Shared API Call Insights for Optimized Malware Detection in Portable Executable Files

Mehdi Kmiti, Jallal Eddine Moussaoui, Khalid El Gholami, Yassine Maleh
LaSTI Laboratory-National School of Applied Sciences Khouribga, Sultan Moulay Slimane University, Beni Mellal, Morocco

***Abstract*—Malware analysis is essential for understanding malicious software and developing effective detection strategies. Traditional detection methods, such as signature-based and heuristic-based approaches, often fail against evolving threats. To address this challenge, this study proposes a static analysis–based malware detection system that employs thirteen classifiers, including Logistic Regression, K-Nearest Neighbors (KNN), Support Vector Machine (SVM), Naive Bayes, Decision Tree, Linear Discriminant Analysis (LDA), Quadratic Discriminant Analysis (QDA), Random Forest, Extra Trees, Gradient Boosting, AdaBoost, and LightGBM. The framework is built on a balanced dataset of 1,318 Windows Portable Executable (PE) files (674 malware, 644 benign), where the features are derived from shared API calls between benign and malicious files to ensure relevance and reduce redundancy. Experimental results show that the Extra Trees classifier achieved the highest accuracy of 98.14%, highlighting its effectiveness in detecting malware. Overall, this study provides a robust, data-driven approach that enhances static malware detection and contributes to strengthening cybersecurity against emerging threats.**

***Keywords*—Malware detection; static analysis; portable executable (PE) files; API calls; extra trees classifier**

I. Introduction

Malware, short for "malicious software", represents a major threat to digital systems, ranging from financial losses and privacy breaches to operational disruptions in critical infrastructure [1]–[4]. Modern malware is increasingly sophisticated, driven by profit, espionage, and cyber warfare, and includes forms such as viruses, worms, trojans, ransomware, and spyware [5], [6]. Recent statistics highlight its scale: more than 90% of digital threats target the Windows operating system, with AV-TEST reporting 1.17 billion threats by mid-2024, while global cybercrime costs are projected to reach $10.5 trillion annually by 2025 [7], [8]. These figures underscore the urgent need for effective malware detection strategies.

Traditional methods, such as signature-based and heuristic-based detection, struggle to keep pace with polymorphic and zero-day malware [9]–[11]. As a result, static analysis, which inspects executables without execution, has emerged as a promising alternative, particularly when combined with machine learning for classification.

Research Gap: Despite promising results, many prior static analysis methods rely on large and redundant feature sets, which increase computational overhead and reduce interpretability. Deep and ensemble learning models achieve high accuracy, but often lack scalability in real-world or resource-constrained environments. There is thus a need for lightweight and interpretable approaches that preserve strong detection performance while reducing redundancy.

To address this gap, we propose a malware detection framework based on shared API calls, which represent the intersection of calls found in both benign and malicious Portable Executable (PE) files. This refined feature space highlights the most relevant behaviors, reduces dimensionality, and supports efficient classification. Using a balanced dataset of 1,318 PE files (674 malware, 644 benign), we evaluate thirteen classifiers and demonstrate that the Extra Trees ensemble achieves the highest accuracy of 98.14%. The contributions of this work lie in constructing a refined dataset of shared API calls that enhances interpretability and efficiency, conducting a comparative evaluation of multiple classifiers, and showing that Extra Trees provides the best trade-off between accuracy and computational cost, making it suitable for practical deployment.

The remainder of this study is organized as follows: Section II reviews related work, covering metadata-based techniques, ensemble and hybrid approaches, and recent deep learning and graph-/transformer-based solutions, followed by the positioning of this study. Section III describes the proposed methodology, including dataset construction, API call extraction, shared-feature representation, preprocessing, the employed machine learning models, training and evaluation protocols, and the computing environment. Section IV presents the experimental results and provides a comprehensive discussion, including a comparative assessment against state-of-the-art (SOTA) approaches, limitations of the proposed method, practical applicability, and future research directions. Finally, Section V concludes the study by summarizing the key findings and outlining directions for future research.

II. Related Work

Research on static malware detection has explored a wide range of features and machine learning models. Existing approaches can be grouped into three main categories: metadata-based methods, ensemble and hybrid models, and deep learning methods, including recent transformer and graph-based solutions.

A. PE Header and Metadata Features

Balram et al. [12] evaluated six classifiers, including Support Vector Machine, Logistic Regression, Random Forest, and XGBoost, on string-based and PE header features. Their findings showed that string features, when combined with XGBoost and hybrid models, provided superior accuracy.

Similarly, Baldangombo et al. [13] analyzed PE headers, DLL names, and API calls, reporting that PE header features alone could achieve up to 99% accuracy with the J48 classifier while maintaining low computational overhead. These studies highlight the effectiveness of handcrafted metadata features but also underscore their limitations in scalability and adaptability to evolving threats.

B. Ensemble and Hybrid Models

Azeez et al. [14] proposed an ensemble framework that integrated fully connected dense Artificial Neural Networks (ANNs) and 1D-CNNs, with Extra Trees serving as a final-stage classifier. Their approach outperformed conventional classifiers in accuracy and robustness. Shijo and Salim [15] combined static and dynamic analysis, using Printable Strings Information (PSI) along with API call sequences extracted from the Cuckoo Sandbox, and achieved 98.7% accuracy with Support Vector Machine (SVM) and Random Forest. Yousuf et al. [16] further improved performance by integrating multiple feature sets, including DLL names, API calls, PE headers, and section attributes, and applying ensemble techniques such as Majority Voting, Stack Generalization, and AdaBoost, achieving 99.5% accuracy. These works demonstrate the benefits of combining diverse features and models, though often at the cost of increased complexity.

C. Deep Learning and Graph-/Transformer-Based Methods

Deep learning techniques have also been widely adopted in malware detection. Bensaoud and Kalita [17] proposed a CNN–LSTM model trained on 8-gram sequences, achieving an SOTA accuracy of 99.91%. More recent studies (2021 to 2025) have introduced advanced architectures based on transformers and graph neural networks (GNNs). Trizna et al. [18] applied a Transformer model with self-attention to dynamic API-call behaviors, improving detection accuracy while enhancing interpretability through attention visualization. Similarly, Seneviratne et al. [19] leveraged Vision Transformers in a self-supervised setup on image-based malware representations, achieving around 97% accuracy.

In the graph-based domain, GNNs are increasingly used to capture structural relationships in executables. Mohammadiana et al. [20] combined graph reduction techniques with GNNExplainer to enable efficient and interpretable malware detection on function and control-flow graphs. Shokouhinejad et al. [21] proposed a stacked GNN framework where multiple base learners are trained on PE control-flow graphs and integrated via an attention-based meta-learner, significantly improving both classification accuracy and explainability. Additionally, Tang [22] introduced a hybrid GNN-based model that fuses static and dynamic features, reporting a 4 to 7% performance gain over existing baselines on the EMBER and VirusShare datasets.

These works demonstrate the potential of deep learning, transformers, and GNNs for achieving high detection accuracy while also improving interpretability. However, their computational requirements are often high, making them less practical for deployment in resource-constrained environments.

D. Positioning of this Work

While prior studies demonstrate that traditional machine learning, ensembles, and deep learning can achieve high malware detection accuracy, they often rely on large or redundant feature sets or demand high computational resources. In contrast, the proposed framework focuses on a lightweight subset of shared API calls between benign and malicious executables. This design reduces redundancy, improves interpretability, and provides a practical balance between accuracy and efficiency, making it suitable for real-world deployment in static malware detection.

III. Methodology

This section details the data sources and selection criteria, the API-call extraction pipeline, the construction of the shared-call feature space, the preprocessing steps, the learning models and hyperparameters, and the training and evaluation protocol used in this study. The objective is to provide a complete and reproducible description of the static-analysis workflow. Table I presents the dataset overview.

A. Data Sources and Selection Criteria

The corpus comprises 1,318 Windows PE files: 674 malware samples obtained from VirusShare (2012 to 2024) and 644 benign executables collected from the C:\Windows\System32 directory of a Windows 10 installation. Duplicates at the file and feature-vector levels were removed. To limit class bias and facilitate clear performance estimation, the dataset was balanced via random undersampling of the majority class, yielding 714 unique samples (357 benign, 357 malware). The final feature matrix contains 559 columns: 558 binary API-call indicators and one label column. Rationale: a balanced setting supports robust classifier comparison without prevalence-driven bias, while focusing on static features enables scalable, execution-free analysis [23]-[27].

TABLE I Dataset Overview

_c_exit	SetLastError	VariantUnit	CopyFileW	…	Class
1	0	0	1	…	1
0	0	1	0	…	1
1	1	0	1	…	0
0	1	1	1	…	0
0	0	1	0	…	1

B. API-Call Extraction Pipeline

API calls were obtained through a two-stage static workflow:

1) Import table parsing using pefile to enumerate imported functions and their DLLs; normalization unified function aliases (e.g., CopyFileA/W) and handled forwarded exports.

2) Disassembly using the Capstone framework to scan code sections for call instructions and resolve additional API references not declared in the import table (e.g., late binding, indirect calls). Extracted names were canonicalized to DLL.Function form (e.g., KERNEL32.CopyFileW), lower-cased, and filtered to exclude non-WinAPI stubs and compiler intrinsics. The output of this stage is a per-file multiset of API

call identifiers [28]-[35]. Fig. 1 shows the steps of data collection process.

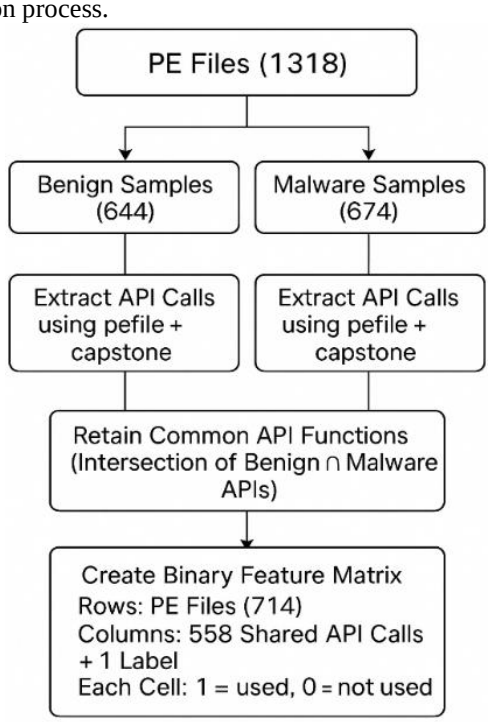

Fig. 1. Data collection process steps.

C. Shared-Call Feature Construction

Let B and M denote the sets of unique API calls observed in benign and malware subsets, respectively. The shared feature set is defined as:

$$S = B \cap M$$

Each file i is represented by a binary vector $x_i \in \{0,1\}^{|S|}$, where $x_{i,j} = 1$ if API $s_j \in S$ appears in file i, and 0 otherwise. Stacking all N files yields $X \in \{0,1\}^{N \times |S|}$. Limiting features to S reduces redundancy and emphasizes behaviorally relevant calls common to both classes.

Pseudo-code (shared-call construction)

1) Extract API sets A_f for each file f
2) Compute $B = \bigcup_{f \in \text{benign}} A_f$, $M = \bigcup_{f \in \text{malware}} A_f$
3) Set $S = B \cap M$
4) For each file f, form $x_f[j] = 1\{s_j \in A_f\}$ for all $s_j \in S$
5) Concatenate all x_f to obtain X.

a) Practical advantage: This representation filters out API calls unique to one class, thereby reducing dimensionality, mitigating dataset sparsity, and emphasizing behaviorally significant features. Compared with using the full API space, this method enhances computational efficiency, supports faster training, and highlights generalizable patterns for malware detection.

D. Preprocessing

Duplicate vectors were removed to ensure one instance per unique API profile. Missing indicators were treated as absence (0). The dataset was then shuffled and split into 70% training / 30% testing with stratification by class. No numerical scaling was applied to binary indicators. Where appropriate (e.g., SVM, LR), class weights were set to balanced [36]-[42]. Fig. 2 presents the data preprocessing steps.

E. Machine Learning Models and Parameters

To evaluate the effectiveness of shared-API-call-based static malware detection, a diverse set of classification algorithms was implemented, covering traditional learners, ensemble methods, and neural networks. Each classifier was trained on the binary feature vectors derived from Section III-C, with performance assessed via 5-fold stratified cross-validation and a 30% independent test set. Hyperparameters were selected based on widely adopted defaults or light grid search to balance performance and computational efficiency.

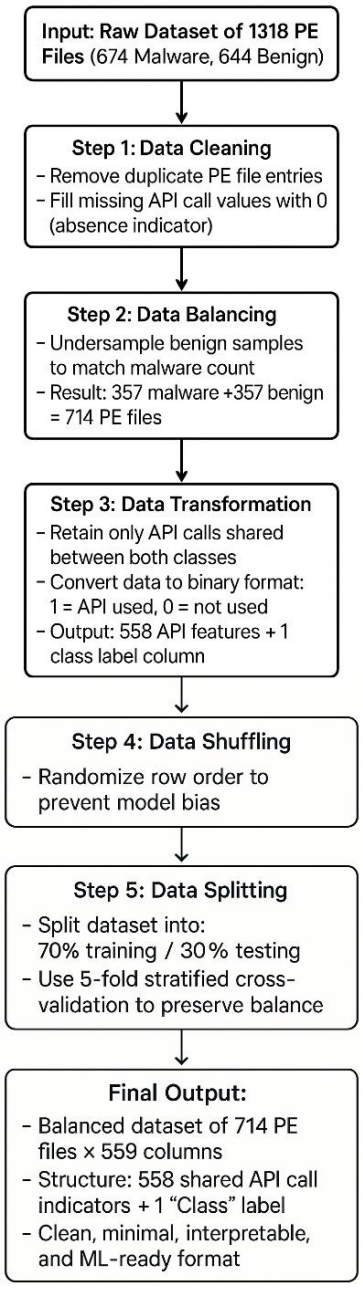

Fig. 2. Data preprocessing steps.

The classifiers are grouped into three categories: traditional baselines, ensemble learners, and deep learning models. A summary of libraries and configurations are presented in Table II to Table IV.

1) Traditional machine learning models: These models serve as interpretable baselines, providing insight into linear separability and low-complexity decision boundaries.

TABLE II TRADITIONAL MACHINE LEARNING MODELS

Model	Library	Hyperparameters / Notes	Practical Rationale
Logistic Regression	scikit-learn	solver='liblinear', penalty='l2', C=1.0, random_state=42	Robust linear baseline; efficient on sparse binary features
K-Nearest Neighbors (KNN)	scikit-learn	n_neighbors=5, weights='uniform', metric='minkowski'	Captures local neighborhoods; sensitive to feature sparsity
SVM	scikit-learn	C=1.0, kernel='rbf', gamma='scale'	Strong nonlinear decision boundary; benchmark for high-dimensional data
Naive Bayes	scikit-learn	Default parameters	Probabilistic, fast baseline for binary data
Desicion Tree	scikit-learn	criterion='gini', random_state=42	Provides interpretability; prone to overfitting
Linear Discriminant Analysis (LDA)	scikit-learn	Default parameters	Tests linear discriminants; assumes Gaussian feature distribution
Quadratic Discriminant Analysis (QDA)	scikit-learn	Default parameters	Quadratic boundary; serves as generative baseline

2) Ensemble learning models: Ensembles aggregate multiple weak learners to improve generalization. These models are particularly suited for binary, high-dimensional spaces, like API call vectors.

TABLE III ENSEMBLE LEARNING MODELS

Model	Library	Hyperparameters / Notes	Practical Rationale
Random Forest	scikit-learn	n_estimators=100, random_state=42	Reduces variance via bagging; robust baseline
Extra Trees	scikit-learn	n_estimators=100, random_state=42	Increased randomization; faster training
Gradient Boosting	scikit-learn	n_estimators=100, learning_rate=0.1, random_state=42	Strong baseline for structured data; balances bias/variance
AdaBoost	scikit-learn	n_estimators=50, learning_rate=1.0, random_state=42	Emphasizes difficult samples; reduces bias
LightGBM	LightGBM	Default parameters	Gradient boosting optimized for large sparse binary features

3) Deep learning models: Deep learning models are increasingly applied in malware detection due to their ability to capture complex, nonlinear relationships among features. In this study, a feedforward ANN was implemented as a representative deep learning baseline. The ANN directly ingests binary API-call feature vectors, enabling it to learn higher-order dependencies beyond the capacity of linear or tree-based classifiers.

TABLE IV DEEP LEARNING MODELS

Model	Library	Hyperparameters / Notes	Practical Rationale
ANN	Keras / TensorFlow	• Input: 559 nodes (one per API call) • Hidden layers: [256, 128], activation='relu' • Dropout: 0.5 • Output: 1 node, activation='sigmoid' • Optimizer: Adam • Loss: Binary Crossentropy • Epochs: 50 • Batch size: 32	Baseline neural network for binary feature vectors

F. Training and Evaluation Protocol

The dataset was partitioned into 70% training and 30% testing with stratified sampling to preserve class balance. Model selection and stability were assessed using 5-fold stratified cross-validation on the training portion; final results were reported on the independent test set. Unless otherwise stated, a fixed threshold of 0.5 on the predicted probability/score was used to compute threshold-dependent metrics (OA, TNR, FPR). AUC was computed from the ROC curve and is threshold-independent. All experiments used identical splits and fixed random seeds for reproducibility.

Metrics. Four complementary measures were reported:

a) Overall Accuracy (OA): proportion of correctly classified samples.

b) True Negative Rate (TNR): proportion of benign files correctly identified as benign.

c) False Positive Rate (FPR): proportion of benign files misclassified as malware.

d) Area Under the ROC Curve (AUC): probability that a randomly chosen malware sample receives a higher score than a randomly chosen benign sample; operationally computed from the ROC via the trapezoidal rule.

All equations are referenced:

$$OA = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}} \quad (1)$$

$$TNR = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FP}} \quad (2)$$

$$FPR = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}} = 1 - TNR \quad (3)$$

$$AUC = \int_0^1 TPR(\tau) \times \mathrm{dFPR}(\tau)\ , With\ TPR = \frac{\mathrm{TP}(\tau)}{\mathrm{TP}(\tau) + \mathrm{FN}(\tau)} \quad (4)$$

G. Reproducibility and Computing Environment

The experiments were conducted on a workstation equipped with an Intel Core i5-12450H CPU (8 cores, up to 4.4

GHz), 16 GB DDR4 RAM, an NVIDIA RTX 2050 GPU (4 GB), and a 512 GB SSD, running Windows 11.

The malware detection pipeline was implemented in Python 3.9. Key libraries and frameworks included:

pefile for parsing PE headers and extracting imported functions.

Capstone disassembly engine for recovering additional API call references from executable code sections.

pandas for data handling and feature structuring.

scikit-learn for implementing classical machine learning classifiers and evaluation procedures.

TensorFlow/Keras for training and evaluating the ANN.

All random seeds were fixed across experiments to ensure reproducibility. Identical training or test splits were maintained for all models to guarantee fair comparison.

IV. Results and Discussion

A. Results

Thirteen classifiers were evaluated to identify the most effective algorithms for malware detection using shared API-call features. Table V summarizes the performance across four key metrics: Overall Accuracy (1), True Negative Rate (2), False Positive Rate (3), and Area Under the ROC Curve (4) [43]-[46].

The Extra Trees classifier achieved the highest overall accuracy (OA = 98.14%) and a strong AUC of 98.19%, confirming its suitability for handling high-dimensional, sparse binary feature vectors. The ANN achieved the highest AUC (99.50%), reflecting excellent discriminatory power between benign and malicious samples, although its OA was slightly lower at 97.67%.

Other strong performers included Logistic Regression (OA = 97.67%, AUC = 97.75%) and AdaBoost (OA = 97.67%, AUC = 97.69%). Gradient Boosting also performed competitively (OA = 97.21%, AUC = 97.31%), while Random Forest and LightGBM both reached OA = 96.74%. Traditional classifiers such as Naïve Bayes, LDA, and QDA underperformed, with OA below 85%, highlighting their limitations in modeling nonlinear feature interactions.

Notably, the Decision Tree classifier achieved an FPR of 0% (3), although its OA and AUC (95.35% and 95.61%) were lower than ensemble-based methods.

Table V shows the performance comparison of classifiers using FPR (3), TNR (2), OA (1), and AUC (4).

TABLE V Performance Comparison of Detectors

Algorithm	FPR (%)	TNR (%)	AUC (%)	OA (%)
Naïve Bayes	3.25	13.49	83.82	83.26
Logistic Regression	0.46	1.86	97.75	97.67
LDA	20	17.67	62.05	62.33
SVM	0.93	2.79	96.38	96.28
QDA	19.06	14.42	66.11	66.51
KNN	0.93	4.18	95.06	94.88
Decision Trees	0	4.65	95.61	95.35
ANN	0.93	1.39	99.50	97.67
Random Forest	0.46	2.79	96.87	96.74
Extra Trees	0.46	1.39	98.19	98.14
AdaBoost	0.93	1.39	97.69	97.67
Gradient Boosting	0.46	2.32	97.31	97.21
LightGBM	0.46	2.79	96.87	96.74

To evaluate the contribution of the proposed approach relative to prior studies, Table VI presents a comparative analysis. It contrasts the performance of the shared API-call-based Extra Trees classifier results with traditional string- and header-based approaches, as well as recent SOTA methods such as deep ensembles, hybrid static–dynamic frameworks, and attention-based models.

TABLE VI Comparative Analysis of the Proposed Method and Existing Malware Detection Approaches

Study	Features used	Model / Architecture	OA (%)	Interpretability	Complexity	Notes
Proposed Method (Extra Trees)	Shared API call vectors	Extra Trees Classifier	98.14	High	Low	Lightweight and fast; effective for sparse binary features
Balram, et al. [12]	String	Hybrid (LR/XGB)	98	Moderate	High	Highest accuracy; longer execution time
Baldangombo et al. [13]	PE Headers	J48	99	High	Moderate	Best standalone feature set; uses PCA
Azeez et al. [14]	PE Headers	Ensemble (7 NNs + ExtraTrees)	100	Low	Very High	Best performance; requires stacked ensemble training
Shijo and Salim [15]	Integrated (PSI + API 3/4-grams)	SVM	98.7	Low	Very High	Best accuracy; combines static/dynamic strengths
Yousuf et al. [16]	PE Header + PE Section (Integrated)	Random Forest, Stack Gen	99.5	Medium	Moderate-High	Uses IG/PCA for feature selection; combines static structural features.
Bensaoud and Kalita [17]	API Calls + Opcodes (8-grams)	CNN-LSTM-3	99.91	Medium	High	Combines CNN for spatial features and LSTM for sequential patterns.
Trizna et al. [18]	Dynamic API call traces	Self-Attention (Nebula)	99.8	Medium	High	SOTA dynamic analysis; resource-intensive
Seneviratne et al. [19]	Malware images (transformed binaries)	Vision Transformer (Self-Supervised)	97	Low	Very High	Requires GPU-heavy training; less interpretable

Mohammadian et al. [20]	Graph-based API dependency features	Graph Reduction + Explainable Learning	95.4	High	High	Improves explainability; complex preprocessing
Shokouhinejad et al. [21]	Graph-based features	Explainable Ensemble Learning	86.14	High	High	Balances interpretability with high accuracy
J. Tang [22]	Static + Dynamic fusion	Hybrid ML Ensemble	> 95	Medium	Very High	Strong accuracy but requires both runtime and static environments

The Receiver Operating Characteristic (ROC) curve for the Extra Trees classifier is shown in Fig. 3, illustrating its balance of high OA and AUC with low FPR.

Fig. 3 presents the ROC curve of the Extra Trees classifier, evaluated with OA (1), AUC (4), TNR (2), and FPR (3).

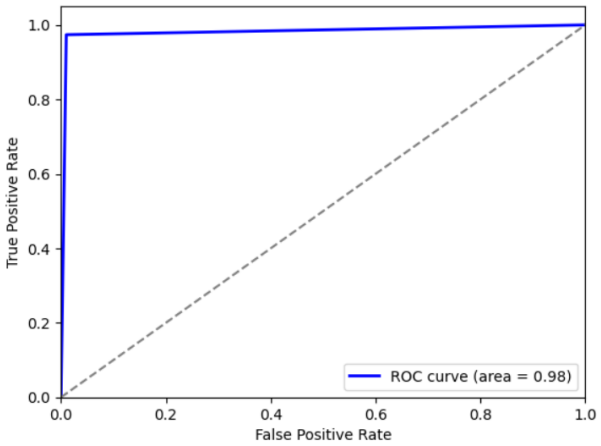

Fig. 3. ROC curve using extra trees classifier.

B. Discussion

The results confirm the effectiveness of tree-based ensemble models for static malware detection in sparse, binary feature spaces. The Extra Trees classifier offers the best balance between accuracy and interpretability, while the ANN model demonstrates superior AUC performance, suggesting that deep learning techniques can better capture subtle feature interactions.

In comparison with existing studies, the proposed framework offers several practical advantages, notably in terms of lightweight architecture, reduced feature redundancy, and improved computational efficiency. For instance, Azeez et al. [14] employed a complex ensemble of seven neural networks combined with Extra Trees to attain near-perfect detection accuracy, resulting in significant training overhead and deployment challenges. In contrast, the present approach circumvents such complexity, enhancing scalability and ease of integration. Furthermore, while Shijo and Salim [15] relied on sandboxed environments to extract dynamic behavioral features, the proposed static-only methodology eliminates the need for runtime execution, thereby simplifying implementation while maintaining competitive detection performance.

Balram et al. [12] and Baldangombo et al. [13] reported strong results using string- and header-based features but did not emphasize interpretability or feature reduction. Yousuf et al. [16] achieved an impressive 99.5% accuracy through feature fusion and advanced ensemble techniques; however, this came at the cost of increased system complexity.

More recent advances, such as self-attention models [18], [19] and explainable graph-based frameworks [20], [21], achieve high accuracy and improved interpretability but require significant computational resources, making them less suitable for lightweight or real-time deployments. Hybrid feature fusion approaches, such as that proposed by Tang [22], demonstrate the value of combining static and dynamic features but introduce additional integration challenges.

Overall, the proposed method achieves high accuracy (98.14% overall accuracy) and excellent discrimination (99.50% AUC) while using a compact and interpretable feature set of 558 shared API calls.

LIMITATIONS

As with most static approaches, the method may be less effective against heavily obfuscated or polymorphic malware and could face challenges related to concept drift, where emerging malware families exhibit behaviors not represented in the training data. Additionally, the moderate dataset size may limit generalization to large-scale, real-world scenarios.

PRACTICAL APPLICABILITY

Despite these limitations, the proposed framework is fast, lightweight, and interpretable, making it suitable for endpoint security systems and resource-constrained environments, where dynamic analysis is infeasible.

FUTURE WORK

Future research should focus on extending the analysis to larger and more imbalanced datasets would provide stronger evidence of generalizability. Moreover, integrating hybrid static–dynamic features could enhance resilience against obfuscation and adversarial evasion. Finally, exploring advanced architectures, such as CNN-LSTM hybrids or transformer-based models, may offer additional improvements in robustness and scalability.

V. CONCLUSION

This study presented a lightweight and interpretable malware detection framework that leverages shared API call behaviors extracted from benign and malicious PE files. Thirteen classifiers were systematically evaluated on a balanced dataset of 1,318 samples, demonstrating that high detection accuracy can be achieved with carefully selected static features. Among the evaluated models, the Extra Trees classifier achieved the best overall accuracy (98.14%), while maintaining high interpretability and computational efficiency. The ANN reached the highest AUC (99.50%), highlighting the ability of shallow deep learning models to capture nonlinear feature interactions. Overall, tree-based ensemble methods proved most effective for high-dimensional binary API-call features, offering a strong balance between detection power, efficiency, and interpretability.

In comparison with recent SOTA approaches such as hybrid CNN-LSTM models, graph-based learning, and transformer-based architectures, the proposed framework provides a more practical and resource-efficient solution. While those models often achieve higher accuracy on larger datasets, they typically introduce greater computational overhead and reduced interpretability. In contrast, the presented framework achieves competitive results while remaining lightweight, making it suitable for deployment in resource-constrained or real-time environments.

Despite these strengths, the study has several limitations. The exclusive reliance on static analysis may reduce robustness against highly obfuscated or adversarial malware, while the use of a moderately sized, balanced dataset limits generalizability to real-world malware distributions, which are often large-scale and highly imbalanced. Issues such as concept drift, adversarial evasion, and scalability in high-volume environments remain open challenges.

Future work should extend this research by validating the framework on larger and more diverse datasets, integrating dynamic and hybrid features, and investigating advanced ensemble or deep learning architectures such as CNN-LSTMs and transformer-based models. Additional directions include enhancing resilience against adversarial evasion techniques, addressing concept drift in evolving threat landscapes, and optimizing scalability for high-throughput detection scenarios. These improvements would strengthen both the theoretical and practical impact of the proposed approach, ensuring its continued relevance in modern malware detection.

REFERENCES

[1] Symantec Corporation, “Internet Security Threat Report,” Rep. 22, Mountain View, CA, USA, Apr. 2017. [Online]. Available: https://www.symantec.com/security-center

[2] S. Anderson, “Understanding the financial impact of cyber-attacks on businesses,” *J. Inf. Secur. Appl.*, vol. 40, pp. 133–142, Mar. 2018, doi: 10.1016/j.jisa.2018.02.009.

[3] W. Stallings and L. Brown, *Computer Security: Principles and Practice*, 4th ed. Boston, MA, USA: Pearson, 2018. [Online]. Available: https://www.pearsonhighered.com/assets/preface/0/1/3/2/0132775069.pdf

[4] M. Ahmad, “A study on ransomware attacks and their impact on the cybersecurity landscape,” *Int. J. Adv. Res. Comput. Sci.*, vol. 10, no. 4, pp. 285–292, Dec. 2019, doi: 10.26483/ijarcs.v10i4.6477.

[5] D. Harley and A. Lee, *Viruses Revealed*. New York, NY, USA: Osborne/McGraw-Hill, 2001. [Online]. Available: https://archive.org/details/virusesrevealed00harl

[6] R. A. Grimes, *Hacking the Hacker: Learn from the Experts Who Take Down Hackers*. Indianapolis, IN, USA: Wiley, 2017.

[7] IBM Security, *Cost of a Data Breach Report*, Armonk, NY, USA: IBM, 2024. [Online]. Available: https://www.ibm.com/reports/data-breach

[8] AV-TEST Institute, *Malware Statistics*, Magdeburg, Germany, 2024. [Online]. Available: https://www.av-test.org/en/statistics/malware/

[9] W. Tang *et al*., “Hybrid analysis framework for large-scale malware detection using machine learning,” *Comput. Secur.*, vol. 98, p. 102014, May 2020, doi: 10.1016/j.cose.2020.102014.

[10] A. Aghaei-Foroushani and M. Zincir-Heywood, “A comprehensive study of evasive malware techniques and their implications for current detection tools,” *Comput. Commun.*, vol. 145, pp. 241–260, Dec. 2019, doi: 10.1016/j.comcom.2019.06.014.

[11] S. Krishnan, Y. C. Tian, J. Sun, and H. Y. Liu, “Heuristic-based detection of code obfuscation malware,” *IEEE Access*, vol. 8, pp. 7198–7207, Dec. 2020, doi: 10.1109/ACCESS.2020.2972442.

[12] N. Balram, G. Hsieh, and C. McFall, “Static malware analysis using machine learning algorithms on APT1 dataset with string and PE header features,” in *Proc. Int. Conf. Comput. Sci. Comput. Intell. (CSCI)*, Las Vegas, NV, USA, Dec. 2019, pp. 90–95, doi: 10.1109/CSCI49370.2019.00022.

[13] U. Baldangombo, N. Jambaljav, and S.-J. Horng, “A static malware detection system using data mining methods,” *arXiv*, arXiv:1308.2831, Aug. 2013. [Online]. Available: http://arxiv.org/abs/1308.2831

[14] N. A. Azeez, O. E. Odufuwa, S. Misra, J. Oluranti, and R. Damasevicius, “Windows PE malware detection using ensemble learning,” *Appl. Sci.*, vol. 11, no. 4, Art. no. 1477, 2021, doi: 10.3390/app11041477.

[15] P. V. Shijo and A. Salim, “Integrated static and dynamic analysis for malware detection,” *Procedia Comput. Sci.*, vol. 46, pp. 804–811, 2015, doi: 10.1016/j.procs.2015.02.149.

[16] M. I. Yousuf, I. Anwer, A. Riasat, K. T. Zia, and S. Kim, “Windows malware detection based on static analysis with multiple features,” *PeerJ Comput. Sci.*, vol. 9, p. e1319, Apr. 2023, doi: 10.7717/peerj-cs.1319.

[17] A. Bensaoud and J. Kalita, “CNN-LSTM and transfer learning models for malware classification based on opcodes and API calls,” *Knowl.-Based Syst.*, vol. 290, p. 111543, Apr. 2024, doi: 10.1016/j.knosys.2024.111543.

[18] D. Trizna, “Nebula: Self-attention for dynamic malware analysis,” arXiv, vol. abs/2310.10664, Oct. 2023. [Online]. Available: https://arxiv.org/abs/2310.10664

[19] S. Seneviratne, R. Shariffdeen, S. Rasnayaka, and N. Kasthuriarachchi, “Self-supervised vision transformers for malware detection,” arXiv, vol. abs/2208.07049, Aug. 2022. [Online]. Available: https://arxiv.org/abs/2208.07049

[20] H. Mohammadian, G. Higgins, S. Ansong, R. Razavi-Far, and A. A. Ghorbani, “Explainable malware detection through integrated graph reduction and learning techniques,” arXiv, vol. abs/2412.03634, Dec. 2024. [Online]. Available: https://arxiv.org/abs/2412.03634

[21] H. Shokouhinejad, R. Razavi-Far, G. Higgins, and A. A. Ghorbani, “Explainable ensemble learning for graph-based malware detection,” arXiv, vol. abs/2508.09801, Aug. 2025. [Online]. Available: https://arxiv.org/abs/2508.09801

[22] J. Tang, “Fusion of static and dynamic features for malware detection,” Proc. ACE Conf., pp. 1–8, Jul. 2025. [Online]. Available: https://direct.ewa.pub/proceedings/ace/article/view/24689

[23] P. Dong, Z. Wang, and J. Cheng, “A new static malware detection method using features of PE headers,” *J. Comput. Syst. Sci.*, vol. 89, pp. 449–464, May 2017, doi: 10.1016/j.jcss.2017.05.001.

[24] F. Schuster *et al*., “Malware Zoo: A dataset of executable and online malware samples for academic and industrial research,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, pp. 781–796, 2017, doi: 10.1145/3133956.3134073.

[25] VirusShare, “Comprehensive Repository of Malware Samples.” [Online]. Available: https://virusshare.com

[26] F. Rafique and M. K. Luhach, “PE file format features for malware detection: A systematic review,” *IEEE Access*, vol. 10, pp. 3035–3044, Jan. 2022, doi: 10.1109/ACCESS.2022.3140032.

[27] A. Mosse, J. Howard, and G. Hill, “Analysis of API calls in portable executable (PE) files for threat analysis,” in *Proc. ACM SIGSAC Conf. Artif. Intell. Cybersecurity*, pp. 55–64, 2019.

[28] E. Andriesse, X. Fratantonio, A. Sasse, and C. Kruegel, “Practical binary code analysis using Capstone,” in *Proc. IEEE Secur. Privacy Workshops (SPW)*, pp. 1–7, 2018, doi: 10.1109/SPW.2018.00010.

[29] V. Talib and S. Girkar, “Detection of malware through API call sequence and classification techniques,” *J. Netw. Comput. Appl.*, vol. 50, pp. 129–143, Jun. 2018, doi: 10.1016/j.jnca.2017.03.015.

[30] G. D'Angelo and R. Di Pietro, "Unsupervised detection of malware via dynamic and static disassembly analysis," in *Proc. IEEE TrustCom*, pp. 91–99, Aug. 2016, doi: 10.1109/TrustCom.2016.016.

[31] Capstone, "Capstone: Lightweight Multi-Architecture Disassembly Framework." [Online]. Available: https://www.capstone-engine.org

[32] M. Saud, U. Mahbub, and A. Anwar, "Analysis of portable executable (PE) file metadata for malware classification using static analysis," in *Proc. ACM SIGSAC Conf. Artif. Intell. Cybersecurity*, pp. 104–113, 2019, doi: 10.1145/3313407.3314009.

[33] S. Kariyappa and H. Ambre, "Static malware analysis with PE parsing: Exploring Python libraries like pefile for threat modeling," *Int. J. Adv. Comput. Sci. Appl. (IJACSA)*, vol. 10, no. 4, pp. 45–54, Apr. 2019, doi: 10.14569/IJACSA.2019.010042.

[34] P. Sun and Y. Wang, "Binary malware classification using supervised machine learning and API feature analysis," in *Proc. ACM Int. Conf. Inf. Knowl. Manage. (CIKM)*, pp. 1555–1562, 2017, doi: 10.1145/3132847.3133071.

[35] F. Santos, C. G. Comin, and A. Capurro, "Static analysis of Windows malware using Capstone disassembly for API call extraction," *IEEE Access*, vol. 8, pp. 6504–6517, Jan. 2020, doi: 10.1109/ACCESS.2019.2963872.

[36] W. Tang, S. Yan, and Z. Yan, "Hybrid data preprocessing for robust malware detection using API features," *IEEE Access*, vol. 9, pp. 34177–34187, Mar. 2021, doi: 10.1109/ACCESS.2021.3060497.

[37] D. Borkin, A. Némethová, G. Michaľčonok, and K. Maiorov, "Impact of data normalization on classification model accuracy," Res. Pap. Fac. Mater. Sci. Technol. Slovak Univ. Technol., vol. 27, no. 45, pp. 79–84, Sep. 2019, doi: 10.2478/rput-2019-0029.

[38] D. Mohaisen and O. Alrawi, "AMAL: High-level comprehensive malware analysis and data preprocessing framework," in *Proc. ACM SIGSAC Symp. Inf., Comput. Commun. Secur. (Asia CCS)*, pp. 101–111, Jun. 2014.

[39] P. C. Zhu and Y. X. Zhou, "Practical data augmentation and cleaning for malware analysis," *J. Comput. Theor. Nanosci.*, vol. 14, no. 1, pp. 134–141, Jan. 2017, doi: 10.1166/jctn.2017.6861.

[40] D. Kong and G. Yan, "Discerning malware program structure through API sequence similarity," in *Proc. IEEE Secur. Privacy Symp.*, pp. 512–522, May 2013, doi: 10.1109/SP.2013.40.

[41] N. Japkowicz and M. Shah, *Evaluating Learning Algorithms: A Classification Perspective*. Cambridge, U.K.: Cambridge Univ. Press, 2011.

[42] F. Schuster and G. Jacob, "Machine learning-based data cleaning in malware detection pipelines," *Comput. Secur.*, vol. 70, pp. 79–87, Sep. 2017, doi: 10.1016/j.cose.2017.06.009.

[43] P. Vemuri, B. Mukherjee, and M. M. S. Poonkodi, "Comparative analysis of machine learning classifiers for malware detection," *IEEE Trans. Emerg. Topics Comput.*, vol. 5, no. 4, pp. 476–489, Dec. 2017, doi: 10.1109/TETC.2016.2602621.

[44] H. Zhang, W. Dai, and Y. Xu, "Malware detection using random forest and decision tree classifiers," *Comput. Secur.*, vol. 60, pp. 23–31, Oct. 2016, doi: 10.1016/j.cose.2016.04.004.

[45] J. Brownlee, *Master Machine Learning Algorithms: Discover How They Work and Implement Them from Scratch*, 2nd ed. Victoria, Australia: Machine Learning Mastery, 2019.

[46] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.*, pp. 785–794, Aug. 2016, doi: 10.1145/2939672.2939785.