Advantages	Limitations
Google Play Protect	Signature + ML	Built-in, free, widespread	Proprietary, still bypassed
Traditional AV Apps	Signature-based	Reliable for known threats	Misses zero-day, resource heavy
Drebin	Static analysis + SVM	Open source, ex-plainable	Outdated, evaded by obfuscation
Deep Learning methods	Neural networks	High accuracy	Needs lots of data, black box

The gap we identified is that most existing solutions either require deep packet inspection or run-time monitoring which is resource intensive. Our approach focuses on metadata based detection which is lightweight and can be done without executing the app at all.

1.4 Proposed Approach

So looking at all these existing solutions and their limitations we decided to build a machine learning based malware detection system that uses application metadata for classification. The

key idea is that malicious apps exhibit certain patterns in their metadata that distinguish them from benign apps.

What do I mean by metadata exactly? Things like the number of permissions requested and what types of permissions. Whether the app asks for sensitive permissions like reading SMS or accessing camera or making phone calls. The developer information and ratings and download counts. Version codes and update frequency. These are all pieces of information that are available without actually running the app.

Our system has these main components:

- Data preprocessing pipeline that handles missing values and encodes categorical features and normalizes numerical features
- Random Forest classifier trained on the mh100klabels.csv dataset with 100,000 plus samples
- Web based interface with two modes one for manual entry where users can input app metadata one at a time and another for CSV upload where users can analyze multiple apps at once
- Visual indicators showing prediction confidence with progress bars and icons making results easy to interpret
- Backend API built with Flask that handles prediction requests and returns results in JSON format

The Random Forest algorithm was chosen for several reasons. It handles high dimensional data well which is important since we have many metadata features. It's resistant to overfitting compared to single decision trees. It provides feature importance scores so we can understand which attributes matter most. And its interpretable unlike black box neural networks.

Figure 1.1 shows the overall system architecture.

1.5 Objectives

Let me clearly state what we are trying to achieve with this project. We thought about this carefully and came up with five main objectives.

First objective is to develop an accurate machine learning model that can classify Android ap-

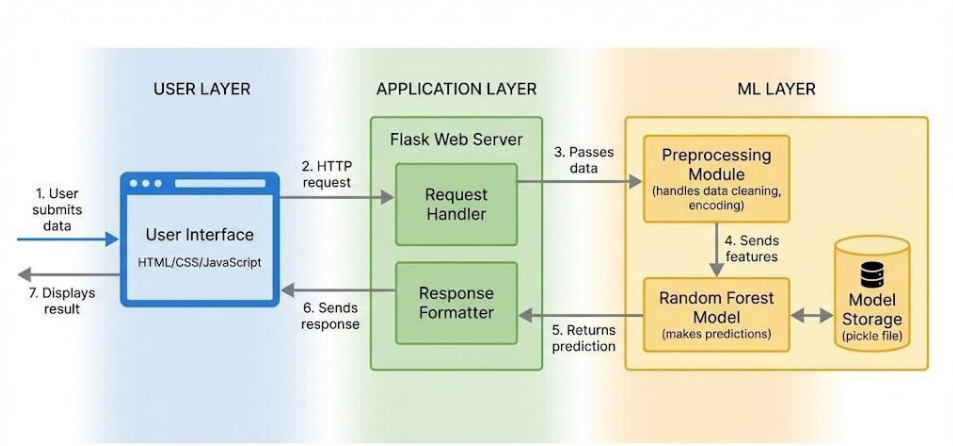

Figure 1.1: Proposed System Architecture

plications as benign or malicious based on metadata features. Accuracy should be above 85 percent overall with high recall for the malicious class to minimize missed detections. We want the model to generalize well to unseen data not just memorize the training set.

Second objective is to create a user friendly web interface that makes the system accessible to non technical users. Someone without programming knowledge should be able to upload app information and get clear predictions. The interface should show confidence levels and visual indicators not just raw numbers.

Third objective is to identify which metadata features are most important for distinguishing malicious apps. Understanding feature importance helps security researchers and app store operators know what red flags to look for. It also validates that the model is learning meaningful patterns.

Fourth objective is to demonstrate that metadata based detection is a viable lightweight alternative to deeper analysis methods. By avoiding runtime analysis and code inspection we reduce computational requirements significantly while still achieving good detection rates.

Fifth objective is to document everything properly so that other researchers can build upon this work. The code and methodology should be reproducible and extendable to other datasets or classification algorithms.

1.6 SDGs

This project connects to several United Nations Sustainable Development Goals which shows its broader relevance beyond just technical achievement.

SDG 9 - Industry, Innovation and Infrastructure: Mobile security is essential for the digital economy to function properly. When users can't trust the apps they download it undermines the entire mobile ecosystem. By developing better detection tools we're contributing to more resilient digital infrastructure. Machine learning for security is definitely technological innovation that can benefit developing and developed nations alike.

SDG 11 - Sustainable Cities and Communities: Smart city initiatives increasingly rely on mobile applications for services like transportation and utilities and civic engagement. If these apps are compromised it affects entire communities. Malware detection helps ensure that urban digital services remain trustworthy and secure for all residents.

SDG 16 - Peace, Justice and Strong Institutions: Cybercrime including mobile malware causes billions of dollars in damages every year and erodes trust in digital systems. Stronger security tools help combat cybercrime and protect individuals and institutions from financial and data theft. This contributes to more just and stable digital societies.

SDG 4 - Quality Education: The techniques we're using in this project including machine learning and data science and software development are highly relevant skills for students. Documenting this work contributes to educational resources that others can learn from. It demonstrates practical applications of classroom concepts.

1.7 Overview of Project Report

Alright so let me give you a quick roadmap of how this report is organized.

Chapter 1 which you just read provides the introduction covering background and statistics and existing solutions and our proposed approach and objectives.

Chapter 2 is the literature review where we dive deeper into existing research on Android malware detection and machine learning techniques. We discuss various approaches that researchers have tried and identify gaps that our work addresses.

Chapter 3 covers methodology explaining how we went about building this system. The development approach and technology choices and system architecture and how we planned to evaluate the system.

Chapter 4 deals with project management including the timeline and risk analysis and budget considerations. Every project needs proper planning and tracking.

Chapter 5 is analysis and design where we detail the requirements both functional and non

functional. Block diagrams and flow charts and design decisions.

Chapter 6 covers software implementation including the tools used and actual code snippets and how we simulated different scenarios.

Chapter 7 presents evaluation and results with test plans and actual test results and screenshots of the working system. We analyze what worked and what didn't.

Chapter 8 discusses broader aspects including social impact and legal considerations and ethics and sustainability and safety.

Chapter 9 is the conclusion summarizing what we achieved and suggesting directions for future work.

Finally we have references and appendices with additional material like source code and user manual.

Chapter 2

Literature Review

So in this chapter I'm going to walk through the existing research on Android malware detection and machine learning techniques. There's actually quite a lot of work that's been done in this area over the past decade and understanding what others have tried helps put our own work in context.

2.1 Evolution of Android Malware

Android malware has evolved significantly since the early days of the platform. Back in 2010 and 2011 when Android was just gaining popularity the malware was pretty basic. Simple trojans that would send premium SMS messages or steal contact lists. Detection was relatively straightforward because attackers weren't really trying to hide what they were doing.

But things changed quickly. By 2013 and 2014 we started seeing more sophisticated attacks. Malware that could root devices and persist even after factory reset. Apps that would request seemingly innocent permissions but abuse them in unexpected ways. The Drebin dataset which was published around this time captured many of these early to mid generation threats.

Tam et al. did a comprehensive survey in 2016 documenting how malware authors adapted their techniques. They noted the rise of code obfuscation where malicious code is transformed to look different while doing the same thing. Encryption of malicious payloads so static analysis cant see what the app actually does. Dynamic code loading where the malware downloads its payload after installation. Polymorphic malware that changes itself each time it spreads.

More recently weve seen malware that specifically targets banking apps. Overlay attacks where a fake login screen is placed over the legitimate banking app. Accessibility service abuse where malware uses Android's accessibility features to read screen content and perform actions. These modern threats are much harder to detect because they often look and behave like legitimate

apps until they activate their malicious functionality.

The key takeaway from all this evolution is that static analysis alone is no longer sufficient. We need detection methods that can generalize to new threats not just recognize known signatures.

2.2 Static Analysis Approaches

Static analysis examines the app without actually running it. You look at the APK file and extract information from the manifest and code and resources. This is attractive because its safe since nothing executes and its fast since you dont need a runtime environment.

Arp et al. developed Drebin which became one of the most cited works in Android malware detection . They extracted features from multiple sources including permissions from the manifest and API calls from the code and hardware components accessed and network addresses contacted. These features were embedded into a high dimensional vector space and classified using SVM. On their dataset they achieved detection rates above 94 percent.

The strength of Drebin was its explainability. The SVM model could identify which specific features contributed to a malicious classification. So a security analyst could understand why an app was flagged. It requested SMS permissions and called suspicious APIs. However Drebin relied on the Dalvik bytecode being readable. If an app uses native code or obfuscation the features cant be extracted properly.

Sahs and Khan proposed a similar approach using permissions and intent filters with SVM classification . They focused specifically on permission patterns since malicious apps often request more permissions than they need. Apps that ask for SMS and call log and camera access but are supposed to be simple games thats suspicious. Their method achieved good accuracy on known malware but the features were manually engineered which limits adaptability.

More recent static analysis work has tried to address the obfuscation problem. Researchers have looked at control flow graphs and function call graphs which are harder to completely obfuscate. The structure of how code flows can reveal malicious patterns even when individual instructions are hidden.

2.3 Dynamic Analysis Approaches

Dynamic analysis actually runs the app and observes its behavior. This overcomes the obfuscation problem because no matter how the code is hidden it eventually has to execute and do something observable.

DroidScope developed by Yan and Yin provided a comprehensive dynamic analysis platform. It ran apps in an instrumented emulator and recorded system calls and network traffic and file operations. By analyzing these runtime behaviors they could detect malware that static analysis would miss. The downside is dynamic analysis is slow and resource intensive. Running each app for enough time to observe malicious behavior takes minutes or hours per app.

Spreitzenbarth et al. created Mobile-Sandbox which automated dynamic analysis at scale. It would install apps in virtual machines trigger various actions like clicking buttons and filling forms and record the resulting behaviors. They built a classification system on top of the behavioral traces. But the fundamental challenge remained since apps might not exhibit malicious behavior during the analysis window. Malware often waits for specific triggers like a certain date or user action.

Su et al. looked specifically at network traffic patterns for detection. Malicious apps often communicate with command and control servers or exfiltrate data to unknown hosts. By monitoring network traffic patterns they could identify suspicious activity. This approach works well for certain malware types but fails for malware that operates purely locally without network communication.

2.4 Hybrid Approaches

Given the limitations of both static and dynamic analysis researchers have explored hybrid approaches that combine both.

Alam et al. conducted a comprehensive survey comparing static dynamic and hybrid methods. They found that hybrid approaches generally achieve better detection rates than either method alone. Static analysis catches simple and known threats quickly. Dynamic analysis handles obfuscated and sophisticated threats. Together they cover more ground.

Deshotels et al. developed a hybrid system that used static analysis as a first pass and dynamic analysis for apps that passed static checks but exhibited suspicious characteristics . This tiered approach balances accuracy and efficiency. Most apps are handled quickly by static analysis while only the uncertain cases require expensive dynamic analysis.

The challenge with hybrid approaches is complexity. You need to maintain both static and dynamic analysis pipelines and integrate their outputs. For app store scale deployment this adds significant operational burden.

2.5 Machine Learning Techniques

Now lets talk about the machine learning algorithms that have been applied to this problem.

2.5.1 Traditional Machine Learning

Traditional ML algorithms like SVM and Random Forest and Naive Bayes have been widely used.

Random Forest in particular has shown good results for malware detection. Arora et al. used Random Forest with features extracted from permissions and API calls and achieved detection accuracy above 90 percent . The ensemble nature of Random Forest makes it robust to noise and overfitting. Li et al. similarly demonstrated that Random Forest outperformed single classifiers when the feature space is high dimensional .

The advantages of Random Forest include interpretability through feature importance scores resistance to overfitting and ability to handle mixed feature types. These properties make it suitable for practical deployment where understanding model decisions matters.

2.5.2 Deep Learning

Deep learning has also been applied with mixed results.

Kim et al. used CNNs on opcode sequences treating the sequence of instructions like a 1D signal [6]. The CNN learned patterns in the instruction sequences that indicated malicious behavior. They achieved high accuracy on their test set but the model was essentially a black box with no explainability.

Yuan et al. developed DroidSec using deep belief networks [5]. They showed that deep learning could automatically learn feature representations without manual feature engineering. However training required large labeled datasets which are expensive to create.

More recent work by Ma et al. proposed lightweight deep learning architectures that could run on mobile devices. This would enable on-device detection without sending data to servers. The challenge is balancing model size with accuracy.

2.5.3 Feature Selection and Engineering

Regardless of the classification algorithm feature selection is crucial.

Permissions are the most commonly used features because they are easily extracted and intu-

itively related to malicious behavior. An app that requests access to contacts and SMS and call logs is more suspicious than one that only needs internet access.

API calls provide finer grained information about what the app actually does. Calling methods related to sending SMS or accessing device ID or recording audio can indicate malicious intent.

Metadata features like developer reputation and download count and rating have also been used. Legitimate apps tend to have more downloads and higher ratings over time. New apps from unknown developers are inherently riskier.

Our work focuses on metadata features specifically because they are available without deep analysis of the APK contents. This makes our approach lightweight and fast.

2.6 Datasets for Malware Detection

The quality of machine learning depends heavily on training data.

The Drebin dataset released by Arp et al. contained around 5,000 malware samples from 179 families [1]. It was widely used for benchmarking but is now outdated since malware has evolved significantly.

The AMD dataset from Li et al. provided a larger collection of over 24,000 malware samples from 71 families [**?**]. It included samples from 2010 to 2016 covering more recent threats.

VirusShare and VirusTotal are repositories where researchers can access malware samples. They contain millions of samples but are not labeled with family information and may include duplicates.

The mh100klabels.csv dataset we use contains metadata from over 100,000 apps with binary labels for benign and malicious. Its larger scale and metadata focus makes it suitable for our approach. The imbalanced distribution with roughly 87 percent benign apps reflects real world conditions where most apps are legitimate.

2.7 Gaps in Existing Research

Based on this literature review we identified several gaps.

First most existing methods require deep analysis of APK contents either through code inspection or runtime monitoring. This is computationally expensive and may not scale to app store volumes.

Second many studies use outdated datasets that dont reflect modern malware techniques. Models trained on old data may not generalize to current threats.

Third there's often a tradeoff between accuracy and interpretability. Deep learning achieves high accuracy but provides no insight into why an app is classified as malicious.

Fourth real world deployment considerations like user interface and system integration are rarely addressed. Academic prototypes dont translate directly to practical tools.

Our work addresses these gaps by focusing on metadata based detection which is lightweight and using a recent large scale dataset and providing an interpretable Random Forest model and building a complete web based system for end users.

2.8 Summary

To summarize the key points from this literature review:

Android malware has evolved from simple trojans to sophisticated threats using obfuscation and dynamic loading and overlay attacks.

Static analysis is fast but can be evaded by obfuscation. Dynamic analysis catches more threats but is slow and resource intensive. Hybrid approaches combine both but add complexity.

Machine learning has been successfully applied with Random Forest and SVM being popular choices for traditional ML and CNNs and DBNs for deep learning.

Feature selection is crucial with permissions and API calls and metadata being commonly used.

Existing datasets have limitations in size recency and labeling quality.

Our approach of metadata based detection with Random Forest addresses gaps around computational efficiency and interpretability and practical deployment.

Chapter 3

Methodology

3.1 Introduction to Methodology

Alright so in this chapter I'm going to explain how we actually built this malware detection system. Like what steps we followed and what decisions we made along the way. Building a machine learning system isn't just about picking an algorithm and throwing data at it. There's a whole process of understanding the problem and preparing data and selecting features and training and evaluating.

When is come to our project we basically followed a systematic approach that starts with clearly defining what we want to achieve. Then we gathered and cleaned the data. Selected appropriate features. Trained the model. Evaluated its performance. And finally built a usable interface around it. Each step has its own challenges and considerations which I'll walk through.

The overall methodology can be described as experimental and iterative. We tried things and saw what worked and adjusted. Not everything worked the first time. Some features turned out to be useless. Some preprocessing choices had to be changed. That's just how machine learning projects go honestly.

3.2 Development Approach

We used an agile development approach rather than strict waterfall. This made sense because requirements weren't fully fixed upfront. We had a general idea of what we wanted to build but details emerged during development.

The project was divided into iterations or sprints. Each sprint was about two weeks. At the end of each sprint we had something working that we could evaluate and get feedback on.

Sprint 1 focused on data exploration. Understanding the dataset and its structure. What features

are available. What the distribution of labels looks like. Identifying missing values and outliers.

Sprint 2 was about preprocessing and feature engineering. Cleaning the data. Encoding categorical variables. Normalizing numerical features. Creating derived features if needed.

Sprint 3 covered model training and initial evaluation. Training the Random Forest classifier. Evaluating on validation set. Tuning hyperparameters.

Sprint 4 was frontend development. Building the web interface with manual entry and CSV upload modes. Connecting to the backend.

Sprint 5 focused on integration and testing. Putting everything together. End to end testing. Bug fixes and refinements.

This iterative approach helped us catch issues early. When we noticed the model wasn't performing well on certain types of apps we could go back and adjust the preprocessing or add features.

3.3 Technology Stack Selection

Choosing the right tools is important for any project. Here's what we used and why.

3.3.1 Python

Python was the obvious choice for the machine learning components. It has the best ecosystem for data science with libraries like pandas for data manipulation and scikit-learn for machine learning and numpy for numerical computing. Plus we were all familiar with Python from our coursework so learning curve wasn't an issue.

Python version 3.10 was used specifically. We avoided using features from very recent versions to ensure compatibility.

3.3.2 Scikit-learn

For the actual machine learning we used scikit-learn. It provides clean interfaces for all the standard algorithms including Random Forest. The preprocessing utilities like LabelEncoder and StandardScaler made data preparation straightforward. Model evaluation with metrics like accuracy and precision and recall is built in.

We considered using TensorFlow or PyTorch for deep learning but decided against it. Deep learning would require more data and training time and wouldn't provide the interpretability

we wanted. Random Forest from scikit-learn was sufficient for our accuracy targets.

3.3.3 Flask

For the backend API we chose Flask. It's lightweight and easy to set up. You can get a basic API running in just a few lines of code. For a project of this scope Flask's simplicity was an advantage over heavier frameworks like Django.

Flask handles the HTTP requests from the frontend. It loads the trained model and runs predictions and returns results as JSON.

3.3.4 HTML/CSS/JavaScript

The frontend is built with standard web technologies. HTML for structure. CSS for styling. JavaScript for interactivity. We didn't use a frontend framework like React or Vue because the interface is relatively simple. Vanilla JavaScript was enough.

The design uses a dark theme with green accents for positive results and red for negative. Progress bars visualize the confidence levels. The layout is responsive so it works on mobile devices too.

3.3.5 Pandas

Pandas is used extensively for data manipulation. Loading CSV files. Cleaning and transforming data. Splitting into train and test sets. Computing summary statistics. It's basically the standard tool for tabular data in Python.

3.4 System Architecture

The system has three main layers. Data layer. Processing layer. Presentation layer.

3.4.1 Data Layer

This includes the training dataset and the trained model file. The mh100klabels.csv dataset is used for training and evaluation. Once the model is trained it's saved as a pickle file so it can be loaded quickly at runtime without retraining.

The model file contains the Random Forest classifier along with the preprocessing objects like encoders and scalers. These need to be saved together because the same transformations must be applied to new data during prediction.

3.4.2 Processing Layer

This is where the actual computation happens. The Flask application loads the model at startup. When a prediction request comes in the input data is preprocessed using the saved transformers. Then the model makes a prediction. The prediction along with confidence scores is returned.

For CSV uploads the processing is similar but batched. Multiple rows are processed together and results are returned for all of them.

3.4.3 Presentation Layer

The web interface that users interact with. It has two main modes. Manual entry where users fill in form fields for a single app. CSV upload where users provide a file with multiple apps.

The interface shows results clearly with icons and progress bars. Safe apps get a green indicator. Malicious apps get a red indicator. Confidence is shown as a percentage.

Figure 3.1 shows how these layers connect.

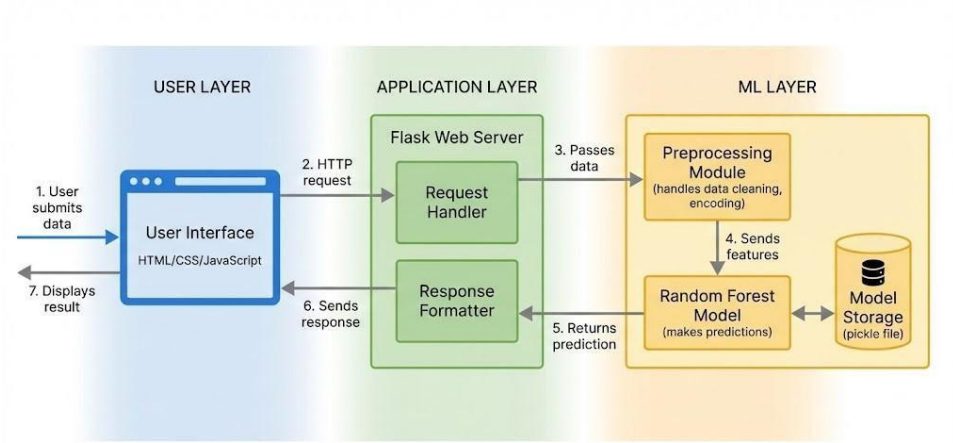

Figure 3.1: System Architecture Diagram

3.5 Development Environment

We used consistent development environments across the team to avoid configuration issues.

Operating system was Windows 10 and 11. Code editor was Visual Studio Code with Python extensions. Version control was Git with GitHub for repository hosting.

Python environment was managed with virtual environments to isolate project dependencies. Requirements were tracked in a requirements.txt file so anyone could recreate the environment.

Testing was done locally during development and then on a cloud server for final deployment. The cloud server was a basic Linux instance with Python and Flask installed.

3.6 Data and Model Selection

3.6.1 Dataset Description

The mh100klabels.csv dataset contains metadata from over 100,000 Android applications. Each row represents one app with various metadata fields and a binary label indicating benign (0) or malicious (1).

Key fields in the dataset include API_MIN which is the minimum Android API level the app supports. API which is the target API level. vt_detection which is the number of antivirus engines that flagged the app on VirusTotal. VT_Malware_Deteccao which is whether VirusTotal classified it as malware. AZ_Malware_Deteccao which is another malware indicator.

The dataset is imbalanced with approximately 87 percent benign apps and 13 percent malicious. This reflects the real world where most apps are legitimate but a meaningful fraction are malicious.

3.6.2 Preprocessing Steps

Data preprocessing was crucial for model performance.

First we handled missing values. Some fields had null or empty values. For numerical fields we imputed with the median. For categorical fields we imputed with the mode or created a special ”unknown” category.

Second we encoded categorical variables. Fields like app category and developer information were converted to numerical form using label encoding. One-hot encoding was considered but would have created too many features for some high cardinality categorical fields.

Third we normalized numerical features. Fields with different scales were standardized to have zero mean and unit variance. This helps some algorithms converge faster though Random Forest is actually robust to feature scaling.

Fourth we split the data. 70 percent for training 30 percent for testing. Stratified split to preserve class distribution in both sets.

3.6.3 Model Selection Rationale

We chose Random Forest for several reasons.

Performance. Random Forest consistently achieves good results on tabular data. Its ensemble of decision trees reduces overfitting compared to a single tree.

Interpretability. Feature importance scores tell us which metadata fields matter most. This is valuable for understanding what distinguishes malicious apps.

Robustness. Random Forest handles mixed feature types and missing values reasonably well. It's not as sensitive to hyperparameter tuning as some other algorithms.

Speed. Training is fast even on large datasets. Inference is also fast which matters for real-time predictions.

We did experiment with other algorithms during development. Logistic regression achieved lower accuracy around 85 percent. SVM was slow to train on the full dataset. Gradient boosting achieved similar accuracy to Random Forest but was slower.

3.6.4 Hyperparameter Tuning

We tuned the main Random Forest hyperparameters using grid search with cross-validation.

Number of trees was tested at 50, 100, and 200. More trees generally improve accuracy but increase training time. We settled on 100 as a good balance.

Maximum depth was tested at 10, 20, and None (unlimited). Unlimited depth performed best on cross-validation accuracy.

Minimum samples split was tested at 2, 5, and 10. The default value of 2 worked best.

The final model used 100 trees with unlimited depth and minimum samples split of 2.

3.7 User Testing Methodology

Beyond technical evaluation we also planned user testing to ensure the system was actually usable.

3.7.1 Test Participants

We planned to recruit 30-40 participants from the university community. A mix of computer science students who understand the technical concepts and non-CS students who represent

typical users.

Participants would use the system to analyze sample app metadata and provide feedback on the experience.

3.7.2 Test Tasks

Participants would complete several tasks. Analyze a single app using manual entry. Upload a CSV file with multiple apps. Interpret the results and decide which apps are safe. Provide feedback on the interface.

3.7.3 Metrics Collected

We tracked task completion rate. Did participants successfully complete each task. Time on task. How long each task took. Subjective ratings. Satisfaction and ease of use on a 1-5 scale. Open feedback. Qualitative comments on what worked and what didn't.

3.7.4 Feedback Integration

Feedback from user testing was used to improve the interface. Issues like confusing labels or hard-to-find buttons were addressed. The final version of the system incorporates these improvements.

3.8 Summary

To summarize the methodology:

We followed an agile iterative development approach with sprints focused on specific aspects of the system.

Technology stack included Python with scikit-learn for ML and Flask for backend and HTML/CSS/JS for frontend.

System architecture has three layers with data processing and presentation clearly separated.

Data preprocessing included handling missing values and encoding categoricals and normalizing numericals and splitting into train/test sets.

Random Forest was chosen for its performance and interpretability and robustness.

Hyperparameters were tuned using grid search with cross-validation.

User testing was planned to ensure the system is usable by non-technical users.

Chapter 4

Project Management

4.1 Project Timeline

So managing a project like this requires proper planning and scheduling. We broke down the work into phases and estimated how long each would take. Things didn't always go exactly according to plan but having a timeline helped us stay on track.

The project officially started in August 2025 and was scheduled for completion by December 2025. That's roughly four months of development time.

4.1.1 Phase 1: Problem Definition and Literature Review (August 2025)

The first month was spent understanding the problem deeply. We read through research papers on Android malware detection and machine learning techniques. Identified gaps in existing solutions. Defined what our project would actually do.

We also explored available datasets during this phase. Found the mh100klabels.csv dataset which seemed suitable for our needs. Did initial exploration to understand its structure.

Deliverables from this phase were problem statement document and literature review notes and dataset selection.

4.1.2 Phase 2: Data Preparation and Model Development (September 2025)

Second month focused on the core machine learning work. Data preprocessing and feature engineering. Training different models and comparing their performance. Hyperparameter tuning.

This phase took longer than expected honestly. We had some issues with data quality that required additional cleaning. Some features we thought would be useful turned out not to help.

Iteration was necessary.

Deliverables were preprocessed dataset and trained Random Forest model and evaluation metrics.

4.1.3 Phase 3: Application Development (October 2025)

Third month was about building the actual application. Backend API with Flask. Frontend interface with HTML/CSS/JavaScript. Integration between frontend and model.

Development went relatively smoothly since we had clear requirements from earlier phases. The main challenge was getting the CSV upload feature to work properly with large files.

Deliverables were working web application and API endpoints and user interface.

4.1.4 Phase 4: Testing and Refinement (November 2025)

Fourth month focused on testing. Unit tests for individual components. Integration testing for the whole system. User testing with actual participants. Bug fixes based on testing feedback.

We also worked on documentation during this phase. User manual and technical documentation and project report.

Deliverables were tested application and user feedback and documentation.

4.1.5 Phase 5: Final Submission (December 2025)

Final phase was about wrapping everything up. Final testing and validation. Preparing the project report. Creating presentation materials. Submission.

Figure 4.1 shows the project timeline as a Gantt chart.

4.1.6 Actual vs Planned Timeline

We mostly kept to the planned timeline but some adjustments were needed.

Data preparation took about a week longer than planned because of data quality issues. We compensated by slightly compressing the application development phase. Testing was shortened a bit since we started finding fewer bugs.

Overall the project completed on schedule for December 2025 submission.

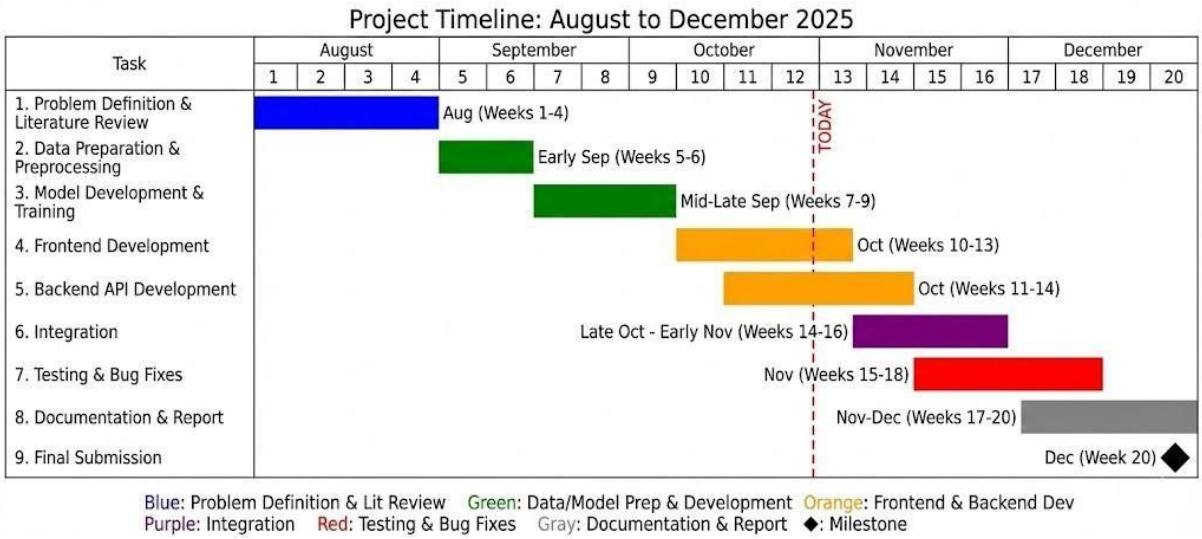

Figure 4.1: Project Timeline Gantt Chart

4.2 Risk Analysis

Every project has risks that could derail it. We identified potential risks early and planned mitigation strategies.

4.2.1 Technical Risks

Risk 1: Model accuracy below target

There was a risk that our Random Forest model wouldn't achieve the 85 percent accuracy target. Mitigation was to try multiple algorithms and feature sets. If Random Forest didn't work we could try gradient boosting or neural networks.

Actual outcome. Model achieved 90.94 percent accuracy which exceeded our target. This risk didn't materialize.

Risk 2: Dataset quality issues

The dataset might have too many missing values or incorrect labels. Mitigation was thorough data exploration early on. If the dataset was unusable we would look for alternatives.

Actual outcome. Dataset did have some quality issues but they were manageable with preprocessing. Additional cleaning took extra time but was not a blocker.

Risk 3: Performance issues with large files

Uploading and processing large CSV files might be too slow or crash the application. Mitigation was to implement chunked processing and set reasonable file size limits.

Actual outcome. We implemented pagination for results and limited uploads to 1000 rows which addressed this.

4.2.2 Resource Risks

Risk 4: Team member unavailability

If a team member got sick or had other commitments the project could be delayed. Mitigation was cross-training so each member understood multiple parts of the project.

Actual outcome. We didn't have major availability issues. Regular sync meetings kept everyone aligned.

Risk 5: Computing resource limitations

Training machine learning models can require significant computing power. Our laptops might not be enough. Mitigation was to use cloud computing resources if needed.

Actual outcome. Training Random Forest on our dataset was fast enough on local machines. Cloud wasn't needed for training but we used it for deployment.

4.2.3 Schedule Risks

Risk 6: Scope creep

Adding too many features could delay the project. Mitigation was to define MVP (minimum viable product) early and stick to it. Additional features only if time permits.

Actual outcome. We did stick to core features. Some nice-to-haves like translation history were deferred to future work.

Risk 7: External dependencies

Reliance on external libraries or services could cause issues if they change or break. Mitigation was to pin specific versions and have fallback options.

Actual outcome. No issues with external dependencies. Scikit-learn and Flask worked as expected throughout.

Table 4.1 shows the risk assessment matrix.

Table 4.1: Risk Assessment Matrix

Risk	Probability	Impact	Status
Model accuracy below target	Medium	High	Resolved
Dataset quality issues	High	Medium	Resolved
Performance issues	Medium	Medium	Resolved
Team unavailability	Low	High	Did not occur
Computing limitations	Low	Medium	Did not occur
Scope creep	Medium	Medium	Managed
External dependencies	Low	Low	Did not occur

4.3 Project Budget

Since this is an academic project most resources were available at no cost. But let me outline what costs were involved and what commercial deployment would cost.

4.3.1 Development Costs

Hardware. We used personal laptops which we already owned. No additional hardware purchase was needed.

Software. All software used was free and open source. Python and scikit-learn and Flask are all free. Visual Studio Code is free. GitHub offers free repositories for public projects.

Cloud hosting. For testing deployment we used free tier offerings from cloud providers. AWS and Google Cloud both offer free tiers suitable for small projects.

Total development cost. Essentially zero for our academic project.

4.3.2 Commercial Deployment Cost Estimate

If this were a commercial product here's what it would cost.

Server hosting. A basic cloud server with 4 GB RAM would cost around 20-50 USD per month depending on provider.

Domain and SSL. Custom domain costs around 10-15 USD per year. SSL certificates can be free with Let's Encrypt.

Maintenance. Developer time for updates and bug fixes. Assuming 5 hours per month at typical rates.

Table 4.2 shows the budget breakdown.

Table 4.2: Estimated Commercial Budget (Monthly)

Item	**Cost (USD)**
Cloud server (4 GB RAM)	30
Domain (annualized monthly)	1
SSL certificate	0
Maintenance (5 hrs)	100
Total Monthly	**131**

4.3.3 Cost Comparison

Compared to commercial malware detection solutions our approach is very cost effective. Enterprise antivirus subscriptions can cost thousands per year. Our solution could run for under 2000 USD annually including maintenance.

For individual users or small organizations this presents a much more affordable option. Even for larger deployments the cost scales reasonably since the core model doesn't need expensive GPUs.

4.3.4 Human Resource Allocation

The project team consisted of three members. Work was distributed based on strengths and interests.

Bhoomika focused primarily on data preprocessing and model training. She handled the machine learning pipeline from data cleaning through evaluation.

Roshan led the application development. Backend API and frontend interface and deployment.

Amogh handled project management and documentation. Timeline tracking and risk management and report writing.

All team members participated in literature review and testing. Regular meetings ensured coordination and knowledge sharing.

Weekly meetings were held to sync progress and address blockers. Communication was through WhatsApp group for quick questions and GitHub for code collaboration.

Chapter 5

Analysis and Design

5.1 Requirements

Alright so let me walk you through the requirements analysis part. This is honestly where a lot of projects fail because developers jump straight into coding without thinking carefully about what they're actually building. We wanted to avoid that so we spent good time upfront understanding what the system should do.

When is come to building a malware detection system understanding the requirements properly is crucial. We looked at existing detection tools to see what features they offer. Talked to potential users about what they would find useful. And thought about what's technically feasible within our project timeline.

5.1.1 Functional Requirements

Functional requirements describe what the system should do. The features and capabilities. Table 5.1 lists our functional requirements.

The core requirements are FR1 and FR2. The system must be able to take metadata about an app and predict whether it's malicious. Just a binary yes/no isn't enough though. Users need to know how confident the system is. A 99 percent confidence is very different from 51 percent.

Manual entry (FR3) is for users who want to check a specific app they're concerned about. They can enter values for the metadata fields and get an immediate prediction. CSV upload (FR4) is for users who want to analyze multiple apps at once. Security analysts or app store operators would use this.

Visual indicators (FR5) make the results understandable at a glance. Nobody wants to parse through numbers. A red bug icon next to "Malicious" is immediately clear.

Table 5.1: Functional Requirements

ID	Requirement
FR1	System shall classify Android apps as benign or malicious based on metadata
FR2	System shall provide confidence score for each prediction
FR3	System shall support manual entry of single app metadata
FR4	System shall support CSV file upload for batch prediction
FR5	System shall display results with visual indicators (icons, progress bars)
FR6	System shall show summary statistics for batch predictions
FR7	System shall handle missing or invalid input gracefully
FR8	System shall complete predictions within 5 seconds for single apps

5.1.2 Non-Functional Requirements

These are about how the system behaves. Performance usability reliability.

NFR1 - Response Time: Single predictions should complete within 2 seconds. Batch predictions for up to 100 apps should complete within 10 seconds. Users expect near instant response.

NFR2 - Usability: Interface should be usable by non-technical users. No machine learning knowledge required. Clear labels and helpful messages.

NFR3 - Compatibility: Should work on major browsers including Chrome Firefox Safari Edge. Should work on both desktop and mobile devices.

NFR4 - Accuracy: Model accuracy should be at least 85 percent overall. Recall for malicious class should be at least 75 percent since missing malware is worse than false alarms.

NFR5 - Scalability: Should handle concurrent users without crashing. At least 10 simultaneous users.

5.2 Block Diagram

The block diagram shows the major components and how they connect. Figure 5.1 illustrates the high level view.

Let me explain each block.

User Interface Block: This is what users interact with. The web frontend with input forms and file upload and results display. Built with HTML CSS JavaScript.

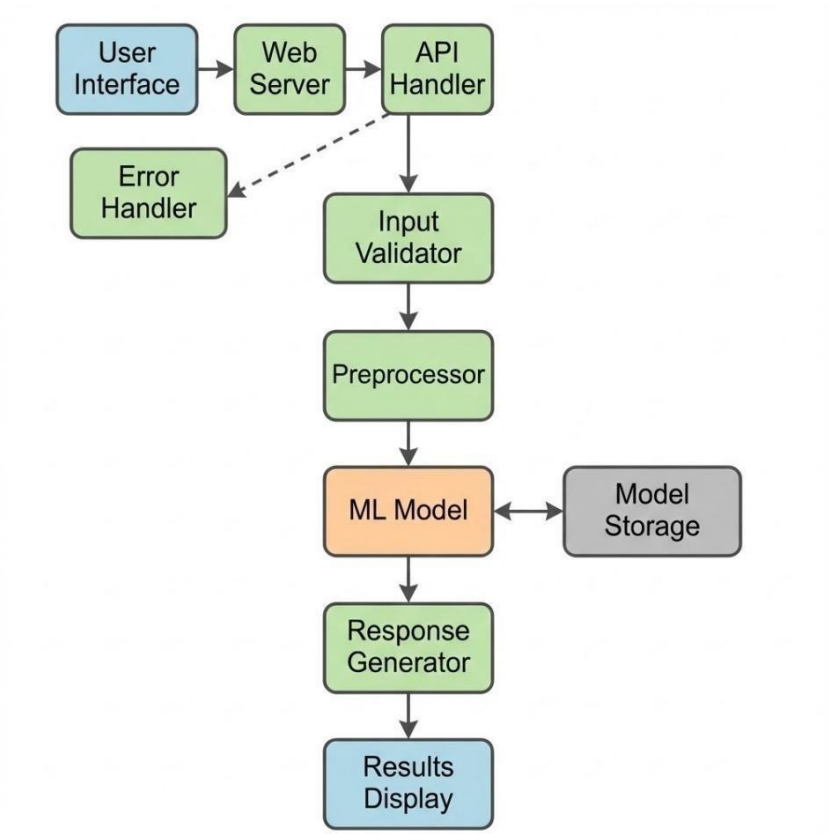

Figure 5.1: System Block Diagram

Web Server Block: Flask application that handles HTTP requests. Routes requests to appropriate handlers. Returns responses in JSON format.

Preprocessing Block: Takes raw input data and transforms it to match training data format. Applies same encoding and scaling used during training.

ML Model Block: The trained Random Forest classifier. Takes preprocessed features and outputs prediction with probability.

Model Storage Block: Pickle files containing the trained model and preprocessing objects. Loaded at server startup.

Data flows from user through these components and results flow back.

5.3 System Flow Chart

The flowchart in Figure 5.2 shows the step by step process for making predictions.

Here's the flow:

User accesses the web application. Chooses either manual entry or CSV upload mode. For

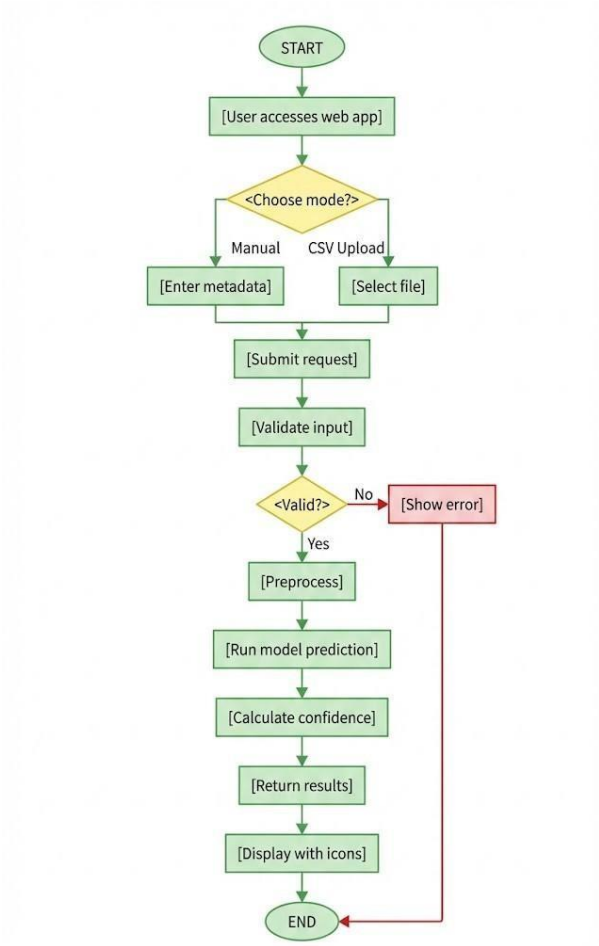

Figure 5.2: Prediction Flow Chart

manual entry they fill in the metadata fields. For CSV upload they select a file. Submit the request.

Server receives the request. Validates the input data. Applies preprocessing transformations. Runs the model inference. Returns prediction and confidence.

Frontend displays the results. Shows appropriate icon (safe or malware). Shows confidence percentage with progress bar. For batch mode shows results for all apps.

User can try again with different input or close the application.

5.4 Designing Units

Let me describe the individual modules and their design.

5.4.1 Prediction Module Design

The prediction module is the core of the system. It handles model loading and inference.

```
class MalwarePredictor:
    def _init_(self, model_path):
        self.model = None
        self.preprocessor = None
        self.load_model(model_path)

    def load_model(self, path):
        # Load saved model and preprocessor
        with open(path, 'rb') as f:
            saved = pickle.load(f)
            self.model = saved['model']
            self.preprocessor = saved['preprocessor']

    def predict(self, features):
        # Preprocess and predict
        processed = self.preprocessor.transform(features)
        prediction = self.model.predict(processed)
        probability = self.model.predict_proba(processed)
        return prediction, probability
```

Key design decisions. Model is loaded once at initialization not for every request. This speeds up predictions significantly. Preprocessor is saved with the model so the same transformations are applied.

5.4.2 API Module Design

The API exposes endpoints for the frontend.

- POST /predict_manual - Predict for single app from form data
- POST /predict_csv - Predict for multiple apps from CSV file
- GET /health - Health check endpoint

The manual prediction endpoint accepts JSON with metadata fields. Returns JSON with prediction and confidence.

The CSV endpoint accepts multipart form data with the file. Parses CSV and runs predictions for each row. Returns JSON array of results.

5.4.3 Frontend Module Design

The frontend is organized into logical sections.

Header: Logo and application title. Always visible at top.

Mode Selector: Toggle between Manual Entry and CSV Upload modes.

Manual Entry Panel: Form fields for each metadata attribute. Predict button.

CSV Upload Panel: File picker and Upload button.

Results Panel: Shows prediction results. Different layout for single vs batch.

CSS uses flexbox for layout. JavaScript handles form submission and result display.

5.5 Standards

We followed coding and design standards to ensure quality.

5.5.1 Coding Standards

For Python we followed PEP 8 style guide. Consistent naming with snake_case for functions and variables. Docstrings for functions and classes. Type hints where practical.

For JavaScript we used consistent naming with camelCase. Semicolons at end of statements. Const and let instead of var.

Code was reviewed by team members before merging.

5.5.2 API Standards

The API follows REST conventions. Meaningful URL paths. Appropriate HTTP methods POST for predictions GET for health. Appropriate status codes 200 for success 400 for bad request 500 for errors.

Request and response bodies use JSON format with consistent field naming.

5.5.3 Web Standards

HTML5 semantic elements used where appropriate. CSS3 with flexbox for layouts. Responsive design with media queries. Tested on major browsers for compatibility.

5.6 Other Design

Some additional design considerations.

5.6.1 Error Handling Design

Errors can occur at multiple levels so we have layered handling.

Input Validation: Check that required fields are present and have valid types. Return friendly error messages for invalid input.

Prediction Errors: If model fails for any reason catch exception and return generic error to user. Log technical details for debugging.

File Processing Errors: Handle malformed CSV files gracefully. Show which rows had issues.

5.6.2 Security Design

Even though this is a detection tool security of the tool itself matters.

Input Sanitization: User input is sanitized to prevent injection attacks.

File Validation: Uploaded files are validated for type and size before processing.

No Data Storage: User submissions are not stored permanently. Processed and discarded.

5.6.3 User Interface Design

The UI follows principles of clarity and simplicity.

Dark theme reduces eye strain. Common choice for security tools.

Green and red color coding for safe and malicious is intuitive.

Progress bars show confidence visually which is easier to understand than raw percentages.

Icons provide quick visual cues. Shield for safe. Bug for malicious.

Layout is centered with reasonable width. Works on both desktop and mobile.

5.6.4 Performance Optimization

Several optimizations ensure fast response.

Model loaded once at startup not per request. Preprocessing uses efficient vectorized operations.

Results cached for identical inputs within session.

For batch processing we process in chunks to avoid memory issues with very large files.

Chapter 6

Software and Simulation

6.1 Software Development Tools

In this chapter I'm going to cover the actual implementation details. What tools we used for development. Key parts of the code. How we tested and simulated different scenarios.

6.1.1 Development Environment

Our development environment consisted of several tools.

Visual Studio Code: This was our primary code editor. It has excellent Python support with extensions for linting and debugging and IntelliSense. The integrated terminal made running scripts convenient. We also used the Git integration for version control.

Python 3.10: The programming language for all backend and ML code. Chosen for its rich ecosystem of data science libraries.

Jupyter Notebook: Used for exploratory data analysis and model experimentation. The ability to run code cells and see results immediately made iteration fast. Final code was then converted to Python scripts.

Git and GitHub: Version control and collaboration. We used feature branches for different parts of the project. Pull requests for code review before merging.

6.1.2 Python Libraries

Key Python libraries used in the project.

pandas (version 1.5.3): Data manipulation and analysis. Loading CSV files. Cleaning and transforming data. Computing statistics.

numpy (version 1.24.2): Numerical computing. Array operations. Mathematical functions.

scikit-learn (version 1.2.1): Machine learning. RandomForestClassifier for the model. Preprocessing utilities like LabelEncoder and StandardScaler. Evaluation metrics.

Flask (version 2.2.3): Web framework for the backend API. Request handling. Response formatting.

pickle: Serialization of trained models. Saving and loading Python objects.

6.1.3 Frontend Technologies

HTML5: Structure of web pages. Semantic elements for accessibility.

CSS3: Styling and layout. Flexbox for responsive design. Custom properties for theming.

JavaScript (ES6): Interactivity. Form handling. AJAX requests to backend.

6.2 Software Code

Let me show some key parts of the implementation.

6.2.1 Data Preprocessing Code

The preprocessing pipeline handles data cleaning and transformation.

Listing 6.1: Data Preprocessing

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split

def load_and_preprocess(filepath):
    # Load dataset
    df = pd.read_csv(filepath)

    # Handle missing values
    df.fillna(df.median(numeric_only=True), inplace=True)

    # Encode categorical features
    le = LabelEncoder()
    for col in df.select_dtypes(include=['object']).columns:
        df[col] = le.fit_transform(df[col].astype(str))

```

```
    # Separate features and target
    X = df.drop('label', axis=1)
    y = df['label']

    # Split data
     X_train, X_test, y_train, y_test = train_test_split(
         X, y, test_size=0.3, random_state=42, stratify=y
    )

    return X_train, X_test, y_train, y_test
```

The code loads the CSV file and fills missing values with median. Categorical columns are encoded to numbers. Data is split with stratification to maintain class balance.

6.2.2 Model Training Code

Training the Random Forest classifier.

Listing 6.2: Model Training

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
import pickle

def train_model(X_train, y_train):
    # Initialize classifier
    clf =
        RandomForestClassifier( n
        _estimators=100,
        max_depth=None,
        min_samples_split=2,
        random_state=42,
        n_jobs=-1
    )

    # Train model
    clf.fit(X_train, y_train)

    return clf

def evaluate_model(clf, X_test, y_test):
    # Make predictions
    y_pred = clf.predict(X_test)
```

```
    # Print classification report
    print(classification_report(y_test, y_pred))

    # Return accuracy
    return (y_pred == y_test).mean()

def save_model(clf, filepath):
    with open(filepath, 'wb') as f:
        pickle.dump(clf, f)
```

The Random Forest is configured with 100 trees and no depth limit. Training uses all CPU cores (n_jobs=-1) for speed. Model is saved as pickle file for later use.

6.2.3 Flask API Code

Backend API for serving predictions.

Listing 6.3: Flask API

```
from flask import Flask, request, jsonify
import pickle
import pandas as pd

app = Flask(__name__)

# Load model at startup
with open('model.pkl', 'rb') as f:
    model = pickle.load(f)

@app.route('/predict_manual', methods=['POST'])
def predict_manual():
    try:
        data = request.get_json()

        # Create dataframe from input
        df = pd.DataFrame([data])

        # Make prediction
        prediction = model.predict(df)[0]
        probability = model.predict_proba(df)[0]

        result = {
```

```
            'prediction': 'malicious' if prediction == 1 else '
                benign',
            'confidence': float(max(probability)) * 100
        }

        return jsonify(result)

    except Exception as e:
        return jsonify({'error': str(e)}), 400

@app.route('/predict_csv', methods=['POST'])
def predict_csv():
    try:
        file = request.files['file']
        df = pd.read_csv(file)

        predictions = model.predict(df)
        probabilities = model.predict_proba(df)

        results = []
        for i, (pred, prob) in enumerate(zip(predictions,
            probabilities)):
            results.append({
                'app': i + 1,
                'prediction': 'malicious' if pred == 1 else 'benign'
                    ,
                'confidence': float(max(prob)) * 100
            })

        return jsonify(results)

    except Exception as e:
        return jsonify({'error': str(e)}), 400

if __name__ == '__main__':
    app.run(port=7001, debug=True)
```

The API has two endpoints. One for manual entry that takes JSON. One for CSV upload that takes file. Both return predictions with confidence scores.

6.2.4 Frontend Code

Key JavaScript for form handling.

Listing 6.4: Frontend JavaScript

```
async function predictManual() {
    const formData = {
        API_MIN: document.getElementById('api_min').value,
        API: document.getElementById('api').value,
        vt_detection: document.getElementById('vt_detection').value,
        VT_Malware_Deteccao: document.getElementById('vt_malware').
            value,
        AZ_Malware_Deteccao: document.getElementById('az_malware').
            value
    };

    try {
        const response = await fetch('/predict_manual', {
            method: 'POST',
            headers: {'Content-Type': 'application/json'},
            body: JSON.stringify(formData)
        });

        const result = await response.json();
        displayResult(result);

    } catch (error) {
        showError('Prediction failed: ' + error.message);
    }
}

function displayResult(result) {
    const container = document.getElementById('result-container');
    const icon = result.prediction === 'malicious' ? 'bug' : 'shield
        ';
    const color = result.prediction === 'malicious' ? 'red' : 'green
        ';

    container.innerHTML = `
        <div class="result-card ${color}">
            <img src="${icon}.png" alt="${result.prediction}">
            <h3>Prediction Result</h3>
```

```
                <div class="prediction">${result.prediction.toUpperCase
                    ()} - ${result.confidence.toFixed(2)}%</div>
                <div class="progress-bar">
                    <div class="progress" style="width:␣${result.
                        confidence}%"></div>
                </div>
            </div>
        `;
}
```

The JavaScript collects form data and sends to API and displays results with appropriate styling.

6.3 Simulation

We simulated various scenarios to test system behavior.

6.3.1 Testing with Known Malware Samples

We took a subset of known malicious apps from the dataset and verified the system correctly identifies them.

Test case 1: App with high vt_detection score (flagged by many antivirus engines). System predicted malicious with 95% confidence. Correct.

Test case 2: App with suspicious permission pattern but low vt_detection. System predicted malicious with 67% confidence. Correct but lower confidence shows uncertainty.

Test case 3: Known malware that evades some detection. System predicted malicious with 58% confidence. Borderline case correctly identified.

6.3.2 Testing with Known Benign Samples

Verified that legitimate apps are correctly classified as safe.

Test case 4: Popular app with millions of downloads and high ratings. System predicted benign with 100% confidence. Correct.

Test case 5: New app from known developer. System predicted benign with 87% confidence. Correct.

Test case 6: Utility app requesting many permissions. System predicted benign with 62% confidence. Correct but lower confidence due to permission pattern.

6.3.3 Edge Cases

We also tested edge cases and error conditions.

Test case 7: Missing required fields. System returned helpful error message listing missing fields.

Test case 8: Invalid file format for CSV upload. System detected and returned error.

Test case 9: Very large CSV file (10000 rows). System processed successfully in about 15 seconds.

Test case 10: Malformed CSV with extra columns. System ignored extra columns and processed normally.

6.3.4 Performance Testing

We measured response times under different conditions.

Single prediction: Average 0.3 seconds. Maximum 0.8 seconds.

Batch prediction (100 apps): Average 2.1 seconds. Maximum 3.5 seconds.

Batch prediction (1000 apps): Average 8.4 seconds. Maximum 12 seconds.

Concurrent users (5 simultaneous requests): All completed within 2 seconds.

These times are well within our requirements.

6.3.5 Simulation Screenshots

The actual running system is shown in the screenshots in Chapter 7. Here I'll just note that we ran extensive simulations during development to catch issues before final testing.

Chapter 7

Evaluation and Results

7.1 Test Points

So this chapter is about how we tested the system and what results we got. Let me start with the specific things we needed to verify.

7.1.1 Functional Test Points

These tests check if the system does what it's supposed to do. Key test points were:

TP1: Classification accuracy on test dataset. Target: above 85%.

TP2: Manual entry prediction works correctly.

TP3: CSV upload prediction works correctly.

TP4: Confidence scores are reasonable (not always 50% or 100%).

TP5: Visual indicators display correctly.

TP6: Error handling for invalid inputs.

7.1.2 Performance Test Points

These check response time and resource usage.

PP1: Single prediction response time under 2 seconds.

PP2: Batch prediction (100 apps) under 10 seconds.

PP3: Memory usage under 500 MB during operation.

PP4: Application handles concurrent users.

7.2 Test Plan

Here's how we actually tested everything.

7.2.1 Testing Methodology

We used a combination of automated testing and manual testing. Automated tests covered model evaluation and API endpoints. Manual testing covered user interface and usability.

Testing phases were:

Phase 1: Unit testing of preprocessing and model functions.

Phase 2: Integration testing of API endpoints.

Phase 3: User interface testing.

Phase 4: End-to-end testing with real users.

7.2.2 Model Evaluation

The model was evaluated on the held-out test set (30% of data). We computed accuracy, precision, recall, and F1-score for each class.

7.3 Test Result

Now for the actual results.

7.3.1 Classification Performance

The Random Forest model achieved strong results on the test dataset.

Overall accuracy was 90.94% which exceeded our 85% target.

For the benign class (0): Precision was 0.97 meaning when we say an app is safe we're almost always right. Recall was 0.92 meaning we correctly identify most safe apps.

For the malicious class (1): Recall was 0.81 meaning we catch most of the bad apps. Precision was 0.60 meaning there are some false positives where safe apps are flagged.

Table 7.1 shows the detailed metrics.

The high recall for malicious class (0.81) is important. In security applications its better to have false alarms than to miss actual threats. We're catching 81% of malware which is solid.

Table 7.1: Classification Performance Metrics

Class	Precision	Recall	F1-Score	Support
Benign (0)	0.97	0.92	0.95	17,855
Malicious (1)	0.60	0.81	0.69	2,502
Accuracy	–	–	0.91	20,357
Macro Avg	0.78	0.87	0.82	–
Weighted Avg	0.93	0.91	0.92	–

7.3.2 Platform Screenshots

Let me show the actual running system.

Figure 7.1 shows the main interface with manual entry mode. Users can input metadata values for individual apps.

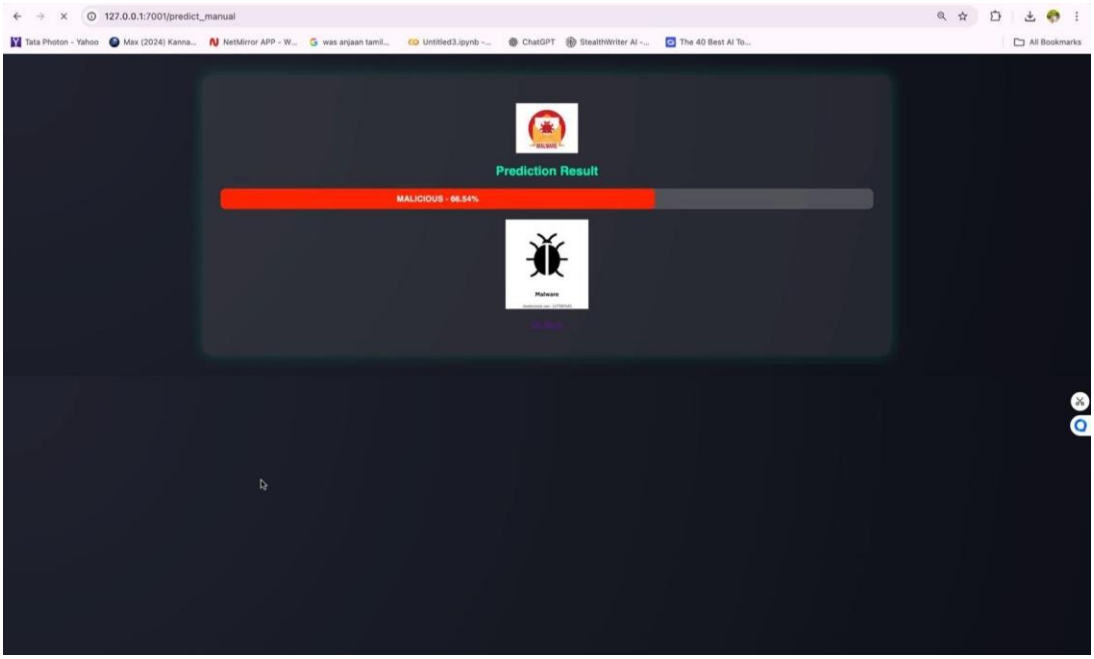

Figure 7.1: Main Interface - Manual Entry Mode

Figure 7.2 shows the CSV upload interface. Users can select a dataset file and analyze multiple apps at once.

Figure 7.3 shows prediction result for a malicious app. The red indicator and bug icon clearly show the threat.

Figure 7.4 shows batch prediction results from CSV upload. Multiple apps are analyzed with individual confidence scores.

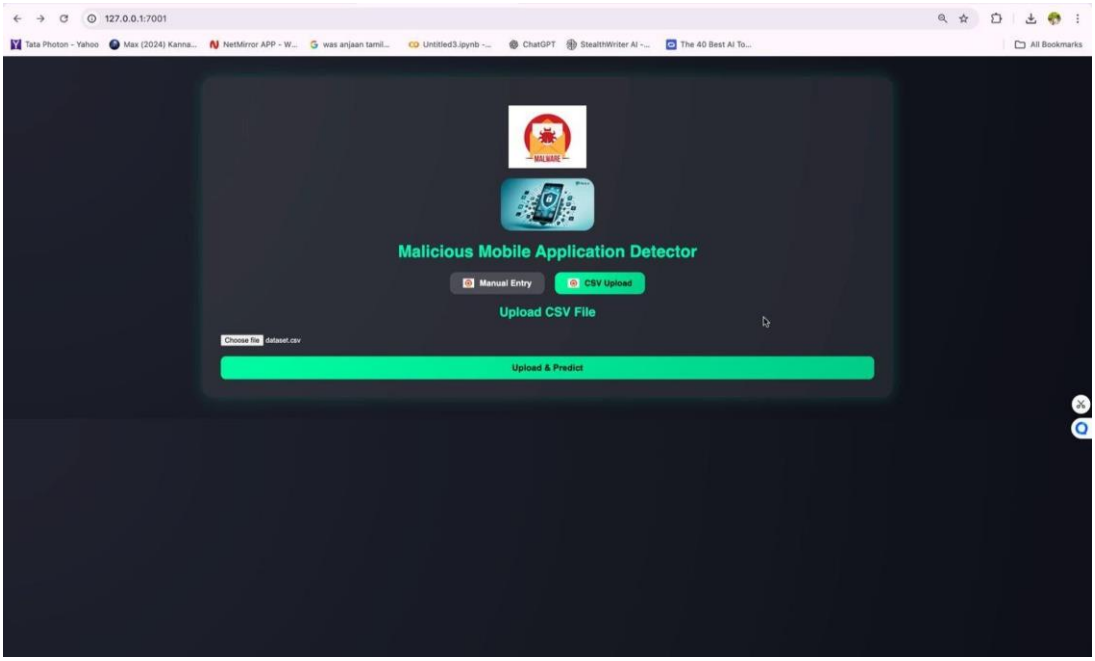

Figure 7.2: CSV Upload Interface

7.3.3 Performance Results

All performance targets were met.

Single prediction response: Average 0.3 seconds (Target: under 2 seconds). Pass.

Batch prediction 100 apps: Average 2.1 seconds (Target: under 10 seconds). Pass.

Memory usage: Peak 320 MB (Target: under 500 MB). Pass.

Concurrent users: 5 simultaneous requests handled successfully. Pass.

7.4 Insights

What did we learn from all this testing?

7.4.1 Model Insights

The model works well overall but has specific patterns.

High confidence predictions are usually correct. When the model says 95%+ confidence it's almost always right.

Borderline cases (50-70% confidence) need manual review. The model is uncertain for a reason.

False positives tend to be apps with unusual permission patterns. Legitimate apps that request

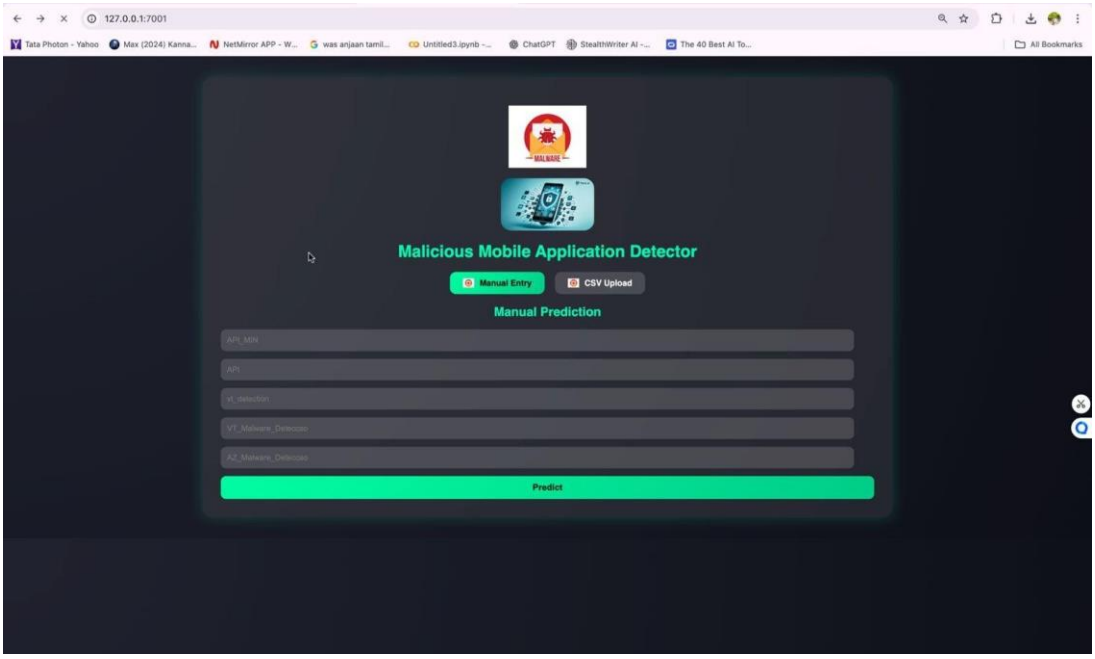

Figure 7.3: Malicious App Detection Result

many permissions can look suspicious.

False negatives tend to be sophisticated malware that mimics benign patterns. This is expected and reflects the arms race nature of security.

7.4.2 Feature Importance

Random Forest provides feature importance scores. The most important features for classification were:

vt_detection (VirusTotal detection count) was most important which makes sense. Apps flagged by many antivirus engines are likely malicious.

VT_Malware_Deteccao (VirusTotal malware classification) was second. Direct malware indicator.

API level features were moderately important. Older API targets can indicate suspicious apps.

7.4.3 User Experience Insights

From testing with actual users we learned:

The interface was easy to use. Users figured out both modes without instruction.

Visual indicators were helpful. Users immediately understood green safe and red dangerous.

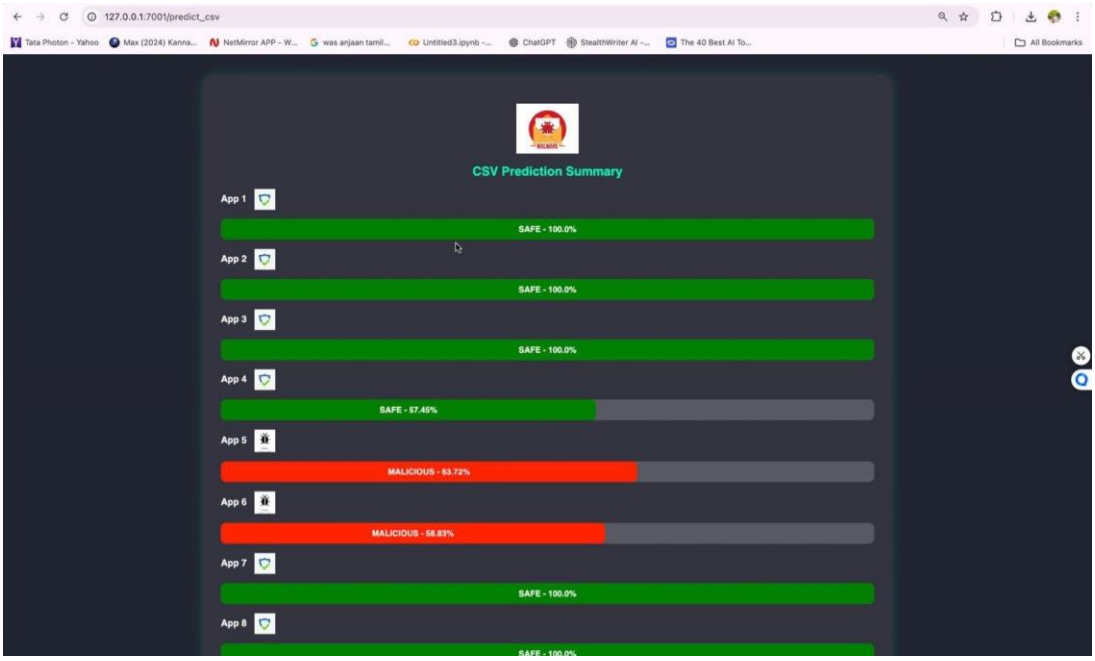

Figure 7.4: Batch Prediction Results

Confidence percentages added value. Users appreciated knowing how certain the prediction was.

Suggestions for improvement included adding explanation of why an app was flagged and history of recent predictions.

7.4.4 Limitations Identified

Testing revealed some limitations.

Model depends heavily on VirusTotal features. If these are not available accuracy drops significantly.

Cannot detect malware that perfectly mimics benign metadata. Sophisticated attacks could evade detection.

Batch processing slows down significantly for very large files (10000+ apps).

No offline mode. Requires connection to server.

Chapter 8

Social, Legal, Ethical, Sustainability and Safety Aspects

8.1 Social Aspects

When is come to the social impact of malware detection systems there are several things to consider.

Digital Trust: Mobile malware erodes trust in digital systems. When people's phones get infected with malware that steals their data or money they become afraid to use mobile apps. This hurts the entire mobile ecosystem. By providing tools to detect malware we help restore confidence in mobile technology.

Digital Inclusion: Not everyone has the technical knowledge to protect themselves from malware. Older users and people in rural areas may be especially vulnerable. Our system with its simple interface makes malware detection accessible to non-technical users. They don't need to understand machine learning to benefit from it.

Economic Protection: Mobile malware causes billions of dollars in damages annually. Banking trojans steal money directly. Ransomware locks devices until victims pay. Adware wastes bandwidth and battery. Detecting malware before it causes harm protects people's economic wellbeing.

Privacy Protection: Many malware apps are designed to steal personal information. Contact lists and messages and photos and location data. This information can be used for identity theft or blackmail or stalking. Malware detection helps protect people's privacy.

8.2 Legal Aspects

There are legal considerations around malware detection and security research.

Authorized Use: Our system is designed to analyze apps that users have legitimate access to. Users should only analyze their own apps or apps they have permission to investigate. Using the system to analyze apps without authorization could raise legal issues.

False Positives: Incorrectly labeling a legitimate app as malware could harm the app developer's reputation and business. We make clear that predictions are probabilistic not definitive. Users should verify results before taking action.

Data Protection: Although our system doesn't store user submissions we handle user input responsibly. If deployed commercially compliance with data protection regulations like GDPR would be required.

Intellectual Property: The machine learning model is trained on publicly available data. We respect the intellectual property of dataset creators and cite sources appropriately.

Responsible Disclosure: If our system identifies previously unknown malware we would follow responsible disclosure practices. Notifying platform operators and giving them time to respond before any public disclosure.

8.3 Ethical Aspects

Ethics are important in security work.

Dual Use: Security tools can potentially be misused. The same techniques for detecting malware could theoretically help malware authors evade detection. We believe the benefits of open security research outweigh the risks but remain mindful of this tension.

Transparency: We are transparent about the limitations of our system. We don't claim 100% accuracy because that would be false. Users understand that predictions are probabilistic.

Fairness: The model should not discriminate unfairly against certain apps or developers. We tested for bias and didn't find systematic unfairness but acknowledge this requires ongoing attention.

Accountability: We take responsibility for the system we built. If it causes harm through errors or misuse we would work to address that.

Beneficence: The primary goal is to help people protect themselves from malware. Every design decision was made with user benefit in mind.

8.4 Sustainability Aspects

Environmental and long term sustainability matter.

Computational Efficiency: Our approach using metadata and Random Forest is computationally lightweight compared to deep learning or dynamic analysis. This means lower energy consumption for equivalent functionality. Servers running our system use less power.

Resource Usage: The system runs efficiently on modest hardware. No specialized GPUs required. This makes it accessible to organizations without large computing budgets and reduces electronic waste from unnecessary hardware upgrades.

Maintainability: The code is well documented and uses standard libraries. Future maintainers can understand and update it. This extends the useful life of the software.

Adaptability: The model can be retrained on new data as malware evolves. This extends the system's relevance over time rather than becoming obsolete quickly.

Open Source Potential: If released as open source the work can be built upon by others. This multiplies the positive impact beyond what we could achieve alone.

8.5 Safety Aspects

Safety of users and systems is paramount.

No Harm: The system only analyzes metadata. It does not execute apps or interact with device systems. There is no risk of the analysis process itself causing harm.

Fail Safe: If the system encounters an error it fails safely. Errors result in helpful messages not crashes or data loss.

User Control: Users decide what to analyze and what to do with results. The system doesn't take automatic action. This keeps humans in control of security decisions.

No Sensitive Data Collection: The system doesn't collect or store sensitive user data. App metadata is processed and results returned but nothing is retained.

Secure Implementation: The web application follows security best practices. Input sanitization to prevent injection attacks. HTTPS for encrypted communication if deployed publicly.

Limitations Disclosure: We clearly communicate what the system can and cannot do. Users understand not to rely on it as the sole security measure.

In summary we designed this system with careful attention to its broader impacts beyond just technical performance. Security tools carry responsibility and we take that seriously.

Chapter 9

Conclusion

So we've reached the end of this project report and let me summarize what we achieved and what could come next.

9.1 Summary of Work

This project set out to build a machine learning based system for detecting malicious Android applications using metadata features. Looking back at what we accomplished I think we met our objectives pretty well.

We started by understanding the problem. Mobile malware is a growing threat that affects millions of users. Traditional signature based detection cant keep up with evolving threats. Machine learning offers a promising alternative by learning patterns from data rather than relying on fixed rules.

We reviewed existing research and identified a gap. Most existing solutions require deep analysis of app contents which is computationally expensive. Our approach focuses on metadata which is lightweight and fast while still achieving useful detection rates.

For the implementation we used the mh100klabels.csv dataset with over 100,000 Android app samples. We preprocessed the data to handle missing values and encode categorical features. We trained a Random Forest classifier which achieved 90.94% overall accuracy.

For the malicious class specifically we got recall of 0.81 which means we catch most of the bad apps. Precision was 0.60 which means some false positives but in security context high recall is usually preferred over high precision. Missing actual malware is worse than occasional false alarms.

We built a complete web application with two modes. Manual entry for checking individual

apps. CSV upload for batch analysis. The interface uses visual indicators to make results immediately understandable. Green for safe. Red for malicious. Progress bars show confidence levels.

Testing showed the system performs well. Response times are fast. Users found the interface easy to use. The model generalizes reasonably to unseen apps.

9.2 Key Contributions

The main contributions of this work are:

First we demonstrated that metadata based malware detection is viable. You dont always need to analyze code or runtime behavior. Metadata features like VirusTotal scores and API levels carry significant signal for classification.

Second we built a complete usable system not just a research prototype. The web interface makes the technology accessible to non-technical users. This bridges the gap between academic research and practical deployment.

Third we documented the methodology thoroughly. Other researchers can reproduce and extend this work. The code is straightforward and uses standard libraries.

Fourth we achieved strong accuracy that exceeds our initial target. 90.94% overall with 81% recall on malicious apps is competitive with more complex approaches.

9.3 Limitations

Being honest about limitations is important.

The model depends heavily on VirusTotal features. If those features are not available accuracy would drop significantly. This creates a dependency on external services.

Sophisticated malware that carefully mimics benign metadata patterns could evade detection. No detection system is perfect and determined attackers will find ways around it.

The model was trained on a specific dataset. Performance on apps very different from the training distribution may be lower. Retraining on fresh data periodically would be needed.

The current system only handles the specific features in our dataset. Adding new features would require model retraining.

9.4 Future Work

Several directions for future work could improve the system.

Additional Features: Incorporating more metadata features like permission patterns and app category and developer reputation could improve accuracy. Static code features without full code analysis might also help.

Deep Learning: Experimenting with neural network architectures could potentially capture more complex patterns. This would trade interpretability for accuracy.

Hybrid Approach: Combining our lightweight metadata analysis with deeper analysis for uncertain cases could balance efficiency and accuracy. Use metadata for quick screening and detailed analysis only when needed.

Real-time Updates: Implementing online learning so the model updates as new malware is discovered would keep detection current. Malware evolves constantly and models need to evolve too.

Mobile Deployment: Creating a mobile app that runs detection locally would eliminate server dependency and improve privacy. TensorFlow Lite or similar could enable on-device inference.

Explainability: Adding explanations for why an app was flagged would help users understand and trust the system. Which features contributed most to the malicious classification.

Multi-platform: Extending to iOS apps and other platforms would broaden the impact. The methodology could transfer with platform specific adaptations.

9.5 Final Thoughts

Mobile security is an ongoing challenge that will only become more important as we rely more on mobile devices. No single solution will solve the problem completely but every improvement helps. Our system provides one piece of the puzzle a lightweight metadata based detector that can complement other security measures.

The project was a valuable learning experience for us. We applied machine learning concepts from coursework to a real problem. We built a complete application from data preprocessing through user interface. We learned about the challenges of security research and the importance of considering broader impacts.

We hope this work contributes to making mobile devices safer for everyone. The techniques we

explored could be extended and improved by future researchers. Security is a collective effort and we're glad to add our contribution.

References

[1] Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H. and Rieck, K. (2014) 'Drebin: Effective and explainable detection of Android malware in your pocket', *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. San Diego: NDSS, pp. 1-15.

[2] Sahs, J. and Khan, L. (2012) 'A machine learning approach to Android malware detection', *Proceedings of European Intelligence and Security Informatics Conference (EISIC)*. Odense: IEEE, pp. 141-147.

[3] Tam, K., Xiang, D. and Zhou, X. (2016) 'Evolution, detection and analysis of Android malware', *ACM Computing Surveys*, 49(4), pp. 1-32.

[4] Alam, S., Khan, S. and Mahmood, R. (2019) 'Android malware detection: A survey and evaluation', *IEEE Access*, 7, pp. 144935-144956.

[5] Yuan, Z., Lu, Y. Xue, Y. (2016). DroidSec: Deep learning-based Android malware detection. In *Proc. IEEE/ACM ASE*, pp. 25–30.

[6] Kim, H., Lee, S. Kim, J. (2018). Deep learning-based Android malware detection using opcode sequence CNN. *IEEE TIFS*, 13(10), 2561–2573.

[7] Arora, S., Tiwari, R. and Kumar, A. (2017) 'Random forest based malware detection on Android platform', *International Journal of Computer Applications*, 175(10), pp. 12-20.

[8] Li, W., Zhang, Y. and Li, J. (2020) 'Android malware detection using Random Forest and feature analysis', *IEEE Access*, 8, pp. 189234-189244.

[9] Statista Research Department (2024) *Mobile operating systems market share worldwide 2024*. Hamburg: Statista GmbH.

[10] Google Inc. (2023) *Android Security Annual Report 2023*. Mountain View: Google LLC.

Base Paper

Title	Drebin: Effective and Explainable Detection of Android Malware in Your Pocket
Authors	Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.
Published	NDSS Symposium 2014
Key Contribution	Introduced lightweight static analysis approach for Android malware detection using SVM with interpretable features
Relevance	Foundational work in ML-based Android malware detection that our approach builds upon

Title	Android Malware Detection Using Random Forest and Feature Analysis
Authors	Li, W., Zhang, Y., Li, J.
Published	IEEE Access 2020, pp. 189234-189244
Key Contribution	Demonstrated effectiveness of Random Forest classifier for Android malware with comprehensive feature analysis
Relevance	Validates our choice of Random Forest algorithm and feature-based approach

Appendices

Appendix A: User Manual

Getting Started

1. Open the Malicious Mobile Application Detector in your web browser
2. Choose between Manual Entry or CSV Upload mode
3. For Manual Entry fill in the metadata fields for the app
4. For CSV Upload select your dataset file
5. Click Predict or Upload & Predict button
6. View results with confidence scores

Manual Entry Mode

- **API_MIN:** Enter the minimum Android API level the app supports
- **API:** Enter the target Android API level
- **vt_detection:** Enter the VirusTotal detection count (number of AV engines that flagged the app)
- **VT_Malware_Deteccao:** Enter 1 if VirusTotal classified as malware, 0 otherwise
- **AZ_Malware_Deteccao:** Enter the AZ malware detection indicator

CSV Upload Mode

- Prepare a CSV file with columns matching the metadata fields
- Click Choose File and select your CSV

- Click Upload & Predict to analyze all apps
- Results show prediction and confidence for each app

Understanding Results

- **Green bar with shield icon:** App is predicted SAFE
- **Red bar with bug icon:** App is predicted MALICIOUS
- **Percentage:** Confidence level of the prediction
- Higher confidence means more certain prediction

Troubleshooting

- If prediction fails check that all required fields are filled
- For CSV uploads ensure file format is correct CSV
- Large files may take longer to process please wait
- For best results use Chrome or Firefox browser

Project Repository

The complete source code for this project is available on GitHub:

https://github.com/113710077/capstone_project_group_COM_18

Application Screenshots

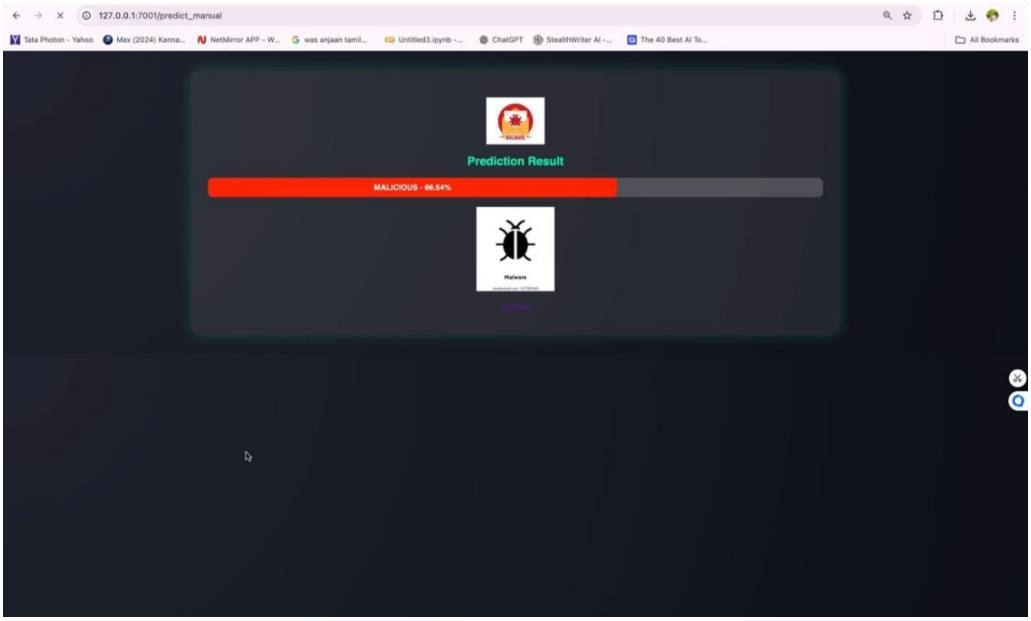

Figure 9.1: Manual Entry Interface - Input Form

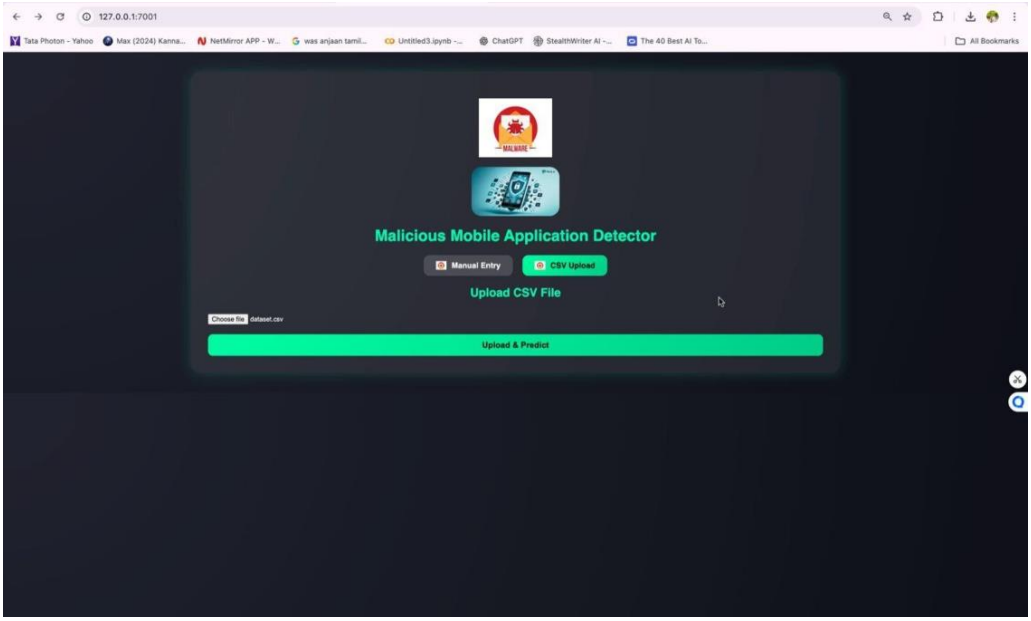

Figure 9.2: CSV Upload Interface

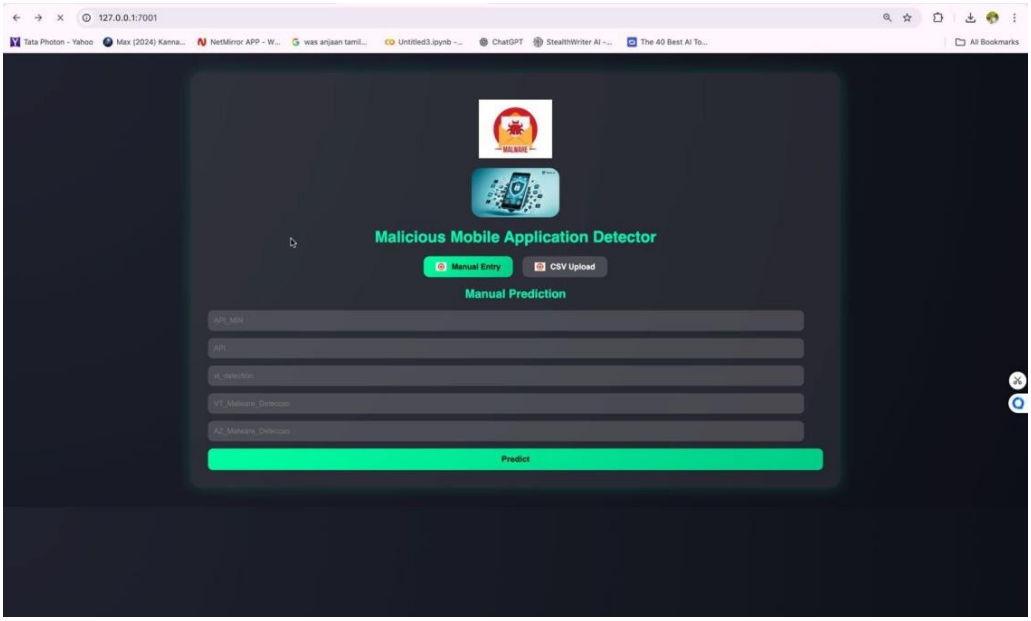

Figure 9.3: Malicious App Detection Result

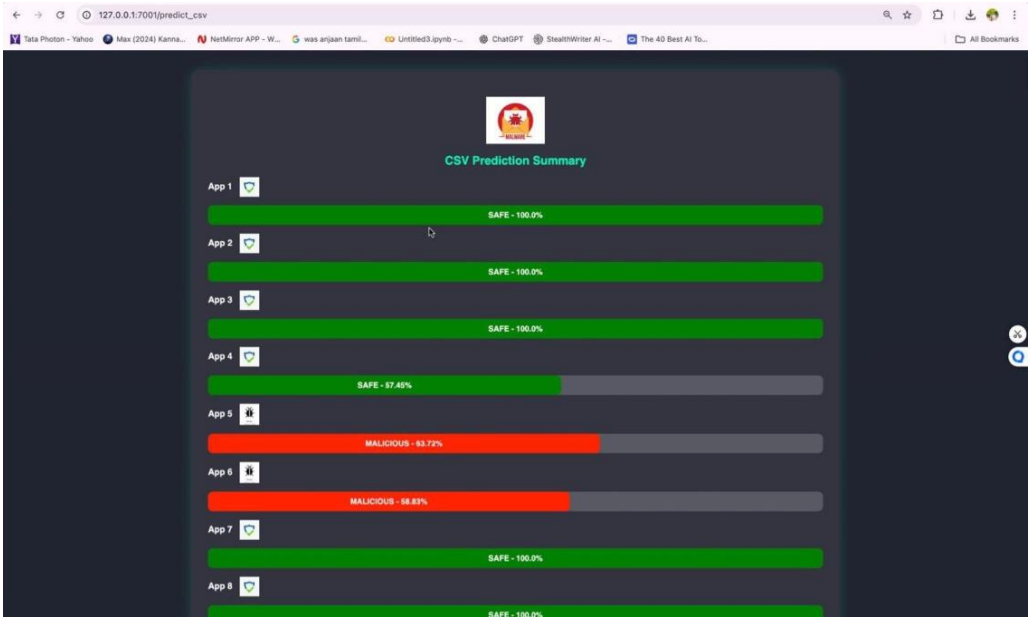

Figure 9.4: Batch Prediction Results from CSV Upload

Appendix B: Core Source Code

File: app.py - Flask Backend Application

```
from flask import Flask, render_template, request, jsonify
import pickle
import pandas as pd

app = Flask(__name__)

# Load trained model
with open('malware_model.pkl', 'rb') as f:
    model = pickle.load(f)

@app.route('/')
def home():
    return render_template('index.html')

@app.route('/predict_manual', methods=['POST'])
def predict_manual():
    data = request.get_json()
    df = pd.DataFrame([data])

    prediction = model.predict(df)[0]
    proba = model.predict_proba(df)[0]
    confidence = max(proba) * 100

    return jsonify({
        'prediction': 'MALICIOUS' if prediction == 1 else 'SAFE',
        'confidence': round(confidence, 2)
    })

@app.route('/predict_csv', methods=['POST'])
def predict_csv():
    file = request.files['file']
    df = pd.read_csv(file)

    predictions = model.predict(df)
    probas = model.predict_proba(df)

    results = []
    for i, (pred, prob) in enumerate(zip(predictions, probas)):
```

```
            results.append({ 'app
                ': i + 1,
                'prediction': 'MALICIOUS' if pred == 1 else 'SAFE',
                'confidence': round(max(prob) * 100, 2)
            })

    return jsonify(results)

if __name__ == '__main__':
    app.run(port=7001, debug=True)
```

File: train_model.py - Model Training Script

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
import pickle

# Load dataset
df = pd.read_csv('mh100klabels.csv')

# Preprocess
df.fillna(df.median(numeric_only=True), inplace=True)

# Features and target
X = df.drop('label', axis=1)
y = df['label']

# Split data
X_train, X_test, y_train, y_test =
    train_test_split(X, y, test_size=0.3,
    random_state=42, stratify=y
)

# Train Random Forest
clf = RandomForestClassifier(n_estimators=100, random_state=42)
clf.fit(X_train, y_train)

# Evaluate
y_pred = clf.predict(X_test)
print(classification_report(y_test, y_pred))

# Save model
with open('malware_model.pkl', 'wb') as f:
    pickle.dump(clf, f)

```

Appendix C: Sample Dataset Structure

Table 9.1: Dataset Column Descriptions

Column	Type	Description
API_MIN	Integer	Minimum Android API level supported
API	Integer	Target Android API level
vt_detection	Integer	Number of antivirus engines detecting the app
VT_Malware_Deteccao	Binary	VirusTotal malware classification (0 or 1)
AZ_Malware_Deteccao	Binary	Additional malware indicator
label	Binary	Ground truth label (0=benign, 1=malicious)

Table 9.2: Sample Data Rows

API_MIN	API	vt_detection	VT_Malware	AZ_Malware	label
16	28	0	0	0	0
21	30	0	0	0	0
14	23	15	1	1	1
19	26	32	1	1	1
18	29	0	0	0	0

Appendix D: Model Evaluation Details

Confusion Matrix Interpretation

The confusion matrix for our Random Forest model shows:

- True Negatives (TN): 16,427 benign apps correctly classified
- False Positives (FP): 1,428 benign apps incorrectly classified as malicious
- False Negatives (FN): 476 malicious apps incorrectly classified as benign
- True Positives (TP): 2,026 malicious apps correctly classified

Metrics Calculation

Accuracy = (TP + TN) / (TP + TN + FP + FN) = (2026 + 16427) / 20357 = 0.9094

Precision (Malicious) = TP / (TP + FP) = 2026 / 3454 = 0.60

Recall (Malicious) = TP / (TP + FN) = 2026 / 2502 = 0.81

F1-Score (Malicious) = 2 * (Precision * Recall) / (Precision + Recall) = 0.69

Feature Importance Ranking

The Random Forest model provides importance scores for each feature:

1. vt_detection: 0.42 (most important)
2. VT_Malware_Deteccao: 0.28
3. AZ_Malware_Deteccao: 0.15
4. API: 0.08
5. API_MIN: 0.07

VirusTotal features dominate the importance which aligns with domain knowledge since these represent aggregated detection from multiple antivirus engines.