Large Language Model-Based Malware Detection for the Windows Operating System

Charles Clark[1] and Niusen Chen[2,*]

[1]*Department of Computer Sciences, University of Wisconsin-Madison, 1210 W Dayton St, Madison, WI 53706, USA*

[2]*Department of Computer Science & Engineering, University of Nevada, Reno, 1071 Evans Ave, Reno, NV 89512, USA*

Abstract: Malware detection in Windows systems remains challenging due to the rapid evolution and increasing complexity of malicious programs. Traditional static, dynamic, and machine learning approaches struggle to adapt to new or obfuscated threats. In this work, we propose a large language model (LLM) based framework for detecting malware at the application layer of the Windows operating system. By learning behavioral patterns from system calls triggered during program execution, the proposed framework allows the LLM to capture the semantic relationships between normal and malicious behaviors. We implement a prototype of the framework and evaluate its performance through experiments.

Keywords: Malware detection, Windows, large language models.

1. INTRODUCTION

In recent years, the number of malware variants has increased rapidly. Reports show that by 2024, there were more than 1.2 billion distinct malware samples worldwide, and over 100 million new malware strains appeared in 2023 [36]. On average, researchers detected about 400,000 new malicious files each day in 2023, which was higher than in the previous year [17, 27]. This rapid growth in both the number and diversity of malware makes it highly challenging to detect and defend against such threats.

Currently, Windows remains the most widely used operating system for both personal and enterprise environments [37], which also makes it the primary target for malware attacks. Studies show that a large majority of malware samples are designed to exploit Windows platforms due to their dominant global market share and extensive third-party software ecosystem [24]. As a result, ensuring malware detection and defense in Windows systems is of critical importance to maintaining user security and system integrity.

Malware detection in Windows systems has been extensively studied using static, dynamic, and machine learning approaches. Static analysis inspects executable files without running them, relying on signatures, opcode sequences, or control-flow graphs to identify known malicious patterns [26, 32, 34]. Although efficient, it is ineffective against obfuscated, packed, or polymorphic malware. Dynamic analysis instead observes runtime behaviors in sandboxed environments by monitoring API/system calls, file operations, and network activities [10, 35]. This method is more resilient to code obfuscation but suffers from high computational cost and poor scalability, and can be easily evaded by anti-sandbox techniques. To improve automation, recent studies apply machine learning and deep learning models that learn discriminative features from API call sequences, byte n-grams, or PE-header metadata [21, 31, 33]. While these models achieve higher accuracy than traditional rule-based detectors, they still depend on handcrafted features and labeled datasets, and often fail to generalize to unseen or adversarial malware. Overall, static, dynamic, and ML/DL-based methods remain limited in adaptability and semantic understanding.

Large language models (LLMs) have grown rapidly in recent years. LLMs provide a new way to solve malware detection by learning patterns directly from large amounts of data. Models like the generative pre-trained transformer (GPT) [14] can understand both text and code, helping them find hidden or unusual behaviors that older methods may miss. After being trained for security tasks, LLMs can be used for automatic threat detection, malware analysis, and vulnerability discovery. For example, the BERT [19] model is very good at understanding the meaning and order of data, making it useful for recognizing malware patterns.

In this work, we propose a LLM-based framework for detecting malware operating at the application layer

*Address correspondence to this author at the Department of Computer Science & Engineering, University of Nevada, Reno, 1071 Evans Ave, Reno, NV 89512, USA; E-mail: niusenc@unr.edu

of the Windows operating system. The key insight behind our design is that when a device is compromised by malware, the malicious program typically exhibits distinctive behaviors through specific API calls, such as attempting to gain root privileges, modifying files, or altering system configurations. These API calls in turn trigger a series of low-level system calls that reflect the malware's behavior. By collecting and analyzing these system calls, we fine-tune a pre-trained LLM to capture the behavioral characteristics that distinguish malicious activities from benign ones. In addition to its technical value, behavior-based malware detection also carries legal and forensic significance. System call traces captured during execution are used in digital forensics to reconstruct attack steps and document malicious activity. Reliable detection at this level helps organizations preserve meaningful evidence for incident investigations and supports clearer attribution when responding to security breaches. Moreover, many regulatory frameworks require auditable and explainable detection mechanisms. By modeling behavioral patterns directly from system level signals, our approach can strengthen evidence collection, improve incident response, and help organizations meet these compliance expectations.

Contributions. Major contributions of this work are:

- We collected 114 malware and benign samples running on the Windows operating system and recorded the system calls generated during their execution.
- We designed a malware detection framework by finetuning a pretrained large language model on the dataset we developed.
- We evaluated the performance of our framework.

2. BACKGROUND

2.1. Malware

Malware is malicious software designed to harm, disrupt, or exploit computer systems and networks. Typical malicious behaviors include modifying or deleting files without authorization, creating or injecting code into processes, establishing unauthorized network connections, stealing sensitive data, logging user activities, encrypting files for ransom, and concealing its presence through evasion techniques. By monitoring these abnormal or suspicious behaviors, such as unexpected file operations, network traffic, or process activities, security systems can detect and defend against malware even when its code is obfuscated or previously unseen.

2.2. Application Programming Interface Calls and System Calls

Application programming interface (API) calls [29] are high-level functions that applications use to request services from the operating system or other software components. System calls provide the basic interface between user application and the OS kernel. For instance, the Windows API function CreateFile internally calls the system call NtCreateFile to access the file system. The relationship between API call and system call is described in Figure **1**.

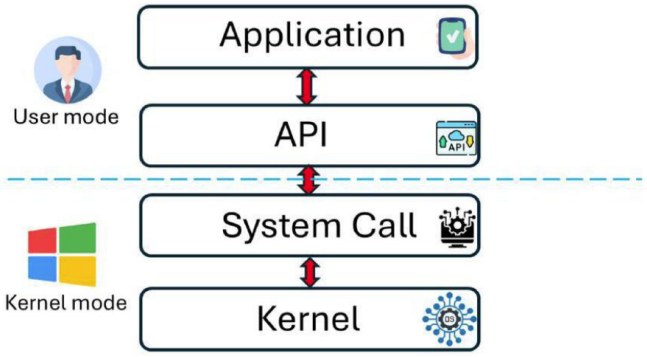

Figure 1: Relationship between API calls and system calls.

2.3. Large Language Models

Large language models (LLMs) are neural network systems trained on a massive amount of data in order to understand and generate human language. Almost all modern LLMs use Transformer architecture. Transformer architecture is a neural network architecture that allows models to process sequences of text in parallel, significantly speeding up training and improving contextual understanding.

3. SYSTEM AND ADVERSARIAL MODEL

We consider a computing device running the Windows operating system, such as a desktop or workstation, that executes multiple user-level applications and background services. Malware may compromise one or more applications at the application layer through malicious email attachments or Trojan horses [40]. Once activated, the malware can carry out various malicious actions, such as credential theft, file encryption, or launching additional payloads.

In our adversarial model, we assume an attacker operating entirely at the application layer. The attacker

cannot alter kernel-level system call behavior, but can modify the malware's user-level execution patterns to evade behavioral detection. This includes obfuscating control flow, adding benign API calls, and delaying or splitting malicious actions across multiple processes. We also assume the attacker is aware of LLM-based detection approaches and may attempt to shape the observable behavior of the application to appear more benign, although they do not have white-box access to our model. We do not consider evasion attacks based on adversarial API-call manipulation, nor attacks that exploit LLM vulnerabilities such as adversarial prompts or poisoned inputs.

4. TERMINOLOGY

- Token - A token is the fundamental unit of data used by language models and serves as a building block for processing text. Importantly, one API call does not necessarily correspond to a single token. For example, the API call NtCreateFile may be split into two tokens: *Nt* and CreateFile.

- Process - A running instance of an executable program. A single executable may create multiple processes, each with its own sequence of API calls.

- Chunk - A contiguous subset of system calls from a process. The subset typically has a limited length (e.g., 512 for the RoBERTa-base model). Each chunk serves as an individual training sample.

5. DESIGN

5.1. Overview

To detect application-level malware on Windows, a detector should monitor the system calls issued by programs and use those traces as input to a classifier. This task requires two main steps: 1) collect and extract sequences of system calls from both benign software and malicious samples; and 2) determine whether a trace indicates malware using a pre-trained classifier. We use large language models as the classifier. Since general-purpose large language models are not trained for malware detection, we fine-tune them on the collected system-call traces so that they can learn to distinguish malicious behavior from normal activity. Our design includes the following components: data collection, data preparation, and model training. We will elaborate on each of them below.

5.2. Data Collection

We have collected 114 malware samples and 114 benign samples. Each sample is an executable file that can be executed on the Windows. The malware samples were obtained from MalwareBazaar [3], a platform that provides the latest samples and threat intelligence. The benign samples were collected from FOSSHUB [2], NirSoft [5], SourceForge [7], and Microsoft [4]. As illustrated in Figure **2**, the malware and benign samples are executed in an isolated sandbox environment. After each execution, the system is restored to its initial state to ensure a clean environment. This procedure is repeated until all samples (both malware and benign) have been tested. After each run, a JSON report is generated containing detailed information about the process and the API calls invoked during the execution.

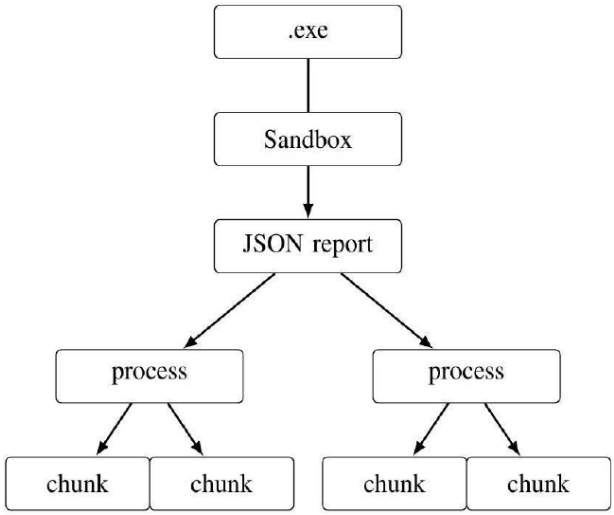

Figure 2: Data pipeline for a single executable.

From each JSON report we extract the processes and their ordered API call names. The sequences are partitioned into fixed-length JSON chunks, with a partial overlap between consecutive chunks to preserve contextual continuity across boundaries. We use a 64 token overlap for 512 and 256 token chunks, dropping it to a 32 token overlap for 128 token chunks. Processes with less than 20 calls were dropped to ensure each sample contains sufficient information for meaningful classification. Very short behavioral traces can introduce noise and potentially reduce model performance. Each chunk is associated with the following three metadata fields.

- Label - Indicates the class used for supervised learning: 0 for benign samples and 1 for malicious samples.

- Analysis_ID - A unique identifier assigned to each executable file. Every executable executed within the sandbox is associated with a distinct ID, ensuring that each process can be precisely traced back to its originating executable.

- Process_ID - A unique identifier assigned to each process generated by an executable during execution. Since a single executable may have multiple processes, each process is assigned a distinct process ID to enable precise tracking.

5.3. Data Preparation

After generating the JSON chunks and adding the metadata, we performed several preprocessing steps to prepare the dataset for model training. The main steps are as follows.

Chunk Aggregation. All generated chunks were randomly merged into a single JSON Lines (JSONL) file. A JSONL file stores multiple JSON objects, with one object per line.This format makes it efficient to store and process large datasets.

Deduplication. A deduplication step was applied across the entire JSONL file to remove redundant entries. This process eliminated duplicate chunks to ensure that only unique behavioral data were retained for model training.

Dataset Splitting. The dataset was divided into training, validation, and testing subsets, using a 60%, 20% and 20% split, respectively. To avoid data leakage, chunks were grouped by their Analysis_ID before splitting. This ensures that all chunks originating from the same application remain within the same subset.

5.4. Model Training

We fine-tune the model using the AdamW optimization algorithm, a variant of Adam (Adaptive Moment Estimation) that decouples weight decay from gradient updates, leading to better generalization and training stability. We do not cap or down-sample the larger class. Instead, during training, we apply a process-uniform sampler, where each chunk is sampled with weight $w_i = 1/|C(p)|$, where $|C(p)|$ denotes the number of chunks in process p. We include a small class term in the training sampler so that benign and malware examples are presented at roughly the same rate each epoch. This approach prevents the majority class from dominating and improves learning for the minority class. The re-balancing is applied only during training.

6. IMPLEMENTATION AND EVALUATION

6.1. Experimental setup

Data collection was performed in CAPEv2 [1], an open-source sandbox derived from the Cuckoo v1 sandbox. CAPEv2 was run inside an isolated Windows 10 virtual machine. CAPEv2 collects detailed data such as behavioral logs, file modifications, network traffic and memory dumps. In this work, we focus on the API calls and their corresponding processes. To avoid bias, we also removed processes containing fewer than 20 API calls and duplicate processes. A summary of the collected data is presented in Table **1**.

Table 1: Number of Processes before and after Filtering

Class	Original	< 20 calls	Duplicate	Final
Malware	405	30	9	366
Benign	145	7	3	135

The model used for training and detection is RoBERTabase [6], a pretrained Transformer-based language model originally released in 2019. It was fine-tuned on our training dataset using the Hugging Face Transformers library and PyTorch within a Google Colab environment equipped with an NVIDIA T4 GPU. The parameters we used to train our model are shown in Table **2**.

Table 2: Training Parameters used for Fine-Tuning the RoBERTa-Base Model

Parameter	Value
Learning rate	1*e* - 5
Weight decay	0.01
Batch size	4
Gradient accumulation	4
Max training epochs	18

We applied early stopping using the validation F1 score with a patience of five epochs. Training was

stopped when the F1 score did not improve for five consecutive epochs, and the model was restored to the checkpoint with the highest F1. In our experiments, training typically completed in about twelve epochs. We also used a cosine learning rate schedule with a warm-up ratio of 10%, and a fixed random seed was applied to ensure reproducibility.

6.2. Metrics

To evaluate the effectiveness of our proposed detection model, we first define four key terms: true positive (TP), false negative (FN), false positive (FP), and true negative (TN). Based on these definitions, we leverage several widely used classification metrics, including accuracy (Acc), precision, recall, F1 score, false positive rate (FPR), and false negative rate (FNR). The definitions of these metrics are as follows:

Accuracy (Acc): The ratio of correctly classified samples to the total number of samples.

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

Precision: The ratio of correctly predicted positive samples to all predicted positives.

$$Precision = \frac{TP}{TP + FP}$$

Recall: The ratio of correctly predicted positive samples to all actual positives.

$$Recall = \frac{TP}{TP + FN}$$

F1 Score (F1): The harmonic mean of precision and recall, reflecting a balance between them.

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

False Positive Rate (FPR): The ratio of negative samples incorrectly predicted as positive.

$$FPR = \frac{FP}{FP + TN}$$

False Negative Rate (FNR): The ratio of positive samples incorrectly predicted as negative.

$$FNR = \frac{FN}{FN + TP}$$

6.3. Behavioral Analysis

As shown in Table **3**, distinct behavioral patterns can be observed between malware and benign software. One significant observation is that malware has fewer API calls per process (2,910) compared to benign software (5,234). The main reason is that malware usually executes shorter and more focused sequences of API calls, often designed to achieve specific malicious goals such as privilege escalation or file modification. In contrast, benign software tends to perform more diverse and long-running operations, including user interactions, background activities, and complex system functions. For example, malware may quickly encrypt files using a short sequence of file access and encryption APIs, while a word processor performs many different operations such as editing and saving documents.

Table **3** also reveals differences beyond API-call counts. Malware processes generate substantially fewer tokens overall (13,132 on average) compared to benign processes (26,669). This reduced volume suggests that malware tends to operate in compact execution bursts, often minimizing unnecessary system interactions to reduce detection surface. Benign software, by comparison, produces longer and more heterogeneous execution traces because it performs multiple functions, maintains state, and interacts with various subsystems over time.

Table 3: Tokens and API Calls by Class

Class	Total tokens	Avg tokens/Proc	Avg calls/Proc
Malware	4,806,313	13,132	2,910
Benign	3,600,180	26,669	5,234

6.4. Evaluation

6.4.1. Model Performance Under Different Chunk Sizes

After training, we calibrate probabilities using temperature scaling on the validation set in order to minimize negative log-likelihood (NLL).

$$NLL_{binary} = -\frac{1}{M}\sum_{i=1}^{M} [y_i log\ p_i + (1 - y_i) log\ (1 - p_i)]$$

We then sweep decision thresholds on the validation set in order to maximize F1 for each aggregation. The chosen temperature and thresholds are applied once to the test set before evaluation. The trained model is employed to evaluate the malware

detection performance on the testing dataset. To investigate the effect of chunk length on the detection rate, we vary it to 512, 256, and 128, respectively. 512 corresponds to the maximum sequence length used during the pre-training of RoBERTa-base [6]. Chunk sizes of 256 and 128 tokens are also commonly used when fine-tuning Transformer-based models [50, 51]. The model outputs a malware probability for each individual chunk. We will refer to each of these as a chunk-level probability. To produce a single process-level probability the corresponding chunk-level probabilities are aggregated together using three methods:

Prob_mean: the average of all chunk probabilities.

$$Prob\_mean = \frac{1}{N}\sum_{i=1}^{N} p_i$$

Prob_p95: the 95th percentile of the chunk probabilities, which emphasizes strong evidence while mitigating noise.

$$Prob\_p95 = Percentile_{95}(\{p_1, p_2, \ldots, p_N\})$$

Prob_max: the maximum chunk probability, capturing the strongest signal but making the prediction more sensitive to outliers.

$$Prob\_max = \max_{1 \le i \le N} p_i$$

The same aggregation methods are then used to produce an application-level probability, where all process-level probabilities corresponding to that application are combined to yield a final prediction.

The results are shown in Table **4**. We can detect that, at the application level, prob_mean with the default 512 token chunks remains the most balanced and deployment ready aggregation method, achieving 93% accuracy, 96% precision, 92% recall, 0.94 *F*1, 5% *FPR*, and 8% *FNR*. When the chunk size is reduced to 256 tokens, prob_p95 improves markedly—its false positive rate drops from 63% to 36% while maintaining perfect recall, demonstrating better precision-recall tradeoffs. At 128 tokens, performance for prob_mean closely mirrors that of the 512-token configuration, with only marginal differences across all metrics, suggesting that the model's decision stability is largely invariant to smaller chunk sizes. Prob_max remains nearly unchanged across all window sizes, reinforcing that chunk length has minimal influence on its strongest activation signals. Overall, prob_mean (512) and prob_mean (128) show consistent, reliable generalization, while prob_p95 (256) continues to offer the best high recall configuration.

6.4.2. Confusion Matrices at Tuned Thresholds

A confusion matrix is a table used to evaluate the performance of a classification model by comparing its predictions to the true labels. This provides deeper insight into how the model behaves, beyond summary metrics such as accuracy or F1. Tables **5**, **6**, and **7** below show the respective confusion matrix at the application level using a 512 chunk size for prob_mean, prob_p95 and prob_max. From the results, the confusion matrices show strong separability between malware and benign applications. prob_mean offers the most balanced performance, with very few false positives and false negatives. In contrast, prob_p95 and prob_max produce higher true-positive rates but also more false positives, indicating a tendency to over-predict malware.

6.4.3. Receiver Operating Characteristic Curve and Area Under Curve

The Receiver Operating Characteristic (ROC) curve provides a graphical summary of a model's ability to distinguish between classes across all possible decision thresholds. The Area Under the Curve (AUC) corresponds to the probability that the model assigns a higher risk score to a randomly selected malware sample than to a randomly selected benign sample.

We plot the ROC curves for application-level classification using the prob_mean, prob_p95, and prob_max aggregation methods. The corresponding results are shown in Figures **3**, **4**, and **5**, respectively. The results show that prob_mean achieves an AUC of 0.989, indicating excellent separability between malware and benign applications. In other words, the model assigns a higher risk score to a randomly chosen malware application than to a randomly chosen benign application in 98.9% of cases. Both prob_p95 and prob_max (Figures **4** and **5**) achieve an AUC of 0.949, demonstrating strong separability but consistently lower performance compared to prob_mean. These findings reinforce that prob_mean provides the most stable and deployment-ready aggregation method.

6.4.4. Deployment Feasibility

We also evaluated the runtime performance of the detector . With a chunk size of 512, the model achieves a mean per-chunk latency of 2.82 ms, a throughput of

Table 4: Test Performance at Tuned Decision Thresholds for Three Chunk Sizes (512, 256, and 128 Tokens)

Level /	Acc	Precision	Recall	F1	FPR	FNR	Threshold
Chunk size = 512							
Process Level							
prob_mean	0.94	0.92	1.00	0.96	0.26	0.00	0.5
prob_p95	0.84	0.83	1.00	0.91	0.65	0.00	0.55
prob_max	0.82	0.81	1.00	0.89	0.74	0.00	0.7
Application Level							
prob_mean	0.93	0.96	0.92	0.94	0.05	0.08	0.8
prob_p95	0.70	0.63	1.00	0.77	0.63	0.00	0.55
prob_max	0.65	0.60	1.00	0.75	0.73	0.00	0.5
Chunk size = 256							
Process Level							
prob_mean	0.91	0.94	0.94	0.94	0.17	0.06	0.7
prob_p95	0.90	0.89	1.00	0.94	0.39	0.00	0.8
prob_max	0.82	0.81	1.00	0.89	0.74	0.00	0.9
Application Level							
prob_mean	0.91	0.95	0.86	0.91	0.05	0.13	0.75
prob_p95	0.83	0.75	1.00	0.86	0.36	0.00	0.9
prob_max	0.65	0.60	1.00	0.75	0.73	0.00	0.9
Chunk size = 128							
Process Level							
prob_mean	0.95	0.96	0.97	0.97	0.13	0.03	0.6
prob_p95	0.90	0.90	0.99	0.94	0.34	0.01	0.85
prob_max	0.86	0.87	0.97	0.91	0.48	0.03	0.9
Application Level							
prob_mean	0.93	0.92	0.96	0.94	0.09	0.04	0.65
prob_p95	0.93	0.89	1.00	0.94	0.14	0.00	0.9
prob_max	0.78	0.71	1.00	0.83	0.45	0.00	0.9

Table 5: Confusion Matrix for prob_mean at the Application Level (512 Chunk Size)

	Actual Malware	Actual Benign
Predicted Malware	22	1
Predicted Benign	2	21

Table 6: Confusion Matrix for prob_p95 at the Application Level (512 Chunk Size)

	Actual Malware	Actual Benign
Predicted Malware	24	14
Predicted Benign	0	8

Table 7: Confusion Matrix for prob_max at the Application Level (512 Chunk Size)

	Actual Malware	Actual Benign
Predicted Malware	24	16
Predicted Benign	0	6

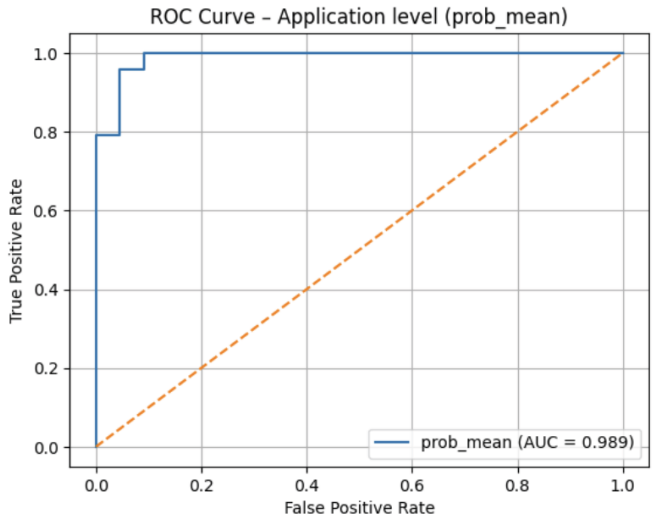

Figure 3: ROC curve for application-level classification using the prob_mean aggregation method (512 chunk size).

355.1 chunks per second, and uses about 1.6 GB of memory during inference. On average, each process generates 35.36 chunks, and each application generates 72.26 chunks. This leads to an estimated end-to-end detection time of about 100 ms per process and about 204 ms per application. In both cases, the full classification completes in well under 250 ms, which is suitable for near–real-time use.

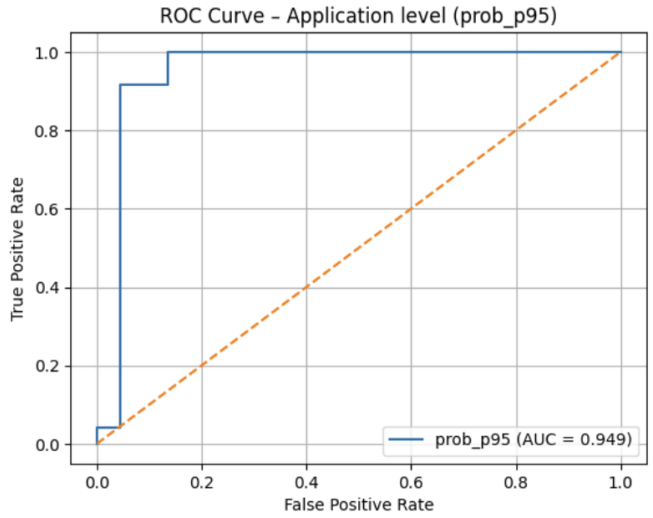

Figure 4: ROC curve for application-level classification using the prob_p95 aggregation method (512 chunk size).

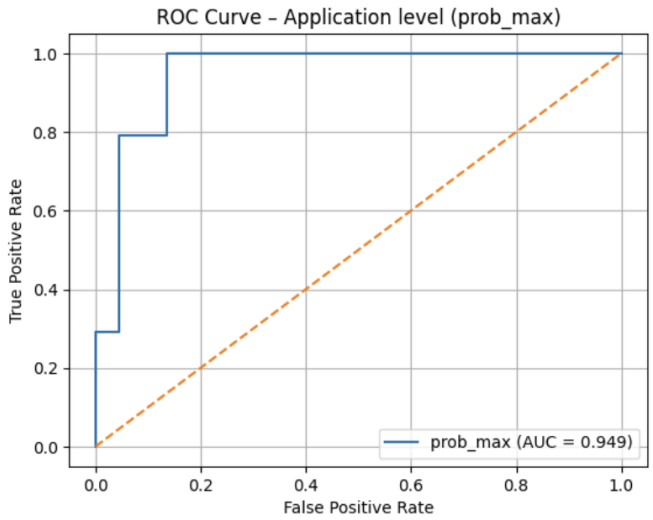

Figure 5: ROC curve for application-level classification using the prob_max aggregation method (512 chunk size).

7. DISCUSSION

7.1. Limitation on Detection Scope

One limitation of our work is that it focuses solely on detecting malware at the application layer. Our approach analyzes system calls and behavioral patterns generated by user-level processes, enabling effective identification of malicious activities within applications. However, it does not address threats that

compromise the operating system itself, such as kernel-level rootkits [22] or other low-level attacks that can bypass user space monitoring. Defending against such OS-level malware requires additional mechanisms, such as hypervisor based monitoring [9], or hardware-assisted security techniques [15], which are beyond the scope of this work.

7.2. Real-Time Detection Constraint

Although the proposed model effectively detects malicious behaviors from system call sequences, the inference process of large language models can be computationally intensive. This may introduce noticeable latency when analyzing continuous streams of system calls in real-world environments. As a result, deploying the model for real-time malware detection becomes challenging, especially on systems with limited computing resources. Future work could explore lightweight model architectures or hardware acceleration techniques to reduce overhead.

7.3. Platform Dependency

Our model and data collection process are specifically designed for the Windows operating system, leveraging system calls and behaviors unique to this environment. As a result, applying the same model to other platforms, such as Linux, macOS, or Android, is non-trivial due to differences in system call interfaces. Future work may focus on integrating cross-platform datasets to enhance the model's generalization and adaptability.

7.4. Bias from Excluding Short Malware Traces

We removed duplicate processes and processes containing fewer than 20 API calls to ensure each sample provides sufficient and meaningful behavioral data. However doing so may introduce bias by under-representing short malware traces. Such traces, while short, may be relevant for security and by excluding these the model becomes biased towards longer behavioral patterns. Future work should incorporate these short traces to more accurately reflect real world data.

7.5. Dataset limitations

A key limitation of this work is the relatively small dataset, consisting of 114 malware and 114 benign executables. Although each executable generates multiple processes and system-call chunks, the overall behavioral diversity remains limited, which may restrict the model's ability to generalize to broader real-world malware families. The small dataset size also increases the risk of overfitting, particularly when fine-tuning large language models that may memorize recurring behavioral patterns rather than learning robust semantic distinctions. Expanding the dataset to include more diverse malware families is an important direction for future work to enhance both robustness and generalization.

7.6. Forensic and Evidentiary Implications

Our malware-detection framework also carries meaningful implications for digital forensics and legal proceedings. The system call traces and model-generated classification logs can serve as reproducible behavioral records that support incident reconstruction, attribution, and timeline analysis during investigations. Because these logs document how an application behaved at the system call level, they can be preserved as digital evidence and referenced in legal or organizational review processes. The model's outputs can also be audited because the preprocessing steps and decision rules are fixed and easy to check. Basic explainability further shows which behaviors led to a decision, which helps improve trust and supports the use of these results as evidence.

7.7. High FPR for prob_p95 and prob_max

Both aggregations are sensitive to spikes or bursts of suspicious behavior, leading to an increase in falsely classified benign files. However as a trade off they achieve perfect or near perfect recall at all chunk sizes. Due to this high sensitivity prob_mean is recommended for deployment, achieving better stability while maintaining accuracy, precision and recall.

8. RELATED WORK

8.1. Signature-Based Malware Detection

A signature is a distinctive feature of malware that encapsulates its structural characteristics and uniquely identifies each sample. Signature-based detection is one of the most widely adopted techniques in commercial antivirus systems, where predefined signatures are used to recognize and block known malware. F. Zolkipli and Jantan designed a malware detection framework that combines signature-based methods, a genetic algorithm, and an automatic signature generator [44]. Tang *et al*. developed a bioinformatics approach that aligns sequences, removes noise, and converts results into simplified

regular-expression signatures compatible with existing intrusion detection systems [39]. Borojerdi and Abadi introduced MalHunter, a detection system that uses sequence clustering and alignment to automatically generate behavior-based signatures for polymorphic malware [13].

8.2. Behavior-Based Malware Detection

The behavior-based malware detection approach monitors program activities using analysis tools and determines whether a program exhibits malicious or benign behavior. Fukushima *et al*. [20] proposed a behavior-based detection approach capable of identifying both unknown and encrypted malware on Windows systems. Christodorescu *et al*. [16] proposed a semantics-aware malware detection approach, observing that certain malicious behaviors consistently appear across all variants of a malware family. A supervised machine learning model was proposed in [30], employing a kernel-based SVM with weighting measures that calculate the frequency of each library call to detect Mac OS X malware.

8.3. Heuristic-Based Malware Detection

Heuristic-based malware detection is a complex approach that leverages prior knowledge, rules, and machine learning techniques to identify malicious software. Arnold and Tesauro proposed an automatically generated heuristic framework for detecting Win32 viruses [8]. Their approach builds multiple neural network classifiers capable of identifying previously unknown Win32 malware. Yanfang *et al*. [42] proposed the Intelligent Malware Detection System (IMDS), which employs objective-oriented association (OOA) mining based on Windows API call analysis. Naval *et al*. [28] proposed a dynamic malware detection system that captures system calls and constructs a graph to identify semantically relevant paths among them.

8.4. Transformer-Based and LLM-based Malware Detection

Recent advances in transformer architectures have inspired a new class of malware detection techniques that reduce dependence on handcrafted features. Raff *et al*. [31] introduced MalConv, a deep learning architecture capable of processing raw executable bytes to detect malware without manual feature engineering. Feng *et al*. proposed LLM-MalDetect [46], which leverages a fine-tuned large language model to integrate permissions, API calls, and string-based semantic features extracted from Android APKs to, achieving higher accuracy. Their results demonstrate that LLMs can capture richer behavioral and contextual information than traditional ML/DL approaches. Zhou *et al*. [47] proposed SRDC, a semantics-based ransomware detection and classification framework that combines internal feature semantics with external LLM-generated knowledge to improve robustness and generalization. Their results show that SRDC significantly outperforms traditional ML/DL methods.

8.5. Other Malware Detection Technologies

Other malware detection technologies include model checking-based malware detection [11, 23, 25], deep learning-based malware detection [12, 18, 43], and cloud-based malware detection [38, 41].

8.6. Forensic and Regulatory Perspectives on Automated Detection

Beyond technical detection methods, prior work has highlighted the forensic and regulatory importance of automated malware-analysis systems. Behavioral artifacts such as system-call traces, audit logs, and execution traces provide reproducible digital evidence that supports incident reconstruction and attribution in forensic investigations [48]. Likewise, regulatory frameworks such as NIST SP 800-53 [49] emphasize the need for transparent, auditable, and timely detection mechanisms to meet incident-response obligations. These works demonstrate that automated detection systems play a significant role not only in identifying malicious behavior but also in ensuring evidentiary integrity and regulatory compliance.

9. CONCLUSION

This paper presents a framework for binary classification between malicious and benign Windows applications. A pretrained RoBERTa model was fine-tuned using a structured dataset of API call traces. Among the tested aggregation strategies, prob_mean demonstrated the most reliable and stable performance across all chunk sizes, with the default 512-token configuration achieving 92% recall and a 0.94 F 1 at the application level. The 128 -token configuration showed nearly identical results, indicating strong consistency and robustness. Prob_p95 remained the second most promising method, particularly at 256 tokens, where it achieved perfect recall (1.00) with improved precision compared to larger chunks. Overall, the results show that the proposed framework

effectively distinguishes malware from benign behavior on Windows systems and remains stable under varying input sizes, making it well-suited for real-world deployment.

Future work can expand the system in several ways. Adding stronger explainability would show which behaviors shape the model's decisions. Connecting the detector with forensic tools could improve incident reconstruction and evidence handling. It will also be important to test the model against adversarial behavior. Additionally, aligning the system with policy and audit requirements can support deployment in real environments.

REFERENCES

[1] Cape sandbox book. https://capev2.readthedocs.io/en/latest/installation/host/installation.html

[2] Fosshub. https://www.fosshub.com/categories.html.

[3] Malwarebazaar. https://bazaar.abuse.ch/.

[4] Microsoft sysinternals suite. https://learn.microsoft.com/en-us/sysint ernals/downloads/sysinternals-suite

[5] Nirsoft. https://www.nirsoft.net/.

[6] Liu, Yinhan, *et al*. "Roberta: A robustly optimized bert pretraining approach." arXiv preprint arXiv:1907.11692 (2019).

[7] Sourceforge. https://sourceforge.net/.

[8] William Arnold and Gerald Tesauro. Automatically generated win32 heuristic virus detection. In Proceedings of the 2000 international virus bulletin conference, 2000.

[9] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. A survey on hypervisor-based monitoring: approaches, applications, and evolutions. ACM Computing Surveys (CSUR), 48(1):1-33, 2015. https://doi.org/10.1145/2775111

[10] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In NDSS, volume 9, pages 8-11, 2009.

[11] Philippe Beaucamps, Isabelle Gnaedig, and Jean-Yves Marion. Abstraction-based malware analysis using rewriting and model checking. In European Symposium on Research in Computer Security, pages 806-823. Springer, 2012. https://doi.org/10.1007/978-3-642-33167-1_46

[12] Yoshua Bengio *et al*. Learning deep architectures for ai. Foundations and trends® in Machine Learning, 2(1):1-127, 2009. https://doi.org/10.1561/2200000006

[13] Haniye Razeghi Borojerdi and Mahdi Abadi. Malhunter: Automatic generation of multiple behavioral signatures for polymorphic malware detection. In ICCKE 2013, pages 430-436. IEEE, 2013. https://doi.org/10.1109/ICCKE.2013.6682867

[14] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, *et al*. Language models are fewshot learners. Advances in neural information processing systems, 33:1877-1901, 2020.

[15] Niusen Chen, Wen Xie, and Bo Chen. Combating the os-level malware in mobile devices by leveraging isolation and steganography. In Applied Cryptography and Network Security Workshops, 2021. https://doi.org/10.1007/978-3-030-81645-2_23

[16] Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant. Semantics-aware malware detection. In 2005 IEEE symposium on security and privacy (S&P'05), pages 32-46. IEEE, 2005. https://doi.org/10.1109/SP.2005.20

[17] Malware statistics. https://controld.com/blog/malware-statistics-trends/

[18] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. Large-scale malware classification using random projections and neural networks. In 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, pages 3422-3426. IEEE, 2013. https://doi.org/10.1109/ICASSP.2013.6638293

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers), pages 4171-4186, 2019. https://doi.org/10.18653/v1/N19-1423

[20] Yoshiro Fukushima, Akihiro Sakai, Yoshiaki Hori, and Kouichi Sakurai. A behavior based malware detection scheme for avoiding false positive. In 2010 6th IEEE workshop on secure network protocols, pages 79-84. IEEE, 2010. https://doi.org/10.1109/NPSEC.2010.5634444

[21] William Hardy, Lingwei Chen, Shifu Hou, Yanfang Ye, and Xin Li. D14md: A deep learning framework for intelligent malware detection. In Proceedings of the International Conference on Data Science (ICDATA), page 61. The Steering Committee of The World Congress in Computer Science, Computer ..., 2016.

[22] Greg Hoglund and James Butler. Rootkits: subverting the Windows kernel. Addison-Wesley Professional, 2006.

[23] Andreas Holzer, Johannes Kinder, and Helmut Veith. Using verification technology to specify and detect malware. In International Conference on Computer Aided Systems Theory, pages 497-504. Springer, 2007. https://doi.org/10.1007/978-3-540-75867-9_63

[24] Kozak, Matous, *et al*. "Updating Windows malware detectors: Balancing robustness and regression against adversarial EXEmples." Computers & Security 155 (2025): 104466. https://doi.org/10.1016/j.cose.2025.104466

[25] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Proactive detection of computer worms using model checking. IEEE transactions on dependable and secure computing, 7(4):424-438, 2008. https://doi.org/10.1109/TDSC.2008.74

[26] Jeremy Z Kolter and Marcus A Maloof. Learning to detect malicious executables in the wild. In Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, pages 470-478, 2004. https://doi.org/10.1145/1014052.1014105

[27] Infosecurity Magazine. Daily malicious files soar 3% in 2023. https: //www.infosecurity-magazine.com/news/daily-malicious-files-soar-3-2023/

[28] Smita Naval, Vijay Laxmi, Muttukrishnan Rajarajan, Manoj Singh Gaur, and Mauro Conti. Employing program semantics for malware detection. IEEE Transactions on Information Forensics and Security, 10(12):2591-2604, 2015. https://doi.org/10.1109/TIFS.2015.2469253

[29] Joshua Ofoeda, Richard Boateng, and John Effah. Application programming interface (api) research: A review of the past to inform the future. International Journal of Enterprise Information Systems (IJEIS), 15(3):76-95, 2019. https://doi.org/10.4018/IJEIS.2019070105

[30] Hamed Haddad Pajouh, Ali Dehghantanha, Raouf Khayami, and KimKwang Raymond Choo. Intelligent os x malware

threat detection with code inspection. Journal of Computer Virology and Hacking Techniques, 14(3):213-223, 2018. https://doi.org/10.1007/s11416-017-0307-5

[31] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. Malware detection by eating a whole exe. arXiv preprint arXiv:1710.09435, 2017.

[32] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. information Sciences, 231:64-82, 2013. https://doi.org/10.1016/j.ins.2011.08.020

[33] Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features. In 2015 10th international conference on malicious and unwanted software (MALWARE), pages 11-20. IEEE, 2015. https://doi.org/10.1109/MALWARE.2015.7413680

[34] M Zubair Shafiq, S Momina Tabish, Fauzan Mirza, and Muddassar Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In International workshop on recent advances in intrusion detection, pages 121-141. Springer, 2009. https://doi.org/10.1007/978-3-642-04342-0_7

[35] PV Shijo and AJPCS Salim. Integrated static and dynamic analysis for malware detection. Procedia Computer Science, 46:804-811, 2015. https://doi.org/10.1016/j.procs.2015.02.149

[36] Spacelift. Malware statistics 2024. https://spacelift.io/blog/malwar e-statistics

[37] StatCounter. Desktop operating system market share worldwide. https://gs.statcounter.com/os-market-share/desktop/worldwide

[38] Hao Sun, Xiaofeng Wang, Rajkumar Buyya, and Jinshu Su. Cloudeyes: Cloud-based malware detection with reversible sketch for resource-constrained internet of things (iot) devices. Software: Practice and Experience, 47(3):421-441, 2017. https://doi.org/10.1002/spe.2420

[39] Yong Tang, Bin Xiao, and Xicheng Lu. Using a bioinformatics approach to generate accurate exploit-based signatures for polymorphic worms. computers & security, 28(8):827-842, 2009. https://doi.org/10.1016/j.cose.2009.06.003

[40] S Typel and G Baur. Theory of the trojan-horse method. Annals of physics, 305(2):228-265, 2003. https://doi.org/10.1016/S0003-4916(03)00060-5

[41] Ram Mahesh Yadav. Effective analysis of malware detection in cloud computing. Computers & Security, 83:14-21, 2019. https://doi.org/10.1016/j.cose.2018.12.005

[42] Yanfang Ye, Dingding Wang, Tao Li, Dongyi Ye, and Qingshan Jiang. An intelligent pe-malware detection system based on association mining. Journal in computer virology, 4(4):323-334, 2008. https://doi.org/10.1007/s11416-008-0082-4

[43] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. Droidsec: deep learning in android malware detection. In Proceedings of the 2014 ACM conference on SIGCOMM, pages 371-372, 2014. https://doi.org/10.1145/2619239.2631434

[44] Mohamad Fadli Zolkipli and Aman Jantan. A framework for malware detection using combination technique and signature generation. In Computer Research and Development, International Conference on, pages 196-199. IEEE Computer Society, 2010. https://doi.org/10.1109/ICCRD.2010.25

[45] Pearce, Hammond, *et al*. "Examining zero-shot vulnerability repair with large language models." 2023 IEEE Symposium on Security and Privacy (SP). IEEE, 2023. https://doi.org/10.1109/SP46215.2023.10179420

[46] Feng, Ruirui, *et al*. "LLM-MalDetect: A Large Language Model-Based Method for Android Malware Detection." IEEE Access (2025). https://doi.org/10.1109/ACCESS.2025.3565526

[47] Zhou, Ce, *et al*. "SRDC: Semantics-based Ransomware Detection and Classification with LLM-assisted Pre-training." Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 39. No. 27. 2025. https://doi.org/10.1609/aaai.v39i27.35080

[48] Casey, Eoghan. Digital evidence and computer crime: Forensic science, computers, and the internet. Academic press, 2011.

[49] NIST SP 800-53. https://csrc.nist.gov/pubs/sp/800/53/r5/upd1/final

[50] Chen, Aokun, *et al*. "Contextualized medication information extraction using transformer-based deep learning architectures." Journal of biomedical informatics 142 (2023): 104370. https://doi.org/10.1016/j.jbi.2023.104370

[51] TensorFlow. (2023). Fine-tune BERT on a downstream task. Retrieved from https://www.tensorflow.org/tfmodels/nlp/fine_tune_bert

Received on 27-10-2025 Accepted on 29-11-2025 Published on 17-12-2025

https://doi.org/10.65879/3070-5789.2025.01.09