

Dragonfly: A Lightweight Offline Antivirus System for Executable Malware Detection Using Machine Learning

HeisnamBidyalyuxmiDevi¹, Sarungbam Usharani Devi², ThongamBobo³, UshamSanjotaChanu⁴

^{1,2,3,4} NIELIT Deemed to be University, Imphal
heisnambidyalyuxmidevi@gmail.com

Abstract. Antivirus products often depend on signature-based and heuristic-based methods, which are often unable to manage the ever-increasing malware variants. Dragonfly is a Windows-focused, fully offline antivirus system designed for practical end-users without network connectivity. The detection models are implemented in Python and operate with an embedded inference pipeline for deploying the models with an Electron-based inference and a file watcher for real-time scanning on end-user machines, it also utilizes a custom static feature engineering and vectorization pipeline based on raw Portable Executable (PE) measurements. The representation includes byte-level distributions, entropy structure, string semantics, PE structural features, and hashed API interaction patterns. The detection models are trained on the EMBER 2027 benchmark and evaluated on the EMBER 2028 dataset, which allows generalization under concept drift. We evaluate Logistic Regression, Support Vector Machines, Random Forests, xgboost, and lightgbm using cross-validation and temporal testing. While Gradient boosting methods achieve the strongest overall detection performance, scalability and deployment constraints at the EMBER scale motivate the selection of lightgbm as a balanced trade-off between accuracy, inference latency, and resource usage, with an accuracy of 89.93% and an f1-score of 89.48%. Inference cost, memory consumption, and confusion matrix analysis also support this selection. Model inference is integrated through an embedded Python managed by a native C++ helper, enabling low-latency inter-process communication with the user interface. Experimental results show that Dragonfly delivers a reliable static malware antivirus that can achieve practical throughput through careful feature engineering, principled evaluation, and modular system design.

Keywords: Endpoint security, Feature engineering, LightGBM, Malware detection, Machine learning, Offline antivirus, Portable Executable (PE), Static analysis, System integration, Temporal evaluation.

(Article history: Selected papers from NIELIT's International Conference on Communication, Electronics and Digital Technologies (NICE-DT'26), 29-30 Jan, 2026, Gorakhpur, India)

1 Introduction

The quick changes in malware behavior make it more dangerous for computer users all over the world. This puts data privacy, system integrity and overall Cybersecurity at a lot of risk [27]. Traditional antivirus software primarily uses a database of predefined signatures and heuristic rules to recognize known malware patterns or behavior [2]. It was successful against known threats, but it has trouble identifying sophisticated malware that is constantly evolving, which makes it less dependable in sophisticated cyberattacks [2][6].

By enabling a detection system that can learn intricate patterns that differentiate between malicious and benign software, machine learning has emerged as a promising method to address these issues [16][27]. Through the analysis of characteristics like hashed API interaction patterns, entropy structure, and byte-level distributions. A machine learning-based antivirus program can detect valuable hidden malware [27]. However, the majority of current research on machine learning-based malware detection concentrates on model deployment [22, 25], which is frequently server or cloud-based. The end user's practicality is limited by this reliance on central architecture, which raises privacy concerns, latency issues, and primarily depends on network availability [5][6].

Dragonfly was created as a cross-language, fully deployable malware detection platform with an emphasis on practical usability in order to close the gap in the current research. Without requiring extensive sandbox execution, it incorporates adaptive features inspired by dynamic behavior understanding while retaining the efficiency of static analysis [11]. Additionally, it was sufficiently light to be integrated into the local system without requiring server and network connectivity, filling a significant gap in previous research [12].

The feature extraction and vectorization pipeline, the model training and evaluation framework, and the cross-platform user interface with a native inference engine are the three main parts of Dragonfly [11]. All these components allow for the functionality of offline malware detection, which is applicable in real-world scenarios. Dragonfly utilizes its own static feature representation, which is based on raw measurements of the Portable Executable (PE) file. This representation focuses on different aspects of the binary, such as statistics, entropy, strings, structural PE, and API interactions. Different machine learning models are evaluated using stratified cross-validation and temporal testing. The performance of the models is evaluated using various

metrics accuracy, precision, recall, f1 score, roc_auc and based on balanced trade-off between detection effectiveness and deployment feasibility, rather than optimizing accuracy alone [16][26][27].

For the deployment, model inference is carried out using an embedded Python runtime, which is handled by a native C++ helper process. It communicates with the Electron-based user interface using inter-process communication. Multithreaded scanning and asynchronous execution enable file analysis and user interaction to occur concurrently.

Major contribution and novelty of the paper are listed below:

- A Scalable Static Feature Engineering and Vectorization Pipeline for Windows Malware Detection.
- A well-founded model selection strategy with detection performance and deployment constraints.
- A modular offline antivirus architecture that combines native inference, asynchronous scanning, and a user-friendly interface.

The remainder of this paper was organized as follows. Section 2 review related work on machine learning based malware detection using various techniques. Section 3 represent the overall system of Dragonfly architecture including data preprocessing, model selection, model reimplementation, IPC, multithreading background watcher and architectural claim. Section 4 provides experimental evaluation and discussion. Finally conclude in Section 5.

2 Related work

The main direction of malware detection research more focused on machine learning and deep learning due to their strong ability to recognize hidden and complex malicious behavior patterns that traditional signature based systems fail to detect [23][24]. The advantages of this approach allows them to identify new malware variants by learning structural or behavioral similarities, which makes them more adaptive than rule based detection systems [17].

Static Analysis Approach

Many earlier approaches rely on static analysis, where the executable was examined without running it. In this approach static features such as byte n-grams, PE headers, opcodes, and printable strings are used [1].

Schultz et al. [1] demonstrated that static feature based learning could detect unseen malware effectively. In similar fashion, Vinod et al. [4] combined both PE header and opcode features, achieving an accuracy near 97%. Bai et al. [7] applied feature selection techniques like CfsSubsetEval and WrapperSubsetEval to reduce dimensionality to improve model efficiency, claiming 99.1% accuracy with Random Forest. Belaoued and Mazouzi [13] built a real-time detection system based on PE Optional Header (OH) interpretation, achieving 97% accuracy using the Rotation Forest model [14]. These works give the strong foundation of the capability of static analysis, but the issue most of them face was obfuscated malicious files (malware that hides its true malicious code through packing, encryption, or code reordering which often bypasses static models) [5].

Dynamic Analysis Approaches

Rather than analysing only the data which are available without executing dynamic analysis methods execute the file into a sandbox environment like Cuckoo Sandbox and observe program behavior [3]. API call sequences, system calls, and resource usage are among the most used features [22]. Salehi et al. [14] proposed MAAR, a scalable method that categorizes API call arguments and achieved over 99% accuracy. Qiao et al. [5] built an automated framework that converts API call reports into sequential data, showing high precision with reduced computational cost. Some advanced malware even detects the sandbox and behaves differently to hide its malicious behavior, which reduces detection reliability [28][8]. Moreover, it was resource-heavy and not practical for large scale deployment [18].

Hybrid Approaches

Both static and dynamic approaches has their limitations so hybrid approaches that combine both was used to overcome this. Shijo and Salim [9] used printable string features with API call sequences and achieved 98.7% detection accuracy. Islam et al. [15] applied hybrid features such as function length frequency and API parameters, which improve overall classification results. These approaches are robust, yet they still remain research oriented due to the fact that they are not directly deployable for real world cross platform and highly dependent on specific dataset or operating system.

Advanced Machine and Deep Learning Approaches

Recently due to advancement of ensemble learning and deep learning, this approach improve generalization [19].

Azeez et al. [19] used ensemble tree models (Extra Trees) on Windows PE datasets, achieved 99% accuracy. Chen et al. [20] introduced a boosting-based malware family classifier using multiple tree models and feature fusion. These studies highlight the potential of deep models for automatic feature learning instead of making and extracting complex features manually. However, large labelled dataset, high computational power, and long training times, making them hard to apply on limited resource environments or cross platform systems [21].

From the literature, it was clear that each approach static, dynamic, or hybrid has its own strength and weakness. Static analysis was lightweight but weak against obfuscation, dynamic analysis was more realistic but time consuming, hybrid systems improve accuracy but often remain confined to single platforms and deep learning approach demand high volume of data-set [27][19][34].

Unlike most prior works that discuss above, Dragonfly emphasizes practical implementation, cross compatibility, and scalability providing a bridge between academic malware detection research and its real world deployment potential.

3 Proposed Framework: Dragonfly

3.1 System Overview

The proposed system named Dragonfly was designed as an offline and privacy-preserving machine learning based antivirus system that can be directly deployed on end-user machines. Its objective was to integrate machine-learning models and lightweight security tools. The overall system architecture follows a systematic pipeline as shown in Fig. 1. In which executable files are processed through feature extraction, preprocessing and vectorization, followed by various model training and evaluation. And the selected model was served by an embedded python environment as a detection engine. C++ helper is responsible to communicate between the detection engine, Electron-based UI and background watcher service through an inter-process communication (IPC) layer.

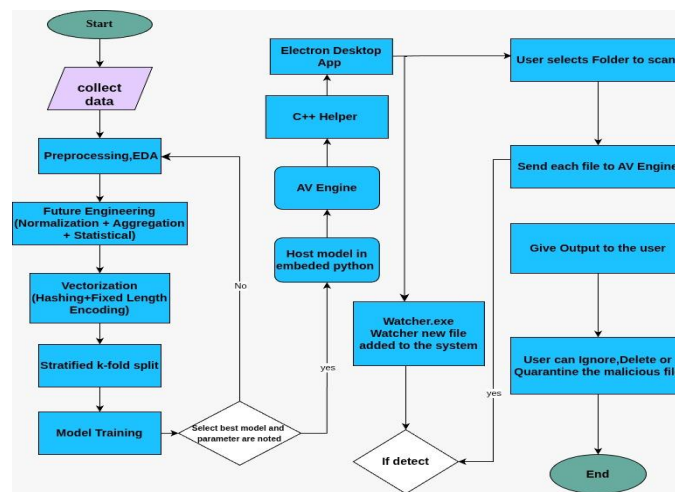


Fig. 1. Flow chart for Dragonfly's overall system architecture.

3.2 Dataset and Preprocessing

To ensure experimental validity and improved generalization, Dragonfly is evaluated using the EMBER dataset [29], a large-scale benchmark for static Windows malware detection. EMBER provides raw Portable Executable (PE) measurements extracted from real benign and malicious binaries.

In this work, samples from the EMBER 2017 release are used for model training, while evaluation is performed on EMBER 2018 dataset. The assessment of robustness against concept drift where malware characteristics change over time rather than being randomly mixed between training and testing sets is made possible by this temporal split, which more accurately reflects realistic deployment scenarios.

In Place of the original EMBER feature vectors directly [29], we design a custom vectorization pipeline that reorganizes and aggregates the raw PE measurements into a structured representation suitable for efficient deployment. Each input executable is transformed into a fixed-length numeric vector capturing complementary byte-level statistics, entropy patterns, string characteristics, structural PE properties, and imported API interactions. The resulting feature vectors are stored in memory-mapped format to support large-scale training and low-overhead access during experimentation. Table 1 show the final preprocess dataset description.

The final processed EMBER 2017 dataset includes 490,000 samples with a near- balanced class distribution, providing a realistic and sufficiently large foundation for evaluating both detection performance and system-level efficiency.

Dataset	EMBER 2018 (process)
Total Sample	490,000

Feature dimensionality	1,149
Benign sample	250,000
Malware sample	240,000
Benign ratio	51.02%
Malware ratio	48.98%
Feature storage format	Memory-mapped Num-Py arrays (.data)
X storage size	2.10 GB
y storage size	478.52 KB
Total storage size	2.10 GB

Table 1. Preprocess dataset description

3.3 Feature Engineering and Vectorization

Dragonfly uses a custom feature engineering and vectorization strategy built on top of raw static PE measurements. This design focus on two main goal: (i) to capture complementary structural and semantic characteristics of Windows executables (ii) to produce a fixed-length efficient representation for large scale training and low latency inference.

The raw descriptors from PE parsing are heterogeneous in nature, spanning byte-level statistics, textual characteristic, structural metadata, and imported API information. These descriptors are therefore reorganized and transformed into a unified numerical vector through normalization, aggregation, and sketching operations. The final feature vector contains 1,149 dimensions which includes the following feature groups.

Byte Distribution Features.

Normalized byte histograms characterize the global distribution of the executable and are complemented by simple statistical summaries to distinguish control, printable, and high-byte regions. These features reveal code density, instruction distribution patterns, and packing behavior while remaining invariant to file.

Entropy Structure Features.

Byte entropy histograms summarize local randomness patterns within the binary. Aggregate statistics over the entropy distribution highlight the presence of compressed or encrypted regions commonly associated with obfuscation.

String-Based Semantic Features.

String metadata is represented through normalized ratios derived from string counts, lengths, entropy, and the occurrences of paths, URLs, and registry references. This helps capture semantic characteristics without being dominated by file size.

PE Structural Features.

Structural characteristics of the executable are summarized using size ratios, section-level entropy statistics, and selected header flags, which provide a compact description of layout irregularities and execution-related properties.

Import Interaction Sketch.

Imported libraries and functions are represented by a hashing, based interaction sketch that combines the two and maps the result to a fixed, size vector with logarithmic scaling. This helps to retain the semantics of behavior while at the same time handling the issue of feature space dimensionality and sparsity.

These feature groups provide a multi-level static representation without dynamic analysis.

3.4 Model Selection and Training Strategy

Dragonfly evaluates multiple machine learning models that are commonly used for static malware detection such as LightGBM, XGBoost, Support Vector Machines, Random Forests and Logistic Regression. The objective is not to create a new learning algorithm, but to identify a model that performs well in detection while being practical for large-scale training and low-latency deployment. Stratified cross-validation is used on the training data, and performance is reported using precision, recall, accuracy, ROC-AUC and F1-score.

Tree-based gradient boosting methods provide the strongest detection performance on the proposed feature representation. However, differences arise when considering scalability and resource requirements. In particular, training XGBoost on the complete set of 1,149 dimensional feature vectors at EMBER scale results in prohibitive memory consumption. To enable a fair comparison, XGBoost is therefore evaluated using a PCA-reduced feature, which significantly lowers memory usage yet preserving variance.

Under this setting, LightGBM delivers performance that is comparable to or better than XGBoost on PCA-transformed features, while also demonstrating faster convergence and lower inference overhead. In contrast to XGBoost, LightGBM is able to train directly on the full feature representation without dimensionality reduction which improves suitable for large-scale static analysis.

Random Forests provides competitive accuracy but require higher memory overhead during inference, whereas linear models such as Logistic Regression and SVM produce fast predictions at the cost of low accuracy, reduced recall on complex sam-

ples. These findings indicate that LightGBM offers the best trade-off between system integration limitations, scalability, and detection effectiveness. As a result, LightGBM is a practical option for deployment in an offline protection system.

3.5 Inference engine and Native Integration

Dragonfly performs model inference using an embedded Python runtime to maintain numerical consistency with the trained models and to avoid duplication of learning logic across languages. The inference process is isolated from the user-facing application through a lightweight native integration layer.

A helper program made with C++ to control the Python interpreter that is built into it. This helper program also helps the inference engine and the Electron-based interface talk to each other.

Named pipes is used to communicate which lets them send and receive information, like file path and prediction results quickly. Communication is implemented using named pipes, allowing asynchronous and low-latency exchange of file-path and inference result without relying on network socket. This way the Dragonfly system can use native components and a flexible inference backend and keep everything organized and separate. The system is designed to make it easy to add things and keep the various parts working well each other. While Python is currently handling the actual inference computation, obtaining that through a native C++ bridge gives migration path to future optimization, including partial or full replacement of the Python runtime if required.

3.6 Watcher, Multi-Threading, and User Interface

Dragonfly has a separate watcher process watching the file system and does injects events to the backend for detection .It queues detection events to a thread pool, where concurrent scanning and classification along with maintaining efficient resource utilization.The Interface communicates to the backend via named pipes, that enabled real-time display of scan output and further quarantine management, all of these make the framework a fully interactive desktop security application.

4 Experimental Evaluation and Discussion

4.1 Experimental Setup

The experimental setup and evaluation procedure are covered in this section. All the tests here are meant to check if Dragonfly's main design ideas really hold. Experiments were conducted on a machine running Windows 11 with an Intel Core i5 CPU and 16 GB RAM .The software stack included Python, scikit-learn, XGBoost, LightGBM, and C++ for helper executable.

4.2 Model Performance Evaluation

This section reports quantitative detection results obtained using the experimental setup described in Section 4.1. Table 2 presents cross-validation results for all evaluated models on the training data, including accuracy, precision, recall, F1-score, and ROC-AUC. Tree-based ensemble methods consistently outperform linear baselines across metrics, with gradient boosting approaches achieving thestrongest overall performance. Table 3 reports final test results on EMBER 2018, highlighting generalization behavior under temporal shift. LightGBM model achieved strongprecision-recall balance and low false-positive rates under temporal evaluation. Fig.2 shows the corresponding confusion matrix, Fig.3 presents the ROC curve and Fig.4 shows the precision recall curve for the final model.

Model	ROC-AUC (mean $\pm$ std)	F1-score (mean $\pm$ std)	Recall (mean $\pm$ std)
LightGBM	0.9994 $\pm$ 0.0000	0.9898 $\pm$ 0.0001	0.9883 $\pm$ 0.0003
Logistic Regression	0.5732 $\pm$ 0.0134	0.2026 $\pm$ 0.0570	0.1202 $\pm$ 0.0434
SVM	0.8685 $\pm$ 0.0286	0.8040 $\pm$ 0.0359	0.8090 $\pm$ 0.0394
Random Forest	0.9970 $\pm$ 0.0001	0.9718 $\pm$ 0.0003	0.9654 $\pm$ 0.0011
XGBoost	0.5346 $\pm$ 0.0012	0.5318 $\pm$ 0.0031	0.5539 $\pm$ 0.0062

Table 2.Cross-validation results for the evaluated model.

Model	Accuracy	Precision	Recall	F1-Score	ROC-AUC
LightGBM	0.8993	0.9366	0.856	0.8948	0.9701
Logistic Regression	0.5137	0.6260	0.0673	0.1216	0.4110
SVM	0.7269	0.6586	0.9418	0.7751	0.8159
Random Forest	0.8029	0.9015	0.6800	0.7752	0.9320
XGBoost	0.8882	0.9321	0.8374	0.8822	0.9660

Table 3.Final test result

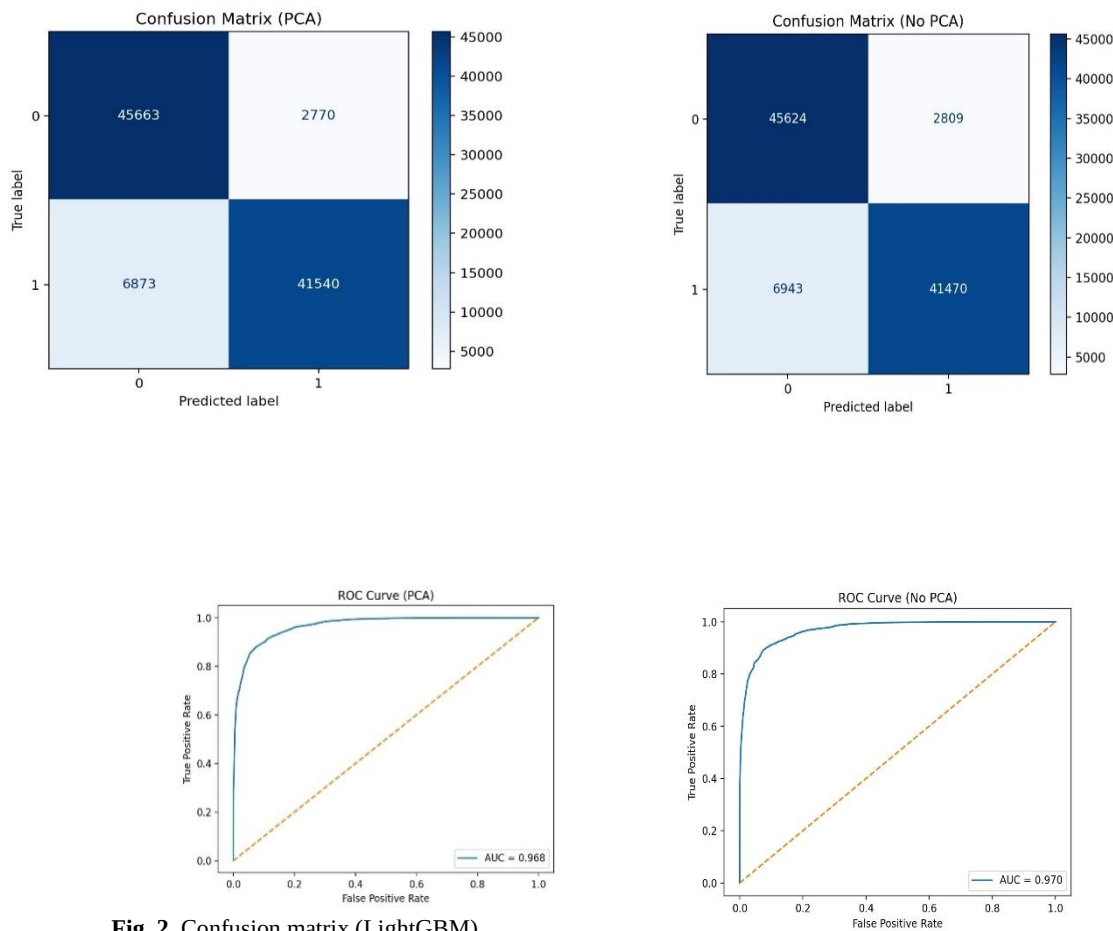
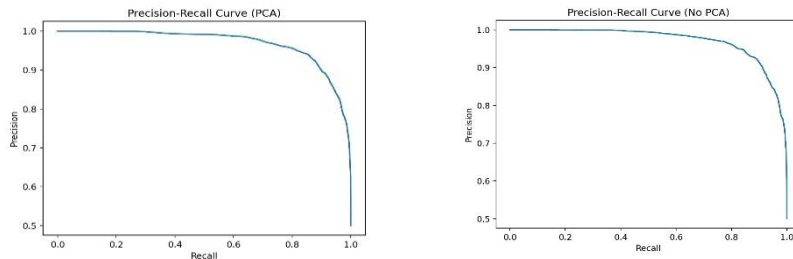


Fig. 2. Confusion matrix (LightGBM)

Fig. 3. ROC AUC

Fig. 4. Precision-Recall curve

4.3 Runtime and Resource Analysis

Inference latency and memory usage are measured for deployment feasibility across models. Table 4 summarizes inference cost and memory usage for all evaluated models. All measurements are performed under identical conditions on 1000 sample.

Linear models have the lowest inference cost and memory use, but achieve weaker detection performance.

Random Forests lead to higher memory usage and higher latency than the other models.

LightGBM achieve a good balance between detection performance and runtime efficiency. The deployment oriented model selection based on computational efficiency in addition to detection performance.

Model	Mean Inference (ms)	Std Inference (ms)	Max Inference (ms)	Load Time (ms)	Peak Memory (MB)
LightGBM	0.1421	0.1667	3.8252	28.06	145.72
LR	0.0739	0.0071	0.1225	15.34	143.88
SVM	0.0827	0.0279	0.4249	35.03	143.63
RF	41.7711	6.4088	120.9054	382.89	403.06
XGBoost	0.5657	1.1652	23.8548	521.73	299.54

Table 4.Inference cost and memory usage

4.4 Robustness and Limitations

Dragonfly depends solely on static analysis and thus does not engage in dynamic execution or behavioral analysis. As such, it is not capable of offering full protection against an advanced packing, encryption, or obfuscation technique that aims to hide malicious activities until runtime.

However, despite these shortcomings, the proposed feature representation does exhibit a few characteristics that are suggestive of obfuscation and relate to the nature of abnormal entropy distributions, compressed section patterns, and unusual structural layouts. Empirical studies have shown that malware with high levels of entropy is identified with greater recall than malware with low entropy, although the former is still not as easily detectable as the latter, and this indicates a degree of robustness against generally packed malware.

However, malware that is highly obfuscated or multi-layer packed is still difficult to detect using completely static analysis tools. Dragonfly is thus best suited as a first-line defense tool that can filter out a large number of malicious samples before more expensive dynamic or behavioral analysis is conducted. The rationale for this is more a reflection of deployment considerations than a lack of robustness against adversaries.

4.5 System Integration and Interface Demonstration

Dragonfly incorporate static malware detection into a desktop app that monitor whatever files or folders user choose. It runs the analysis in the background asynchronously. Whenever a file event pops up, the feature extraction and inference is quietly done, then pipes the results straight to the userinterface in realtime. Fig. 4 shows snapshots from the deployed working interface. Built on Electron, the UI lets user to perform quick scans, custom scan or full scan, and it tracks live progress and displayed total scanned files, estimated time of completion (ETA). Using encryption logic, quarantine functionality is used to safely encrypt and isolate the detected malicious files. It also offers a backup option for their recovery or permanent removal if needed.

As seen in Fig.5, the Dragonfly architecture was stress-tested on enormous directory trees with a ton of nested folders to really push it. Even then without UI freeze, and the live counters for file progress and detection logs updated smoothly in real time. This proved the robustness and usability of Dragonfly's multithreaded design in real workloads also.

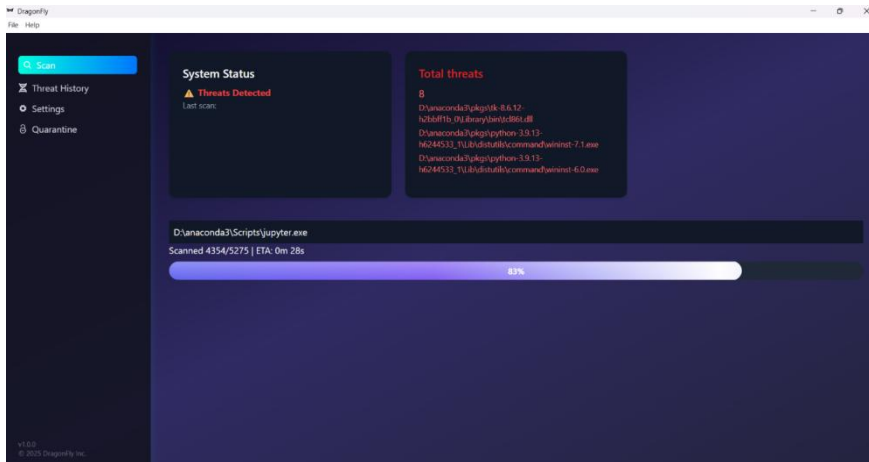


Fig. 5.The Dragonfly interface displays the total number of scanned files, the scanning progress, and the estimated time of arrival.

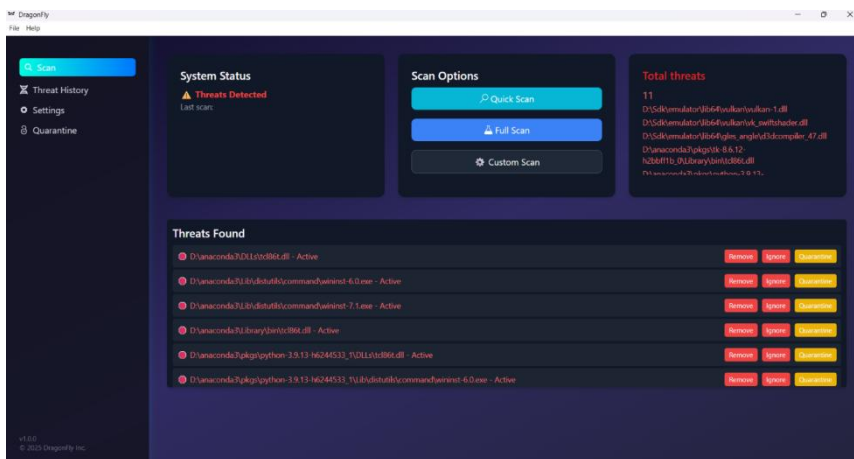


Fig. 6.The Dragonfly interface scanned the directory and found big bad files and quarantine management.

4.6 Discussion

The results show that the proposed Dragonfly framework is able to effectively perform static malware detection, yet it is appropriate for endpoint devices. In the proposed framework, the use of the large scale benchmark with temporal validation ensures that the generalization ability is measured realistically and the proposed feature representation ensures that the results are stable for different types of models.

The comparison of models indicates that there is a trade-off between performance and deployment constraints.

The gradient boosting methods perform well overall.

LightGBM strikes a good balance between practicality and competitive performance with lower time and resource overhead, which is critical for continuous offline analysis.

The system-level evaluation also verifies that the detection pipeline can be integrated into a responsive desktop application. Asynchronous execution and multithreaded design enable stable execution with realistic workloads without compromising user interaction. The static analysis-based approach employed by Dragonfly means that it cannot deal with sophisticated packing and runtime-based obfuscations. Thebest approach to using Dragonfly is as a first-line, lightweight defense that filters malicious files before subjecting them to more expensive dynamic analysis.Futureresearch can focus on using a hybrid approach and optimizations using native inferences.

5 Conclusion

This paper introduced Dragonfly, a fully offline based malware detection system, which is designed to be deployed on end-points. Our aim was to move static machine learning-based detection out of the lab and into the field without compromising user privacy.

In a large-scale benchmark with temporal evaluation, the effectiveness of well-designed static features has been demonstrated, which can generalize well while being efficient. LightGBM has been demonstrated as a viable deployment model. Dragonfly has a modular structure for its models, which use an embedded Python runtime for inference, assisted by a lightweight C++ helper that interacts with the Electron-based interface. This is to keep the system responsive during long scans while also enabling optimization in the future without affecting the system as a whole.

Even though the system is static-based and does not cover all types of obfuscation, Dragonfly acts as a lightweight first-line defense that closes the gap between malware detection research and actual deployable endpoint security software.

Authors' Contribution Declaration:

HeisnamBidyalyuxmi Devi: Responsible for ensuring the successful completion and coherence of all components.

UshamSanjota Chanu: Original draft writing and conceptualization; Contributing significantly to the building of the core AI framework to the design of our model.

Sarungbam Usharani Devi: System architecture; Helped design the complete system how to structure your labels, integrate knowledge, and implementation structure.

Thongam Bobo Singh: Thongam Bobo Singh: Cyber support and experimentation; managed malicious files, supplied anti-virus logic, and made sure that experimental procedures were carried out safely and validated.

References

1. J. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo, Data mining methods for detection of new malicious executables, Proc. IEEE Symp. Security and Privacy, 2001, pp. 38-49.
2. Umakant Mishra, Methods of Virus Detection and Their Limitations, SSRN, Aug. 2010. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=1916708
3. J. Vennon, Cuckoo sandbox: Automated malware analysis, Cuckoo Sandbox Project, 2013, [Online]. Available: <https://cuckoosandbox.org>
4. P. Vinod and S. V. Kumar, A static feature-based malware detection architecture using feature selection and classification, IEEE, Int. Conf. on Computer and Communication Technology, 2017, pp. 283-288.
5. M. Qiao, Y. Xu, and T. Liu, Automated malware detection framework using API call sequences, IEEE Conf. on BiData, 2018, pp. 1807-1811.
6. F. Guyton, Performance comparison of local versus cloud malware detection on Android using machine learning techniques, IEEE Conference Publication, 2019. [Online]. Available: <https://ieeexplore.ieee.org/document/9020622>
7. J. Bai, X. Guo, and S. Yang, Feature selection for malware detection using random forest, in Proc. IEEE ICICCS, 2019, pp. 132-136.
8. T. Darshan and R. Jaidhar, Combining static and dynamic features for improved malware detection, IEEE Int. Conf. on Computing, Communication and Networking Technologies, 2019, pp. 1-6.
9. M. Shijo and G. Salim, Hybrid malware detection based on static and dynamic API call sequences, Comput. Secur. vol. 85, 2019, pp. 267-278.
10. S. Aditya, S. S. Bendre, and S. Mukherjee, Deep learning for malware detection using API call sequences, IEEE Int. Conf. on Advances in Computing, 2019 pp. 233-238.
11. JagsirSingh, Jaswinder Singh, A survey on machine learning-based malware detection in cloud environment, ScienceDirect, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/abs/pii/S1383762120301442>
12. Y. Chai et al., LGMal: A Joint Framework Based on Local and Global Features for Malware Detection, IEEE Access, vol. 8, 2020, pp. 114627-114637.
13. S. Belaoued and M. Mazouzi, Real-time malware detection system using machine learning on PE header, Int. J. Comput. Appl., 2020, vol. 175, no. 4, pp. 36-41.
14. Y. Salehi, A. Mirjalili, and A. Zareapoor, MAAR: a scalable API call argument analysis for malware detection, IEEE Access, 2020, vol. 8, pp. 177494-177505.
15. M. Islam, A. Rahman, and S. M. Lalm, Hybrid features based malware detection system, IEEE ICCE, 2020, pp. 112-117.
16. J. S. Sraw and K. Kumar, using static and dynamic malware features to perform malware ascription, arXiv preprint arXiv: 2112.02639, Dec. 2021. [Online]. Available: <https://arxiv.org/abs/2112.02639>
17. A. A. Azeez, M. S. Ali, and F. Al-Turjman, Ensemble learning for variant malware detection, IEEE Access, 2021, vol. 9, pp. 106496-106507.
18. Resource consumption: C. Cuckoo Development Team, Dynamic analysis limitations, in Cuckoo Sandbox Documentation, 2021.

19. A. N. Azeez, O. E. Odufuwa, S. Misra, J. Oluranti, and R. Damaševičius, Windows PE malware detection using ensemble learning, *Informatics*, 2021, vol. 8, no. 1, p. 10, doi: 10.3390/informatics8010010.
20. W. Chen, W. Fan, and F. Qiu, Boosting based malware classification with feature fusion, *IEEE Trans. Emerg. Topics Comput.*, 2021 vol. 9, no. 1, pp. 159–170.
21. W. R. Aditya and S. Girinito, Deep learning for malware classification platform using API call sequences, in *Proc. 2021 IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom)*, Makassar, Indonesia, 2021, pp. 106–110.
22. Vinay J, Zuhair Bilal, Rohit K, Syed Ali Zuhair, Cloud based Malware Detection System, *International Journal of Engineering Research & Technology*, May 2022 vol. 11, no. 05.
23. K. S. Rajasekaran, P. S. Mohammed, M. A. Belal, and M. Hamdi, Machine learning for malware detection: a survey, *IEEE Access*, 2022, vol. 10, pp. 12345-12366.
24. P. Vinod, M. Sharma, and V. Jain, Dynamic and static malware analysis using machine learning techniques: a review, *IEEE Trans. Dependable Secure Comput.*, 2022, vol. 19, no. 4, pp. 346-360.
25. Advances in Malware Analysis and Detection in Cloud Computing Environment, *International Journal of Safety and Security Engineering*, Jin. 2023, vol. 14, no. 2, pp. 233-239.
26. Muhammad Shoaib Akhtar, Tao Feng, Evaluation of Machine Learning Algorithms for Malware Detection, *PeerJ Computer Science*, Jan. 2023. [Online]. Available: <https://pmc.ncbi.nlm.nih.gov/articles/PMC9862094/>
27. A. Bensaoud, J. Kalita, M. Bensaoud, A Survey of Malware Detection Using Deep Learning, arXiv: 2407.19153 [cs.CR], Jul. 2024. [Online]. Available: <https://arxiv.org/abs/2407.19153>
28. Sandbox evasion: M. Apriorit, Malware sandbox evasion: Detection techniques & solutions, *Apriorit Blog*, Sep. 2025, [Online]. Available: <https://www.apriorit.com>
29. H. Anderson and P. Roth, “EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models,” arXiv preprint arXiv: 1804.04637, 2018.