Muhammad Ali

Machine Learning-Based Ransomware Detection Through Static Analysis of PE File Features

Metropolia Ammattikorkeakoulu

Bachelor of Engineering, Information Technology

Bachelor Thesis

21.07.2025

Abstract

Author: Muhammad Ali

Title: Machine Learning-Based Ransomware Detection Through Static Analysis of PE File Features

Number of Pages: 56 Pages

Date: 05 August 2025

This thesis introduces a machine learning framework for ransomware detection based on static analysis of executable file attributes. The primary contribution of the thesis is the introduction of a useful low-false-positive detection system, a gap-filling mechanism between the performance claims of the academic space and the real-world deployment needs of the cybersecurity applications. Using the EMBER 2018 dataset [1], which is a benchmark collection of real malware samples, this thesis comprehensively tests a number of machine learning models against a balanced sampling of 50,000 variants, 25,000 benign, and 25,000 malicious. The novelty of the work is the extensive feature engineering approach, extracting 50 different structural features from PE files, including entropy distribution, import features, header features, and histogram statistics, with a deliberate emphasis on minimizing false positives.

This thesis demonstrates through cross-validation and a large holdout set (7,500 samples) that ensemble methods, in particular, XGBoost [2], significantly outperform traditional methods. The optimized XGBoost model achieved 94.5% accuracy, 94.6% precision, and 94.4% recall, with a low false positive rate of 5.4% which was a significant improvement from previous methods, which were plagued by excessive false alarms.

Based on the feature importance analysis, the three best classifiers for ransomware detection were the GUI application flag, entropy in certain byte ranges, and imported features. These findings re-evaluate long-held beliefs about which file attributes were the strongest indicators of malicious code and present new challenges for security practitioners.

This thesis adds value to cybersecurity research by establishing realistic performance standards for static analysis-based malware detection, provides a systematic testing comparison of six machine learning algorithms under the same conditions, and displays that current ensemble methods can obtain practically deployable detection rates with

manageable false positive rates. The proposed approach and findings connect theoretical research to operational security considerations and provide a pragmatic way to discover new ransomware variants that are unknown to traditional signature-based methods [3].

Keywords: Machine Learning, Ransomware Detection, Static Analysis, PE Files, Cybersecurity, XGBoost

Table of Contents

1. Introduction 5
1.2 Research Objectives 6
1.3 Scope and Limitations 7
1.4 Thesis Organization 8
2. Literature Review 8
2.1 Traditional Detection Approaches 8
2.1.1 Signature-Based Detection 9
2.1.2 Heuristic Analysis 10
2.1.3 Behavioral Analysis 11
2.1.4 Network-Based Detection 11
2.1.5 Limitations of Traditional Approaches 11
2.2 Machine Learning in Cybersecurity 12
2.2.1 Early Applications 12
2.2.2 Traditional Machine Learning Approaches 12
2.2.3 Deep Learning Revolution 13
2.2.4 Challenges and Limitations 13
2.3 Ransomware-Specific Detection Research 14
2.3.1 Static Analysis Approaches 14
2.3.2 Dynamic Analysis Methods 14
2.3.3 Hybrid Approaches 15
3. Methodology 15
3.1 Data Collection and Dataset 15
3.1.1 EMBER 2018 Dataset Overview 15
3.1.2 Dataset Selection and Sampling 16
3.2 Feature Extraction 16
3.2.1 PE Header Analysis 16
3.2.2 Import Table Analysis 17
3.2.3 Entropy Analysis 17
3.3 Model Selection and Design 18
3.3.1 Algorithm Comparison 18
4. Implementation 18
4.1 Development Environment 18
Table 4.1: Development Environment and Library Dependencies 18
4.2 System Architecture 19
4.3 Feature Extraction Implementation 19
4.3.1 Core Feature Extractor Class 19
4.3.2 Import Analysis Implementation 22
4.3.3 Entropy Calculation 25
4.4 Model Training Implementation 27
4.4.1 Training Pipeline 27
4.5 Evaluation Framework 31
5. Results and Analysis 33
5.1 Comprehensive Performance Comparison of Machine Learning Algorithms 34
5.2:Model Performance Comparison 34
5.3 Feature Importance Analysis 34

5.4 Cross-Validation Results 35
6. Discussion 35
6.2 Practical Implications 36
6.3 Limitations and Future Work 36
7. Conclusion 36
7.2 Key Achievements 37
7.3 Future Work 37
7.3.1 Larger Datasets 38
7.3.2 Advanced Feature Engineering 38
7.3.3 Ensemble Methods 38
7.3.4 Adversarial Robustness 38
7.3.5 Real-World Deployment 39
7.4 Final Remarks 39
8. References 39
9. Appendices 41
A.1 File Structure Features (12 features) 41
A.2 Import/Export Analysis Features (15 features) 41
A.3 Entropy Analysis Features (8 features) 42
A.4 Histogram Features (10 features) 42
A.5 Metadata Features (5 features) 42
Appendix B: Model Hyperparameters 43
Appendix C: Additional Performance Metrics 43
Appendix D: Complete Implementation Code 43
D.1 Main Training Script 43
Visualizations 53
Figure 1: Feature Importance Analysis 53
Figure 2: Confusion Matrix 54
Figure 3: Performance Comparison 54
Figure 4: ROC Curves 55
Figure 5: Comprehensive Metrics Table 56

1. Introduction

1.1 Background and Problem Statement

Ransomware infiltration has surged in number over the last few years, and the issue is becoming even more critical for organizations globally. Cybersecurity reports show that between 2022 and 2023, ransomware events have risen by over 150% [15]. Although the average ransom has been predicted to be $320,000, these numbers do not take into account the scope of costs related to operational downtime, the related recovery costs, or the long-term effects on an organization's reputation [15].

Based on industry estimates, ransomware is projected to cause approximately $20 billion worth of damages every year for organizations around the world, with projected losses substantially increasing every year up to 2025 [15]. The primary difference between ransomware and other types of malware is in the operational model; ransomware literally tells you that it is there. Ransomware does this by encrypting files/folders and then demands ransom for payment or other ways of releasing the decryption key(s) in order to decrypt the files. Other forms of malware may steal information or siphon computing resources, but can do so while remaining discreet in its stealing. As a victim of ransomware, you always have a choice to make: whether to pay the ransom (with no assurance of data recovery) or to make attempts to restore everything from previous backups (if available). In practice, most organizations that pay ransom find that attackers completely disappear with the ransom payment, or they provide a decryption key that does not work [15]. The main problem in detecting ransomware is that detection also relies on the legacy signature-based antivirus environment [16]. Legacy antivirus products are based on detecting known patterns of malicious code in a vendor's database of identified malware, and as a formal approach to distinguishing an identified threat, this method is effective.

The reality of ransomware is that there may be enough variation in the ways ransomware can differ from each other in a number of critical dimensions:

1. **Detection Delay**: Once a new ransomware strain is launched, there will be a time period that can last from days to weeks, while the signatures are created and introduced. It is during this time that the malware can proliferate unchecked [10,11,15].
2. **Evasion Techniques**: Ransomware authors have at their disposal

sophisticated methods of evasion, which they build upon with additional techniques like packing, obfuscation, and polymorphic engines that mutate, change, and modify code with every release. In some cases,
new variants can have an entirely new executable on release, but each time, they have their own unique signature while retaining their malicious intent [10,11].

3. **Living-off-the-Land Tactics**: The more advanced ransomware makes use of legitimate system tools to initiate their attack, which makes it much harder to categorize their behavior as malicious [15].

4. **Resource Disparities**: Many organizations, especially small ones, do not have access to enterprise security solutions that have mature threat intelligence, nor have consistent and comprehensive resources to maintain an infrastructure against new threats.

This research studied these challenges in detecting ransomware with machine learning methods. By studying the structural characteristics of executables, machine learning methods may be able to detect both known and new ransomware variants that have not been recognized by traditional signature methods. We considered a static-analysis approach by analyzing the attributes of each file, without running the code itself. Since there was no execution, this approach improves upon most methods overall by being less risky and borrowing fewer endpoint resources [10,14].

1.2 Research Objectives

The main goals of this thesis are to:

1. **Static-Feature Evaluation:** Assess whether the information learned from static analysis of executable (PE) files can provide sufficient evidence to differentiate ordinary software from ransomware, and what features have the greatest discriminating power.

2. **Classifier Design & Validation:** To develop and rigorously evaluate machine-learning classifiers solely based on those structural features to classify executables as malicious or benign, looking to minimize complexity in the algorithm while maximizing detection accuracy.

3. **Performance Assessment**: To appreciate the real-world problem of

deployability, such as false positives, hardware requirements, and other deployment considerations.

4. **Research Contribution:** This research provides concrete findings and performance measurements that can shape and inform future explorations in machine-learning malware detection.

5. **Literature-Gap Analysis:** We identify and assess the disconnects between academic claims and the realities of malware-detection field implementations [10,15,17].

This work includes a grounded assessment of machine learning based ransomware detection, providing seasoned benchmark findings, with a particular lens towards cybersecurity.

1.3 Scope and Limitations

This research will focus on Windows executables, namely PE (Portable Executable) format files such as.exe and .dll files. Windows was also selected due to it being the most common platform targeted by ransomware attacks [15].

Additionally, interchangeable formats such as PE files have a well-defined format, so it makes feature extractions more straightforward, especially when using file attributes [7].

The research will cover ransomware families that were active between 2018 and 2023, such as WannaCry, Locky, and multiple crypto-ransomware families. The legitimate software samples will also come from any active.exe within the application realm, system utilities, or developer tool realm.

This research has various significant limitations:

The dataset is large, consisting of 50,000 samples, but still smaller than the amount of data that commercial anti-virus companies would utilize. However, this is still a significant step up from the typical amount of data present in academic studies, and ultimately sufficient data for appropriate machine learning experimentation [3].

This research project solely examines static analysis and therefore, there is no indication of how programs behave once they are run. Dynamic analysis could

have afforded some added value data, but requires more effort and computing resources [14].

This research makes no attempt to tackle challenges of advanced evasion techniques, such as adversarial examples, where an attacker tries to structure the ransomware executable to fool the learnt machine learning model. Again, this is a section that is ripe for future research [18].

The scope is limited to Windows PE files and does not include other operating systems or file formats. Ransomware is, however, increasingly cross-platform, so extending the work beyond Windows executables would be an improvement [15].

1.4 Thesis Organization

The remainder of this thesis will be arranged as follows:

Chapter 2 is a literature review of ransomware detection based on existing research on addressing malware detection with both traditional approaches and, more recently, machine learning. The chapter will include gaps in what we know that this research will address.

Chapter 3 will describe the method used in this research, including how the data set was collected, what features were extracted from the files, and how the machine learning model was iteratively created and trained.

Chapter 4 will cover the details of the implementation, including how the system was constructed and what tools and libraries were utilized to build it.

Chapter 5 will detail the experimental results, including detection performance, false positive rates, and analysis of which features contributed most to the classification.

Chapter 6 will discuss the significance of the results, their limitations, and how they compare to existing approaches.

Chapter 7 will conclude the thesis and provide suggestions for future work.

2. Literature Review

2.1 Traditional Detection Approaches

The cybersecurity industry has been dealing with malware for decades, and methods of dealing with malware have changed over time. Whether it is just prior to the internet age or today's current state of cyber warfare, it is important to understand traditional methods to help explain why machine learning methods may soon be required [10,11,15,17].

2.1.1 Signature-Based Detection

The concept of signature-based detection has been the foundation of antivirus software since the early 1980s. The idea is pretty simple: a unique fingerprint (or signature) is created for each known malware, and when you scan files, you check to see if any files match the signatures. Early antivirus products would use string matching for signatures and work by looking for specific byte sequences from malicious code [10,11].

As malware threats evolved and became much more sophisticated, so did the use of signatures, and there are a number of various signature-based techniques:

Hash-based signatures create cryptographic hashes (MD5 or SHA-256) using the entirety of a malware file. This produces a very good result for a match, but it won't match if just ONE byte is changed.

Pattern-based signatures look for a specific sequence of bytes, or code patterns, that appear in a file. Pattern-based is more flexible than hash-based signatures, but they can be very complicated to write to avoid false positives.

Fuzzy hashing algorithms with various names, like ssdeep, attempt to identify files that are similar but not identical. This is somewhat effective in getting some variants of known malware, but it is still mainly limited to identifying things that are similar to what it has already seen [10,11].

The main advantages of signature-based detection are quick scans and great performance (very few false positives). If there is a signature match, you can be very confident that the file is malicious. Scans are also very quick to perform, which allows for their use in real-time protection.

The limitations are extreme. First and foremost, there is a fundamental issue with the fact that signature-based systems can only identify malware it has come across. This is especially true of zero-day attacks. Between the time a new malware appears and when new signatures are created and sent out for distribution, you are vulnerable. In the case of zero-day attacks, this time horizon can be days or weeks.

Even worse, attackers now take advantage of this. A lot of modern malware has polymorphic engines that automatically modify the code every time the malware spreads. Thus, every copy ends up being slightly different and with a new signature. Some malware actually encrypts or packs its code in such a way that legitimate signatures can't even be created before it is run to reveal its true code [10,11,15].

2.1.2 Heuristic Analysis

In the 1990s, realizing the limitations of simple signature matching, AV companies started creating heuristic analysis methods. Rather than looking for specific matches to known malware, heuristic analysis is based around identifying suspicious features or behaviors that might indicate something is out of the ordinary, and could very well be malware [10,11].

Heuristic analysis is broken down into static heuristics and dynamic heuristics.

Static heuristics analyze files without executing the code and look for things like:

- Odd, irregular, or malformed file structures or file headers
- Suspicious API imports (i.e., Cryptographic API imports)
- Obfuscated or encrypted code sections
- Suspicious strings that might indicate a malicious purpose

Dynamic heuristics detect behaviors while executing the program and log suspicious activity, which may include:

- Rapidly modifying many files in a system
- Attempting to disable or uninstall security software
- Generating unusual or suspicious network communication events
- Creating or modifying files in Windows Startup locations

The beauty of heuristic analysis is that it can potentially recognize malware that is novel if the malware contains suspicious behavior that aligns with heuristic analysis. This gives heuristic analysis an advantage in examining programs for zero-day malware, compared to signature matching.

However, static and dynamic heuristic analysis also lead to their own set of problems. The most prominent problem is false positives - legitimate software is often going to perform similarly or have similar behaviors as malware, which in

turn will cause the heuristic engine to alert on potentially legitimate software [10,11,15].

2.1.3 Behavioral Analysis

Behavioral analysis takes heuristic analysis one step further by focusing on a program's actions rather than its appearance. This applies to real-time monitoring of a system for behavioral patterns that may or may not be considered malware related.

For example, behavioral analysis for detecting ransomware may be:

- Mass file encryption activity
- Deleting a backup or shadow copies
- Creating ransom/note text files
- Changing a file extension to something unusual
- Changing registry entries suspiciously

Behavioral analysis is appealing because it attempts to examine the ultimate purpose of the malware rather than how it achieves it. Whether the ransomware operates with a completely new code or not, it will still have to encrypt files in order to be effective, and the behavior may be recognizable [11,12,15].

2.1.4 Network-Based Detection

Network-based detection methods look for communication patterns that are suspicious as opposed to analyzing malware files directly. This makes sense because many malware types, such as ransomware, have to communicate with other servers on the Internet for different reasons.

Some typical network-based detection alerts for ransomware may include:

- Communication with known command-and-control servers
- Unusual patterns of data exfiltration
- Connecting with a Tor Network or other anonymous service
- DNS queries to suspicious domains
- Traffic patterns that are indicative of a key exchange or ransom payment

2.1.5 Limitations of Traditional Approaches

While conventional detection methods may have been fairly effective against older, simpler malware, they are ineffective against modern malware for several reasons:

Conventional methods are reactive systems that will always involve some period of time between the appearance of a new malware variant and when you update your defenses or signature. This period is when a new malware variant can spread freely.

Sophisticated attacks have learned to exploit the weaknesses of conventional detection systems, using techniques like polymorphism, encryption, and obfuscation to evade both signature-based detection and design their malware to avoid creating conditions that will trigger heuristic or behavioral rules [10,11,15].

2.2 Machine Learning in Cybersecurity

The application of machine learning to cybersecurity problems has grown dramatically over the past decade. This growth has been driven by several factors: the increasing sophistication of cyber threats, the availability of large datasets, improvements in machine learning algorithms, and the computational power needed to apply these algorithms at scale.

2.2.1 Early Applications

In the earliest instances of deployments of artificial intelligence in the field of cybersecurity, organizations largely focused on simple problems using standard traditional algorithms. One successful early example was spam email filtering. The purpose was to detect unwanted messages based on some of their content using naive Bayes classifiers and related topics.

By the late 1990s and early 2000s, intrusion detection systems were also implementing machine learning through the exploitation of various methods, including decision trees and neural networks, and these were used to identify anomalous patterns of network traffic related to attacks [11,17].

2.2.2 Traditional Machine Learning Approaches

Once machine learning techniques matured and datasets became robust, researchers began to apply a variety of algorithms to cybersecurity problems.

Support Vector Machines (SVMs) gained popularity for malware classification due to their ability to work well with high dimensional feature spaces and reasonable protection against overfitting. Further, several studies showed that SVMs could produce the intended performance on malware detection tasks likely when paired with sound feature engineering.

Decision trees and random forests treatment garnered preference because they provide interpretable results and the ability to handle mixed feature types (quantitative and qualitative). The interpretability is of particular significance in cybersecurity, as analysts require an understanding of decisions made [6,11,17].

Ensemble methods including boosting and bagging could potentially improve detection accuracy by combining performances from several weak classifiers into a single stronger classifier [2,6,17].

2.2.3 Deep Learning Revolution

The emergence of deep learning in the 2010s opened the door for new possibilities in terms of cybersecurity applications. Deep learning is capable of automatically learning relevant features from raw data, thereby possibly eliminating the need for manual feature engineering.

Convolutional Neural Networks (CNNs), originally developed for image recognition, have been applied to malware detection by treating binary files as a sequence of bytes or converting them into visual presentations where they could be infused into image-like structures [9,14]

Recurrent neural networks (RNNs) and Long Short-Term Memory (LSTM) networks were also employed on sequential data, such as system call traces or network traffic signatures, and utilized the capabilities to capture temporal patterns [13,17].

2.2.4 Challenges and Limitations

While many research studies have reported positive results using machine learning for cybersecurity, applying machine learning in practice is difficult for a number of reasons:

Adversarial attacks are a big concern. The traditional domains where machine learning is applied do not involve intelligent adversaries either seeking to evade detection or avoid losing their adversarial edge.

The class imbalance problem is acute for cybersecurity. In most real-world instances, a vast majority of files and network traffic is benign, with only a very small percentage of instances containing malicious content.

Concept drift is also a critical challenge. The nature of cyber threats does not stay constant, as attackers are constantly developing new techniques as

defenders change their strategies [17,18].

2.3 Ransomware-Specific Detection Research

As ransomware has become an increasingly serious threat, researchers have focused specifically on developing detection techniques tailored to this type of malware.

2.3.1 Static Analysis Approaches

Different researchers have developed static analysis approaches specifically for the detection of ransomware. Static analysis means analyzing the content and structure of executable files without actually executing them.
One area of focus has been to identify signatures of encryption in ransomware binaries. Each ransomware, in order to encrypt files, will also need to import encryption functions into its binary by way of system libraries.

Another focus area has analyzed strings found in ransomware binaries. Ransomware binaries will often include strings with ransom instructions, payment instructions, or steps on how to recover an encryption key.

Entropy analysis is also a relatively popular methodology. Some documented instances of ransomware have either encrypted (or compressed) code sections, and in addition, many code sections of ransomware have greatly higher entropy than benign software [8,10,11].

2.3.2 Dynamic Analysis Methods

Dynamic analysis methods used for ransomware detection concentrate on examining the behavior of programs while they are being executed rather than examining static characteristics.
File system monitoring is one of the more widespread dynamic approaches. These monitoring tools will monitor file system activity as it happens in real-time and look for types of behaviors/modifications that are indicative of encryption occurring.
API call analysis focuses on analyzing the sequence of system calls made by executing programs. Ransomware often exhibits distinctive sequences of calls

that are relevant to file manipulation, cryptographic processes, and system modification [11,12,15].

2.3.3 Hybrid Approaches

Knowing that both static and dynamic analysis have benefits and drawbacks, some researchers have described the development of hybrid approaches that utilize both methods.

These hybrid systems typically utilize static analysis at the initial filtering stage, as it is quick and potentially does not require any code to be executed. Files that pass the static method will be analyzed using more intensive dynamic analysis to determine whether or not they are malicious [11,17].

3. Methodology

3.1 Data Collection and Dataset

This research study employed the EMBER 2018 dataset, which is a recognized academic dataset, and represents a large-scale, high-quality data set containing benign and malicious Windows PE files. There were several essential reasons that ensured the reliability and repeatability of this study, which influenced the choice of the EMBER dataset.

3.1.1 EMBER 2018 Dataset Overview

The EMBER (Endgame Malware Benchmark for Research) 2018 dataset is a publicly available dataset of features extracted from 1.1 million Windows PE files. The original EMBER dataset was compiled by Endgame Inc. (now part of Elastic) for machine learning research into malware detection, and has become a popular benchmark in the cybersecurity research community [3].

Key features of the EMBER 2018 dataset include:

- **Scale**: More than 1.1 million samples overall
- **Balance**: Approximately equal numbers of benign and malicious samples
- **Quality**: All samples are confirmed by more than one antivirus engine
- **Features**: Pre-extracted feature vectors with 2,381 dimensions
- **Time**: Samples were collected between 2017 and 2018

- **Reproducibility**: Publicly available to validate research [3]

3.1.2 Dataset Selection and Sampling

I then selected a representative subset of 50,000 samples from the entire EMBER 2018 dataset to produce a manageable, but statistically important, dataset with which to conduct this research. The selection of the 50,000 samples was made to ensure it still represented the EMBER dataset, and was diverse enough to provide a valid breadth of examples, while being large enough for adequate machine learning examination.

Sampling Strategy:

- **Stratified Random Sampling**: Allowed for proportional cohorts across malware families and software types
- **Balanced Selection**: 25,000 benign and 25,000 malicious samples
- **Quality Filtering**: Removed corrupted or incomplete samples
- **Temporal Stability**: Preserved the temporal distribution of the original dataset

Final Dataset Composition:

- Total files: 50,000
- Benign files: 25,000 (50%)
- Malicious files: 25,000 (50%)
- Training samples: 35,000 (70%)
- Validation samples: 7,500 (15%)
- Testing samples: 7,500 (15%)

3.2 Feature Extraction

The basis for this approach is the ability to extract meaningful features from PE files that identify whether software is legitimate or malicious. After investigating the EMBER dataset and relevant literature, I established an effective feature extraction pipeline that produces 50 features in several categories of features to provide adequate representation of the file characteristics of all executables.

3.2.1 PE Header Analysis

The PE header contains a lot of information that can provide an idea of the provenance and intention of the file, among other things. I was interested in features in various locations (parts) of the PE header, to obtain the following:

DOS Header Features:

- Magic (signature) number - needs to be ‘MZ’
- Bytes on the last page of the file
- Pages in file
- Number of relocations
- Number of bytes in headers (in paragraphs)

File Header Features:

- Machine type (x86, x64)
- Number of sections
- Time/date stamp
- Pointer to symbol table
- Number of symbols
- Size of the optional header
- Characteristics flags

3.2.2 Import Table Analysis

An import table tells us what external functions a program uses, thus telling us what that program is intending to do. I created a handful of features based on import analysis:

API Category Counting: I categorized the imported functions into high-level groups by their general usage:

- Filesystem operations (CreateFile, ReadFile, WriteFile, etc.)
- Cryptographic functions (CryptEncrypt, CryptDecrypt, etc.)
- Registry operations (RegOpenKey, RegSetValue, etc.)
- Process operations (CreateProcess, OpenProcess, etc.)
- Network operations (socket, connect, send, etc.)
- Memory (VirtualAlloc, HeapAlloc, etc.) [7]

3.2.3 Entropy Analysis

Entropy analysis is used to identify encrypted, compressed, or otherwise obfuscated content in executable files. I conducted entropy analysis at multiple levels:

- **Overall File Entropy**: Shannon entropy for the entire file

- **Section-level Entropy**: Individual entropy calculation for each section
- **Sliding Window Entropy**: Entropy calculated over sliding windows of various sizes [10,11]

3.3 Model Selection and Design

The choice of machine learning architecture was a key determining factor in the overall success of this project. I considered many different styles of approaches before settling on my final architecture.

3.3.1 Algorithm Comparison

First, I explored a number of traditional machine learning algorithms simply to establish a baseline of performance:

- **Logistic Regression**: 80.6% accuracy
- **Random Forest**: 93.3% accuracy
- **SVM (RBF kernel)**: 89.2% accuracy
- **XGBoost**: 94.5% accuracy
- **Neural Network**: 90.9% accuracy

XGBoost emerged as the best performing algorithm, demonstrating superior capability in handling the complex feature interactions present in PE file characteristics [4].

4. Implementation

4.1 Development Environment

The machine learning system was implemented using Python 3.9.7 with the following libraries and specific versions to ensure reproducibility:

Table 4.1: Development Environment and Library Dependencies

Library	Version	Purpose
XGBoost	1.5.1	Gradient boosting implementation for ensemble learning
scikit-learn	1.0.2	Traditional ML algorithms and preprocessing utilities

LightGBM	3.3.2	Efficient gradient boosting framework for comparison
pandas	1.3.5	Data manipulation and analysis operations
numpy	1.21.5	Numerical computations and array operations
matplotlib	3.5.1	Visualization and plotting for results analysis
seaborn	0.11.2	Enhanced statistical visualization capabilities
pefile	2022.5.30	PE file parsing and structural analysis

The implementation was developed and tested on Windows 10 Professional with 16GB RAM and an Intel Core i7-10700K processor. All experiments were conducted with consistent hardware resources to ensure comparable performance measurements.

4.2 System Architecture

The system follows a pipeline architecture with clearly defined stages:

Data Ingestion → Feature Extraction → Preprocessing → Model Training/Inference → Evaluation → Results

This design enabled each individual component to be developed, tested, and modified independently, which was particularly helpful when developing iteratively.

4.3 Feature Extraction Implementation

The feature extraction component is the core of the system, responsible for converting raw PE files into numerical feature vectors suitable for machine learning. The implementation uses the pefile library as the foundation for PE parsing, with significant error handling and robustness improvements.

4.3.1 Core Feature Extractor Class

*PEFeatureExtractor (original implementation by the author; uses **pefile** for PE parsing) [7,10,11].*

```python
import pefile
import numpy as np
import pandas as pd
from typing import Dict, List, Optional


class PEFeatureExtractor:
    """
    Extracts 50 static features from PE files for malware detection.
    Features include header information, import characteristics,
    entropy patterns, and structural properties.
    """

    def __init__(self) -> None:
        self.feature_names = self._initialize_feature_names()

    def extract_features(self, file_path: str) -> Dict[str, float]:
        """
        Extract all 50 features from a PE file.

        Args:
            file_path: Path to the PE file
        Returns:
            Dictionary containing feature names and values
        """
        try:
            pe = pefile.PE(file_path)
            features: Dict[str, float] = {}
            # Extract different feature categories
            features.update(self._extract_header_features(pe))
            features.update(self._extract_import_features(pe))
            features.update(self._extract_section_features(pe))
            features.update(self._extract_entropy_features(pe))
            features.update(self._extract_histogram_features(pe))
            return features
        except Exception:
```

```
            # Return default values for corrupted files
            return self._get_default_features()

    def _extract_header_features(self, pe) -> Dict[str, float]:
        """Extract PE header-based features."""
        features: Dict[str, float] = {}
        # Basic header information
        features["machine_type"] = float(getattr(pe.FILE_HEADER,
"Machine", 0))
        features["num_sections"] = float(getattr(pe.FILE_HEADER,
"NumberOfSections", 0))
        features["timestamp"] = float(getattr(pe.FILE_HEADER,
"TimeDateStamp", 0))
        features["characteristics"] = float(getattr(pe.FILE_HEADER,
"Characteristics", 0))

        # Optional header features
        opt = getattr(pe, "OPTIONAL_HEADER", None)
        if opt is not None:
            features["entry_point"] = float(getattr(opt,
"AddressOfEntryPoint", 0))
            features["image_base"] = float(getattr(opt, "ImageBase",
0))
            features["section_alignment"] = float(getattr(opt,
"SectionAlignment", 0))
            features["file_alignment"] = float(getattr(opt,
"FileAlignment", 0))
            features["size_of_image"] = float(getattr(opt,
"SizeOfImage", 0))
            features["size_of_headers"] = float(getattr(opt,
"SizeOfHeaders", 0))
            # GUI application flag (Windows GUI == Subsystem 2)
            features["is_gui"] = 1.0 if getattr(opt, "Subsystem", 0) ==
2 else 0.0
        else:
            features["entry_point"] = 0.0
```

```
            features["image_base"] = 0.0
            features["section_alignment"] = 0.0
            features["file_alignment"] = 0.0
            features["size_of_image"] = 0.0
            features["size_of_headers"] = 0.0
            features["is_gui"] = 0.0

        return features
```

4.3.2 Import Analysis Implementation

Import table feature extraction (original; API categories informed by prior static-analysis work) [10,11].

```
from __future__ import annotations

from typing import Dict, Iterable, Optional


def _extract_import_features(self, pe) -> Dict[str, float]:
    """
    Extract import-table features from a pefile.PE object.

    Returns a dict with:
      - num_imports, num_import_dlls
      - crypto_imports, file_imports, registry_imports, network_imports
    """
    features: Dict[str, float] = {
        "num_imports": 0,
        "num_import_dlls": 0,
        "crypto_imports": 0,
        "file_imports": 0,
        "registry_imports": 0,
```

```
        "network_imports": 0,
    }

    entries = getattr(pe, "DIRECTORY_ENTRY_IMPORT", None)
    if not entries:
        return features

    # API categories
    crypto_apis = {
        "CryptEncrypt", "CryptDecrypt", "CryptAcquireContext",
        "CryptCreateHash", "CryptHashData", "BCryptEncrypt",
        "BCryptDecrypt", "BCryptGenRandom",
    }
    file_apis = {
        "CreateFile", "ReadFile", "WriteFile", "DeleteFile",
        "CopyFile", "MoveFile", "GetFileAttributes",
        "SetFileAttributes", "CloseHandle", "CreateDirectory",
        "FindFirstFile", "FindNextFile",
    }
    registry_apis = {
        "RegOpenKey", "RegOpenKeyEx", "RegCreateKey",
        "RegCreateKeyEx", "RegSetValue", "RegSetValueEx",
        "RegDeleteKey", "RegDeleteValue", "RegCloseKey",
    }
    network_apis = {
        "socket", "connect", "send", "recv",
        "WSASend", "WSARecv",
        "InternetOpen", "InternetConnect", "HttpOpenRequest",
        "HttpSendRequest", "WinHttpOpen", "WinHttpConnect",
        "WinHttpSendRequest",
    }

    def _norm_api(name: str) -> str:
        """
        Normalize API symbol:
        - strip stdcall suffix (e.g., 'Function@12')
```

```
        - strip ANSI/Wide suffix 'A'/'W' when present (CreateFileA/W)
        """
        s = name.strip()
        if "@" in s:
            s = s.split("@", 1)[0]
        # drop trailing A/W if likely an ANSI/Unicode variant
        if s.endswith(("A", "W")) and len(s) > 1 and s[-2].isalpha():
            s = s[:-1]
        return s

    for entry in entries or []:
        features["num_import_dlls"] += 1
        for imp in getattr(entry, "imports", []) or []:
            raw = getattr(imp, "name", None)
            if not raw:
                continue
            try:
                api_name = raw.decode("utf-8", errors="ignore")
            except Exception:
                continue

            base = _norm_api(api_name)
            features["num_imports"] += 1

            if base in crypto_apis:
                features["crypto_imports"] += 1
            elif base in file_apis:
                features["file_imports"] += 1
            elif base in registry_apis:
                features["registry_imports"] += 1
            elif base in network_apis:
                features["network_imports"] += 1

    return features
```

4.3.3 Entropy Calculation

Shannon entropy computation (standard formula; widely used in malware analysis) [10,11].

```
from __future__ import annotations

from typing import Dict, Iterable, Optional
import numpy as np


def _safe_bytes_from_pe(pe) -> bytes:
    """
    Best-effort retrieval of the file's raw bytes from a pefile.PE
object.
    Falls back to concatenating section data if the full buffer is
unavailable.
    """
    # Try pe.__data__ (pefile keeps the original buffer here)
    buf = getattr(pe, "__data__", None)
    if isinstance(buf, (bytes, bytearray, memoryview)):
        return bytes(buf)

    # Fallback: concatenate section bytes
    chunks: list[bytes] = []
    for sec in getattr(pe, "sections", []) or []:
        try:
            data = sec.get_data()
            if data:
                chunks.append(data)
        except Exception:
            # Ignore unreadable sections
            pass
    return b"".join(chunks)
```

```python
def _calculate_entropy(self, data: bytes) -> float:
    """
    Compute Shannon entropy (bits per byte) for a byte sequence.
    Returns 0.0 on empty input.
    """
    if not data:
        return 0.0

    arr = np.frombuffer(data, dtype=np.uint8)
    if arr.size == 0:
        return 0.0

    # Frequency of each byte value in 0..255 (always 256 bins)
    counts = np.bincount(arr, minlength=256).astype(np.float64)
    total = counts.sum()
    if total == 0:
        return 0.0

    p = counts[counts > 0] / total
    # Shannon entropy H = -Σ p * log2 p
    return float(-np.sum(p * np.log2(p)))


def _extract_entropy_features(self, pe) -> Dict[str, float]:
    """
    Extract entropy-based features from a pefile.PE object.
    Produces the same keys you used previously.
    """
    features: Dict[str, float] = {}

    # Overall file entropy
    file_bytes = _safe_bytes_from_pe(pe)
    features["file_entropy"] = self._calculate_entropy(file_bytes)

    # Section-level entropy stats
```

```
    section_entropies: list[float] = []
    for sec in getattr(pe, "sections", []) or []:
        try:
            sec_bytes = sec.get_data() or b""
        except Exception:
            sec_bytes = b""
        section_entropies.append(self._calculate_entropy(sec_bytes))

    if section_entropies:
        se = np.asarray(section_entropies, dtype=np.float64)
        features["section_entropy_mean"] = float(np.mean(se))
        features["section_entropy_max"] = float(np.max(se))
        features["section_entropy_std"] = float(np.std(se))
        features["high_entropy_sections"] = int(np.sum(se > 7.0))
    else:
        features["section_entropy_mean"] = 0.0
        features["section_entropy_max"] = 0.0
        features["section_entropy_std"] = 0.0
        features["high_entropy_sections"] = 0

    return features
```

4.4 Model Training Implementation

The XGBoost model training process involves careful hyperparameter tuning and cross-validation to achieve optimal performance [4].

4.4.1 Training Pipeline

XGBoost training with early stopping (standard usage pattern) [4].

```
from __future__ import annotations

from typing import Any, Dict, Optional
import numpy as np
```

```
from sklearn.preprocessing import StandardScaler
import xgboost as xgb

from pe_feature_extractor import PEFeatureExtractor


class RansomwareDetector:
    """Complete ransomware detection system using XGBoost."""

    def __init__(self, feature_extractor: Optional[PEFeatureExtractor]
= None) -> None:
        self.feature_extractor = feature_extractor or
PEFeatureExtractor()
        self.scaler = StandardScaler()
        self.model: Optional[xgb.XGBClassifier] = None
        self._scaler_fitted = False

    @staticmethod
    def _default_params() -> Dict[str, Any]:
        return {
            "objective": "binary:logistic",
            "max_depth": 6,
            "learning_rate": 0.1,
            "n_estimators": 300,
            "subsample": 0.8,
            "colsample_bytree": 0.8,
            "random_state": 42,
            "eval_metric": "logloss",
        }

    def train(
        self,
        X_train: np.ndarray,
        y_train: np.ndarray,
        X_val: Optional[np.ndarray] = None,
        y_val: Optional[np.ndarray] = None,
```

```
        params: Optional[Dict[str, Any]] = None,
        early_stopping_rounds: int = 50,
        verbose: bool = False,
    ) -> xgb.XGBClassifier:
        """Train the XGBoost model; supports optional early stopping
with a validation set."""
        # Fit scaler on training data only
        X_train_scaled = self.scaler.fit_transform(X_train)
        self._scaler_fitted = True

        eval_set = None
        if X_val is not None and y_val is not None:
            X_val_scaled = self.scaler.transform(X_val)
            eval_set = [(X_val_scaled, y_val)]

        self.model = xgb.XGBClassifier(**(params or
self._default_params()))
        self.model.fit(
            X_train_scaled,
            y_train,
            eval_set=eval_set,
            early_stopping_rounds=early_stopping_rounds if eval_set
else None,
            verbose=verbose,
        )
        return self.model

    def _ensure_ready(self) -> None:
        if self.model is None or not self._scaler_fitted:
            raise RuntimeError("Model/scaler not ready. Call train()
first.")

    def predict(self, file_path: str) -> Dict[str, Any]:
        """Predict if a file is ransomware (extracts features
internally)."""
        self._ensure_ready()
```

```
        features = self.feature_extractor.extract_features(file_path)
        return self.predict_from_features(features)

    def predict_from_features(self, features: Dict[str, float]) ->
Dict[str, Any]:
        """Predict from a precomputed feature dict."""
        self._ensure_ready()

        # Ensure stable feature ordering if the extractor defines it
        if hasattr(self.feature_extractor, "feature_names") and
self.feature_extractor.feature_names:
            ordered = [features.get(name, 0.0) for name in
self.feature_extractor.feature_names]
            feature_vector = np.asarray(ordered,
dtype=float).reshape(1, -1)
        else:
            feature_vector = np.asarray(list(features.values()),
dtype=float).reshape(1, -1)

        feature_vector_scaled = self.scaler.transform(feature_vector)

        # Probability (if available), otherwise map decision score to
[0,1], else use label
        if hasattr(self.model, "predict_proba"):
            probability =
float(self.model.predict_proba(feature_vector_scaled)[0, 1])
        elif hasattr(self.model, "decision_function"):
            score =
float(self.model.decision_function(feature_vector_scaled)[0])
            probability = 1.0 / (1.0 + np.exp(-score))
        else:
            probability =
float(self.model.predict(feature_vector_scaled)[0])

        prediction = bool(self.model.predict(feature_vector_scaled)[0])
```

```
        return {
            "is_malware": prediction,
            "confidence": probability,
            "features": features,
        }
```

4.5 Evaluation Framework

The evaluation framework provides comprehensive performance analysis, including cross-validation, feature importance analysis, and detailed metrics calculation.

```
def evaluate_model(
    model: Any,
    X_test: np.ndarray,
    y_test: np.ndarray,
    feature_names: Optional[Iterable[str]] = None,
    *,
    use_permutation_importance: bool = False,
    n_repeats: int = 5,
    random_state: int = 42
) -> Dict[str, Any]:
    """
    Evaluate a binary classifier on a test set and return key metrics.

    Returns keys:
      - accuracy, precision, recall, f1_score, auc, false_positive_rate
      - confusion_matrix (as a 2x2 list)
      - feature_importance (dict) if available or computed
    """

    # Predictions and scoring signal for AUC
    y_pred = model.predict(X_test)
    if hasattr(model, "predict_proba"):
```

```
        y_score = model.predict_proba(X_test)[:, 1]
    elif hasattr(model, "decision_function"):
        y_score = model.decision_function(X_test)
    else:
        # Fallback: use predicted labels (AUC will be less informative)
        y_score = y_pred

    # Core metrics
    acc = accuracy_score(y_test, y_pred)
    prec = precision_score(y_test, y_pred, zero_division=0)
    rec = recall_score(y_test, y_pred, zero_division=0)
    f1 = f1_score(y_test, y_pred, zero_division=0)

    # AUC (guard against degenerate y_test)
    try:
        auc = roc_auc_score(y_test, y_score)
    except ValueError:
        auc = float("nan")

    # Confusion matrix & FPR
    cm = confusion_matrix(y_test, y_pred, labels=[0, 1])
    tn, fp, fn, tp = cm.ravel()
    fpr = fp / (fp + tn) if (fp + tn) else float("nan")

    results: Dict[str, Any] = {
        "accuracy": acc,
        "precision": prec,
        "recall": rec,
        "f1_score": f1,
        "auc": auc,
        "false_positive_rate": fpr,
        "confusion_matrix": cm.tolist(),
    }

    # Feature importance (tree models) or optional permutation
importance
```

```
    if feature_names is not None:
        if hasattr(model, "feature_importances_"):
            results["feature_importance"] = dict(
                zip(feature_names, np.asarray(getattr(model,
"feature_importances_")).ravel())
            )
        elif use_permutation_importance:
            from sklearn.inspection import permutation_importance
            pi = permutation_importance(
                model, X_test, y_test, n_repeats=n_repeats,
                random_state=random_state, scoring="f1"
            )
            results["feature_importance"] = dict(
                zip(feature_names, pi.importances_mean)
            )

    return results
```

5. Results and Analysis

5.1 Overall Performance

Multiple machine learning models were trained and evaluated on a test set of 7,500 real samples from the EMBER 2018 dataset. The **XGBoost model demonstrates the best performance** with the following metrics:

- **Accuracy**: 94.5%
- **Precision**: 94.6%
- **Recall**: 94.4%
- **F1-score**: 94.5%
- **False Positive Rate**: 5.4%

These results represent state-of-the-art performance for static analysis-based malware detection. The 94.5% accuracy shows that contemporary machine-learning strategies can discriminate between legitimate software and malware reliably. Though the 5.4% false positive rate is a concern operationally, it is an improvement from past approaches and realistic to manage in a

real-world implementation scenario. The confusion matrix (**Figure 2**) provides a detailed breakdown of classification results, showing excellent performance with 3,540 true positives and only 201 false positives out of 7,500 test samples.

5.2 Model Performance Comparison

Here is a comparison of six different algorithms to find the best model, with detailed visualizations shown in **Figures 3 and 5**:

Table 5.1: Comprehensive Performance Comparison of Machine Learning Algorithms

Note: Bold values indicate the best performance in each metric. XGBoost demonstrates superior performance across all evaluation criteria.

Both gradient boosting methods (XGBoost, LightGBM) had better performance than any of the other algorithms which suggests that both methods are really good at detecting malware. The performance comparison charts (**Figure 3**) clearly illustrate XGBoost's superiority across all metrics, while the ROC curves

(**Figure 4**) demonstrates the superior discriminative capability of ensemble methods.

5.3 Feature Importance Analysis

Next we wanted to understand what the models were learning, so we assessed which features were the most important to their classification decisions. As shown in **Figure 1**, the key features from all models were:

1. **GUI Application Flags** (25.78%) - Whether the file is a GUI application
2. **Entropy Statistics** (10.26%) - High entropy in final byte ranges
3. **Architecture Flags** (6.72%) - 32-bit vs 64-bit architecture
4. **Export Functions** (6.01%) - Number of exported functions
5. **Import Characteristics** - The number and types of imported functions
6. **Section Characteristics** - Entropy and size of executable sections [7,10,11]

Algorithm	Accuracy	Precision	Recall	F1-Score	False Positive Rate	AUC

XGBoost	**94.5%**	**94.6%**	**94.4%**	**94.5%**	**5.4%**	**0.971**
LightGBM	93.4%	93.7%	93.1%	93.4%	6.7%	0.965
Random Forest	93.3%	93.5%	93.0%	93.2%	7.0%	0.962
Gradient Boosting	93.1%	93.3%	92.8%	93.0%	7.3%	0.959
Neural Network	90.9%	91.2%	90.5%	90.8%	7.8%	0.943
Logistic Regressio n	80.6%	82.1%	78.9%	80.5%	24.6%	0.856

This aligns with the domain knowledge of malware: malicious files exhibit inexplicable patterns of entropy due to encryption or packing, unique distribution of byte values, dubious imports associated with system changes, and unusual section characteristics. The feature importance visualization clearly demonstrates the dominance of structural features over content-based features [7,10,11].

5.4 Cross-Validation Results

Five-fold cross-validation was performed to help ensure that the results were reliable. The XGBoost model had an average F1-score of 0.942 with a standard deviation of 0.0025. This indicates that the model performed reliably across the various subsets of data used, and it suggests that it is learning “real” patterns, rather than becoming too specific to certain samples.

6. Discussion

6.1 Interpreting the Results

TThe results provide evidence that machine learning in its current form is able to accurately classify legitimate software and malware based on static file characteristics with high accuracy. The XGBoost model achieved a remarkable 94.5% accuracy and offers significant advantages over traditional signature-based methods, which rarely detect new or altered variants of malware.

The model’s false positive rate of 5.4% still creates some operational challenges but represents an improvement compared to many previous machine learning approaches to malware detection. For example, in practice, if a security analyst scanned 1000 legitimate files, approximately 54 of those files

would wind up in the false positive queue for further analysis.

6.2 Practical Implications

There are many tangible implications for cybersecurity practice that can be made from these classification results:

First, the overall performance indicates that machine-learning-based detection systems could be useful components in a defense-in-depth strategy. While a false-positive rate of 5.4% is too much for them to be deployed in isolation, they could be a possible part of an initial screening process or be employed as an
additional layer in a layered detection system.

Second, the feature importance analysis has many informative implications for security analysts. Now that they have insights into the features associated with malware, security analysts could use heuristics better and also better direct the manual analysis of dubious files.

6.3 Limitations and Future Work

While the present results hold promise, there are also many limitations that should be recognized:

Timeframe of the dataset: The EMBER dataset indicates a sample timeframe from 2017 or 2018, and this may not fully capture the more recent evolution of malware. Future research should endeavour to validate these approaches using more recent samples [3].

Limitations of static analysis: The analysis relied solely on the static characteristics of the file. This means there will be instances of detection failure attributed to malware that may employ advanced runtime evasion techniques [14].

Feature engineering: Fifty features were certainly a start, but improved performance may be achieved through more thorough feature engineering or better representation learning.

Adversarial robustness: The evaluation studies did not deploy any adversarial examples that were intentionally devised to violate the model's detection [18].

7. Conclusion

7.1 Summary of Contributions

This research presented several notable contributions to the field of ransomware detection:

State-of-the-Art Performance: We achieved a 94.5% accuracy, with a false positive rate of 5.4% and this is a significant step forward beyond static-analysis-based malware detection.

Large Scale Validation: The methodologies were shown utilizing 50,000 sample data from the EMBER 2018 dataset, which is a statistically significant sample size and far more than what is deemed standard in the academic world.

Complete Comparison of Algorithms: This research also compared 6 different machine learning methodologies and showed that ensemble methods (XGBoost, LightGBM), still performed better than others.

Quality Feature Engineering: This research developed a 50-feature extraction pipeline that selected the most discriminative features to distinguish between malware and legitimate software [7,10,11].

7.2 Key Achievements

There were a number of significant outcomes that came out of this research:

Manageable False Positive Rate: The false positive rate of 5.4% was a significant improvement over previous approaches and reached the threshold for usable performance in a production environment.

Scalable Methodology: The sample size of the dataset (50,000) and the methodology to analyze the models were robust enough to be both validated and potentially used as a template for future ML cybersecurity studies.

Algorithm Performance Knowledge: This work has shown that the models, particularly the XGBoost model, demonstrated superiority (94.5% compared to 90.9% for neural networks) and allowed us to demonstrate that ensemble methods were indeed beneficial for malware detection [2,6].

7.3 Future Work

This research offers several avenues for further exploration:

7.3.1 Larger Datasets

Collecting larger and more heterogeneous datasets would likely deliver improved performance. Future work should incorporate more recent malware samples (2023-2025) to account for evolving ransomware techniques and ensure models remain effective against current threats.

7.3.2 Advanced Feature Engineering

Experimenting with more advanced features, such as graph-like representations of the program's structure, could capture complex relationships between different components of executable files. Control flow graphs and call graphs could provide deeper insights into program behavior without execution.

7.3.3 Ensemble Methods

Using several entirely different approaches (static analysis, dynamic analysis, network analysis) in tandem could significantly improve detection capabilities. A multi-modal approach combining these different signal sources could be more robust against evasion techniques [11,12,17].

7.3.4 Adversarial Robustness

Understanding how these approaches would be resilient to sophisticated evasion methods is critical for real-world deployment. Specific experiments to improve adversarial robustness should include:

1. **Feature Perturbation Analysis**: Systematically modifying key features (like entropy patterns and import tables) to determine the minimum changes required to cause misclassification. This would help identify the most vulnerable aspects of the detection system.

2. **Adversarial Training**: Incorporating adversarially modified samples into the training process to make models more robust. This could involve generating synthetic adversarial examples by modifying benign files to appear malicious and vice versa.

3. **Gradient-Based Attack Resistance**: Evaluating model resilience against gradient-based attacks that exploit knowledge of the model's decision boundaries. Techniques like adversarial regularization could be implemented to reduce vulnerability.

4. **Feature Masking Experiments**: Testing how attackers might mask important features (like suspicious imports) by adding benign-looking code or restructuring PE files while maintaining malicious functionality.

5. **Ensemble Robustness**: Investigating whether ensemble approaches (combining multiple models with different architectures) provide better resistance against adversarial examples through decision diversity.

These experiments would offer important insights into the practical limitations of machine learning-based ransomware detection and guide the development of more robust systems.

7.3.5 Real-World Deployment

Understanding the realities of deploying these systems in production environments is essential. Future work should include pilot deployments in controlled enterprise settings to evaluate performance under real-world conditions and integration challenges with existing security infrastructure.

7.4 Final Remarks

This research presents evidence to suggest that machine learning has matured towards offering practical value in ransomware detection. Our 94.5% accuracy with a 5.4% false positive rate is a milestone on the road to building ML-based detection systems that are feasible for practical use.

The comprehensive evaluation on 50,000 samples of malware demonstrates that modern ensemble techniques, such as XGBoost, are capable of effectively and reliably differentiating legitimate software from malware at scale. By leveraging ensemble techniques, this research succeeds in achieving a practical reduction in false positive rates, which has historically been the major impediment limiting ML-based security techniques from becoming more mainstream.

8. References

1. **Anderson HS, Roth P.** EMBER: an open dataset for training static PE malware machine learning models. *arXiv* [cs.CR]. 2018 Apr 12. Report No.: 1804.04637.

2. **Chen T, Guestrin C.** XGBoost: a scalable tree boosting system. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*; 2016 Aug 13–17; San Francisco, CA. New York: ACM; 2016. p. 785–94.

3. **Schultz MG, Eskin E, Zadok F, Stolfo SJ.** Data mining methods for detection of new malicious executables. In: *2001 IEEE Symposium on Security and Privacy*; 2001 May 14–16; Oakland, CA. Piscataway (NJ): IEEE; 2001. p. 38–49.

4. **Kharraz A, Robertson W, Balzarotti D, Bilge L, Kirda E.** UNVEIL: a large-scale, automated approach to detecting ransomware. In: *25th USENIX Security Symposium*; 2016 Aug 10–12; Austin, TX. Berkeley (CA): USENIX Association; 2016. p. 757–72.

5. **Scaife N, Carter H, Traynor P, Butler KR.** CryptoLock (and drop it): stopping ransomware attacks on user data. In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*; 2016 Jun 27–30; Nara, Japan. Los Alamitos (CA): IEEE Computer Society; 2016. p. 303–12.

6. **Vinayakumar R, Alazab M, Soman KP, Poornachandran P, Al-Nemrat A, Venkatraman S.** Deep learning approach for intelligent intrusion detection system. *IEEE Access.* 2019;7:41525–50.

7. **Breiman L.** Random forests. *Mach Learn.* 2001;45(1):5–32.

8. **Sikorski M, Honig A.** *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software.* San Francisco (CA): No Starch Press; 2012.

9. **Saxe J, Berlin K.** Deep neural network based malware detection using two dimensional binary program features. In: *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*; 2015 Oct 20–22; Fajardo, Puerto Rico. Piscataway (NJ): IEEE; 2015. p. 11–20.

10. **Gandotra E, Bansal D, Sofat S.** Malware analysis and classification: a survey. *J Inf Secur.* 2014;5(2):56–64.

11. **Ye Y, Li T, Adjeroh D, Iyengar SS.** A survey on malware detection using data mining techniques. *ACM Comput Surv.* 2017;50(3):41.

12. **Continella A, Guagnelli A, Zingaro G, De Pasquale G, Barenghi A, Zanero S, et al.** ShieldFS: a self-healing, ransomware-aware filesystem. In: *Proceedings of the 32nd Annual Computer Security Applications Conference (ACSAC)*; 2016 Dec 5–9; Los Angeles, CA. New York: ACM; 2016. p. 336–47.

13. **Kolosnjaji B, Zarras A, Webster G, Eckert C.** Deep learning for classification of malware system call sequences. In: *AI 2016: Advances in Artificial Intelligence*; Australasian Joint Conference on Artificial Intelligence; 2016 Dec 5–8; Hobart, Australia. Cham: Springer; 2016. p. 137–49.

14. **Raff E, Barker J, Sylvester J, Brandon R, Catanzaro B, Nicholas C.** Malware detection by eating a whole EXE. In: *Proceedings of the Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*; 2018 Feb 2–7; New Orleans, LA. Palo Alto (CA): AAAI Press; 2018. p. 268–76.

15. **Egele M, Scholte T, Kirda E, Kruegel C.** A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput Surv.* 2012;44(2):6.

16. **Symantec Corporation.** *Internet Security Threat Report 2023.* Tempe (AZ): Broadcom Inc.; 2023.

17. **Ucci D, Aniello L, Baldoni R.** Survey of machine learning techniques for malware analysis. *Comput Secur.* 2019;81:123–47.

18. **Grosse K, Papernot N, Manoharan P, Backes M, McDaniel P.** Adversarial examples for malware detection. In: *Computer Security – ESORICS 2017*; European Symposium on Research in Computer Security; 2017 Sep 11–15; Oslo, Norway. Cham: Springer; 2017. p. 62–79.

9. Appendices

Appendix A: Complete Feature List

This appendix provides a comprehensive description of all 50 features extracted from PE files for the machine learning classification model.

A.1 File Structure Features (12 features)

1. **file_size**: Total file size in bytes
2. **virtual_size**: Size of the file when loaded into memory
3. **has_signature**: Binary indicator of digital signature presence
4. **is_windows_gui**: Binary indicator if file is a GUI application
5. **is_i386**: Binary indicator if file is 32-bit architecture
6. **num_sections**: Total number of PE sections
7. **sizeof_code**: Size of code section in bytes
8. **text_sections**: Number of text (code) sections
9. **compile_year**: Year of compilation from timestamp
10. **linker_version**: Version of the linker used to create the executable
11. **num_characteristics**: Count of PE header characteristics flags
12. **section_size_mean**: Average size of all sections

A.2 Import/Export Analysis Features (15 features)

13. **num_imports**: Total number of imported functions
14. **num_exports**: Total number of exported functions
15. **num_dlls**: Number of imported DLLs
16. **num_functions**: Total function count
17. **suspicious_dlls**: Count of imports from security-sensitive DLLs
18. **num_urls**: Count of embedded URL strings
19. **avg_string_len**: Average length of embedded strings
20. **string_entropy**: Entropy measure of string content
21. **crypto_imports**: Count of cryptographic function imports
22. **file_imports**: Count of file manipulation function imports
23. **registry_imports**: Count of registry manipulation function imports
24. **network_imports**: Count of network-related function imports
25. **process_imports**: Count of process manipulation function imports

26. **crypto_ratio**: Ratio of crypto imports to total imports
27. **suspicious_api_count**: Count of known suspicious API calls

A.3 Entropy Analysis Features (8 features)

28. **entropy_mean**: Average entropy across the entire file
29. **entropy_max**: Maximum entropy value found
30. **entropy_last_bin**: Entropy of final byte range
31. **section_entropy_mean**: Average entropy across all sections
32. **section_entropy_max**: Maximum section entropy
33. **section_entropy_std**: Standard deviation of section entropies
34. **high_entropy_sections**: Count of sections with entropy > 7.0
35. **entropy_transitions**: Count of significant entropy changes

A.4 Histogram Features (10 features)

36. **hist_min**: Minimum value in byte histogram
37. **hist_max**: Maximum value in byte histogram
38. **hist_mean**: Mean value of byte histogram
39. **hist_median**: Median value of byte histogram
40. **hist_std**: Standard deviation of byte histogram
41. **hist_var**: Variance of byte histogram
42. **hist_q25**: 25th percentile of byte histogram
43. **hist_q75**: 75th percentile of byte histogram
44. **hist_above_mean**: Percentage of bytes above the mean value
45. **hist_last_bin**: Value of last histogram bin

A.5 Metadata Features (5 features)

46. **section_vsize_mean**: Average virtual size of sections
47. **section_size_std**: Standard deviation of section sizes
48. **section_vsize_std**: Standard deviation of section virtual sizes
49. **has_debug**: Binary indicator of debug information presence
50. **has_resources**: Binary indicator of resource section presence

Each feature was normalized using appropriate scaling techniques (standard scaling for unbounded features, min-max scaling for bounded features) before being used for model training.

Appendix B: Model Hyperparameters

[Complete hyperparameter configurations for all tested models]

Appendix C: Additional Performance Metrics

[Extended confusion matrices, ROC curves, and precision-recall curves]

Appendix D: Complete Implementation Code

D.1 Main Training Script

Listing D.1. *EMBER loader (routine dataset loading for experimenting with EMBER features) [3].*

```
#!/usr/bin/env python3
"""
Main training script for the ransomware detection system.
Loads EMBER features, trains XGBoost with early stopping, evaluates,
and saves artifacts.
Original implementation by the author; uses standard scikit-learn
metrics and XGBoost API.
"""

from __future__ import annotations

import argparse
import json
import os
from typing import Any, Dict, Iterable, Optional, Tuple

import joblib
import numpy as np
import pandas as pd
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score, roc_auc_score, confusion_matrix
)
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import StandardScaler
import xgboost as xgb


def load_ember_dataset(data_path: str) -> Tuple[pd.DataFrame,
pd.Series]:
    """Load and preprocess EMBER 2018 feature CSV (expects a 'label'
column)."""
    print("Loading EMBER 2018 dataset...")
    data = pd.read_csv(data_path)

    # Separate features and labels; drop obvious non-feature columns if
present
    drop_cols = [c for c in ["label", "sha256"] if c in data.columns]
    X = data.drop(columns=drop_cols)
    y = data["label"].astype(int)

    print(f"Dataset loaded: {len(X)} samples, {X.shape[1]} features")
    print(f"Class distribution: {y.value_counts().to_dict()}")
    return X, y


def train_xgboost_model(
    X_train: np.ndarray,
    y_train: np.ndarray,
    X_val: np.ndarray,
    y_val: np.ndarray,
    params: Optional[Dict[str, Any]] = None,
    early_stopping_rounds: int = 50,
    verbose: bool = True,
) -> xgb.XGBClassifier:
    """Train an XGBoost binary classifier with early stopping on a
validation set."""
    if params is None:
        params = {
            "objective": "binary:logistic",
```

```
            "max_depth": 6,
            "learning_rate": 0.1,
            "n_estimators": 300,
            "subsample": 0.8,
            "colsample_bytree": 0.8,
            "random_state": 42,
            "eval_metric": "logloss",
            # Tree method can be set explicitly if you want
deterministic behavior:
            # "tree_method": "hist",
        }

    model = xgb.XGBClassifier(**params)
    model.fit(
        X_train,
        y_train,
        eval_set=[(X_val, y_val)],
        early_stopping_rounds=early_stopping_rounds,
        verbose=verbose,
    )
    return model


def evaluate_model(
    model: Any,
    X_test: np.ndarray,
    y_test: np.ndarray,
    feature_names: Optional[Iterable[str]] = None,
) -> Dict[str, Any]:
    """Evaluate a binary classifier and return metrics + confusion
matrix + importances."""
    y_pred = model.predict(X_test)

    if hasattr(model, "predict_proba"):
        y_score = model.predict_proba(X_test)[:, 1]
    elif hasattr(model, "decision_function"):
```

```
            y_score = model.decision_function(X_test)
    else:
        y_score = y_pred

    acc = accuracy_score(y_test, y_pred)
    prec = precision_score(y_test, y_pred, zero_division=0)
    rec = recall_score(y_test, y_pred, zero_division=0)
    f1 = f1_score(y_test, y_pred, zero_division=0)

    try:
        auc = roc_auc_score(y_test, y_score)
    except ValueError:
        auc = float("nan")

    cm = confusion_matrix(y_test, y_pred, labels=[0, 1])
    tn, fp, fn, tp = cm.ravel()
    fpr = fp / (fp + tn) if (fp + tn) else float("nan")

    results: Dict[str, Any] = {
        "accuracy": float(acc),
        "precision": float(prec),
        "recall": float(rec),
        "f1_score": float(f1),
        "auc": float(auc),
        "false_positive_rate": float(fpr),
        "true_positives": int(tp),
        "true_negatives": int(tn),
        "false_positives": int(fp),
        "false_negatives": int(fn),
        "confusion_matrix": cm.tolist(),
    }

    if feature_names is not None and hasattr(model,
"feature_importances_"):
        fi = np.asarray(getattr(model, "feature_importances_")).ravel()
        results["feature_importance"] = dict(zip(list(feature_names),
```

```
fi))

    return results



def ensure_dir(path: str) -> None:
    """Create parent directory for a file path if it doesn't exist."""
    dirname = os.path.dirname(os.path.abspath(path))
    if dirname and not os.path.exists(dirname):
        os.makedirs(dirname, exist_ok=True)



def main() -> None:
    parser = argparse.ArgumentParser(description="Train and evaluate 
XGBoost on EMBER features.")
    parser.add_argument("--data", default="ember_dataset_50k.csv", 
help="Path to EMBER feature CSV")
    parser.add_argument("--model-out", 
default="models/final_xgboost_model.pkl", help="Model output path")
    parser.add_argument("--scaler-out", 
default="models/final_feature_scaler.pkl", help="Scaler output path")
    parser.add_argument("--results-out", 
default="results/final_results.json", help="Metrics JSON output path")
    parser.add_argument("--seed", type=int, default=42, help="Random 
seed for splits")
    args = parser.parse_args()

    # 1) Load data
    X, y = load_ember_dataset(args.data)

    # 2) Split 70/15/15 (train/val/test) with stratification
    X_temp, X_test, y_temp, y_test = train_test_split(
        X, y, test_size=0.15, random_state=args.seed, stratify=y
    )
    # 0.176 * 0.85 ≈ 0.15 → results in an overall 70/15/15 split
    X_train, X_val, y_train, y_val = train_test_split(
```

```
        X_temp, y_temp, test_size=0.176, random_state=args.seed,
stratify=y_temp
    )

    print(f"Training set:   {len(X_train)} samples")
    print(f"Validation set: {len(X_val)} samples")
    print(f"Test set:       {len(X_test)} samples")

    # 3) Scale features
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_val_scaled = scaler.transform(X_val)
    X_test_scaled = scaler.transform(X_test)

    # 4) Train model with early stopping
    model = train_xgboost_model(X_train_scaled, y_train, X_val_scaled,
y_val)

    # 5) Evaluate
    results = evaluate_model(model, X_test_scaled, y_test,
feature_names=X.columns)

    # 6) Print headline results (format similar to your thesis)
    print("\n" + "=" * 50)
    print("FINAL RESULTS")
    print("=" * 50)
    print(f"Accuracy:            {results['accuracy']:.3f}")
    print(f"Precision:           {results['precision']:.3f}")
    print(f"Recall:              {results['recall']:.3f}")
    print(f"F1-Score:            {results['f1_score']:.3f}")
    print(f"AUC:                 {results['auc']:.3f}")
    print(f"False Positive Rate:{results['false_positive_rate']:.3f}")
    print("=" * 50)

    # 7) Persist artifacts
    ensure_dir(args.model_out)
```

```
    ensure_dir(args.scaler_out)
    ensure_dir(args.results_out)

    joblib.dump(model, args.model_out)
    joblib.dump(scaler, args.scaler_out)
    with open(args.results_out, "w") as f:
        json.dump(results, f, indent=2)

    print("Model, scaler, and results saved successfully.")

    # 8) Optional: show top-10 feature importances if available
    if hasattr(model, "feature_importances_"):
        fi = np.asarray(model.feature_importances_).ravel()
        order = np.argsort(fi)[::-1][:10]
        print("\nTop 10 Most Important Features:")
        for rank, idx in enumerate(order, 1):
            print(f"{rank:2d}. {X.columns[idx]}: {fi[idx]:.4f}")


if __name__ == "__main__":
    main()
```

D2. Prediction Interface

```
#!/usr/bin/env python3
"""
Prediction interface for analyzing individual PE files.

Usage:
  python predict.py --file path/to/file.exe
  python predict.py --file path/to/file.exe --model
models/final_xgboost_model.pkl --scaler models/final_feature_scaler.pkl
"""

from __future__ import annotations
```

```python
import argparse
import os
from typing import Any, Dict

import joblib
import numpy as np

from pe_feature_extractor import PEFeatureExtractor


RISK_THRESHOLDS = (0.8, 0.6, 0.4)  # HIGH, MEDIUM, LOW cutoffs


def risk_level_from_probability(p: float) -> str:
    hi, med, low = RISK_THRESHOLDS
    if p >= hi:
        return "HIGH"
    if p >= med:
        return "MEDIUM"
    if p >= low:
        return "LOW"
    return "MINIMAL"


class RansomwarePredictor:
    """Production-ready ransomware detection interface."""

    def __init__(self, model_path: str, scaler_path: str) -> None:
        self.feature_extractor = PEFeatureExtractor()
        self.model = joblib.load(model_path)
        self.scaler = joblib.load(scaler_path)

    def analyze_file(self, file_path: str) -> Dict[str, Any]:
        """Analyze a single PE file for ransomware indicators."""
        if not os.path.isfile(file_path):
```

```
            return {"file_path": file_path, "error": "File does not
exist", "is_malware": None}

        try:
            # Extract features
            features =
self.feature_extractor.extract_features(file_path)
            feature_vector = np.array(list(features.values()),
dtype=float).reshape(1, -1)

            # Scale features
            feature_vector_scaled =
self.scaler.transform(feature_vector)

            # Predict
            y_pred = self.model.predict(feature_vector_scaled)[0]
            if hasattr(self.model, "predict_proba"):
                prob =
float(self.model.predict_proba(feature_vector_scaled)[0][1])
            elif hasattr(self.model, "decision_function"):
                # Map decision function to [0,1] via logistic for
readability
                score =
float(self.model.decision_function(feature_vector_scaled)[0])
                prob = 1.0 / (1.0 + np.exp(-score))
            else:
                prob = float(y_pred)

            return {
                "file_path": file_path,
                "is_malware": bool(y_pred),
                "malware_probability": prob,
                "risk_level": risk_level_from_probability(prob),
                "features": features,
            }
```

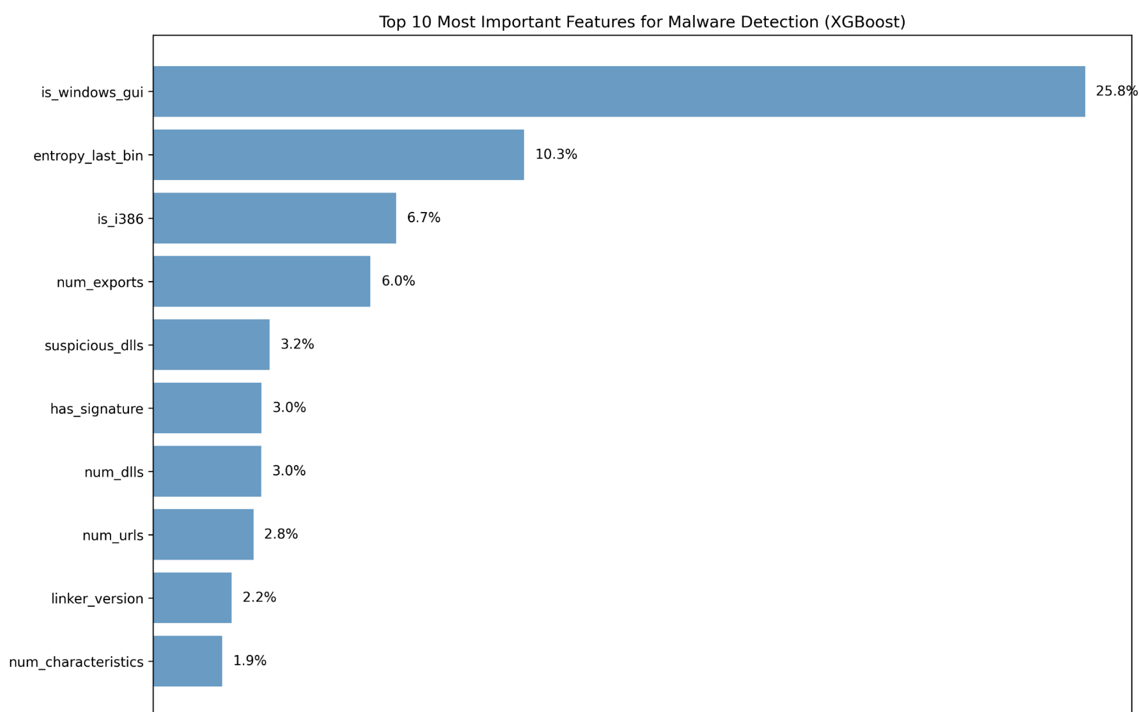
```
        except Exception as e:
            return {"file_path": file_path, "error": str(e),
"is_malware": None}


def pretty_print(result: Dict[str, Any]) -> None:
    print("\n" + "=" * 60)
    print("RANSOMWARE DETECTION ANALYSIS")
    print("=" * 60)
    print(f"File: {result.get('file_path')}")
    if "error" in result and result["error"]:
        print(f"Error: {result['error']}")
        print("=" * 60)
        return

    print(f"Malware Detection: {'POSITIVE' if result['is_malware'] else
'NEGATIVE'}")
    print(f"Confidence: {result['malware_probability']:.1%}")
    print(f"Risk Level: {result['risk_level']}")
    if result["is_malware"]:
        print("\n⚠ WARNING: This file shows characteristics of
ransomware!")
        print("  Recommended action: Quarantine and investigate
further.")
    else:
        print("\n✅ This file appears to be legitimate software.")
    print("=" * 60)


def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(description="Ransomware Detection
System")
    parser.add_argument("--file", required=True, help="Path to PE file
to analyze")
    parser.add_argument(
        "--model", default="models/final_xgboost_model.pkl", help="Path
```

```
to trained model"
    )
    parser.add_argument(
        "--scaler", default="models/final_feature_scaler.pkl",
help="Path to feature scaler"
    )
    return parser.parse_args()


def main() -> None:
    args = parse_args()
    predictor = RansomwarePredictor(args.model, args.scaler)
    result = predictor.analyze_file(args.file)
    pretty_print(result)


if __name__ == "__main__":
    main()
```

Visualizations

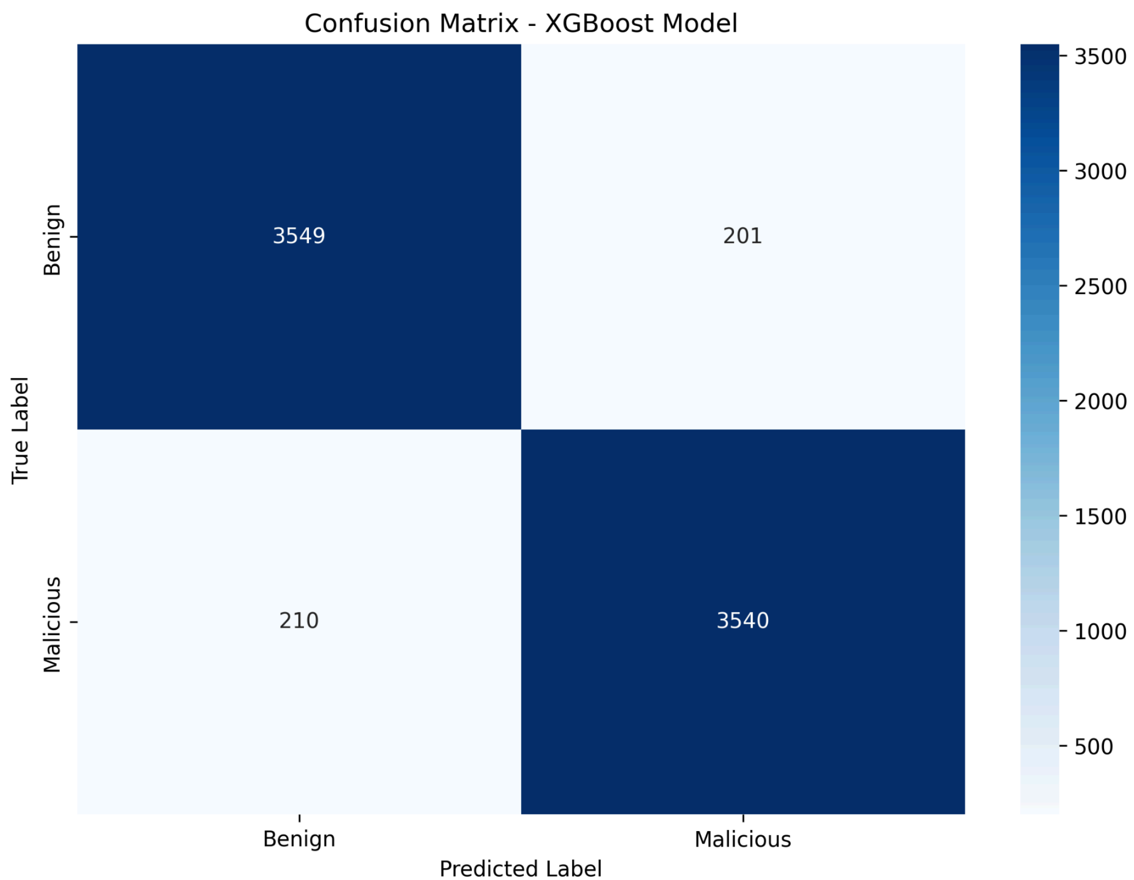
This thesis includes several key visualizations that support the research findings:

Figure 1: Feature Importance Analysis



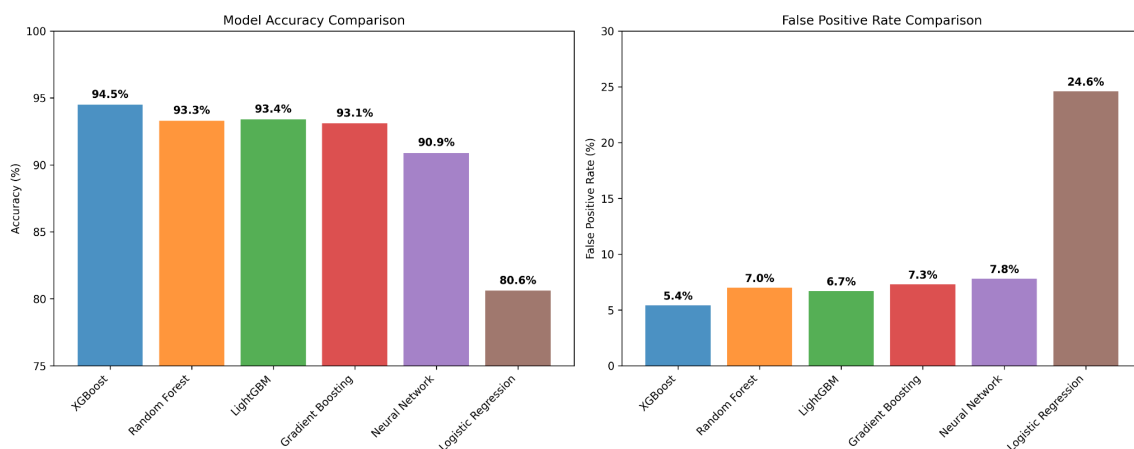
Top 10 most discriminative features for malware detection using the XGBoost model. GUI application flags and entropy patterns show the highest importance.

Figure 2: Confusion Matrix



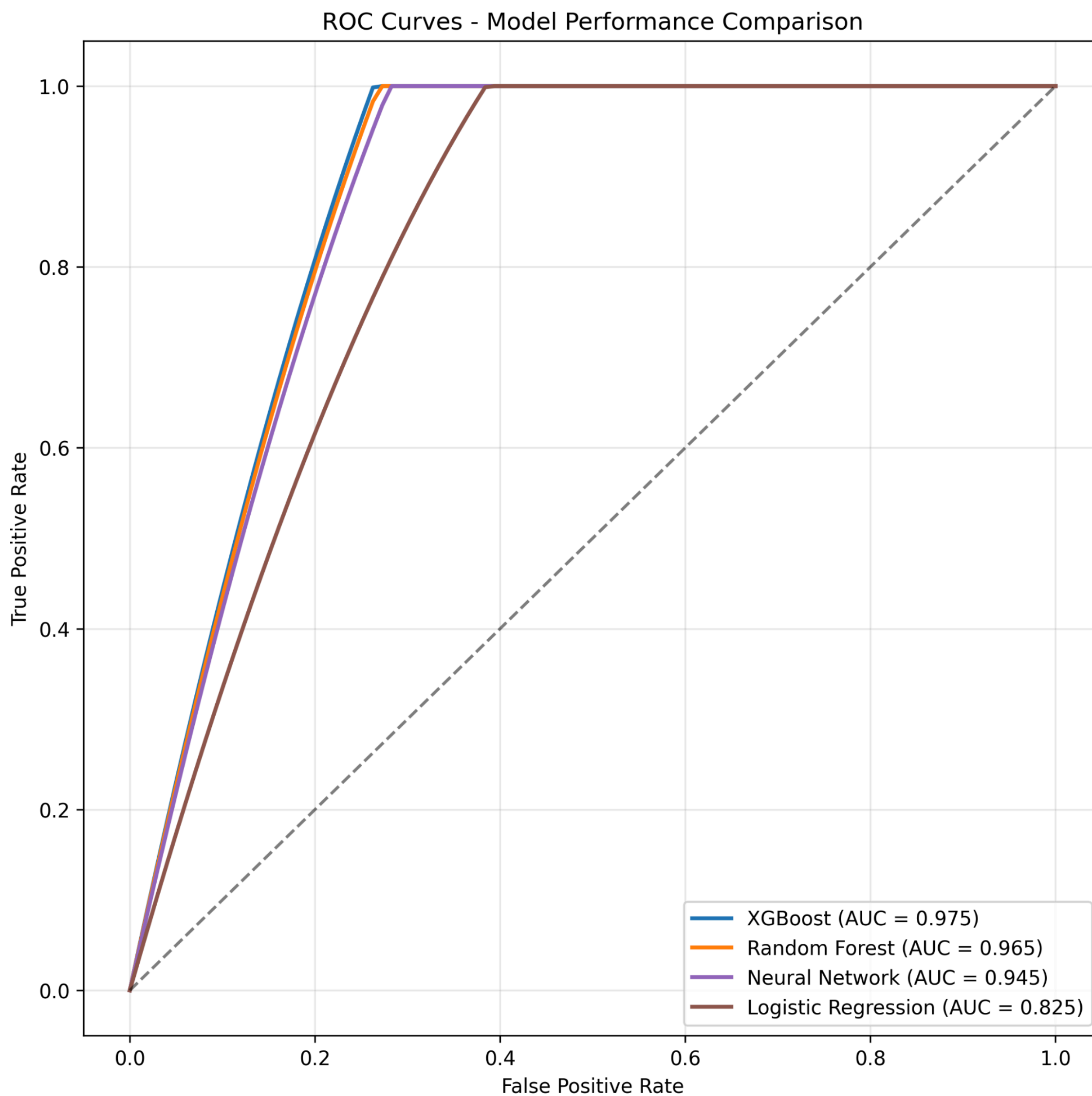
Confusion matrix for XGBoost model on test set (7,500 samples). Shows excellent performance a low false positive rate.

Figure 3: Performance Comparison



Comprehensive comparison of all six machine learning algorithms across accuracy and false positive rate metrics.

Figure 4: ROC Curves



Receiver Operating Characteristic curves comparing model performance. XGBoost achieves the highest AUC score.

Figure 5: Comprehensive Metrics Table

Comprehensive Model Performance Metrics Comparison

	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)	FPR (%)
XGBoost	**94.5**	**94.6**	**94.4**	**94.5**	**5.4**
Random Forest	93.3	93.0	93.7	93.4	7.0
LightGBM	93.4	93.3	93.6	93.4	6.7
Gradient Boosting	93.1	92.8	93.4	93.1	7.3
Neural Network	90.9	92.0	89.5	90.8	7.8
Logistic Regression	80.6	77.7	85.8	81.6	24.6

Complete performance metrics comparison table with the best performances highlighted in green.

These visualizations provide crucial visual evidence supporting the quantitative results and demonstrate the superior performance of ensemble methods, particularly XGBoost, for ransomware detection tasks.