Malware Detection and Categorization Using ML and LLMs

by

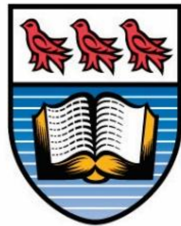
Damanpreet Singh

B.Tech, Punjabi University, Patiala (India), 2022

A Report Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF ENGINEERING

in the Department of Electrical and Computer Engineering



University
of Victoria

© Damanpreet Singh, 2025
University of Victoria

All rights reserved. This report may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

Supervisory Committee

Malware Detection and Categorization Using ML and LLMs

by

Damanpreet Singh

B. Tech, Punjabi University, Patiala (India), 2022

Supervisory Committee

Dr. Issa Traoré, Supervisor

(Department of Electrical and Computer Engineering)

Dr. Sherif Saad, Co-Supervisor

(Department of Electrical and Computer Engineering)

Abstract

The rapid growth of malware attacks has created an urgent need for automated systems capable of accurately detecting and understanding malicious behaviour. This project presents a comprehensive work for Malware Detection and Categorization using Machine Learning and Large Language Models (LLMs). The system's goal is to improve cybersecurity by not just detecting malware but also producing concise, intelligible descriptions of every threat it finds.

The Microsoft Malware Classification dataset, which comprises approximately 21,000 malware samples grouped into nine primary families with corresponding .byte and.asm files, was adopted for the project. Since only malicious samples were present in the original dataset, roughly 15,000 benign files were added to enable binary categorization of malicious and non-malicious programs. XGBoost, LightGBM, SVM (RBF), and KNN were among the machine learning models that were trained and tested independently on both datasets. By applying the SMOTE technique, the dataset's imbalance was reduced, thereby improving classification accuracy and mitigating bias toward the majority malware families. Using the SMOTE technique, class imbalance was addressed. The models were evaluated for both binary classification (malicious vs. benign) and multi-class family prediction, achieving high detection performance.

To enhance interpretability, an LLM-based explanation module was integrated. Following classification, the anticipated malware family is sent to an LLM (through Ollama), which produces a natural-language synopsis outlining the traits, actions, and defenses of the malware. Users can upload files, view predictions, and read the generated explanations in real time thanks to an intuitive Gradio interface.

In order to provide both technical accuracy and human interpretability, the developed system successfully blends large language models for explainable analysis with machine learning for precise detection. By assisting researchers and security analysts in proactive malware defense, this method advances the field of intelligent cybersecurity by bridging the gap between detection and comprehension.

Table of Contents

Supervisory Committee ii

Abstract iii

Table of Contents iv

List of Tables vii

List of Figures viii

Glossary x

Acknowledgments xi

Dedication xii

Chapter 1: Introduction 1

1.1 Context 1

1.2 Project Objectives 2

1.3 Related Work 3

1.4 Report Structure 5

Chapter 2: Dataset and Feature Extraction .... **Error! Bookmark not defined.**

2.1 Dataset Description 7

2.2 Data Preprocessing 9

2.2.1 Data Cleaning and Normalization 10

2.3 Feature Extraction 12

2.3.1 Byte-Level Features (.bytes files) 12

2.3.2 Assembly-Level Features (.asm files) 13

2.4 TF-IDF (Term Frequency – Inverse Document Frequency) 15

2.5 Splitting of the Dataset 15

2.6 Data Balancing (SMOTE) 17

Chapter 3: Machine Learning Model Development 20
3.1 Model Selection 21
3.1.1 Logistic Regression 21
3.1.2 XGBoost 21
3.1.3 LightGBM 22
3.1.4 Support Vector Machine (SVM) 22
3.1.5 K-Nearest Neighbors (KNN) 23
Chapter 4: Experimental Evaluation 24
4.1 Evaluation Metrics 24
4.2 Performance Evaluation on Binary Classification 27
4.3 Performance Evaluation for bytes Mulit-class Classification 29
4.3.1 XGBoost 29
4.3.2 LightGBM 32
4.3.3 KNN 35
4.3.4 SVM 38
4.4 Performance Evaluation for .asm (Multi-class Classification) 41
4.4.1 XGBoost 41
4.4.2 LightGBM 44
4.4.3 KNN 47
4.4.4 SVM 50
4.5 Comparison Between .bytes and .asm Features 53
Chapter 5: LLM Integration and Summary Evaluation 55
5.1 Integration Overview 55

5.2 Gradio User Interface (UI) Integration 56
5.3 Malware Summary using Ollama Integration 57
Chapter 6: Conclusion 61
Bibliography 62

List of Tables

Table 2.1: Malware Families... 09

Table 4.1: Evaluation Hardware Specifications.. 24

Table 4.2: Classification Report for binary classification.. 28

Table 4.3: XGBoost Classification Report on .byte.......................**Error! Bookmark not defined.**

Table 4.4: LightGBM Classification Report on .byte 34

Table 4.5: KNN Classification Report on .byte ....**Error! Bookmark not defined.**7

Table 4.6: SVM RBF Classification Report on .byte... 40

Table 4.7: XGBoost Classification Report on .asm 43

Table 4.8: LightGBM Classification Report on .asm 46

Table 4.9: KNN Classification Report on .asm...........................**Error! Bookmark not defined.**9

Table 4.10: SVM RBF Classification Report on .asm 52

Table 4.11: Comparison of model performance between .asm and .bytes data 54

List of Figures

Figure 1.1: Global Malware and PUA Growth (2008–2024) [1] 1

Figure 2.1: Description of the Dataset 8

Figure 2.2: Structure of .byte file 11

Figure 2.3: Structure of .asm file 11

Figure 2.4: Malware Family Distribution of the dataset 17

Figure 2.5: After appling SMOTE 18

Figure 3.1: Overall Framework of Malware Classification Process 20

Figure 4.1: Confusion Matrix for Binary Classification 28

Figure 4.2: Confusion Matrix for XGBoost Model on .bytes 30

Figure 4.3: Training and Validation Accuracy for XGBoost Model on .bytes 31

Figure 4.4: Confusion Matrix for LightGBM Model on .bytes 33

Figure 4.5: Training and Validation Accuracy for LightGBM Model on .bytes 34

Figure 4.6: Confusion Matrix for KNN Model on .bytes 36

Figure 4.7: Training and Validation Accuracy for KNN Model on .bytes 37

Figure 4.8: Confusion Matrix for SVM RBF Model on .bytes 39

Figure 4.9: Training and Validation Accuracy for SVM RBF Model on .bytes 40

Figure 4.10: Confusion Matrix for XGBoost Model on .asm 42

Figure 4.11: Training and Validation Accuracy for XGBoost Model on .asm 43

Figure 4.12: Confusion Matrix for LightGBM Model on .asm 45

Figure 4.13: Training and Validation Accuracy for LightGBM on .asm 46

Figure 4.14: Confusion Matrix for KNN Model on .asm 48

Figure 4.15: Training and Validation Accuracy for KNN on .asm 49

Figure 4.16: Confusion Matrix for SVM RBF Model on .asm.. 51

Figure 4.17:Training and Validation Accuracy for SVM RBF on .asm....................................... 52

Figure 5.1: Flowchart of the online malware detection process 56

Figure 5.2: Gradio User Interface for Malware Detection and LLM-Based Explanation 57

Figure 5.3: System detect a benign file.. 59

Figure 5.4: System detect a malicious file 60

Glossary

ASM: Assembly File contains the disassembled code of a malware sample

BENIGN: Non-malicious file or program added to enable binary classification

BYTE: Binary Term Expression

SMOTE: Synthetic Minority Oversampling Technique for data balancing

XGBoost: Extreme Gradient Boosting algorithm

LightGBM: Light Gradient Boosting algorithm

SVM: Support Vector Machine classifier

KNN: K-Nearest Neighbors algorithm

TF-IDF: Term Frequency-Inverse Document Frequency

IEEE: Institute of Electrical and Electronics Engineers

Gradio: Gradient Radio Interface

Ollama: Open Large Language Model Application

LLM: Large Language Model

ML: Machine Learning

GPU: Graphics Processing Unit

RAM: Random Access Memory

Acknowledgments

I would like to express my sincere gratitude to Dr. Traoré for his constant guidance, encouragement, and support throughout my master's work. His thoughtful feedback, patience, and genuine interest in my progress greatly shaped my learning experience. I am truly grateful for his mentorship, which inspired me to grow both technically and personally.

I would also like to extend my heartfelt thanks to my parents, grandparents, and sister Mehnoorpreet Kaur, whose unconditional love and encouragement have been my greatest strength. A special thank you to my cousins Navjeevanjot Singh, Harleen Kaur, and Aryan Singh for their constant motivation and belief in me throughout this journey. Their unwavering support made this achievement possible.

Dedication

With love and gratitude, I dedicate my Master of Engineering project to my grandmother, who has always been my light and strength.

Chapter 1: Introduction

1.1 Context

Malware attacks have grown rapidly in both scale and sophistication over the past decade [1][2]. Originally developed as simple viruses and worms, malicious software has now evolved into complex and adaptive threats that target computers, networks, and even critical infrastructure. This growing evolution has transformed the field of cybersecurity and created an urgent need for advanced and intelligent defense systems.

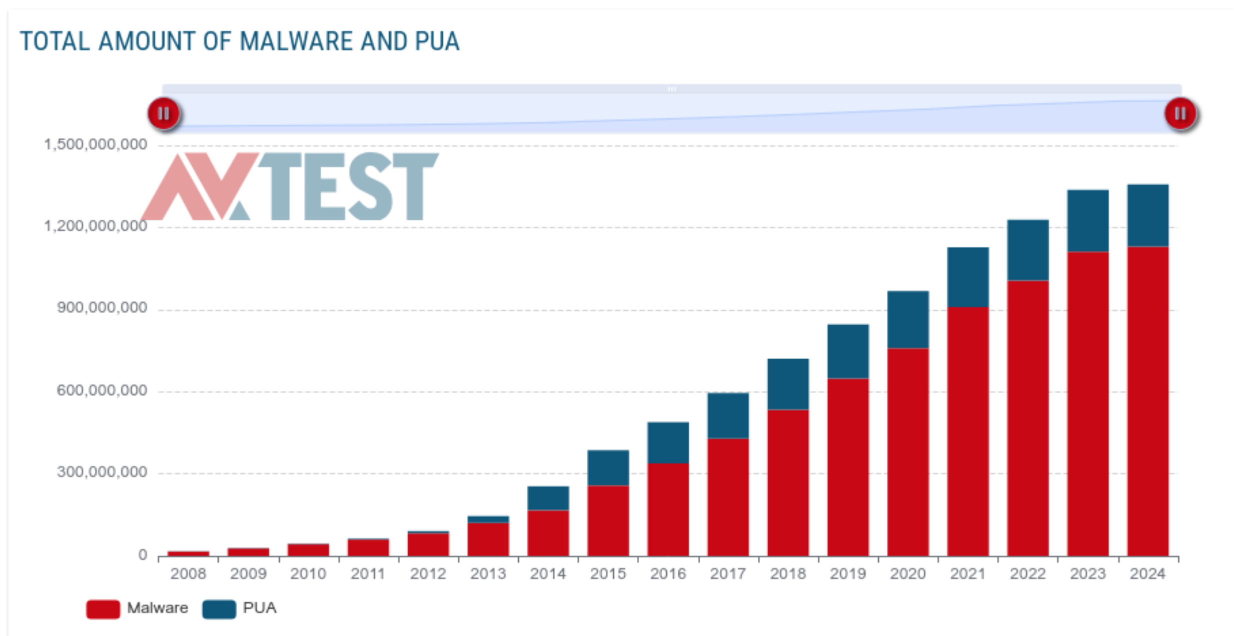


Figure 1.1: Global Malware and PUA Growth (2008–2024) [1]

Figure 1.1 shows the rapid rise in the number of malware and potentially unwanted applications (PUA) detected worldwide from 2008 to 2024 compiled by the AV-TEST Institute [1]. According to the AV-TEST Institute, the number of malware samples has increased dramatically over the past decade, surpassing 1.4 billion in 2024 [1]. This trend highlights the growing scale and sophistication of cyber threats across all digital platforms.

Malware is used in many harmful ways, targeting personal computers, mobile devices, and even large organizations. It can steal sensitive information, encrypt or delete data, slow down systems,

and disrupt critical operations. Cybercriminals use malware for various purposes such as financial fraud, data theft, spying, and spreading false information. As more people and companies depend on digital systems, the risk and impact of malware attacks continue to grow every year [2].

However, the increasing number and complexity of these attacks have introduced major cybersecurity challenges. Modern malware can change its structure or behavior to avoid detection, making traditional antivirus software less effective [2][3]. Hackers use advanced techniques such as code obfuscation, encryption, and polymorphism to disguise malicious programs, allowing them to bypass security filters and infect systems undetected. This constant evolution has made malware detection a difficult and ongoing task [3][4].

Ensuring the safety and integrity of digital systems is now more important than ever. Protecting data, networks, and software from unauthorized access or infection is critical to maintaining trust in technology. A single compromised system can lead to serious consequences, including financial loss, privacy breaches, and large-scale service disruptions.

In this context, machine learning (ML) has become a powerful tool in strengthening cybersecurity. By learning from large datasets of both malicious and safe files, ML algorithms can automatically detect patterns and identify new malware that traditional methods might miss [3]. This approach enables faster, more accurate, and adaptive protection against constantly evolving cyber threat.

1.2 Project Objectives

This project aims to design and build a malware identification and classification system using large language models (LLMs) and machine learning (ML) techniques. The goal is to automatically identify if a given file is dangerous or safe, classify detected malware into the appropriate family, and then generate an intelligible report detailing the virus's behavior and potential dangers.

In the first stage of the project, datasets containing both benign and harmful files were gathered and compiled. The benign files were collected from trusted, secure sources, and the malicious samples are extracted from the Microsoft Malware Classification Challenge (BIG 2015) dataset

[3][4][5]. To capture its binary and assembly-level characteristics for training, each file is analyzed to extract significant features from its .bytes and .asm representations.

After feature extraction, the project builds and evaluates different machine learning models, including Logistic Regression, XGBoost, LightGBM, Support Vector Machine (SVM), and K-Nearest Neighbors (KNN), for binary and multi-class classification. The binary model assesses whether a file is hazardous or not, whereas the multi-class technique identifies the malware family (e.g. Ramnit, Lollipop, Simda or Obfuscator. ACY).

The project also uses an LLM to enhance interpretability. After classifying a file, the LLM automatically generates a human-readable explanation that clarifies the threat, describes how the malware behaves or spreads, and outlines its potential impact. This ensures that both technical and non-technical stakeholders can better understand the detection results, making the system more transparent and user-friendly.

By using this method, the project aims to develop an intelligent, automated, and explicable malware detection system that will enable users better comprehend and react to cyberthreats while also increasing the accuracy of cyber threat identification.

1.3 Related Work

Researchers have turned to machine learning to identify and categorize malware, mainly because old-school signature-based antivirus tools just cannot keep up. Malware keeps changing using tactics like obfuscation, encryption, and morphing into new forms, so, the focus has shifted to building smarter models that learn from the raw features of files. The goal is to catch both what we already know and whatever new threats arise.

Dambra et al. [8] studied the characteristics of malware classifiers. They found that the way you build your dataset, select your features, and design your model can seriously change how well the classifier works [8]. Interestingly, they showed that static features like opcode counts or byte patterns are usually more effective than dynamic ones, at least when the data is balanced across malware families. They also stressed that smart feature selection really helps models generalize

better. They evaluated several machine learning models on these features sets to understand their performance on both malware detection and malware family classification tasks.

Qiao and colleagues built a multi-label malware classification system for Android, using active learning [9]. Instead of forcing each sample into a single malware family, their model recognized that applications often show more than one type of bad behavior. They evaluated 180 labeled malware samples to identify six behavioral categories and created 531 features using APIs, permissions, and intents. The researchers tested 70 classifier combinations, and their model obtained 74% accuracy, which increased to 86.7% after adding 5,396 high-confidence pseudo-labeled examples via active learning.

Anand et al. recently came up with MALITE, a lightweight framework for identifying and classifying malware on devices with limited processing power, such as IoT devices and embedded systems [10]. They convert binary files into images, which are classified through a model combining a neural network and random-forest models. They tested MALITE on seven widely used malware datasets: Malimg, Microsoft BIG, Dumpware10, MOTIF, Drebin, CICAndMal2017, and MalNet, and benchmarked its performance against four state-of-the-art methods. The experimental results demonstrated that both MALITE variants, the neural network-based MALITE-MN and the histogram-driven random forest MALITE-HRF, provide strong detection and classification accuracy while requiring substantially lower memory and computational resources. These results highlight MALITE's effectiveness and practicality for deployment in resource-constrained environments.

Martins et al. proposed an approach that analyzes the structure of executable files by processing the raw bytes and parsing sections of Portable Executable (PE) files considering their semantic and location [11]. Then, they used convolutional neural networks (CNNs) to identify important patterns in these sections. By focusing on the actual code regions and their meaning, their model achieved improved results in categorizing different types of malwares. In the end, they showed that teaching models to recognize these semantic patterns doesn't just boost accuracy it also helps people make sense of what the model is doing when it analyzes malware.

Ponnuru et al. established that even though machine learning models can achieve great accuracy, they are still vulnerable to adversarial manipulations [12]. Their assessment covered important

defensive tactics presented in recent studies and looked at how attackers modify malware files subtly to avoid detection by machine learning. They exposed the flaws in the malware classifiers that are still in use today by examining adversarial strategies and the accompanying defenses. In the end, they showed that stronger defenses are needed if we want these systems to be effective when deployed in real-world environments.

Another study was conducted by Jeoldar et al. who reviewed how LLMs are being used in software and malware security [13]. They examined recent work that applies LLMs to malware code analysis, detection of new variants, and automated threat investigation. Their review also looked at transformer-based methods, key datasets, and specialized LLM models that support static code analysis. Overall, the study highlights current progress, ongoing challenges, and emerging ideas for improving cybersecurity with LLM-based tools.

Most of these studies have helped improve the classification and identification of malware, but they missed something important that is a way to explain their decisions in plain language. Our project helps address such shortcoming by building a malware detection pipeline using shallow ML models combined with LLM. We explored different shallow classifiers and we used the Ollama API to add a large language model that summarizes the results. Now, the system doesn't just classify threats, it explains in clear terms what each malware family does and why it matters.

1.4 Report Structure

The report structure is as follows:

Chapter 1 provides an overview of malware threats, outlines the motivation of the study, states the project objectives, and reviews related work.

Chapter 2 describes the datasets used, explains the preprocessing of malicious and benign files, and details the feature extraction from .bytes and .asm data.

Chapter 3 outlines the machine learning framework implemented for malware detection and family classification. It details the different classification models used and explains the training process for both binary and multi-class classification tasks.

Chapter 4 presents the testing methodologies used to evaluate the system, reports on the results obtained, and assesses the performance of the security monitoring system in detecting and responding to threats.

Chapter 5 presents the integration of the LLM component, which automatically generates readable summaries of the detected malware. It describes how the LLM was integrated with the trained models to enhance interpretability and user understanding

Chapter 6 summarizes the findings of the project and discusses the implications of the results in relation to the project's objectives.

Chapter 2: Dataset and Feature Extraction

This chapter presents the datasets used in our model design and evaluation, and present the data cleaning and preparation approaches, and the steps taken for feature extraction. It introduces the malicious and benign datasets, outlines how the raw .bytes and .asm files were cleaned and normalized, and explains how numerical features were generated from them. The chapter concludes by summarizing the preprocessing and feature extraction workflow that prepares the data for machine learning.

2.1 Dataset Description

The dataset consists of two types of files: some packed with malware, others completely safe. The malware samples come from the Microsoft Malware Classification Challenge dataset (BIG 2015) [7], which is a popular dataset in the community [10][12]. It involves multiple malware samples from different families.

Each malware sample has two parts:

- A **.bytes file**, which contains the raw program code in hexadecimal form.
- An **.asm file**, which contains the program's instructions after being converted into assembly language.

These two files help the system understand how each malware family is built and how it behaves, as illustrated in Figure 2.1.

The dataset contains a total of 21,736 malicious samples, split evenly between .bytes files and .asm files, 10,868 of each, and weighing about 200 GB.

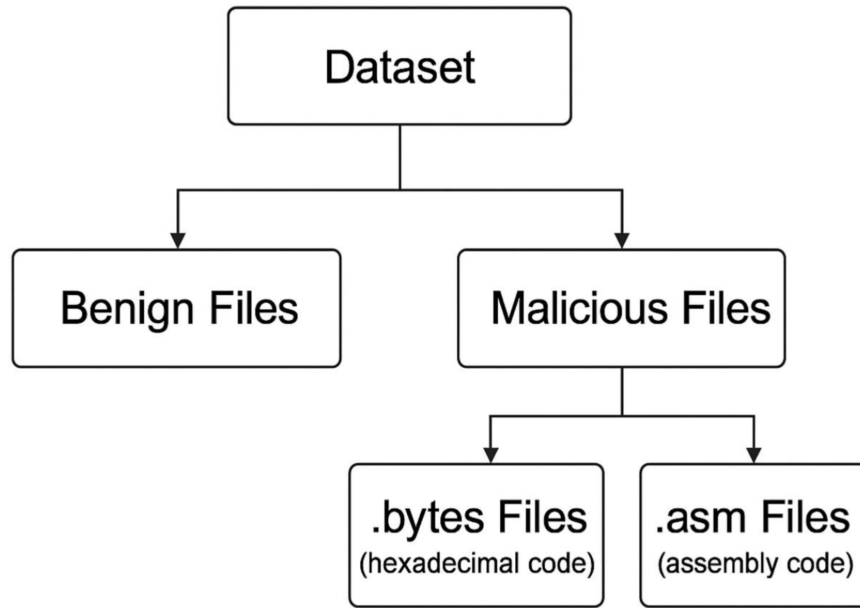


Figure 2.1: Description of the Dataset

The malware samples are from 9 different families. Table 2.1 shows the breakdown of samples among the different families involved.

These malware families exhibit a wide range of behaviors. For example, Ramnit behaves like a worm, spreading automatically across systems. Kelihos_ver3 functions as a backdoor, communicating with remote command-and-control (C&C) servers. Lollipop and Vundo operate as adware and trojans, disguising themselves as legitimate software. Meanwhile, Obfuscator.ACY conceals or encrypts its code to evade detection.

Family Name	Train Samples	Type
Ramnit	1541	Worm
Lollipop	2478	Adware
Kelihos_ver3	2942	Backdoor

Vundo	475	Trojan
Simda	42	Backdoor
Tracur	751	Trojan Downloader
Kelihos_ver1	398	Backdoor
Obfuscator.ACY	1228	Any kind of obfuscated malware
Gatak	1013	Backdoor

Table 2.1: Malware Families

The benign dataset is a compilation of files from trusted sources such as built-in Windows programs, and various open-source applications checked by scanning them using antiviruses.

The benign dataset consists of safe, clean files from common regular software programs from trusted sources such as Notepad, Paint, Calculator, and a broad range of open-source programs. In total, the benign dataset contains 16719 samples.

Each of the files in the benign dataset was scanned with several antivirus tools such as Windows Defender to confirm that it was completely clean. This step was very important to make sure that the system learns correctly what safe software looks like.

To keep the dataset consistent, the same steps were followed for the benign files as for the malware samples. Each file was processed to create a .bytes version and an .asm version, so that both safe and malicious data share the same format. This helps the machine learning models compare them more fairly and identify what makes a program harmful or harmless.

2.2 Data Preprocessing

Before training the machine learning models, the collected data needed to be cleaned, organized, and converted into usable form. Raw .bytes and .asm files usually contain secondary information such as headers, symbols, or text formatting that do not always contribute to learning patterns. If this extra data is not removed, it can slow down the model and reduce accuracy.

The preprocessing stage ensures that all the samples, both benign and malicious, follow the same structure and format. This step is very important because inconsistent data can cause serious problems during training and testing. The goal of preprocessing is to make the dataset clean, balanced, and ready for the next step of feature extraction.

2.2.1 Data Cleaning and Normalization

Preprocessing starts with data cleaning. This involves stripping out anything from the raw files that doesn't help with feature extraction. Raw binaries often contain a large number of low-level metadata, formatting artifacts, and automatically generated content that do not contribute meaningful information to the classification task. Removing unnecessary material ensures that the learning algorithm focus only on patterns that actually describe program behavior.

For the malicious dataset, the files came in .bytes and .asm formats, but they still had a lot of material that are not useful for learning. The .bytes files contained many empty spaces and long stretches of repeated hexadecimal values. These repetitive patterns typically arise from padding, unused memory regions, or compiler-generated structures, which add noise but provide no valuable behavioral insight. The .asm files had memory addresses, random comments, and compiler information none of which show how the program works. So, all that extra noise must be removed, leaving just the parts that reveal what the malware really does. Additionally, removing unused bytes, repeated hexadecimal sequences, memory addresses, and other non-informative elements helps reduce dimensionality, speeds up feature extraction, and prevents the model from learning misleading or irrelevant patterns.

.asm file

```
.text:00401000                                                    assume es:nothing,
ss:nothing, ds:_data,   fs:nothing, gs:nothing
.text:00401000 56                                                 push    esi
.text:00401001 8D 44 24 08                                        lea     eax, [esp+8]
.text:00401005 50                                                 push    eax
.text:00401006 8B F1                                              mov     esi, ecx
.text:00401008 E8 1C 1B 00 00                                     call    ??
0exception@std@@QAE@ABQBD@Z ; std::exception::exception(char const * const &)
.text:0040100D C7 06 08 BB 42 00                                  mov     dword ptr [esi],
offset off_42BB08
.text:00401013 8B C6                                              mov     eax, esi
.text:00401015 5E                                                 pop     esi
.text:00401016 C2 04 00                                           retn    4
.text:00401016                                         ; -------------------------------------
-------------------------------------
.text:00401019 CC CC CC CC CC CC CC                               align 10h
.text:00401020 C7 01 08 BB 42 00                                  mov     dword ptr [ecx],
offset off_42BB08
.text:00401026 E9 26 1C 00 00                                     jmp     sub_402C51
.text:00401026                                         ; -------------------------------------
-------------------------------------
.text:0040102B CC CC CC CC CC                                     align 10h
.text:00401030 56                                                 push    esi
.text:00401031 8B F1                                              mov     esi, ecx
.text:00401033 C7 06 08 BB 42 00                                  mov     dword ptr [esi],
offset off_42BB08
.text:00401039 E8 13 1C 00 00                                     call    sub_402C51
.text:0040103E F6 44 24 08 01                                     test    byte ptr
```

Figure 2.2: Structure of .asm file

.bytes file

```
00401000 00 00 80 40 40 28 00 1C 02 42 00 C4 00 20 04 20
00401010 00 00 20 09 2A 02 00 00 00 00 8E 10 41 0A 21 01
00401020 40 00 02 01 00 90 21 00 32 40 00 1C 01 40 C8 18
00401030 40 82 02 63 20 00 00 09 10 01 02 21 00 82 00 04
00401040 82 20 08 83 00 08 00 00 00 00 02 00 60 80 10 80
00401050 18 00 00 20 A9 00 00 00 00 04 04 78 01 02 70 90
00401060 00 02 00 08 20 12 00 00 00 40 10 00 80 00 40 19
00401070 00 00 00 00 11 20 80 04 80 10 00 20 00 00 25 00
00401080 00 00 01 00 00 04 00 10 02 C1 80 80 00 20 20 00
00401090 08 A0 01 01 44 28 00 00 08 10 20 00 02 08 00 00
004010A0 00 40 00 00 00 34 40 40 00 04 00 08 80 08 00 08
004010B0 10 00 40 00 68 02 40 04 E1 00 28 14 00 08 20 0A
004010C0 06 01 02 00 40 00 00 00 00 00 00 20 00 02 00 04
004010D0 80 18 90 00 00 10 A0 00 45 09 00 10 04 40 44 82
004010E0 90 00 26 10 00 00 04 00 82 00 00 00 20 40 00 00
004010F0 B4 00 00 40 00 02 20 25 08 00 00 00 00 00 00 00
00401100 08 00 00 50 00 08 40 50 00 02 06 22 08 85 30 00
00401110 00 80 00 80 60 00 09 00 04 20 00 00 00 00 00 00
00401120 00 82 40 02 00 11 46 01 4A 01 8C 01 E6 00 86 10
00401130 4C 01 22 00 64 00 AE 01 EA 01 2A 11 E8 10 26 11
00401140 4E 11 8E 11 C2 00 6C 00 0C 11 60 01 CA 00 62 10
00401150 6C 01 A0 11 CE 10 2C 11 4E 10 8C 00 CE 01 AE 01
00401160 6C 10 6C 11 A2 01 AE 00 46 11 EE 10 22 00 A8 00
00401170 EC 01 08 11 A2 01 AE 10 6C 00 6E 00 AC 11 8C 00
00401180 EC 01 2A 10 2A 01 AE 00 40 00 C8 10 48 01 4E 11
00401190 0E 00 EC 11 24 10 4A 10 04 01 C8 11 E6 01 C2 00
```

Figure 2.3: Structure of. bytes file

Figures 2.2 and 2.3 show examples of the raw structure of .bytes and .asm files, respectively, before cleaning. The .bytes file displays hexadecimal values that represent raw machine code, while the .asm file shows disassembled assembly instructions.

For the benign dataset, the files were collected in executable (.exe) format from safe and verified sources. To make them compatible with the malware samples, each .exe file was converted into. bytes and .asm formats using disassembly tools. This step ensured that both benign and malicious samples shared the same structure, making comparison fair and consistent.

After cleaning, data normalization was applied. Different files can vary a lot in size: some malware samples are very large, while some benign programs are quite small. Normalization helps bring all features into a similar range, so that one file type does not dominate the learning process. In this project, Min-Max normalization was used to standardize all feature values. Each feature x_i was scaled to the [0,1] range using the formula.

$$x_i' = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

Where $x_{\min}$ and $x_{\max}$ represent the minimum and maximum values of that feature across all samples. This ensures that both benign and malicious files have the same influence during training. In other words, very large files or features with naturally higher values do not overpower smaller files or low-value features.

2.3 Feature Extraction

To identify relevant features, we convert the raw code into numerical features that describe the program's structure, behavior, or frequency of certain patterns. This process helps transform the unreadable code into measurable values, such as how often a specific instruction appears, how certain bytes are distributed, or what types of operations occur most frequently.

2.3.1 Byte-Level Features (.bytes files)

The .bytes file is a direct dump of the program's binary data written in hexadecimal format.

Each line in a .bytes file contains a memory address followed by a sequence of hexadecimal byte values representing the program raw machine code.

For example, consider the following structure: 00401000 8B FF 55 8B EC 83 EC ?? 53 56 57 8B F1.

The first part (00401000) is simply the memory address. Since it only indicates where the instruction is located in memory, it does not provide any useful information for classification and is therefore ignored during preprocessing. The second part (8B FF 55 8B EC 84 EC ??) contains the hexadecimal byte values that represent the program's actual machine instructions. These bytes are the meaningful component of the .bytes file, because they capture the low-level behavior of the program and are consistent across samples of the sample malware family.

To prepare these values for machine learning, each hexadecimal token is converted into its corresponding decimal value in the range [0-255]. This conversion is done programmatically by interpreting each token as a base – 16 number and converting it into its base-10 representation. For example, 8B becomes 139, FF becomes 255, and any unknown bytes marked as ?? are replaced with a neutral placeholder such as 0.

This conversion produces a clean, numerical sequence for every .bytes file, allowing the raw machine code to be represented as a structured feature vector that machine learning models can analyze consistently across all samples.

2.3.2 Assembly-Level Features (.asm files)

The .asm files are disassembled versions of the program that show what the program does in assembly language. Each line in an .asm file contains the following structure:

Structure: .text:00401000 push ebp mov.

The first part (.text:00401000) is the memory address. The second part (push ebp) contains the opcode (instruction) and its operand. The opcode (like mov, push, call, jmp) tells what operation

the program is performing. We focused only on the opcodes because they describe the actual actions of the programs, such as moving data, jumping to another location, or calling a function.

The extracted opcodes were then cleaned by removing unwanted symbols and converted into numbers by computing the Term Frequency-Inverse Document Frequency (TF-IDF) measures. TF-IDF helps quantify how frequently each opcode appears in a given file relative to how common or rare it is across the entire dataset. This produces a numerical representation where frequently used instructions receive higher global importance. TF-IDF helps the machine learning model understand which instructions are frequent and which ones are rare, enabling it to learn meaningful behavioral patterns that distinguish one malware family from another.

2.4 Term Frequency – Inverse Document Frequency (TF-IDF)

The TF-IDF was computed to convert the assembly instructions extracted from the .asm files into numerical values that a machine learning model can understand. TF-IDF is a common text-representation technique that helps identify how important a particular word, or in this case an instruction (opcode), is within a document compared to all other documents in a dataset [17].

The TF-IDF consists of 2 terms: the term frequency (TF) and the inverse document frequency (IDF).

In our case, the TF is computed as a measure of how often a certain opcode appears in a file. The more frequent an instruction occurs in a document, the greater its TF number is. It allows identifying what the malware is actually doing most of the time. For instance, if the opcode mov keeps occurring in a program, that just means the malware spends a lot of time moving data around between registers.

The TF is defined as follows:

$$TF(t,d) = \frac{f_{t,d}}{\sum_k f_{k,d}}$$

where $f_{t,d}$ is the number of times the opcode t appears in the file d, and the denominator represents the total number of opcodes in that file.

The IDF calculates the instruction's rarity or uniqueness throughout the dataset. Opcodes like **push** or **mov** that occur in nearly every file are seen as less informative and are assigned a lower weight. An instruction like **xor** or **jmp**, on the other hand, gains more value and weight for classification if it only occurs in a small number of malware families.

The IDF is defined as follows:

$$IDF(t) = \log\left(\frac{N}{n_t}\right)$$

Where N is the total number of files in the dataset, and n_t is the number of files that contain the instruction t.

The final TF-IDF score is calculated by multiplying these two values ($TF \times IDF$). This means that instructions that occur frequently in a particular file but rarely in others are given the highest importance [15]. Through this process, every .asm file is transformed into a numerical vector, where each number represents the significance of a specific instruction.

2.5 Splitting of the Dataset

In our experiments, for model training and evaluation, we split the dataset into three parts: training, validation, and testing. By doing this, we make sure the model learns from one chunk of data, gets fine-tuned on another, and encounters a fresh set when it is time to see how well it really performs. This setup keeps the model from just memorizing the data and helps it handle new information better [16][17].

The project consists of a two-stage classification pipeline. In the first stage, a binary classifier (benign vs. malicious) was trained using an 80/20 train–test split, because this task does not involve multiple classes and does not require a separate validation set. In the second stage, a multi-class malware family classifier was trained exclusively on malicious samples. Since this malicious dataset contained nine malware families, a 70/15/15 split (train/validation/test) was used to ensure balanced evaluation across families.

The combined malware dataset consists of 10,868. bytes files and 10,868 .asm files, making a total of approximately 21,700 malware samples. The model's purpose was to learn the distinction between benign and malicious files. The benign dataset was included only in the binary classification stage, where the goal was to distinguish between benign and malicious files. The overall train test split was 80/20, applied after combining the benign and malware samples into a single dataset. In other words, the benign data was not split independently but was part of the same stratified split as the malware data, ensuring both classes appeared proportionally in training and testing. Since this step served only as an initial detection layer, performing a separate split on benign data alone would not have added any benefit.

The .bytes and .asm datasets were processed and split separately because they contain different types of features, numerical for .bytes and textual for .asm. Although both types were ultimately converted into numerical feature vectors for model training, they required different preprocessing steps: raw hexadecimal bytes were converted to decimal values, while assembly instructions .asm were transformed using TF-IDF [15]. By handling each data type on its own, we can build models that really fit the unique features of each one.

To maintain fairness, we used a stratified split, so every subset (i.e. training, validation, and testing) has the same mix of malware families. That way, each family gets equal attention in every subset. We split the data as follows:

- **70% for training** (used to train the model)
- **15% for validation** (used for parameter tuning and model optimization)
- **15% for testing** (used for final performance evaluation)

The splitting was performed using the train_test_split () function from scikit-learn library with the stratify parameter applied to the class labels to preserve the original family distribution. This yielded the following number of samples for each category (.asm or .bytes): 7607. 1630, 1631 for

train, validation, and test, respectively. By extracting 2567 features for each sample, we end up with the following data dimensions:

Train: (7607, 256) **Validation:** (1630, 256) **Test:** (1631, 256)

By keeping the .bytes and .asm datasets separate, the models learned both the structure and behavior of malware. That helped boost accuracy and made the classifications more reliable overall.

2.6 Synthetic Minority Oversampling Technique (SMOTE)

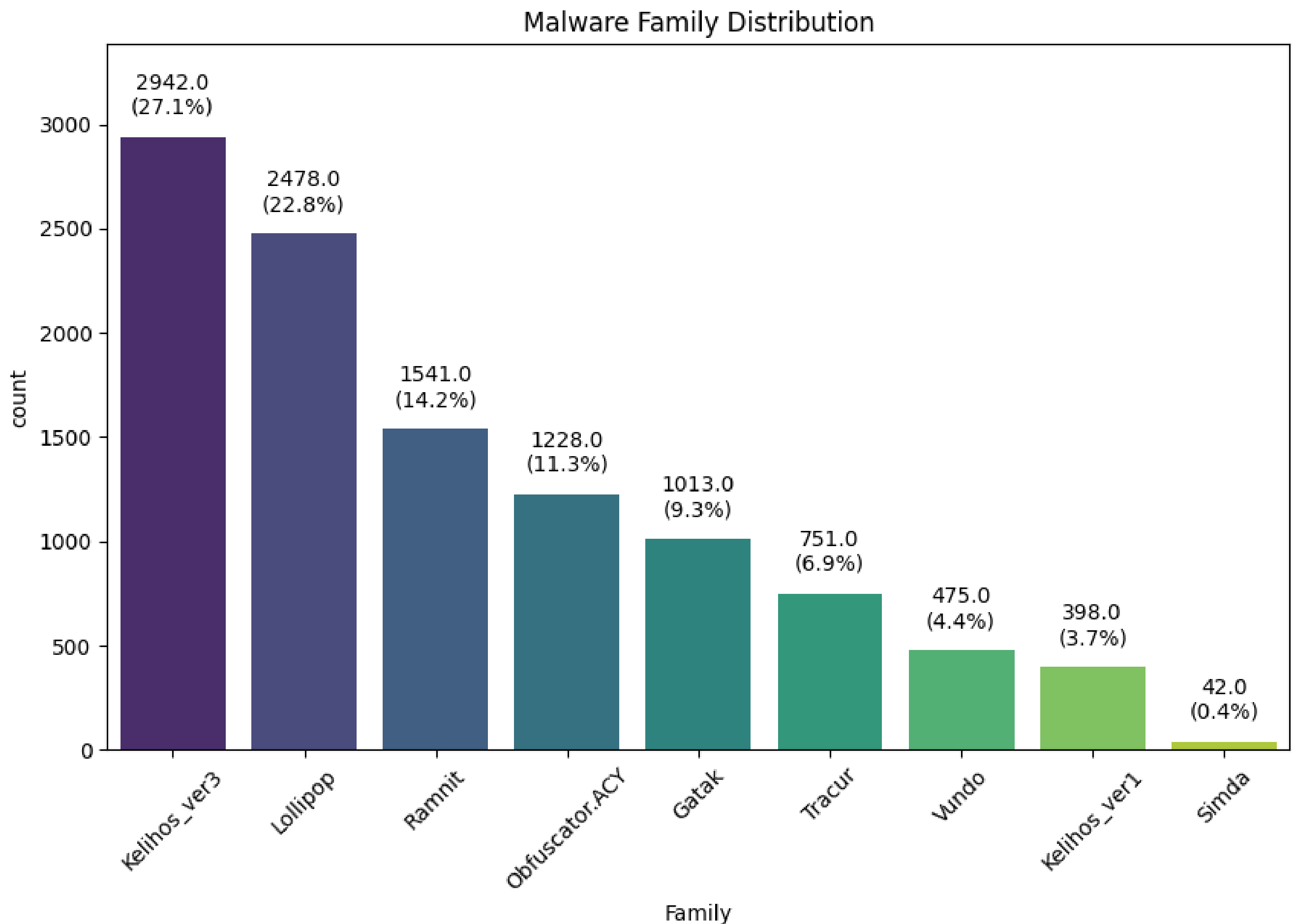


Figure 2.4: Malware family distribution in the dataset

Figure 2.4 shows the overall malware family distribution across all samples. The distribution demonstrates that the dataset is imbalanced, with families such as Kelihos_ver3 (27.1%) and Lollipop (22.8%) containing far more samples than smaller families like Simda (0.4%) and Kelihos_ver1 (3.7%). This observation highlights the importance of using balancing techniques to achieve uniform representation before training.

To address this issue, we used the Synthetic Minority Oversampling Technique (SMOTE). Rather than just copying data from the smaller classes, it mixes similar points from the minority class to create new, realistic examples. This makes the dataset more balanced, so each class gets a fair shot when the model learns [16][17].

In this project, I used SMOTE only on the training set, right after the stratified split of the dataset. The validation and test sets were kept unchanged to preserve the natural class distribution and ensure that the model's performance was evaluated under real-world conditions.

After SMOTE was applied, the training set size increased because each of the nine malware families was balanced to 2059 samples [7]. This resulted in a new, fully balanced training dataset with a total of 18,531 samples, while the validation and testing sets remained unchanged [17][18]. This gave the following new breakdown:

- Balanced Training Set: 2059 x 9 = 18,531
- Validation Set: 1630 samples (unchanged)
- Test Set: 1631 samples (unchanged)

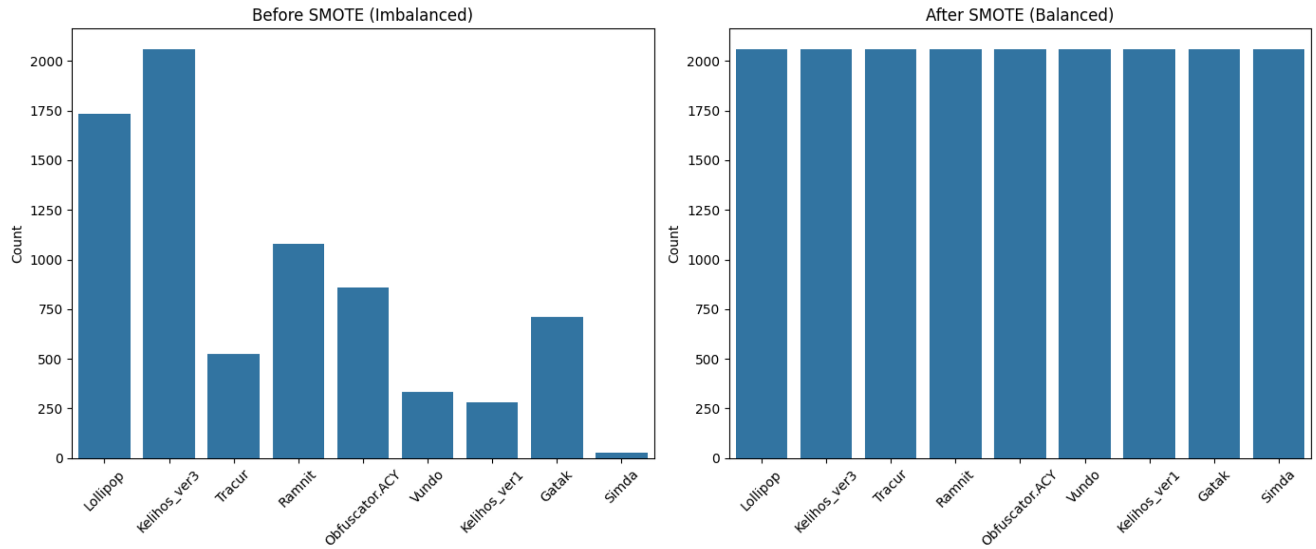


Figure 2.5: After applying SMOTE

Figure 2.5 illustrates the effect of applying SMOTE on the training dataset. Before balancing, a few malware families such as Lollipop and Kelihos_ver3 contained a much larger number of samples, whereas families like Simda and Kelihos_ver1 had very few. After applying SMOTE, the dataset became balanced, with each family having approximately the same number of samples.

This equal representation ensured that the machine learning models could learn features from all malware families without bias toward any dominant class, ultimately improving classification accuracy and generalization performance.

Chapter 3: Machine Learning Model Development

In this chapter, I walk through how I built and trained the machine learning models for malware detection and classification.

Figure 3.1 depicts our proposed framework, which consists of first establishing whether a file is benign or malicious through binary classification, and if the latter is true, determine through multiclass classification the corresponding malware family.

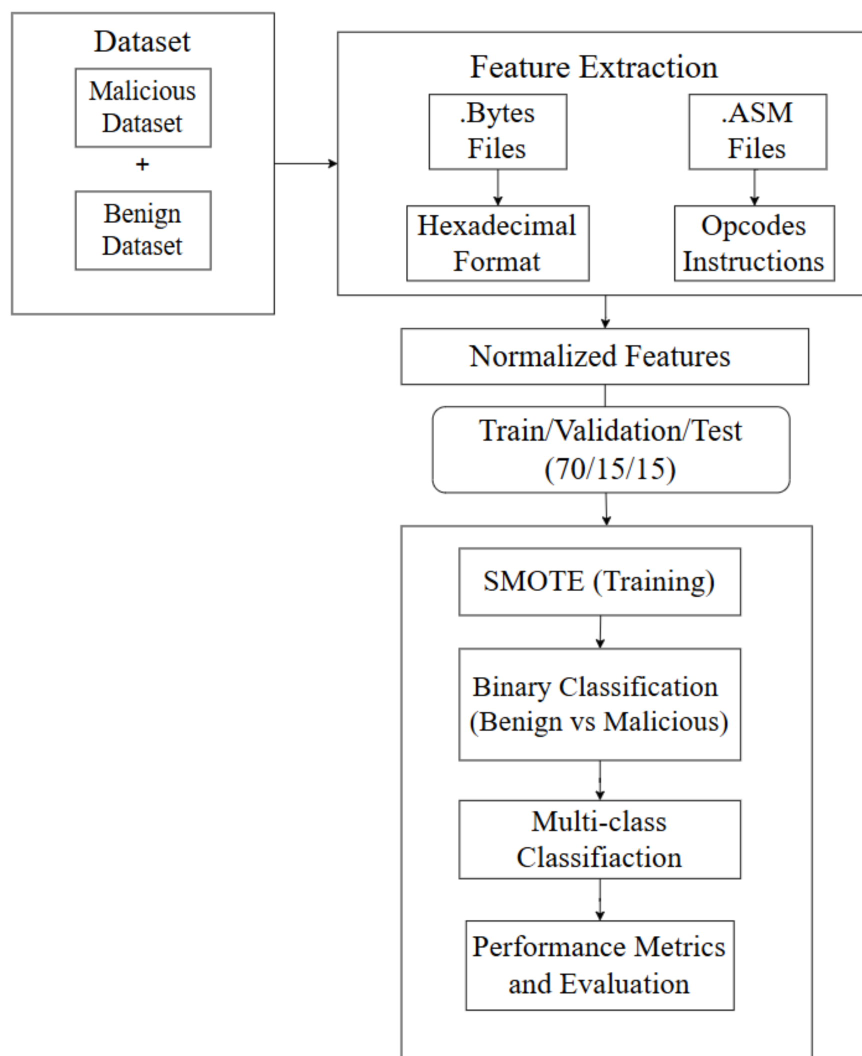


Figure 3.1: Overall framework of malware classification process.

3.1 Model Selection

One of the most important steps in creating an effective malware categorization system is model selection. Finding models that can effectively learn from patterns in both. bytes and.asm feature representations for high accuracy and generalization performance is the aim.

In this project, the classification process was divided into two stages to improve accuracy and interpretability.

- **Stage 1 - Binary Classification:**
  The first stage determines whether a given file is benign or malicious. For this a Logistic Regression model was chosen due to its simplicity, interpretability, and strong performance on linearly separable data [18].

- **Stage 2 – Multiclass Malware Classification**
  Once a file is flagged as malicious, it is passed to the second stage, where the model identifies which malware family it belongs to among the nine classes in our dataset. In this stage, multiple machine learning algorithms were tested and compared to identify the most effective model for malware family classification, including XGBoost, LightGBM, Support Vector Machine (SVM), and K-Nearest Neighbors (KNN) [19].

Random Forest was initially tested as a baseline model, but its performance did not align with the project goals, particularly due to inconsistent results on minority malware families. Therefore, Logistic Regression was selected for the first stage as it provided more stable, consistent, and interpretable outcomes.

3.1.1 Logistic Regression (LR)

LR is a machine learning method that sorts data into two groups like benign and malicious. It looks for patterns between the inputs and the labels you want to predict. In this project, we used LR right at the start to tell apart benign files from malware. It is quick and reliable, so the model can make fast decisions before diving into the complex task of figuring out which malware family a file belongs to [20]. In addition, LR helps to establish a simple but strong baseline for evaluating how well more advanced models perform.

3.1.2 Extreme Gradient Boosting (XGBoost)

XGBoost is a powerful and widely used machine learning algorithm based on decision trees. It builds many small trees one after another, and each new tree tries to fix the mistakes made by the previous ones. By combining the results of all trees, the final model becomes stronger and more accurate than a single tree.

The main idea behind XGBoost is boosting, which means improving performance step by step. Each tree learns from the errors of the earlier trees and gives more attention (weight) to the samples that were classified incorrectly. This helps the model reduce errors and improve its predictions.

In this project, XGBoost was used in the second stage of the system to classify malware into one of the nine families. It was chosen because of its high accuracy, fast computation, and robustness against noise in the data [21].

3.1.3 Light Gradient Boosting Machine (LightGBM)

LightGBM is another tree-based machine learning method, with some similarities with XGBoost. Both build many small decision trees, one after another, to boost accuracy. However, LightGBM is built for speed. It uses less memory too, so, it is effective when working with huge datasets. Instead of growing trees level by level like most methods, LightGBM grows them leaf by leaf.

It always first picks the leaf with the greatest information gain. So, it finds smarter splits and usually needs fewer trees to achieve better performance results. In this project, I used LightGBM for multiclass malware family classification. Speed, low memory use, and the way it handles both .bytes and .asm features made it an appropriate choice [22]. It handles these features well because .bytes and .asm data are high-dimensional and sparse, and LightGBM's histogram-based learning and leaf-wise growth are designed to efficiently process such feature structures.

3.1.4 Support Vector Machine (SVM)

SVM is a supervised learning algorithm that separates data into different groups by finding the best possible boundary, known as hyperplane. It tries to maximize the margin or the distance between the hyperplane and the nearest data points of each class, called support vectors because a

larger margin leads to better generalization and helps the model make more reliable decisions on new, unseen malware samples.

For non-linear data, SVM uses a mathematical trick called a kernel function, which maps the data into a higher-dimensional space where it becomes easier to separate. This transformation allows SVM to handle complex feature relationships commonly found in malware datasets, where patterns are rarely linearly separable. In this project, the Radial Basis Function (RBF) kernel was used because it works well for complex, non-linear patterns in malware data [24].

3.1.5 K-Nearest Neighbors (KNN)

The KNN algorithm is one of the simplest and most intuitive machine learning methods. Instead of learning patterns during training, it stores all the training data and makes decisions when a new sample is introduced. When a new file is tested, the algorithm looks for other samples in the dataset that are most similar to it. These nearby samples are called neighbors. The algorithm then checks which malware family is most common among those neighbors and assigns that family to the new file. In simple terms, KNN assumes that files with similar characteristics are likely to belong to the same class [23].

Chapter 4: Experimental Evaluation

In this chapter, we evaluate the performance of the proposed framework using the dataset introduced earlier. The evaluation is conducted using a laptop, with the system specifications detailed in Table 4.1.

Table 4.1: Evaluation Hardware Specifications

Component	Details
Brand	ASUS ROG Zephyrus
OS Version	Windows 11 Home, Version 24H2 (Build 26100.6584)
Processor	12th Gen Intel® Core™ i7-12700H
Memory Capacity	16 GB DDR4 RAM
CPU Speed	2.3 GHz (up to 4.7 GHz Turbo Boost)
Core Count	14 Cores (6 Performance + 8 Efficiency Cores)

4.1 Evaluation Metrics

To evaluate the performance of the machine learning models, several metrics were employed, including F1-Score, Precision, Recall, Accuracy, Macro Average, and Weighted Average. These metrics help assess how effectively the model identifies malware and distinguishes it from benign files.

Precision indicates how often the model gets it right when it classifies a sample as malware. In other words, when the model flags a file as malicious, precision shows the chance it really is malicious. Precision is computed as the number of files the model correctly classifies as malware called True Positives (TP) divided by the total number of files it classified as malware including both the ones it gets right (True Positives) and the ones it gets wrong called False Positives (FP).

Precision is computed as follows:

$$Precision = \frac{True\ Positive\ (TP)}{True\ Positive\ (TP) + False\ Positive\ (FP)}$$

Higher precision means the model yields a reduced number of false alarms [28].

Recall measures how many actual malware files the model detects. Recall is calculated as the number of true positives divided by the total number of real malware files, both the ones it found (True Positives) and the ones it missed also called False Negatives (FN).

Recall is defined as follows:

$$Recall = \frac{True\ Positive\ (TP)}{True\ Positive\ (TP)\ +\ False\ Negative\ (FN)}$$

Accuracy measures how often the model gets things right. It counts both the malware the model flags correctly (True Positives) and the benign files it identifies as safe also called True Negatives (TN).

Accuracy is computed as follows:

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

The F1-Score mixes Precision and Recall into one number, striking a balance between the two. This really matters when the dataset is unbalanced. The F1-Score helps ensure that the model is just good at finding malware, but also at not missing any.

F1-Score is computed as follows:

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

Macro Average treats all malware families equally, no matter how many samples each one has. It calculates Precision, Recall, or F1-Score for every class and then takes a simple average. This metric is useful when the dataset is imbalanced, because it shows how the model performs on all classes, including the small ones.

Macro Average is computed as follows:

$$Macro\ Average = \frac{1}{K} \sum_{i=1}^{K} M_i$$

Where M_i is the metric value (Precision, Recall or F1-Score) for every class, and K is the total number of classes.

Weighted Average also combines the per-class scores, but it gives more weight to classes that have more samples. This helps create a balanced score that reflects the true distribution of the dataset. It prevents large classes from dominating the results while still considering the smaller ones.

Weighted Average is computed as follows:

$$Weighted\text{-}F1 = \sum_{i=1}^{N} w_i \cdot F1_i$$

$$w_i = \frac{number\ of\ samples\ in\ class\ i}{total\ number\ of\ samples}$$

Where w_i is the proportion of samples that belong to class i. It is calculated as the number of samples in class i divided by the total number of samples across all classes. Here, class i refers to any malware family (e.g., Ramnit, Kelihos_ver3, Obfuscator.ACY, etc.), and total samples represents the total number of malware samples used in the multiclass classification dataset.

4.2 Performance Evaluation on Binary Classification

The first stage of the malware detection framework focuses on binary classification, where the goal is to determine whether a given file is benign or malicious. Logistic Regression was chosen for this task because it is a simple yet powerful algorithm for binary classification problems, capable of handling large datasets efficiently while providing clear probabilistic outputs.

Figure 4.1 shows the confusion matrix for the Logistic Regression model on the dataset. Out of 38,455 total samples, 15934 benign and 20599 malicious files were correctly classified. Only a small number of samples were misclassified: 785 benign samples were wrongly detected as malicious, and 1177 malicious samples were incorrectly marked as benign.

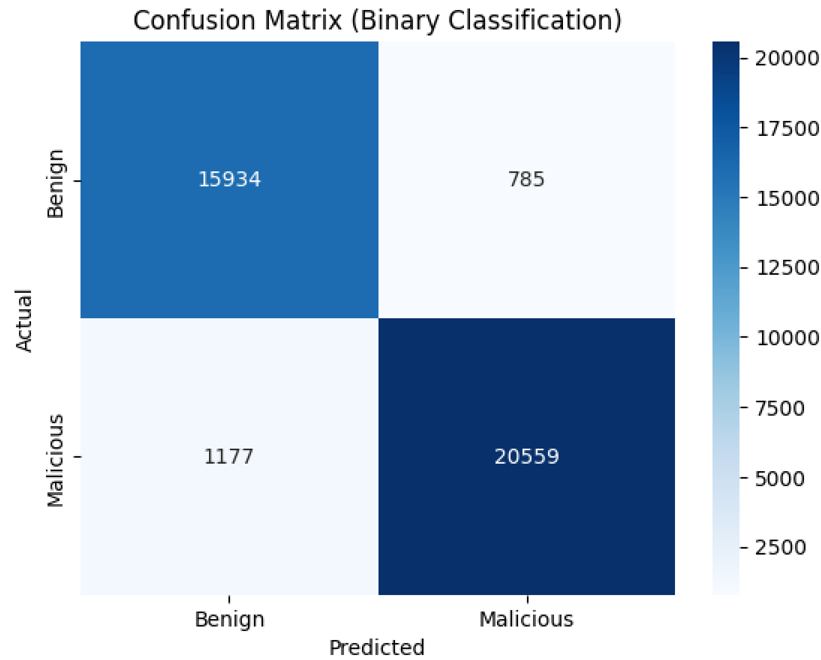


Figure 4.1: Confusion Matrix for Binary Classification using Logistic Regression

The model achieved an overall accuracy of 95%, as shown in the classification report in Table 4.2. The precision, recall, and F1-score values were all around 0.95, indicating consistent and stable performance across both classes. The slightly higher precision for the malicious class (0.96) shows that the model is more confident when identifying infected files, with very few false detections.

	Precision	**Recall**	**F1-Score**	**Support**
Benign	0.93	0.95	0.94	16719
Malicious	0.96	0.95	0.95	21736
accuracy			0.95	38455
Macro avg	0.95	0.95	0.95	38455

Weighted-avg	0.95	0.95	0.95	38455

Table 4.2: Classification Report for binary classification

These results confirm that the Logistic Regression model provides a strong initial layer of detection, effectively filtering benign files from harmful ones before passing the malicious samples to the second-stage multi-class modeling for family classification.

4.3 Performance Evaluation on bytes for Multi-Class Classification

Once the first round of binary classification has identified the malicious files from the benign ones, the focus is to identify the corresponding families. In this subsection, we present the multi-class results using the four different machine learning models considered in the project based on the .bytes sample files.

4.3.1 XGBoost

The Extreme Gradient Boosting (XGBoost) model delivered the best overall performance on the .bytes dataset, achieving high accuracy and stability. Most samples fall along the diagonal, indicating that the model was able to clearly separate the byte-level patterns of diagonal families. Figures 4.2 and 4.3 show the obtained confusion matrix and training/validation accuracy curve, respectively.

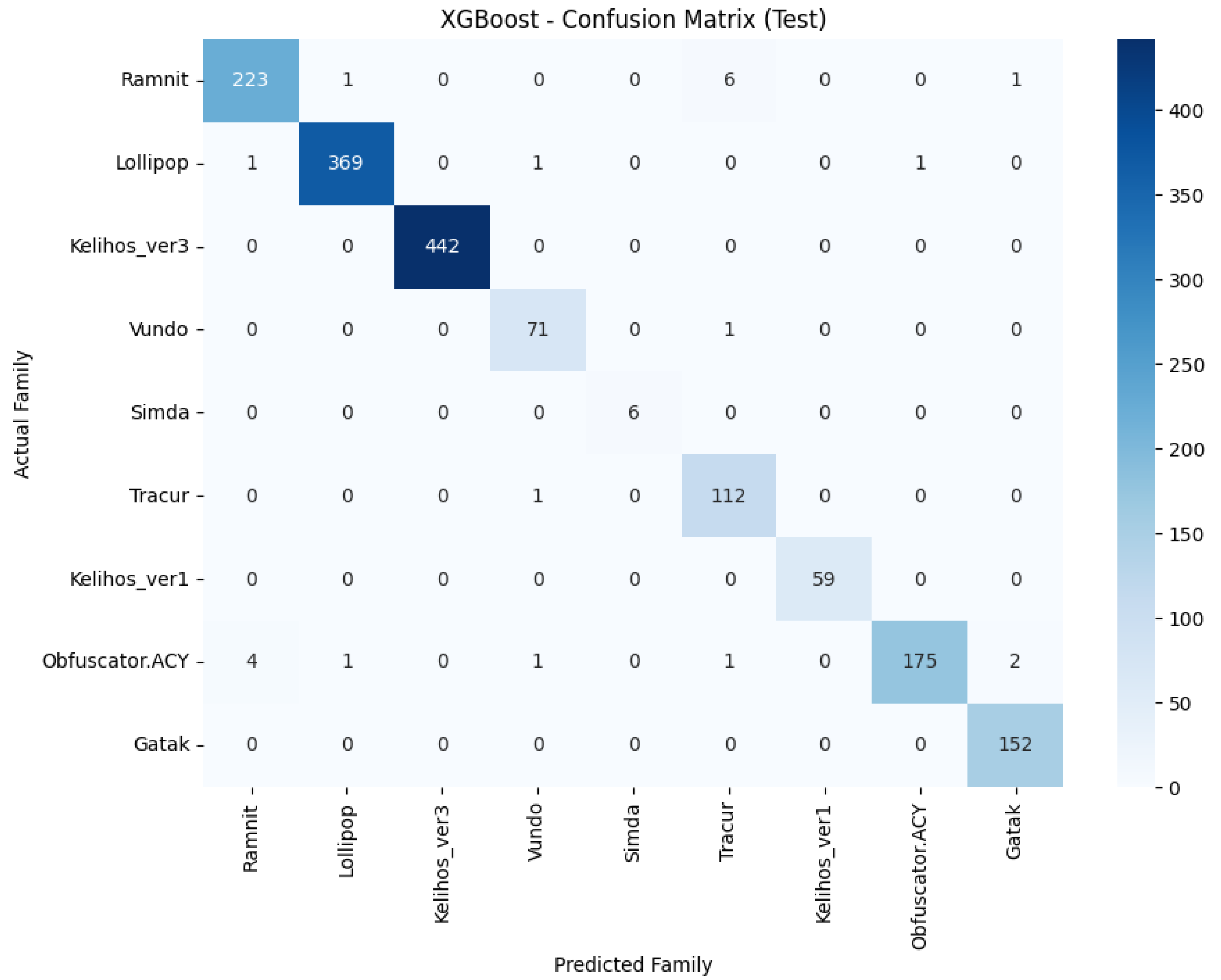


Figure 4.2: Confusion Matrix for XGBoost Model on .bytes samples

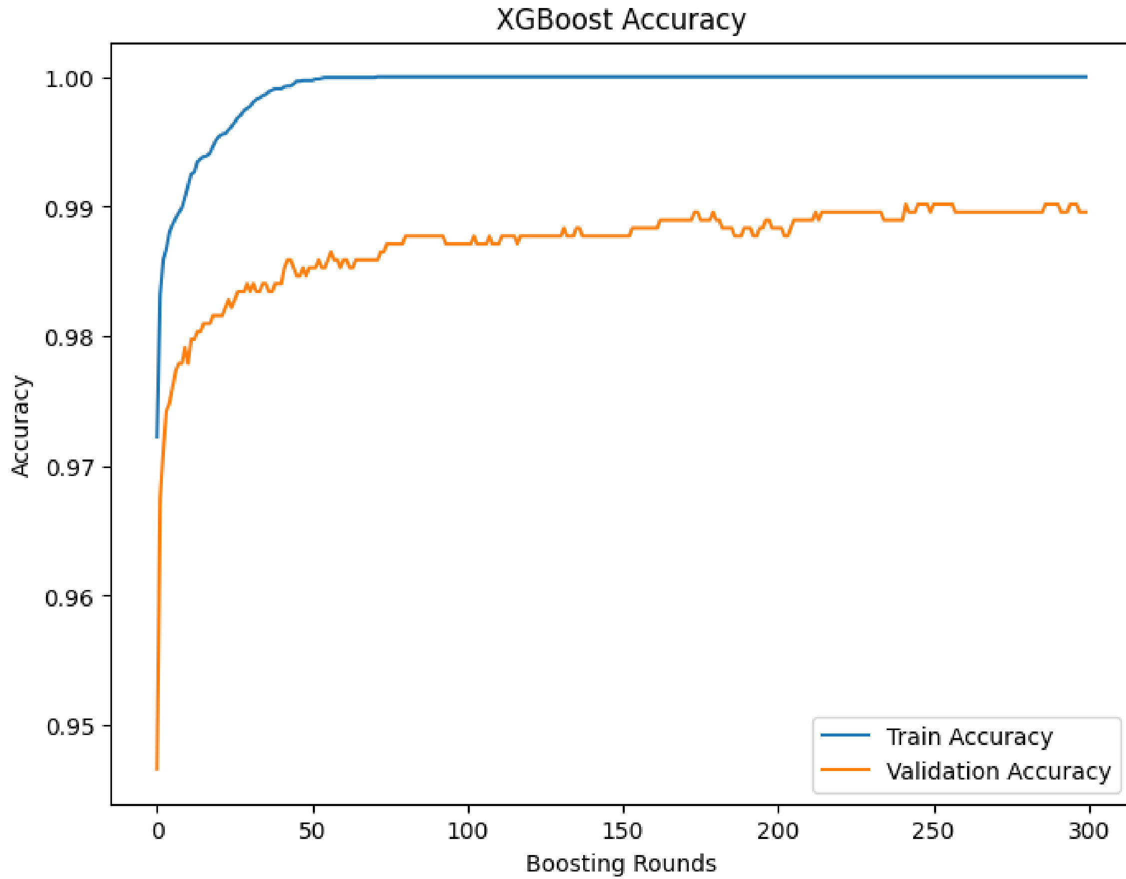


Figure4.3: Training and Validation Accuracy Curve for XGBoost Model

As shown by the per-class classification report depicted by Table 4.3, XGBoost achieved an overall accuracy of 99%, delivering consistently high precision, recall, and F1-scores across all families.

Malware Family	**Precision (%)**	**Recall (%)**	**F1-Score (%)**	**Accuracy (%)**
Ramnit	0.98	0.97	0.98	
Lollipop	0.99	0.99	0.99	

Kelihos_ver3	1.00	1.00	1.00	0.99
Vundo	0.96	1.00	0.98	
Simda	1.00	1.00	1.00	
Tracur	0.94	0.99	0.97	
Kelihos_ver1	1.00	1.00	1.00	
Obfuscator.ACY	1.00	0.95	0.97	
Gatak	0.98	0.99	0.99	

Table 4.3: XGBoost Classification Report on .byte

4.3.2 LightGBM

The Light Gradient Boosting Machine (LightGBM) model also performed strongly on the .bytes dataset, achieving high accuracy and demonstrating stable convergence. The confusion matrix shows strong diagonal dominance, indicating accurate predictions for almost all classes. Figures 4.4 and 4.5 show the confusion matrix and training/validation accuracy curve, respectively, for the LightGBM model on the .bytes dataset.

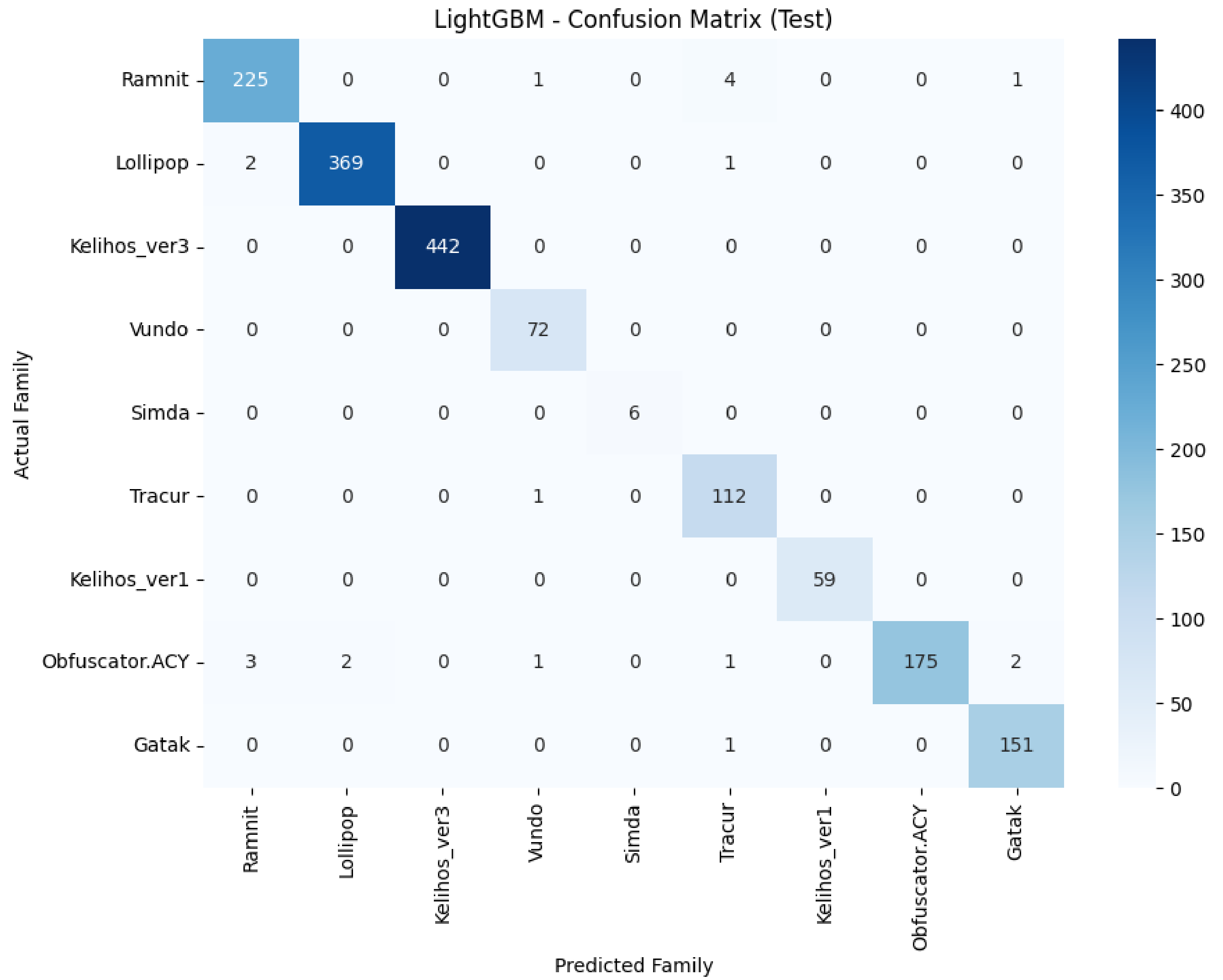


Figure 4.4: Confusion Matrix for LightGBM Model

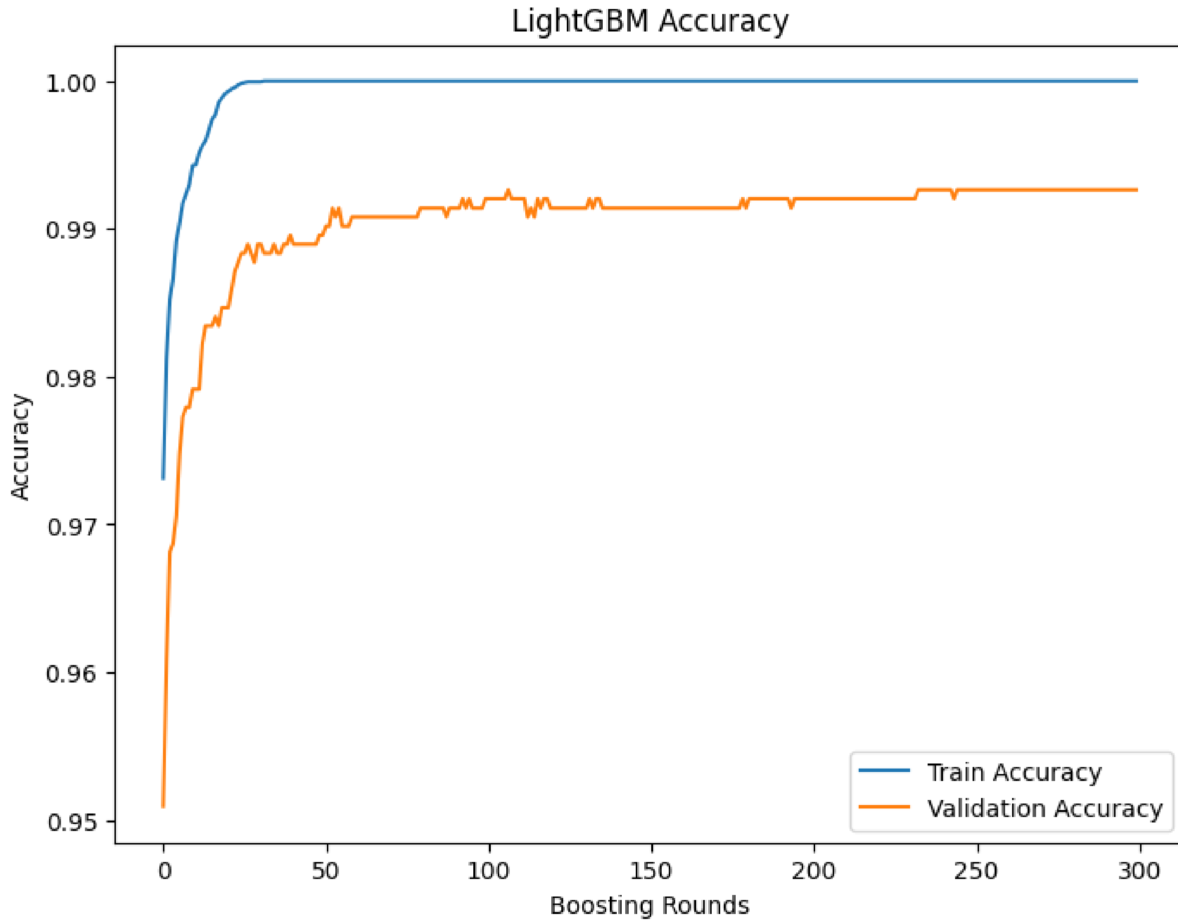


Figure 4.5: Training and Validation Accuracy Curve for LightGBM

As shown by the per-class classification report depicted by Table 4.4, LightGBM achieved an overall accuracy of 98.0%, performing exceptionally well across most families such as Kelihos_ver3, Simda, and Obfuscator.ACY, which showed nearly perfect classification.

Malware Family	Precision (%)	Recall (%)	F1-Score (%)	Accuracy (%)
Ramnit	0.97	0.97	0.97	

Lollipop	0.99	0.99	0.99	0.98
Kelihos_ver3	1.00	1.00	1.00	
Vundo	0.96	1.00	0.97	
Simda	1.00	1.00	1.00	
Tracur	0.94	0.99	0.96	
Kelihos_ver1	1.00	1.00	1.00	
Obfuscator.ACY	1.00	0.95	0.97	
Gatak	0.98	0.99	0.98	

Table 4.4: LightGBM Classification Report on .byte

4.3.3 K-Nearest Neighbor

The K-Nearest Neighbors (KNN) model showed reasonable performance on the .bytes dataset, demonstrating gradual improvements in accuracy as more training samples were provided. The confusion matrix shows that KNN correctly identified most samples from families like Kelihos_ver3 and Vundo. Some mix-ups happened between families with similar patterns, such as Lollipop and Obfuscator.ACY. Figures 4.6 and 4.7 show the confusion matrix and training/validation accuracy curve, respectively, for the KNN model on the .bytes dataset.

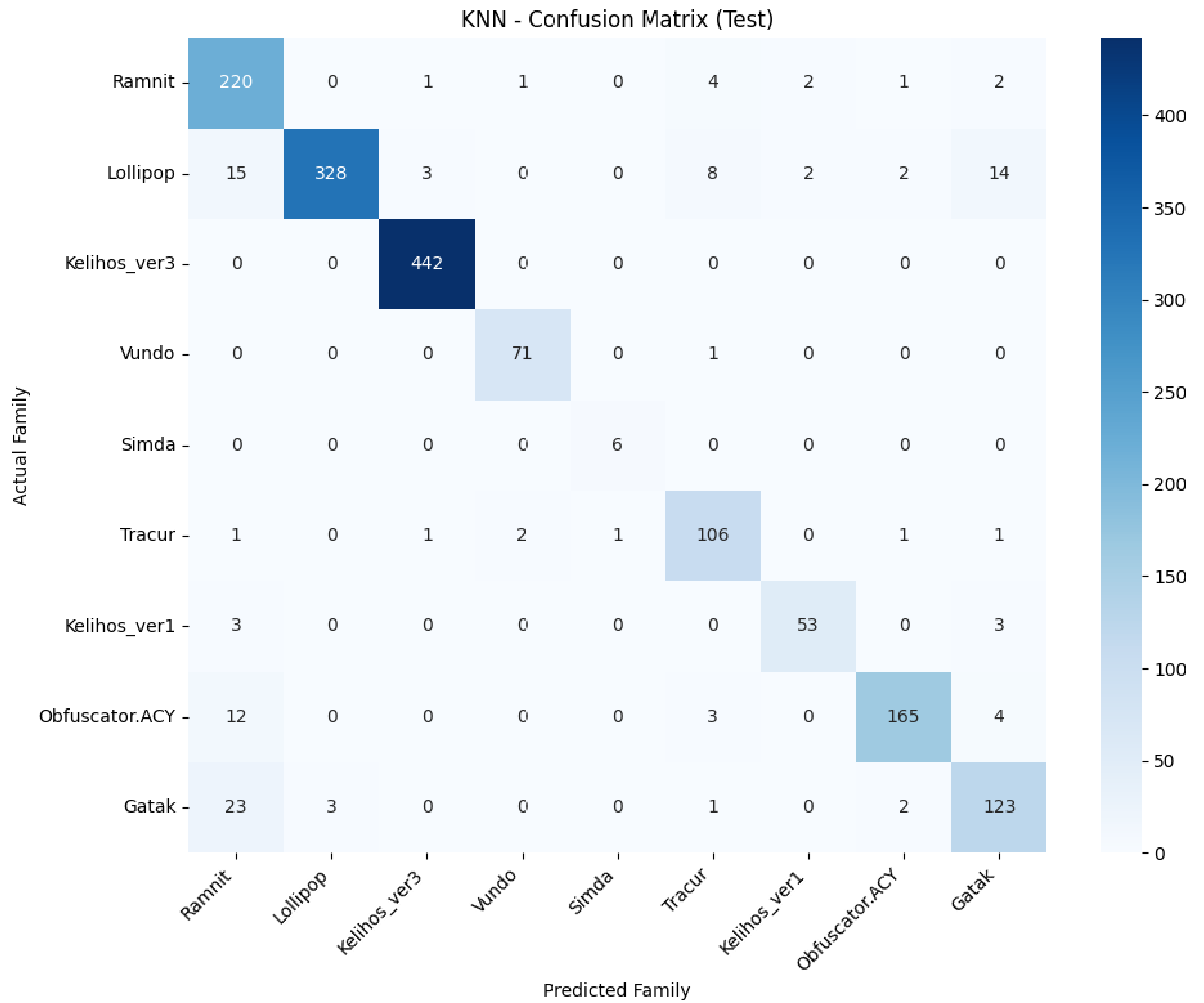


Figure 4.6: Confusion Matrix for KNN Model

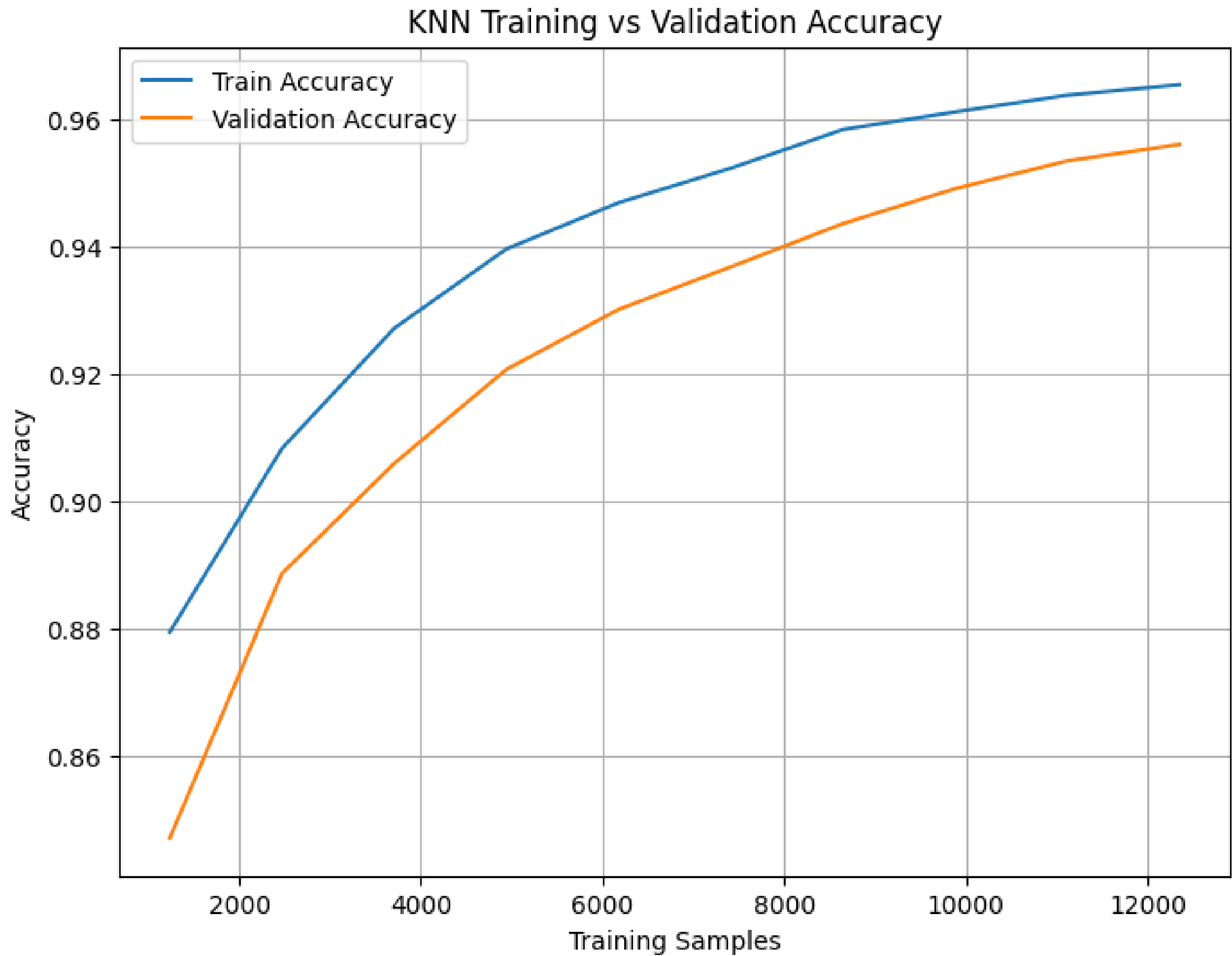


Figure 4.7: Training and Validation Accuracy Curve for KNN

As shown in the classification report depicted in Table 4.5, the overall accuracy is 98% which means the model performed well. However, compared to XGBoost and LightGBM, KNN was slightly less accurate because it can be affected by how features are scaled and how large the dataset is.

Malware Family	Precision (%)	Recall (%)	F1-Score (%)	Accuracy (%)
Ramnit	0.80	0.95	0.87	

Lollipop	0.99	0.88	0.93	
Kelihos_ver3	0.99	1.00	0.99	
Vundo	0.96	0.99	0.97	0.98
Simda	0.86	1.00	0.92	
Tracur	0.86	0.94	0.90	
Kelihos_ver1	0.93	0.90	0.91	
Obfuscator.ACY	0.96	0.90	0.93	
Gatak	0.84	0.81	0.82	

Table 4.5: KNN Classification Report on .byte

4.3.4 SVM

The Support Vector Machine with an RBF kernel showed moderate performance on the .bytes dataset, achieving steadily improving accuracy during training and demonstrating reliable but lower overall performance compared to boosting models. The confusion matrix shows that SVM correctly identified most malware samples, but some overlaps occurred between similar families such as Ramnit and Obfuscator.ACY. The accuracy graph depicted by Figures 4.8 and 4.9 show the confusion matrix and training/validation accuracy curve, respectively, for the SVM model on the .bytes dataset.

This means the model was consistent and learned well, but compared to boosting models like XGBoost and LightGBM, its performance was slightly lower due to higher computational cost and slower training on large feature sets.

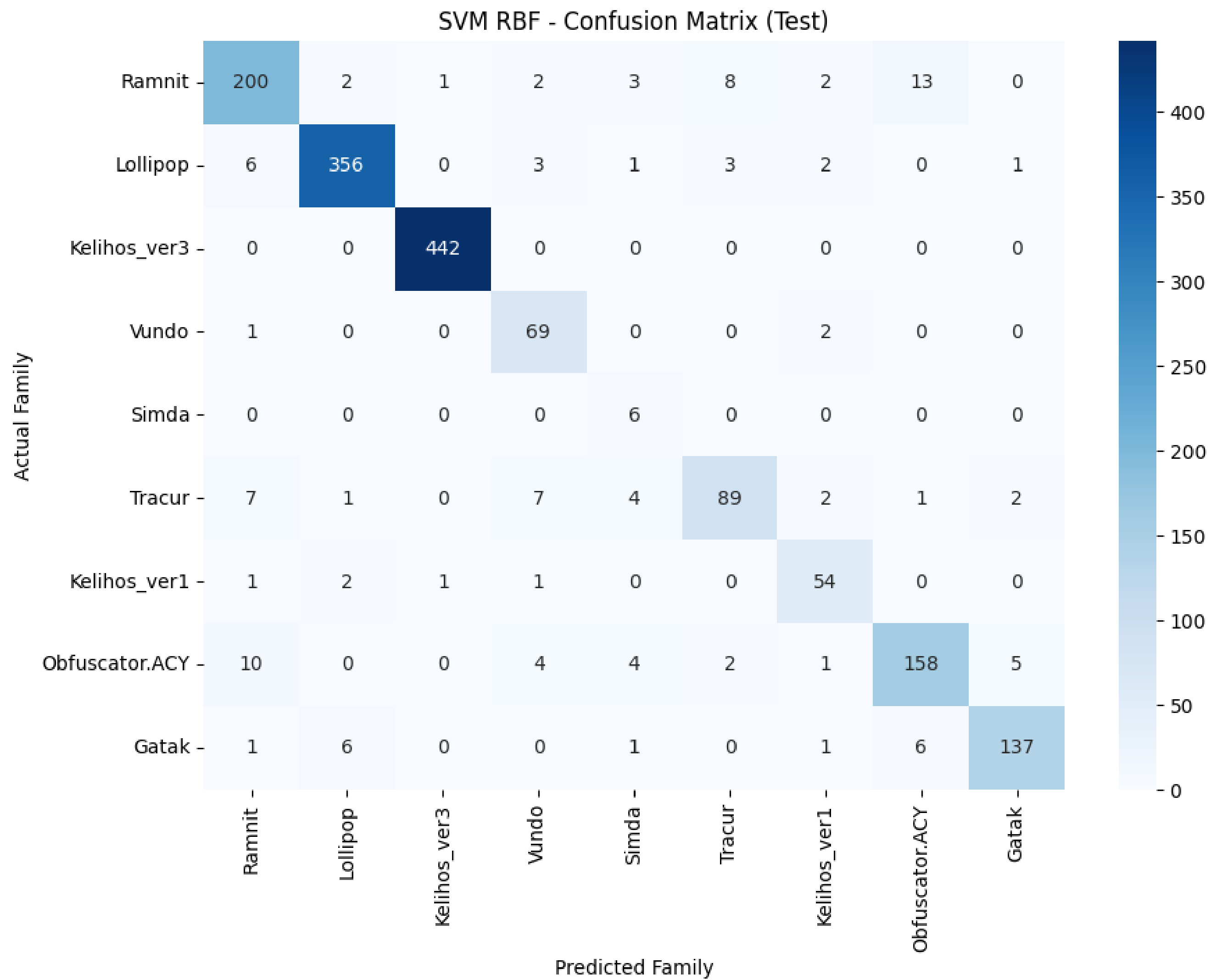


Figure 4.8: Confusion Matrix for the SVM Model

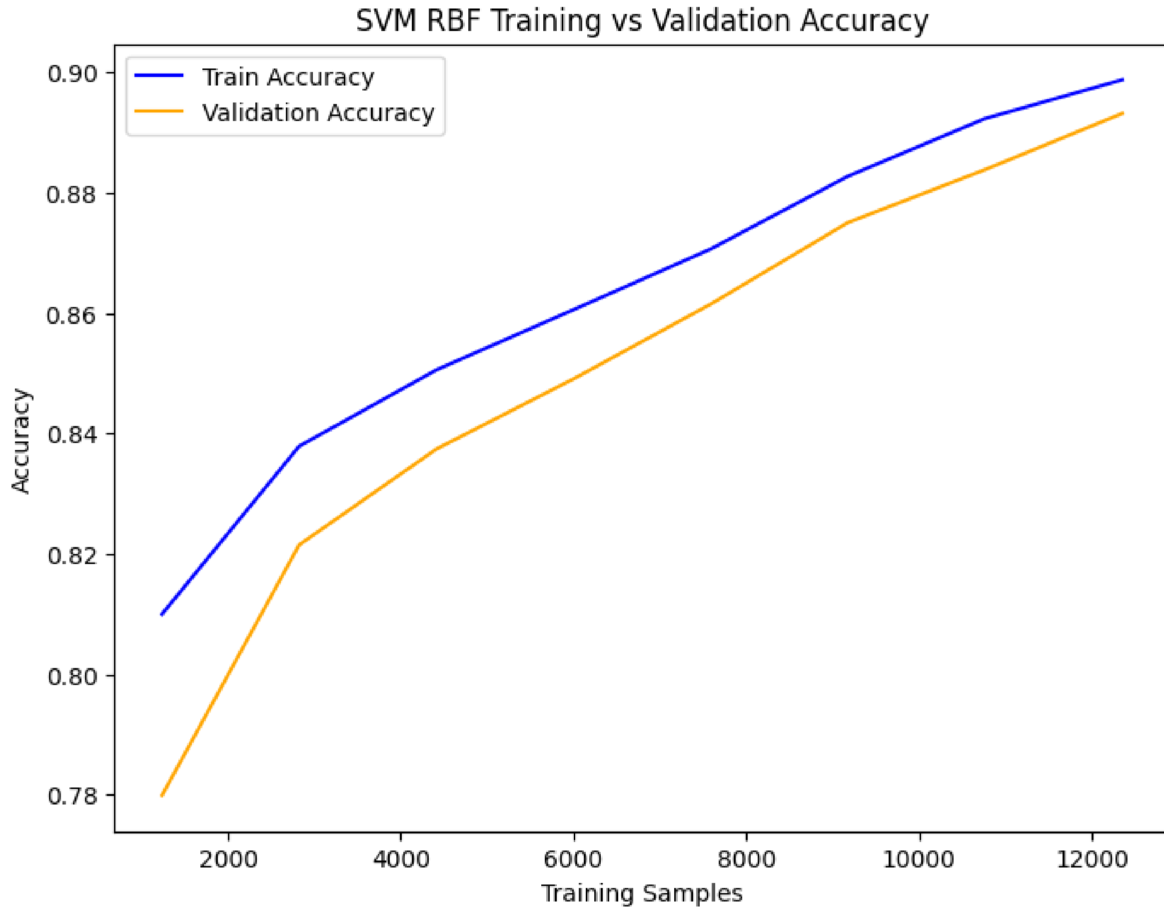


Figure 4.9: Training and Validation Accuracy Curve for SVM

As shown in the classification report in Table 4.6, the overall accuracy is 92%, which means the SVM-RBF model performed reasonably well. However, its accuracy is lower than XGBoost, LightGBM, and KNN because SVM has difficulty handling very large and high-dimensional datasets like the .bytes files.

Malware Family	Precision (%)	Recall (%)	F1-Score (%)	Accuracy (%)
Ramnit	0.88	0.86	0.87	0.92
Lollipop	0.97	0.95	0.96	
Kelihos_ver3	0.99	1.00	0.99	
Vundo	0.80	0.95	0.87	
Simda	0.31	1.00	0.48	
Tracur	0.87	0.78	0.82	
Kelihos_ver1	0.84	0.91	0.87	
Obfuscator.ACY	0.88	0.85	0.87	
Gatak	0.94	0.90	0.92	

Table 4.6: SVM Classification Report on .byte

4.4 Performance Evaluation on .asm for Multi-Class Classification

We present in this subsection the performance results on the .asm dataset using the same four models as previously.

4.4.1 XGBoost

The XGBoost model also performed very well on the .asm dataset, showing strong accuracy and stable learning.

Figures 4.10 and 4.11, and Table 4.7 show the obtained confusion matrix, accuracy curve and classification report, respectively. An overall accuracy of 99% is achieved, which is similar to the performance obtained on .bytes dataset.

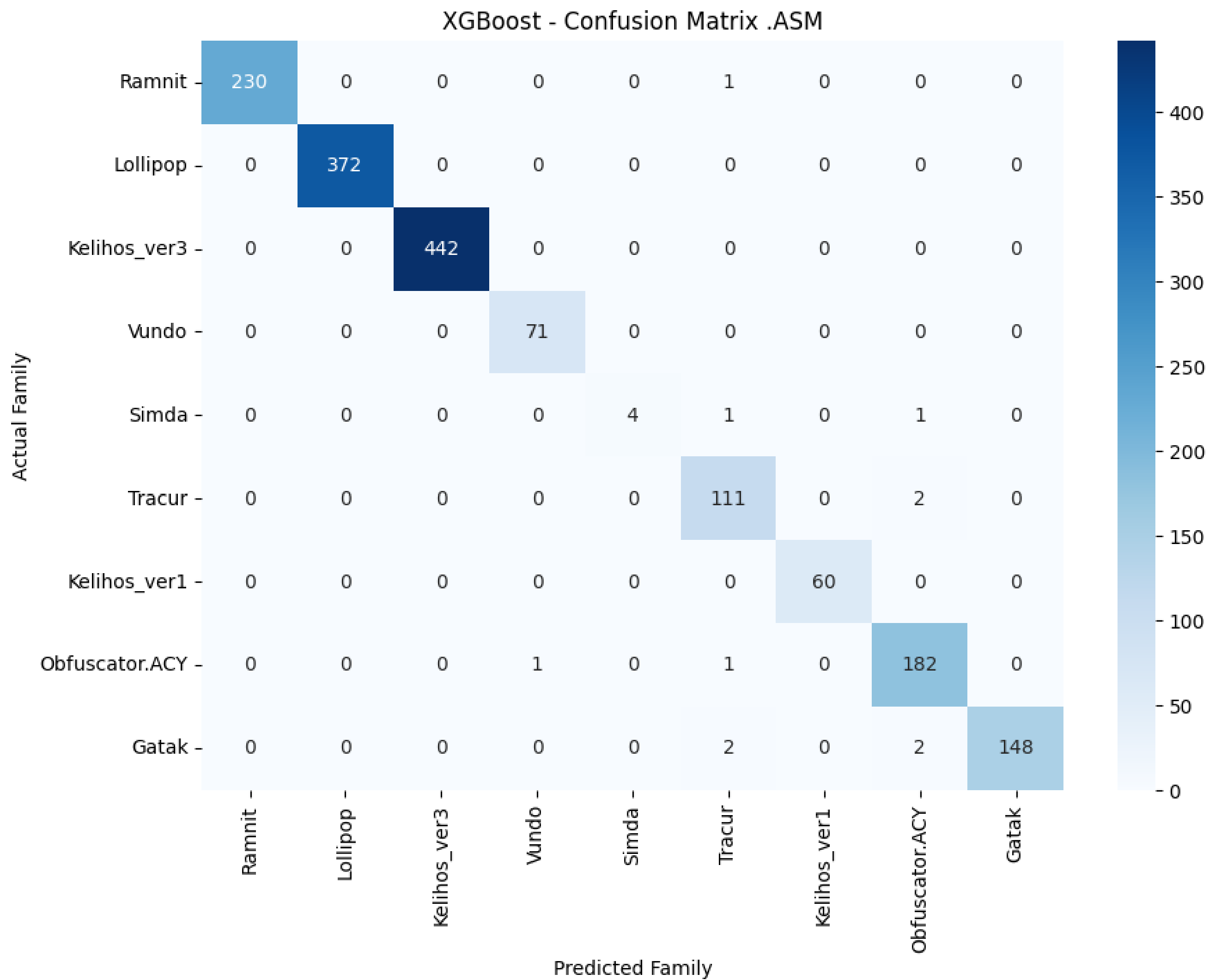


Figure 4.10: Confusion Matrix for XGBoost Model

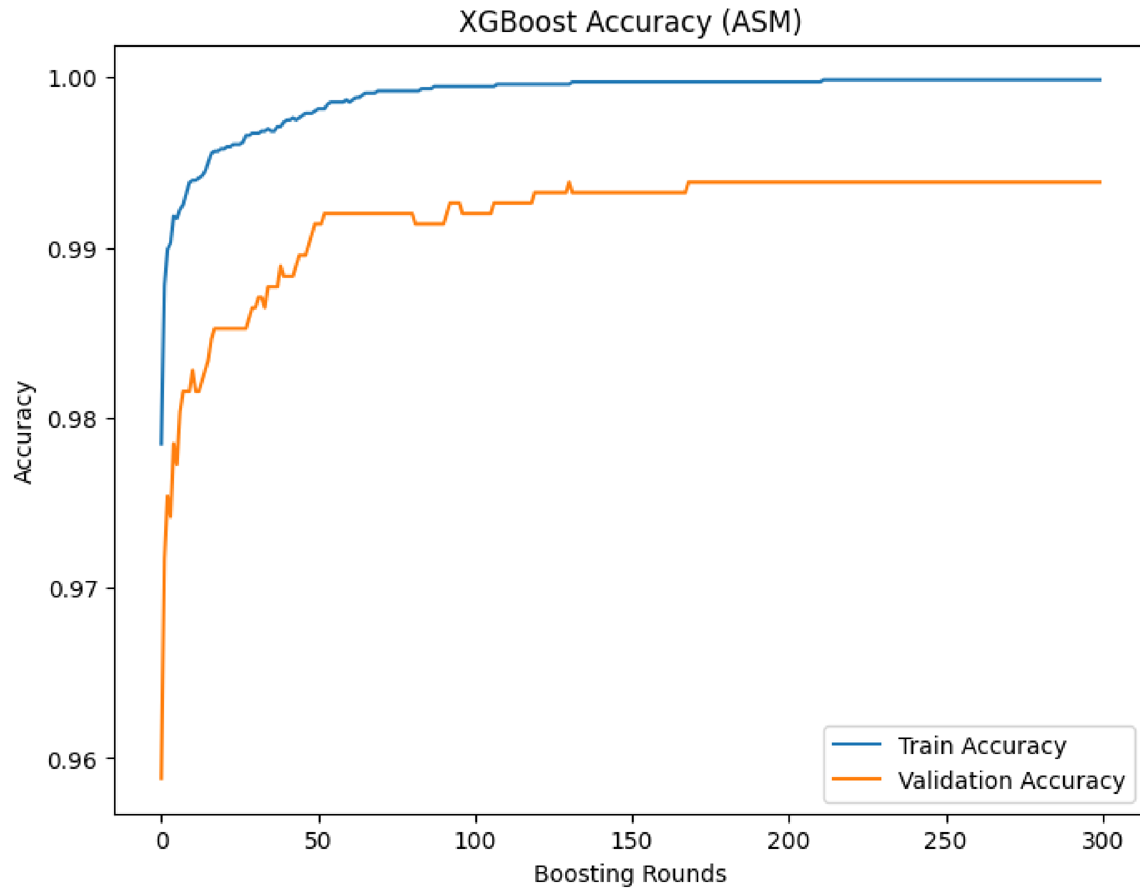


Figure 4.11: Training and Validation Accuracy Curve for XGBoost

Malware Family	Precision (%)	Recall (%)	F1-Score (%)	Accuracy (%)
Ramnit	1.00	1.00	1.00	0.99
Lollipop	1.00	1.00	1.00	
Kelihos_ver3	1.00	1.00	1.00	
Vundo	0.99	1.00	0.99	

Simda	1.00	0.67	0.80	
Tracur	0.96	0.98	0.97	
Kelihos_ver1	1.00	1.00	1.00	
Obfuscator.ACY	0.97	0.99	0.98	
Gatak	1.00	0.97	0.99	

Table 4.7: XGBoost Classification Report on .asm

4.4.2 LightGBM

The LightGBM model also performed strongly on the .asm dataset, showing stable and accurate predications across most malware families. Figures 4.12 and 4.13, and Table 4.8 show the confusion matrix, accuracy curve and classification report, obtained, respectively, for the LightGBM model on .asm samples.

As shown by the confusion matrix, the model correctly classified most malware families. The learning curve tells the same story: both training and validation accuracy increased quickly and settled around 99%. The model learned what it needed to, without overfitting.

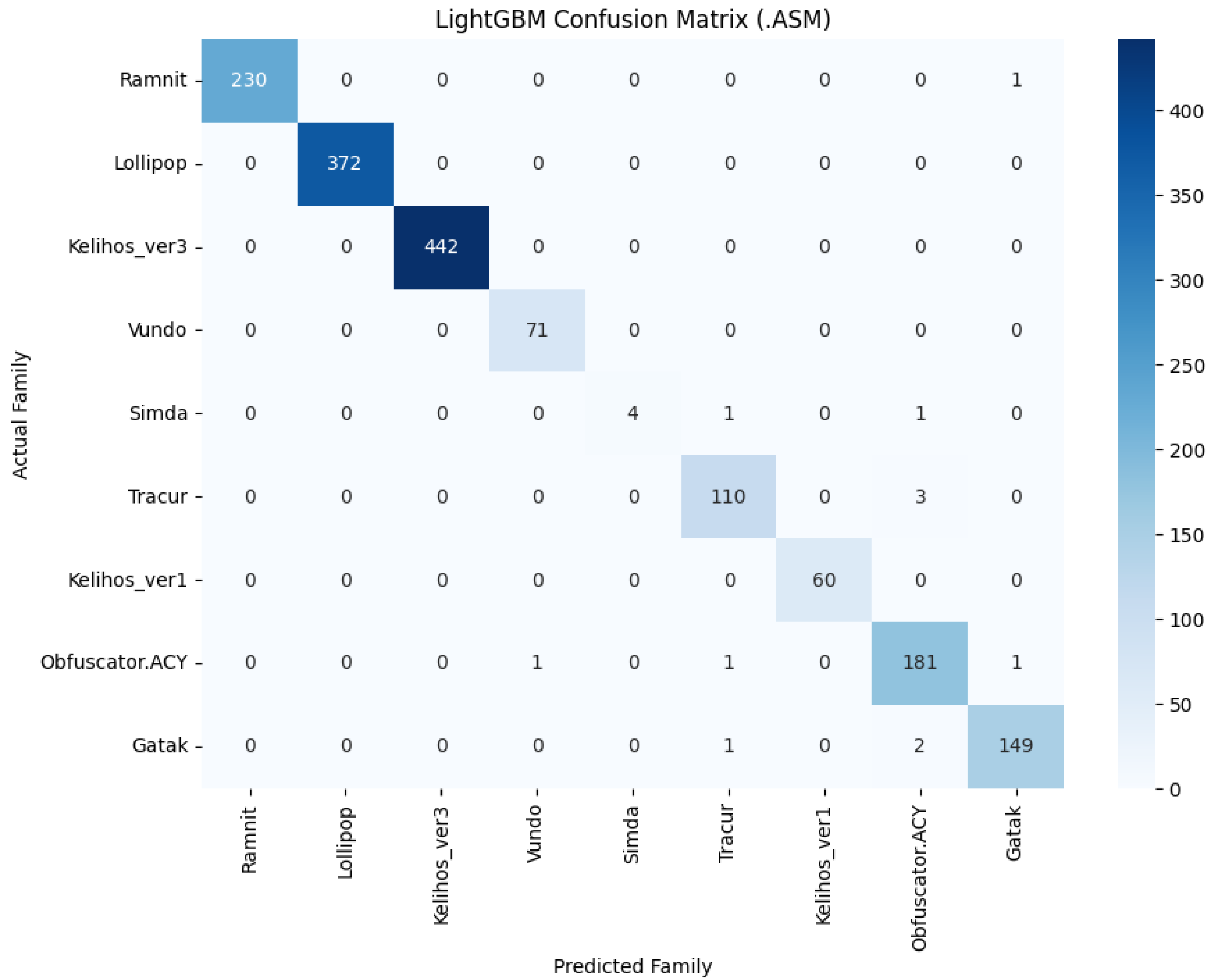


Figure 4.12: Confusion Matrix for LightGBM Model

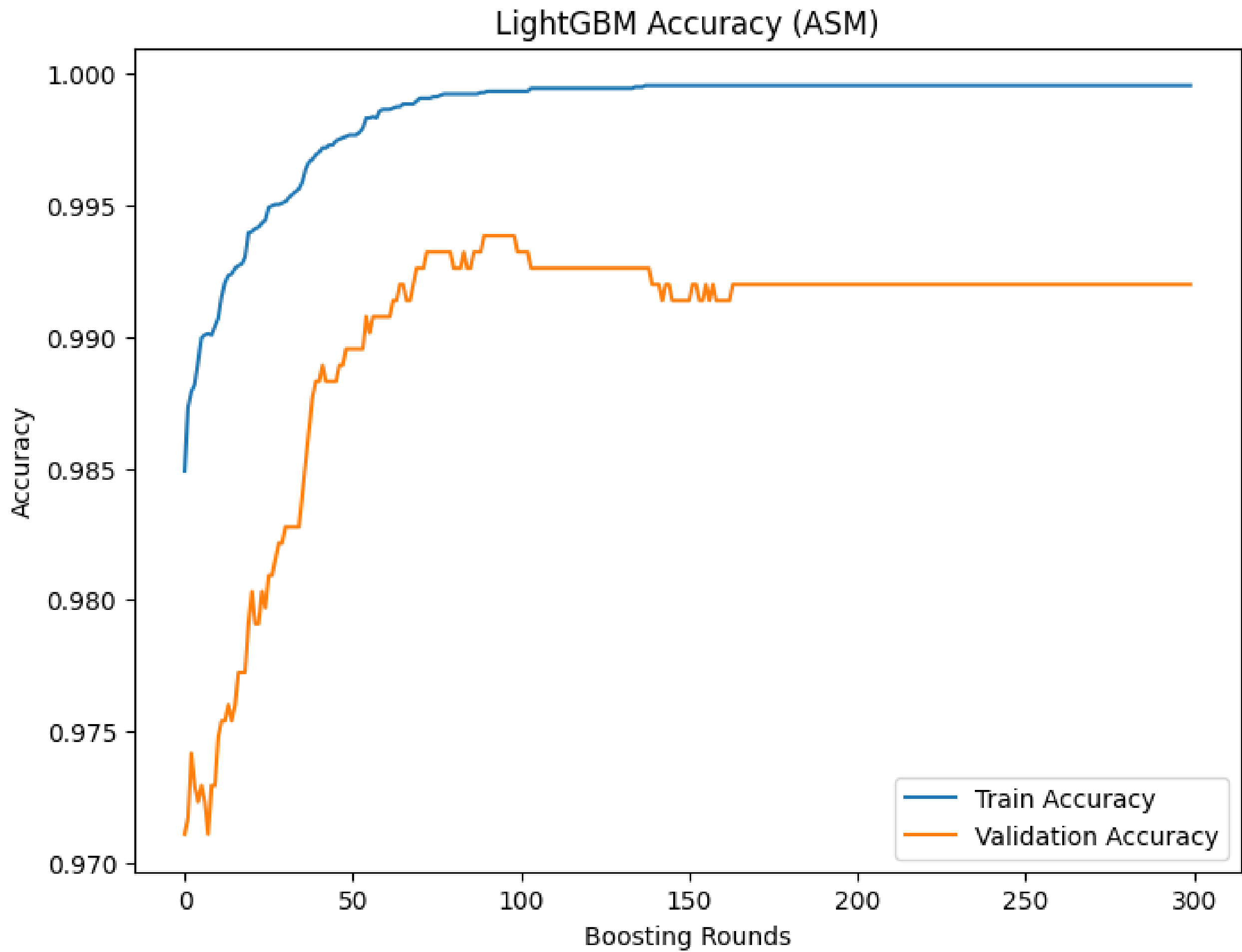


Figure 4.13: Training and Validation Accuracy Curve for LightGBM

Malware Family	Precision (%)	Recall (%)	F1-Score (%)	Accuracy (%)
Ramnit	1.00	1.00	1.00	0.99
Lollipop	1.00	1.00	1.00	
Kelihos_ver3	1.00	1.00	1.00	
Vundo	0.99	1.00	0.99	

Simda	1.00	0.67	0.80	
Tracur	0.97	0.97	0.97	
Kelihos_ver1	1.00	1.00	1.00	
Obfuscator.ACY	0.97	0.98	0.98	
Gatak	0.99	0.98	0.98	

Table 4.8: LightGBM Classification Report on .asm

4.4.3 K-Nearest Neighbors

The KNN model gave good results on the .asm dataset, showing consistent learning and accurate predictions for most malware families.

The model was able to recognize most malware families accurately, with only a few small errors in families that looked very similar. The accuracy curve shows that both training and validation performance kept improving as more samples were added, finally reaching around 98% accuracy. This shows that the model learned effectively and performed well on new, unseen data.

Figures 4.14 and 4.15, and Table 4.9 show the confusion matrix, accuracy curve and classification report, obtained, respectively for the KNN model on .asm samples.

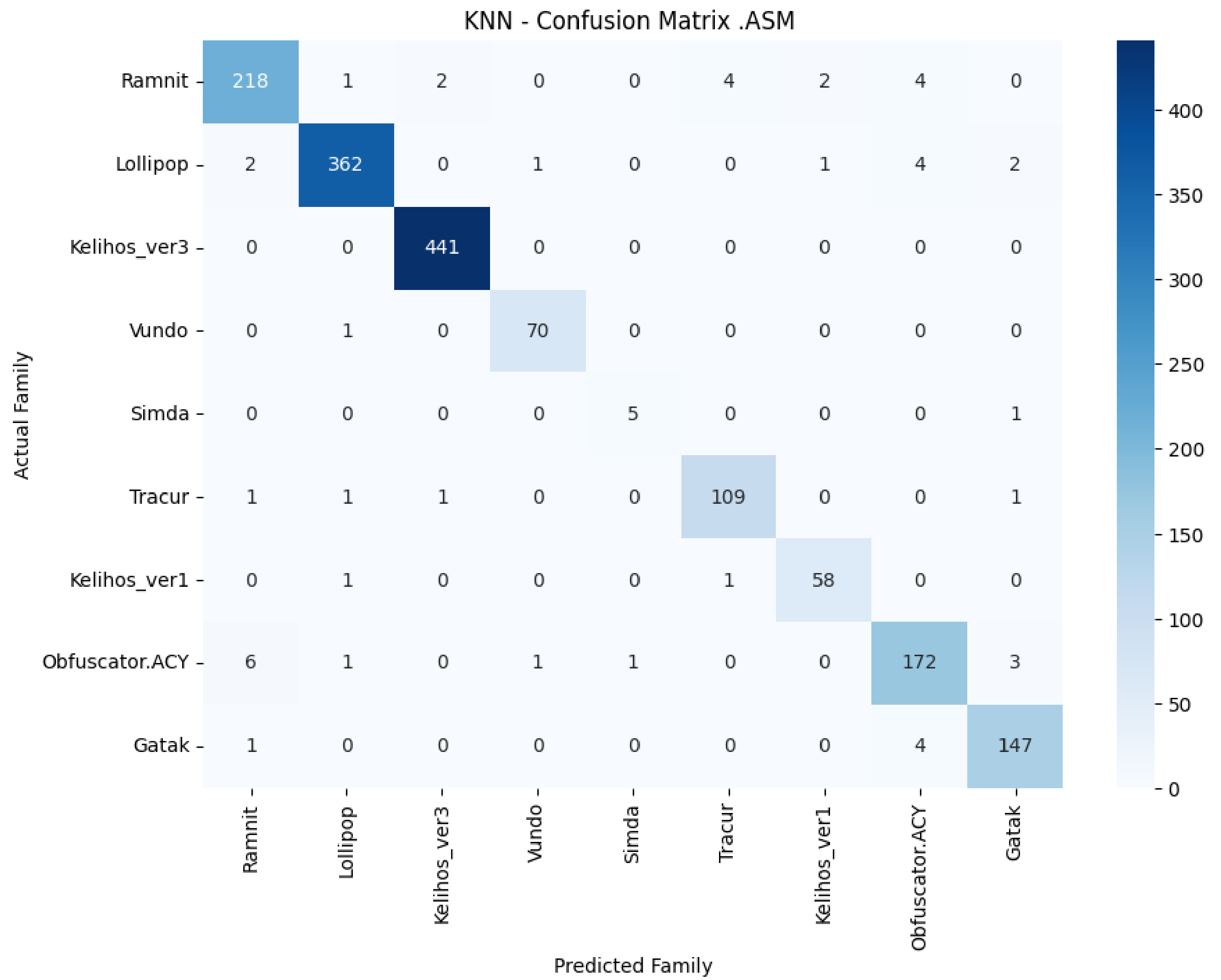


Figure 4.14: Confusion Matrix for KNN Model

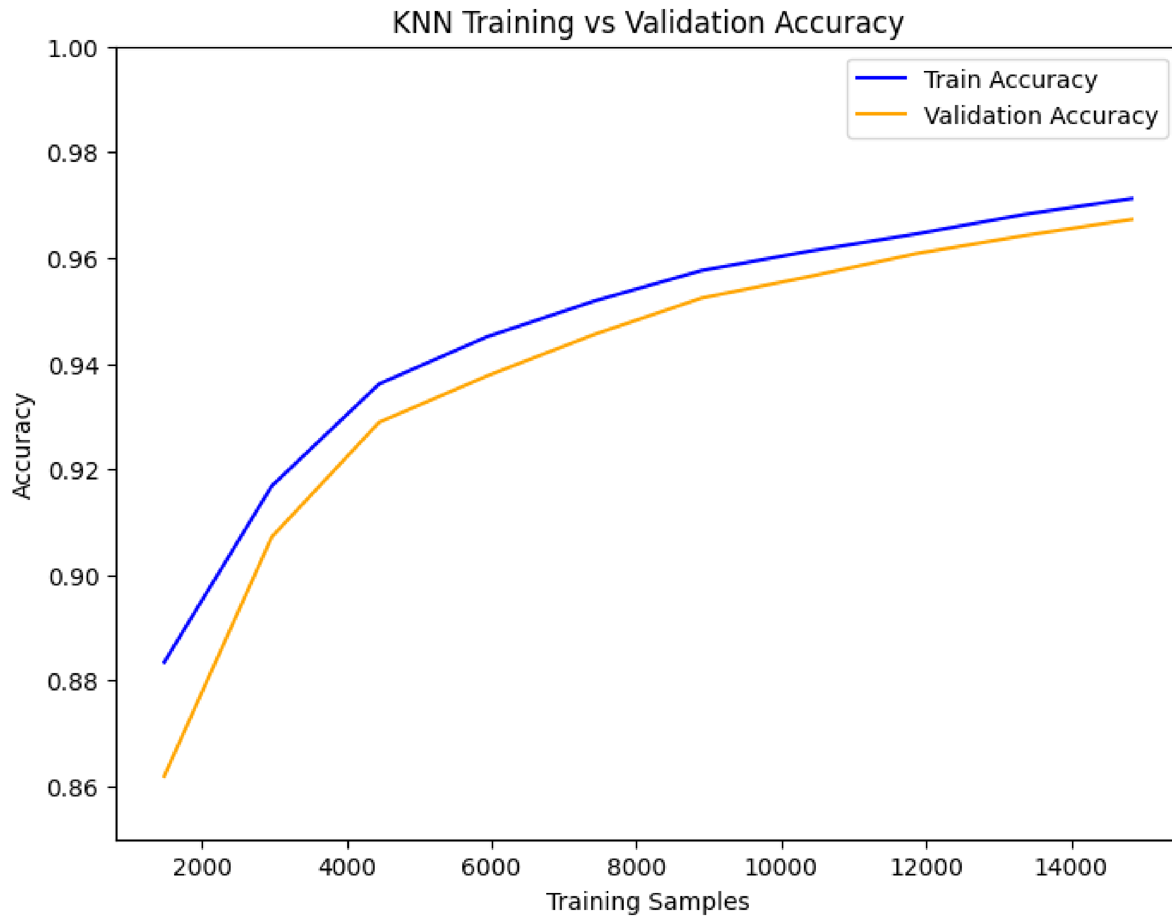


Figure 4.15: Training and Validation Accuracy Curve for KNN

Malware Family	Precision (%)	Recall (%)	F1-Score (%)	Accuracy (%)
Ramnit	0.97	0.96	0.96	
Lollipop	1.00	0.98	0.99	
Kelihos_ver3	0.99	1.00	1.00	
Vundo	1.00	1.00	1.00	0.98

Simda	0.55	1.00	0.71	
Tracur	0.96	0.96	0.96	
Kelihos_ver1	0.97	0.97	0.97	
Obfuscator.ACY	0.95	0.95	0.95	
Gatak	0.98	0.99	0.99	

Table 4.9: KNN Classification Report on .asm

4.4.4 SVM

The SVM achieved moderate performance on the .asm dataset. It was able to identify most malware families correctly, though some confusion appeared between families with very similar patterns. The accuracy graphs show that both training and validation accuracy increased steadily with more samples, reaching around 86%, which indicates the model learned key features but did not generalize as strongly as boosting-based models like XGBoost and LightGBM.

Figures 4.16 and 4.17, and Table 4.10 show the confusion matrix, accuracy curve and classification report, obtained, respectively, for the SVM model on .asm samples.

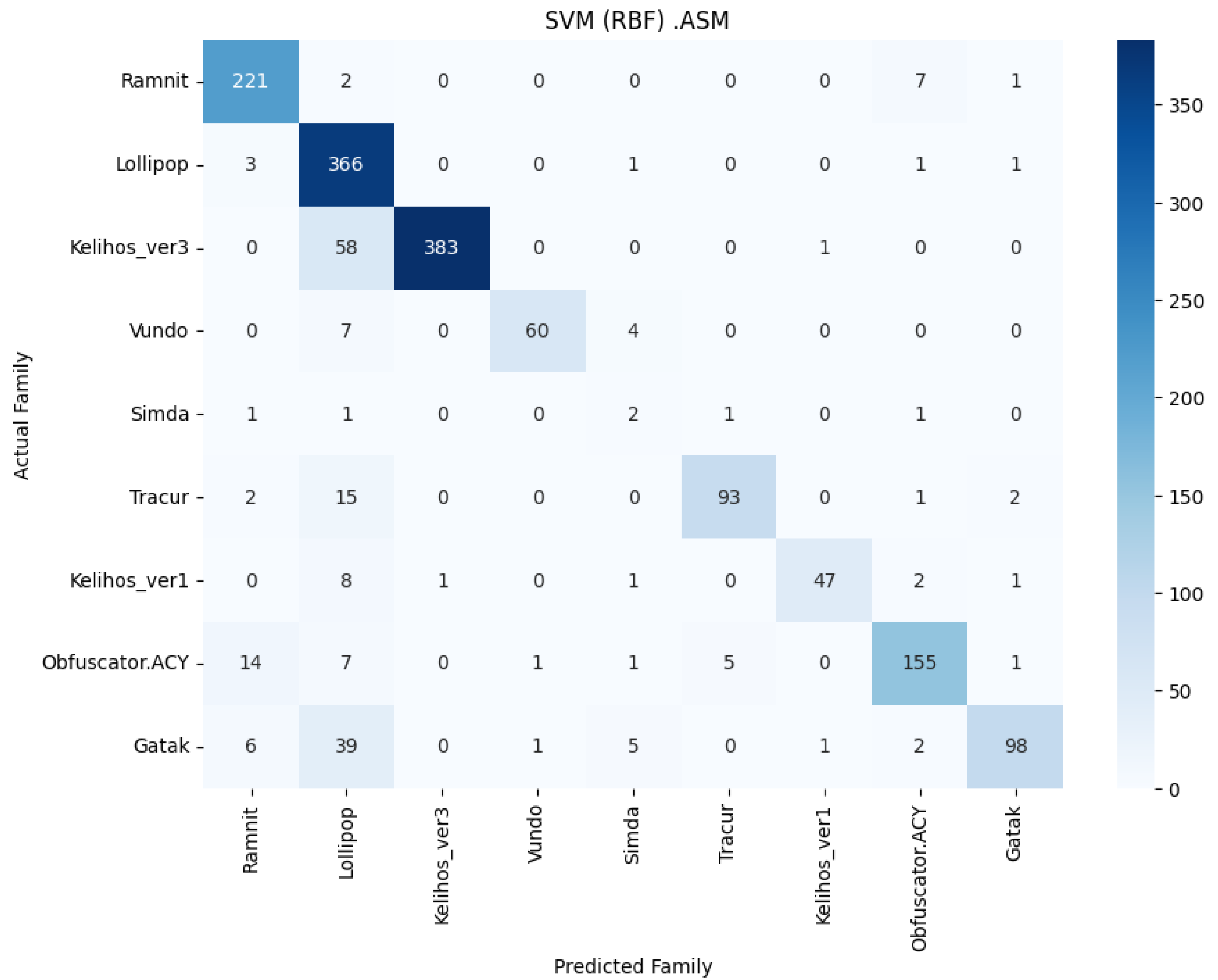


Figure 4.16: Confusion Matrix for SVM-RBF Model

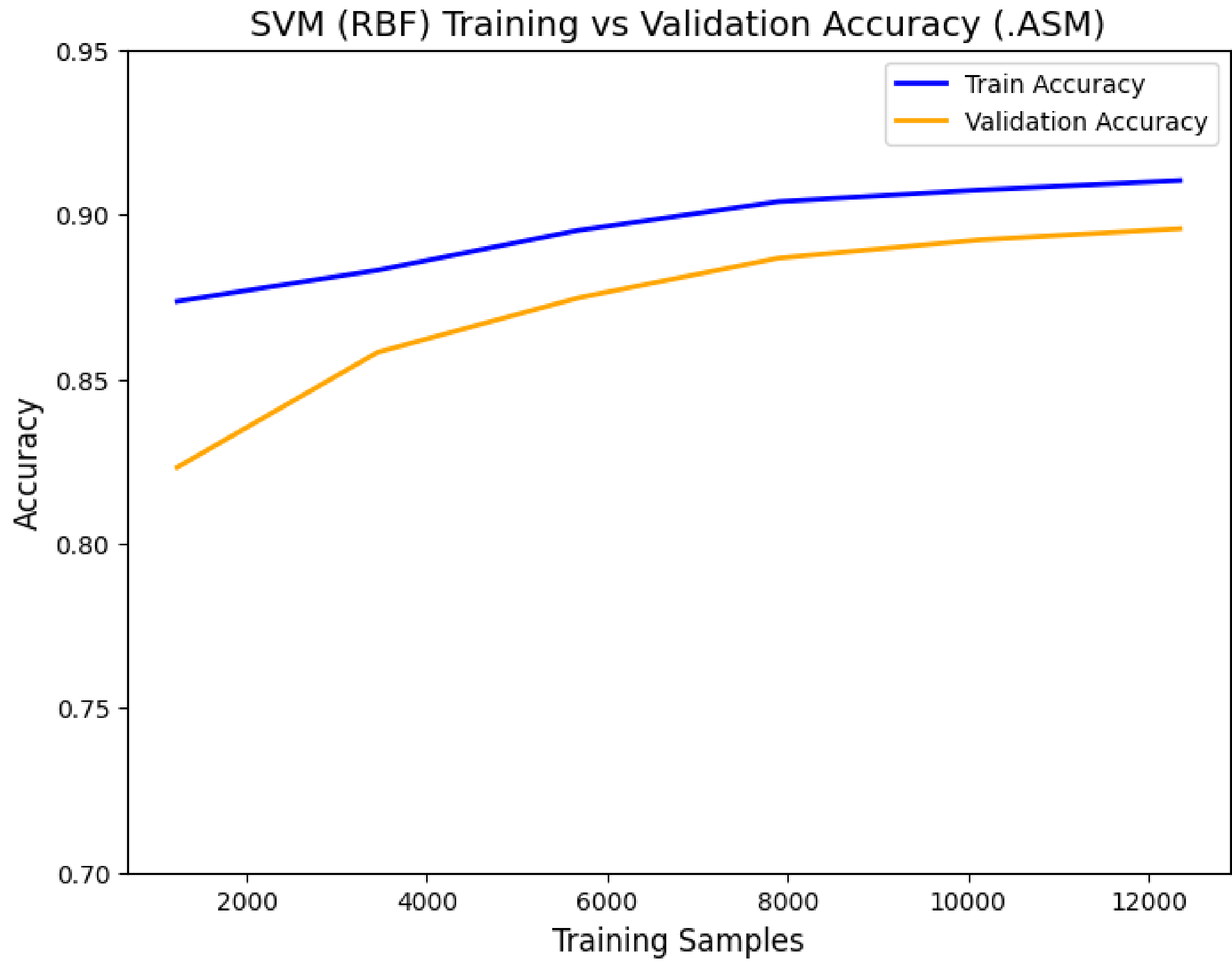


Figure 4.17: Training and Validation Accuracy Curve for SVM-RBF

Malware Family	Precision (%)	Recall (%)	F1-Score (%)	Accuracy (%)
Ramnit	0.89	0.96	0.96	
Lollipop	0.73	0.98	0.84	
Kelihos_ver3	1.00	0.87	0.93	
Vundo	0.97	0.85	0.90	0.87

Simda	0.14	0.33	0.20	
Tracur	0.94	0.82	0.88	
Kelihos_ver1	0.96	0.78	0.86	
Obfuscator.ACY	0.92	0.84	0.88	
Gatak	0.94	0.64	0.77	

Table 4.10: SVM-RBF Classification Report on .asm

4.5 Comparison Between .bytes and .asm Features

Table 4.11 compares the performance of all four models when trained on the .bytes and .asm data. Even though both datasets contain the duplicate malware files, they show very different sides of the program. The .bytes files contain raw hexadecimal code representing the low-level binary data that forms the malware's structure. Because these patterns are simple and easy for the models to detect, algorithms like XGBoost and LightGBM achieved nearly perfect accuracy, around 0.99. These models can easily recognize repeated binary patterns, enabling them to identify different malware families correctly. The KNN model also performed well, with an accuracy of about 0.98 though it was slightly slower on large dataset. The SVM model yielded the lowest results, around 0.92 mainly because it struggles to handle too many complex features at once.

On the other hand, .asm files contain human-readable assembly instructions that describe what the malware does rather than how it looks. This type of data is more complex because programs belonging to the same malware family can still appear different in their assembly representations. As a result, most models showed a slight decrease in accuracy when tested on the .asm dataset.

XGBoost and LightGBM continued to perform very well with accuracies close to 0.99, while KNN remained at 0.98 and SVM dropped at 0.87. This reduction occurred because .asm files introduce more noise and variation, which makes it harder for models to find consistent and precise patterns.

To summarize, the .bytes features help the models understand what the malware looks like internally. In contrast, the .asm features explain how it behaves. The .bytes data provide structure, and the .asm data provides logic. Combining both types of features in the future could lead to more complete and powerful malware detection system that learns from both the internal design and behavioral patterns of malicious programs.

Model	.bytes Accuracy	.asm Accuracy	Remarks/Observations
XGBoost	0.99	0.99	Best Results; learns both structure and behavior effectively
LightGBM	0.98	0.99	Very close to XGBoost; slightly faster and stable
KNN	0.98	0.98	Works well but slower; some confusion in similar families
SVM	0.92	0.87	Performance drops on complex .asm data; less suitable for large feature sets.

Table 4.11: Comparison of Model Accuracy between .asm and .bytes data

Chapter 5: LLM Integration and Summary Evaluation

This chapter presents the integration of a Large Language Model (LLM) into the malware classification system, so it can produce easy-to-read summaries [27].

5.1 Integration Overview

Figure 5.1 depicts the workflow of the integrated framework. The workflow starts with scanning a file to decide whether it is safe or malicious using binary classification. If deemed malicious, the file is categorized using multi-class classification into a malware family. The LLM then produces based on the family a plain-language explanation of what the malware does, how it spreads, and why it is a problem.

I initially tested cloud-based LLMs such as DeepSeek and ChatGPT for generating explanations, but they were not suitable because they require Internet access, rely on external servers, and raise privacy concerns when working with malware files. They also introduced higher latency. For these reasons, Ollama was chosen instead, since it runs fully offline, offers faster local inference, and integrates seamlessly with the lightweight ML pipeline.

We implemented a user interface based on Gradio [25]. The Gradio interface provides a clean, interactive web window allowing uploading files and viewing classification results and automatic summaries all in one place. Gradio is easy to hook up with Python ML models, handles real-time updates, and can be tweaked for both testing and deployment.

This setup allows users to quickly check whether a file is malicious, see the predicted malware family, and read a simple explanation of the threat, all through a unified interface.

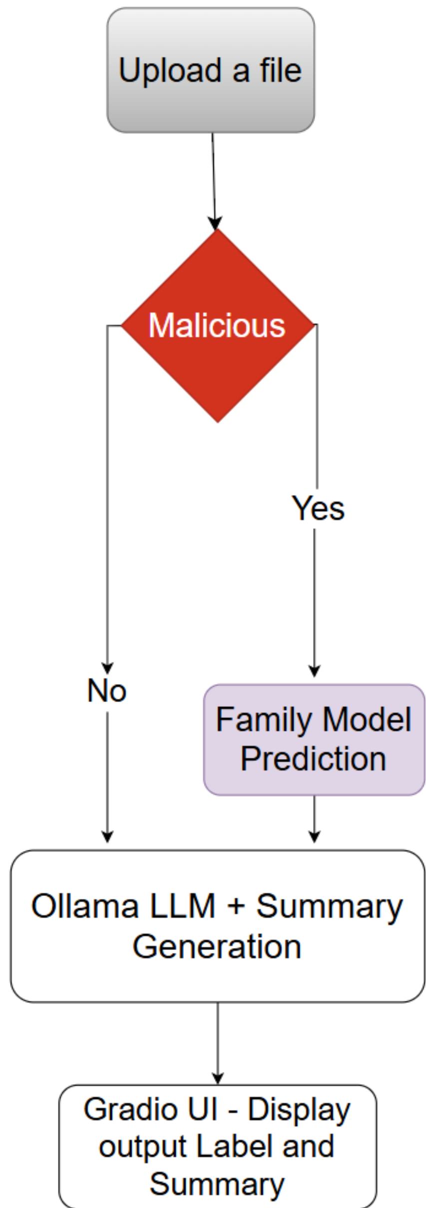


Figure 5.1: Flowchart of the Online Malware Detection Process

5.2 Gradio User Interface

The Gradio interface was utilized to make the system user-friendly and straightforward. Rather than entering commands, the user uploads a file, clicks the Submit button, and receives the forecast

immediately. If the file is harmful, the system indicates which family it belongs to after determining whether it is safe or not. The LLM then writes a brief description of the malware's functions.

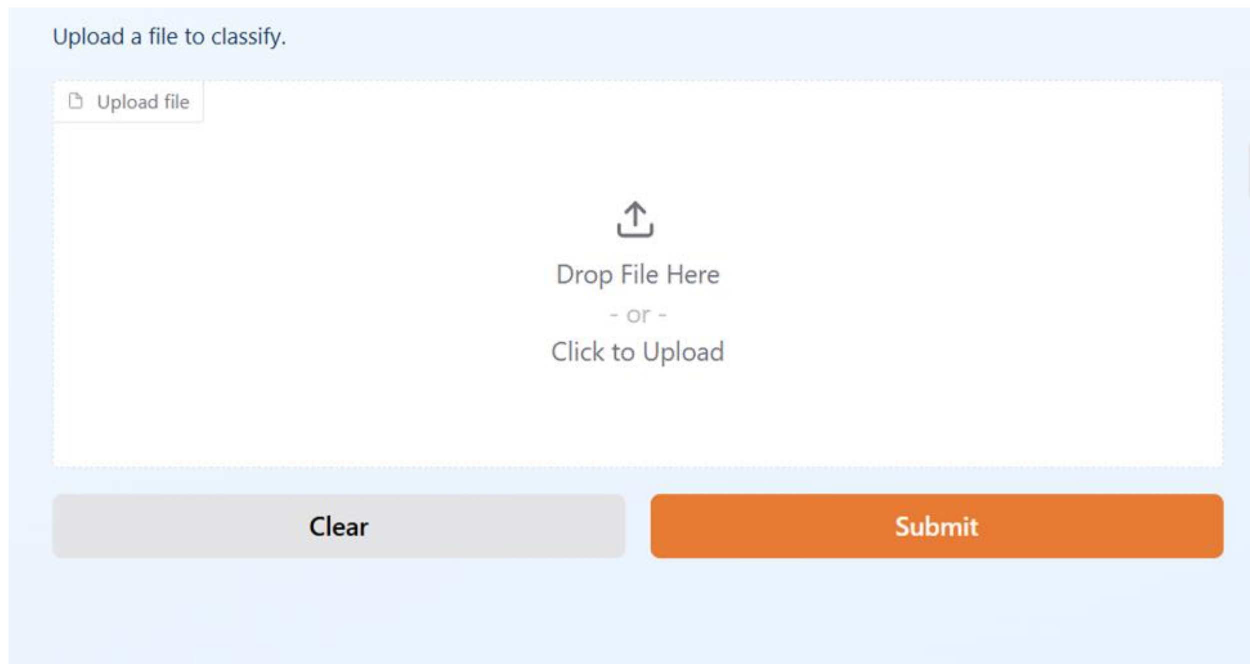


Figure 5.2: Gradio User Interface for Malware Detection and LLM-Based Explanation

The Gradio-based user interface designed for this project is shown in Figure 5.2. Users may click to choose a file or drag and drop one into the upload box, which is part of the interface's simple, clean structure. Buttons to remove the existing input and submit a new file are located beneath it. The findings are displayed on the right, along with the generated summary and the forecast label (Benign or Malicious) [27].

The entire procedure is simplified for both technical and non-technical users using Gradio. It unifies all the system's components onto a single platform, including explanation, prediction, and categorization. Without requiring users to view raw code or logs, this approach enhances usability and facilitates rapid comprehension of the output.

5.3 Malware Summary Generation using Ollama LLM

After the malware detection model checks a file and decides if it is safe or dangerous, it hands things off to the LLM running on Ollama [26] [27]. That is where the system converts the technical result into plain English, so users can actually understand the underpinnings like what the malware does, how it spreads, and how to avoid it. The LLM also provides a few technical details such as which model ran the check, how long it took, how many tokens the process used, and how confident the system is about its decision.

The process flows through four main steps:

1. **File Upload:** The user uploads a file.

2. **Model Prediction:** The trained ML model determines whether the file is benign or malicious.

3. **LLM Summary Generation:** The result is passed to the LLM, which writes a short explanation of the outcome.

4. **Output Display:** The Gradio interface shows both the classification and the summary clearly for the user.

Two examples are illustrated by Figures 5.3 and 5.4. In Figure 5.3 illustrates a safe file. In Figure 5.4 the system detects a malicious file, categorizes it in the malware family Obfuscator.ACY and lays out exactly what makes it dangerous. This time, the system reports 94.5% confidence level and backs it up with a summary explaining why the file is harmless.

All in all, adding the LLM makes the whole process less intimidating and much more user-friendly. Instead of dumping technical jargon, the system explains everything in plain language users can actually understand.

Status: Benign

Confidence: 94.50%

A benign executable file is considered safe because it does not contain any malicious code or instructions that could harm the computer or its data. The file has been digitally signed by its creator, verifying its authenticity and integrity, and has passed through multiple layers of security checks to ensure it is free from viruses, trojans, and other forms of malware. As a result, the executable file can be trusted to run safely on the system without posing any risk of infection or data loss.

LLM Performance Summary

- **Model:** llama3:instruct
- **Total Time:** 15.41 sec (Load: 6.55s)
- **Prompt Tokens:** 29
- **Generated Tokens:** 96
- **Speed:** 11.72 tokens/sec

Flag

Figure 5.3: System detects a benign file

Status: Malicious

Family: Obfuscator.ACY

Confidence: 74.50%

Obfuscator.ACY is a type of malware that disguises itself as legitimate software to evade detection by antivirus programs. It typically spreads through phishing emails or fake updates for popular software like Adobe Flash Player or Java. Once installed, Obfuscator.ACY can steal sensitive information such as login credentials and payment details, as well as take control of the infected device. To protect yourself from this malware, be cautious when opening email attachments or clicking on links, keep your operating system and software up to date, and use strong antivirus software that includes features like behavioral detection and sandboxing. Additionally, avoid using pirated software or unverified download sites, and regularly back up your data in case of infection.

LLM Performance Summary

- **Model:** llama3:instruct
- **Total Time:** 22.71 sec (Load: 9.82s)
- **Prompt Tokens:** 41
- **Generated Tokens:** 141
- **Speed:** 11.88 tokens/sec

Flag

Figure 5.4: System detects a malicious file

Chapter 6: Conclusion

We developed an integrated framework which allows identifying and categorizing malicious files and providing detailed explanation on the corresponding malware. The framework integrates shallow machine learning algorithms and LLM. In the experimental evaluation on the Microsoft Malware Classification dataset, XGBoost and LightGBM came out on top for accuracy for multi-class clarification. KNN, though, stood out because Zit was fast and still performed well, which makes it a solid pick for real-time detection.

One of the next steps is to bring together the .bytes and .asm features into a single hybrid model. By combining both, the system learns not just what malware looks like on the inside its structure but also how it acts, its logic and operations. Blending both perspectives achieves better detection and helps the model handle a broader variety of malware. Another key priority is growing the dataset. Right now, most of the training comes from the Microsoft Malware Classification Challenge, but adding in fresher, more varied samples from sources like VirusShare, MalwareBazaar, or those new Kaggle threat datasets, will improve and strengthen the model. Mixing in samples from different operating systems Windows, Android, Linux will move it closer to a versatile malware detector, not just a technology tied to one platform.

We also plan to explore fusion-based models that combine both .bytes and .asm features into a unified representation. This would allow the system to learn both the internal structure and runtime behavior of malware simultaneously. Additionally, we will compare outputs from different LLMs such as DeepSeek, ChatGPT, and other emerging models and evaluate their explanation quality. Fine-tuning these LLMs on malware-specific corpora is another future direction to improve clarity and domain accuracy of generated summaries.

Finally, we will consider operationalizing deployment by converting the framework into a real-time malware monitoring tool.

Bibliography

[1] AV-TEST Institute, “Malware statistics and trends,” https://www.av-test.org/en/statistics/malware/.

[2] Panja, S, Mondal S, Nag A, Singh J. P, Saikia M. J, & Barman A. K. (2024). “An Efficient Malware Detection Approach Based on Machine Learning Feature Influence Techniques for Resource-Constrained Devices.” *IEEE Access*, vol. 13, pp. 145233–145249, 2024.

[3] Ferdous J; Islam, R; Mahboubi, A; Islam, M.Z. “A Survey on ML Techniques for Multi-Platform Malware Detection: Securing PC, Mobile Devices, IoT, and Cloud Environments”. *Sensors*, vol. 25, no. 4, p. 1153, 2025.

[4] Philip O ’Kane ; Sakir Sezer ; Kieran McLaughlin ; “Obfuscation : The Hidden Malware.” n *Proc. 2011 IEEE International Conference on Cyber Conflict (CYCON)*, Tallinn, Estonia, 2012.

[5] Anderson & Roth, “EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models” arXiv:1804.04637, 2018.

[6] Harang & Rudd, “SOREL – 20M: A Large-Scale Benchmark Dataset for Malicious PE Detection” arXiv:2012.07634, 2020

[7] Royi Ronen; Marian Radu; Corina Feuerstein; Elad Yom-Tov; Mansour Ahmadi “Microsoft Malware Classification Challenge” in Proc *2018 IEEE International Conference on Big Data (Big Data)*, pp. 1005–1011, IEEE, 2018.

[8] Dambra, G., et al. “Decoding the Secrets of Machine Learning in Malware Classification: A Deep Dive into Datasets, Feature Extraction, and Model Performance.” *Proc. 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS),* pp. 60–74, 2023.

[9] Qijing Qiao et al; Ruitao Feng; Sen Chen. “Multi-label Classification for Android Malware Based on Active Learning” arXiv:2410.06444, 2024.

[10] Sidharth Anand; Barsha Mitra; Soumyadeep Dey; Abhinav Rao; Rupsha Dhar; Jaideep Vadiya "MALITE: Lightweight Malware Detection and Classification for Constrained Devices" arXiv:2309.03294, 2023.

[11] Eliel Martins; Javier Bermejo Higuera; Ricardo Sant; Juan Ramon. "Semantic Malware Classification Using Artificial Intelligence Techniques". *Computer Modeling in Engineering & Sciences*, vol. 142, no. 3, pp. 789–812, 2025.

[12] Mahesh Datta Sai Ponnuru; Likhita Amasala; Tanu Sree Bhimavarapu; Guna Chaitanya Garikipati "A Malware Classification Survey on Adversarial Attacks and Defense" arXiv:2312.09636, 2023.

[13] Hamed Jelodar; Samita Bai; Parisa Hamedi; Heasmodin; Roozbeh; Ali Ghorbani "Large Language Model (LLM) for Software Security: Code Analysis, Malware Analysis, reverse Engineering" arXiv:2504.07137

[14] Scott E .Coull; Christopher Gardner "Activation Analysis of a Byte-Based Deep Neural Network for Malware Classification" arXiv:1903.04717, 2019.

[15] Mihui Kim; Haesoo Kim "A Dynamic Analysis Data Preprocessing Technique for Malicious Code Detection with TF-IDF and Sliding Windows" *Electronics*, vol. 13, no. 5, p. 963, 2024

[16] N. V. Chawla, K. W. Bowyer; L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[17] Alberto Fernandez; Garcia; Herrera; Chawla "SMOTE for Learning from Imbalanced Data: Progress and Challenges, Marking the 15-year Anniversary" *Journal of Artificial Intelligence Research*, vol. 61, pp. 863–905, 2018.

[18] Roshan Kumari; Saurabh Kr. Srivastava "Machine Learning: A review on Binary Classification" *International Journal of Computer Science and Engineering*, vol. 5, no. 4, pp. 45–54, 2017.

[19] Grandini; Bagli; Visani "Metrics for Multi-Class Classification : an Overview" arXiv:2008.05756, 2020.

[20] Xiaonan Zou; Tian; Shen "Logistic Regression Model Optimization and Case Analysis" Proc. *IEEE 4th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*,pp. 1453–1457, 2019.

[21] Chen; Liu; Chang-Hua Zhang "XGBoost-Based Algorithm Interpretation and Application on Post-Fault Transient Stability" *Proc. 2018 IEEE Power & Energy Society General Meeting (PESGM)*, pp. 1–5, 2018.

[22] Guolin Ke; Thomas Finley; Wei Chen "LightGBM: A Highly Efficient Gradient Boosting Decision Tree" *Advances in Neural Information Processing Systems (NeurIPS)*,pp. 3146–3154, 2017.

[23] Zhang; Debo Cheng; Zhenyun Deng "A novel KNN algorithm with data-driven k parameter computation " *Pattern Recognition Letters*, vol. 109, pp. 44–54, 2018.

[24] Bor-Chen Kuo; Hsin-Hua Ho; Jin-Shiuh Taur " A Kernel-Based Feature Selection Method for SVM with RBF Kernel for Hyperspectral Image Classification*" IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*,vol. 7, no. 4, pp. 1148–1159, 2014.

[25] Abubakar Abid, Ali Abdalla, Ali Abid, Dawood Khan Alfozan, "Gradio: Hassle-Free Sharing and Testing of ML Models in the Wild" arXiv:1906.02569, 2019.

[26] Saha, B., Rani, N., & Shukla, S. K." Malware: Automating the Comprehension of Malicious Software Behaviours using Large Language Models (LLMs)". arXiv:2504.01145, 2025.

[27] Jelodar H., Bai S., Hamedi P., Razavi-Far R "Large Language Model (LLM) for Software Security: Code Analysis, Malware Analysis, Reverse Engineering". arXiv:2504.07137, 2025.