Detection and Explanation of PowerShell Malware with Large Language Models

Meng Wang[1], Emma Topolovec[1], Beatrice Arana[1], Naga Venkata Prasanna Sai Gupta Chimakurthy Kuladeep[1], Zhaohan Xi[1], Guanhua Yan[1], Xiaokui Shu[2], Scott D. Stoller[3], and Ping Yang[1]

[1] Binghamton University, State University of New York, Binghamton NY, USA
{mwang150,etopolo1,barana1,cgupta2,zxi1,ghyan,pyang}@binghamton.edu
[2] IBM Thomas J. Watson Research Center, Yorktown Heights NY, USA
xiaokui.shu@ibm.com
[3] Stony Brook University, Stony Brook NY, USA
stoller@cs.stonybrook.edu

Abstract. PowerShell-based malware increasingly leverages fileless, in-memory execution and advanced obfuscation techniques to bypass traditional signature-based security solutions. Although machine learning (ML) methods have shown promise in detecting such threats, their limited interpretability often hinders security analysts' ability to understand and validate detection results. Recent advancements in large language models (LLMs) demonstrate their potential for both threat detection and the generation of interpretable explanations. This paper investigates the application of LLMs for detecting malicious PowerShell scripts and producing human-interpretable explanations for their classification decisions. In our approach, LLMs are prompted to classify PowerShell scripts, localize specific code segments responsible for the malicious behavior, and generate natural language explanations that articulate the reasoning behind these classifications. To assess the contribution of the identified segments to the script's maliciousness, we perform an ablation study to measure changes in VirusTotal community scores before and after removing (masking) these segments. A significant reduction in score suggests that the code segments likely contribute to the script's malicious behavior. Our results show that LLMs are capable of identifying and explaining malicious PowerShell scripts, although performance varies across different models.

Keywords: Malware Detection · Large Language Models · Explainable AI.

1 Introduction

PowerShell-based malware has become increasingly prevalent due to its capability to execute stealthily in memory, bypass traditional signature-based antivirus solutions, and exploit PowerShell's native integration within the Windows operating system [31]. Such malware attacks often embed malicious activity within

legitimate scripting operations, posing significant challenges for conventional security tools.

Machine learning-based approaches have shown promise in improving security in areas such as intrusion detection [24,30,37], anomaly detection [3,28,11], and malware classification [21,2,13]. However, a major limitation of these models is the difficulty in interpreting their decisions. While Explainable AI techniques, such as SHapley Additive exPlanations (SHAP) [20], provide insights into the contribution of individual features, it is often difficult for non-experts to interpret their outputs. This is due to the complexity of numerical and graphical representations, the indirect relationship between SHAP values and specific malicious behaviors, and the lack of sufficient contextual information linking SHAP outputs to real-world security events [7].

Recently, Large Language Models (LLMs) have demonstrated capability in reasoning and code analysis, making them a promising tool for malware detection. While LLMs have been applied to a range of security tasks including automated network intrusion detection [38], vulnerability discovery [5], and explainable anomaly detection [1], to the best of our knowledge, they have not been explored for malware detection and explanation.

In this paper, we investigate the use of LLMs to detect malicious PowerShell scripts, localize the code segments responsible for malicious behavior, and generate interpretable, human-readable explanations that help security analysts understand the rationale behind the detection results. Our study uses a dataset of malicious PowerShell scripts compiled in [35]. The maliciousness of each script was verified from VirusTotal, a widely used platform that aggregates malware detection results from over 70 security vendors, including major companies such as Kaspersky Lab, Avira, BitDefender, AVG, ESET, G-Data, Malwarebytes, and McAfee [34]. The Community Score of VirusTotal is defined as the fraction of security vendors that flag a file as malicious.

In our framework, LLMs are first prompted to classify each script as either benign or malicious. For scripts identified as malicious, the models are further prompted to pinpoint specific code segments responsible for the malicious behavior and to generate corresponding natural language explanations that justify the classification. These explanations are intended to assist security analysts in assessing the script's malicious intent. To systematically assess the accuracy and relevance of the code segments identified by LLMs, we perform an ablation-style evaluation, in which the identified code segments are removed (masked) from the original script, and the modified script is submitted to VirusTotal for re-analysis. A substantial reduction in the VirusTotal community score following the removal of these segments indicates that the identified code segments likely contribute to the script's malicious functionality. In addition, we compute BERTScore to evaluate how closely LLM-generated explanations align with VirusTotal reference explanations, and assess how different prompt engineering strategies affect overall model effectiveness.

We conducted experiments to evaluate the performance of five LLMs deployed locally on our university servers: GPT-OSS-20B, Llama 3.3, Qwen-2.5-

Coder-32B, Mixtral-8×22B, and Gemma 3-27B. To assess output consistency, each experiment was repeated three times and the results were averaged. The evaluation reveals substantial performance variation across models. Overall, the performance of each LLM is highly consistent across runs despite the inherent stochasticity of LLM inference, although some variability is observed in individual cases.

Our contributions are summarized as follows:

- We investigate the use of LLMs for detecting malicious PowerShell scripts, identifying code segments responsible for malicious behavior, and generating human-readable explanations of the detection results. To the best of our knowledge, this is the first study to evaluate the application of LLMs in this context.
- We propose an ablation-based evaluation approach to assess LLMs' accuracy in localizing malicious code segments. This approach involves masking the identified segments and measuring the resulting change in the community score on VirusTotal.
- We conduct an evaluation on five LLMs and our results suggest significant variation in performance of different LLMs.

Organization: The rest of this paper is organized as follows. Section 2 provides background information on PowerShell and fileless malware, VirusTotal, and the SHAP explainer. Section 3 presents our evaluation framework, including the datasets as well as the model and input design. Section 4 presents the experimental setup, evaluation metrics, and results. Section 5 discusses related work, and Section 6 concludes the paper and outlines future research directions.

2 Background

PowerShell and Fileless Malware: PowerShell is a task-based command-line shell and scripting language built on the .NET framework [22]. Its support for cmdlets, pipelines, modules, and remote execution makes it a powerful tool for system administration and automation, but also an attractive target for attackers. Fileless malware refers to malicious software that operates entirely in memory, avoiding file-based payloads [31]. By leveraging legitimate system tools such as PowerShell, Windows Management Instrumentation (WMI), or the Windows Registry, attackers can execute malicious actions without leaving clear traces on disk. These "living-off-the-land" (LotL) techniques make fileless attacks difficult to detect using conventional signature-based defenses.

VirusTotal: VirusTotal [34] is a widely used platform that aggregates malware detection results from over 70 antivirus engines, website scanners, sandbox environments, and other analytical tools. Among the metrics provided, VirusTotal computes a community score, which is the proportion of security vendors that classify a given sample as malicious. Formally, if a sample is analyzed by N

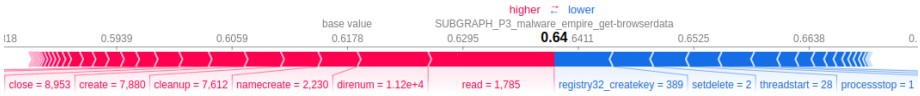

Fig. 1. Local SHAP explanation example.

vendors and k of them flag it as malicious, the community score is defined as $\frac{k}{N}$. For ease of analysis, we express this ratio as a percentage. For instance, if $k = 6$ and $N = 60$, the score is $\frac{6}{60} = 10\%$. A higher community score indicates greater consensus among vendors regarding the maliciousness of the sample. It is worth noting that VirusTotal scores do not constitute a perfect ground truth and may reflect vendor-specific biases. However, it is one of the most widely used and practical evaluation tools available to the security community. Limitations, including vendor bias and the lack of definitive ground truth, are common challenges in security evaluations.

SHAP explainer: SHAP (SHapley Additive Explanations) [20] is a framework for interpreting the predictions of machine learning models. It is based on Shapley values from cooperative game theory, which assigns each feature an importance value that reflects its influence on an individual prediction. Figure 1 gives a local SHAP explanation for a malicious PowerShell script sample produced using the machine learning model in [7] based on ETW features. The model's output for this sample can be decomposed as

$$f(x) \;=\; \phi_0 \;+\; \sum_{i=1}^{M} \phi_i,$$

where

- $f(x)$ is the "prediction value", i.e. the model's output probability for the sample, obtained by adding all feature contributions to the "base value".
- ϕ_0 is the "base value", which is the model's average output probability for the malicious class over the background dataset when no feature information is available.
- ϕ_i is the SHAP value of the i-th feature, indicating how much that feature pushes the prediction up (if $\phi_i > 0$) or down (if $\phi_i < 0$) relative to the "base value".

In this example, the base value is $\phi_0 = 0.6178$ and the prediction is $f(x) = 0.64$. Because the prediction exceeds the default classification threshold of 0.5, the script is labeled as malicious. Each arrow in the figure represents a feature, with length proportional to its SHAP value. The feature "read" has the highest value, indicating it is the most influential factor in this classification.

SHAP also provides a global summary to capture feature importance across multiple predictions. Figure 2 shows a global SHAP summary, which captures

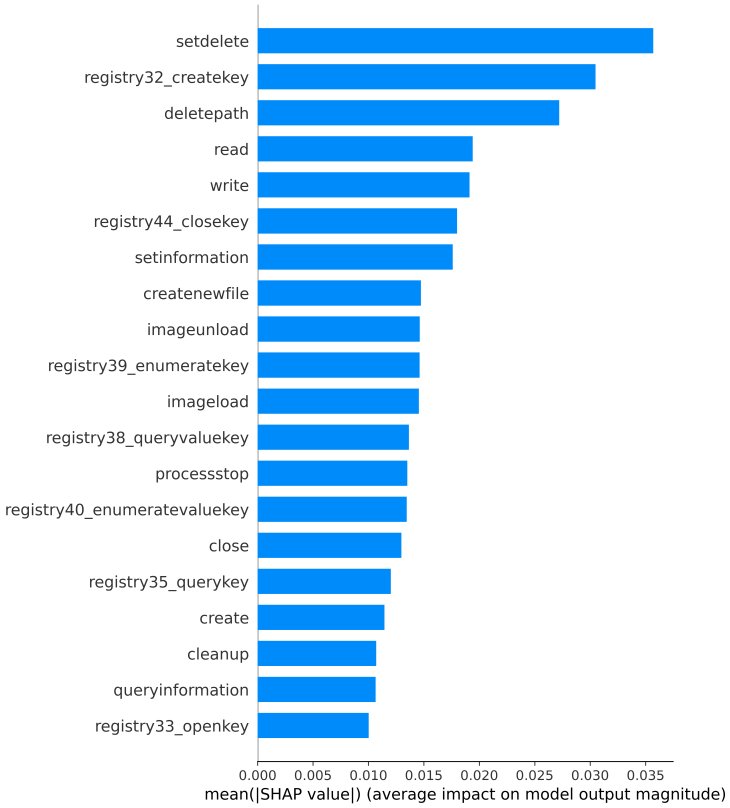

Fig. 2. An example of global SHAP explanation.

feature importance across many predictions. The horizontal axis represents SHAP values, and the vertical axis lists features in descending order of overall impact. The feature "setdelete" has the highest mean SHAP value, indicating that it plays the most significant role in the model's general decision-making process.

It is worth noting that SHAP produces numeric, model-centric outputs that quantify how features influence predictions, which often require domain expertise or familiarity with model interpretability techniques to interpret effectively. In contrast, LLMs generate textual, human-readable explanations that are easier for non-experts to understand.

3 Evaluation Framework

We propose an evaluation framework to assess how LLMs perform in detecting PowerShell malware and generating interpretable explanations for their detections. Figure 3 illustrates the pipeline of our framework. First, we establish a baseline for each malicious script through its community score retrieved from

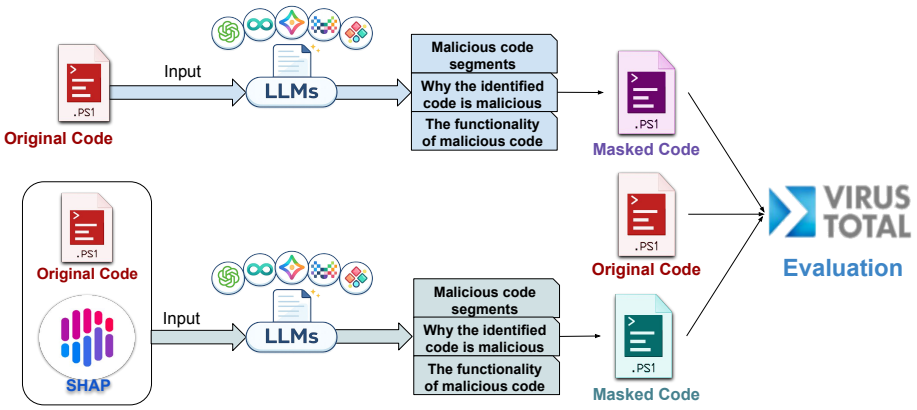

Fig. 3. An overview of our evaluation framework.

VirusTotal. Next, we prompt LLMs to classify PowerShell scripts as either benign or malicious. We then use a multi-step prompting strategy to further task the LLMs with pinpointing the specific code segments responsible for the malicious behavior and generating natural language explanations for each segment. To quantitatively validate the accuracy of these localizations, we measured the change in VirusTotal's community score after masking (removing) the identified segments from the original script. A significant reduction in the community score after masking suggests that the identified segments likely contribute to the script's malicious activities. By pinpointing malicious code segments and generating natural language explanations of their behavior, our framework enables security analysts to verify malicious intent through human-readable insights, potentially reducing the manual effort and time required for analysis.

3.1 Datasets

We used PowerShell scripts from the test dataset in [35]. To ensure consistent inputs to the LLMs, comments were removed from each script to prevent the models from relying on them for malware detection. Blank lines were also removed to allow accurate computation of the proportion of malicious code segments identified by the LLMs.

3.2 Model and Input Design

We conducted two types of experiments: (1) LLMs were provided only with the raw PowerShell script, and (2) LLMs were provided with the script, SHAP explainer results from the machine learning model in [7], as well as textual descriptions of the ETW features in the SHAP results. The SHAP explainer quantifies and ranks the importance of behavioral features extracted from ETW

logs. Each script in our dataset has its own SHAP output, specifying the relative importance of 4-gram events in predicting maliciousness. For each script, we selected the top 30 most important 4-gram events based on SHAP values as prompt inputs. We also identified the top 10 globally important 4-gram events across the test set, representing the events most consistently associated with malicious behavior. These events were provided to the LLMs to help them focus on key indicators of malicious activity.

We propose a multistep prompting strategy to guide LLMs in detecting malicious code segments and generating explanations of their behavior. Instead of combining all instructions into a single prompt, we structure the interaction as a sequence of prompts. This modular design breaks the overall task into smaller, more manageable steps, allowing the LLMs to focus on one aspect at a time, avoid missing important details, and reduce hallucinations. We incorporate prompt engineering techniques such as one-shot prompting and Chain-of-Thought (CoT) reasoning. Each LLM interaction involves a sequence of prompts, summarized below. Prompts 1-6 are used when SHAP explainer results are included; otherwise, prompts 2 and 3 are omitted. This strategy is referred to as the "Mask-all-malicious-code" strategy.

- **Prompt 1**: Define the LLM's role explicitly as a security researcher and analyst.
- **Prompt 2 (with SHAP explainer results only)**: Provide the SHAP explainer results along with descriptions of the features appearing in the SHAP output, including their associated ETW events, providers, and relevant attributes.
- **Prompt 3 (with SHAP explainer results only)**: Provide SHAP explainer results, explicitly listing both script-specific and globally significant malicious 4-gram events.
- **Prompt 4**: Provide the PowerShell script along with a representative example illustrating the expected format for identifying malicious code segments.
- **Prompt 5**: Prompt the LLM to clearly explain the functionality and malicious behavior of the identified malicious code segments.
- **Prompt 6**: Request additional insights or observations regarding the script's malicious intent, behaviors, or potential impact.

We also employed an additional prompt engineering strategy in which LLMs are instructed to assign a confidence label "high", "medium", or "low", to each identified code segment. The models are then prompted to return only segments labeled as high confidence. This approach aims to reduce the size of identified code segments, which enables security analysts to focus on the most relevant malicious behaviors. We refer to this method as the "Mask-high-confidence-code" strategy.

Figure 4 gives an example output from Llama 3.3-70B, in which the LLM identifies potentially malicious code segments and provides an explanation of their functionality.

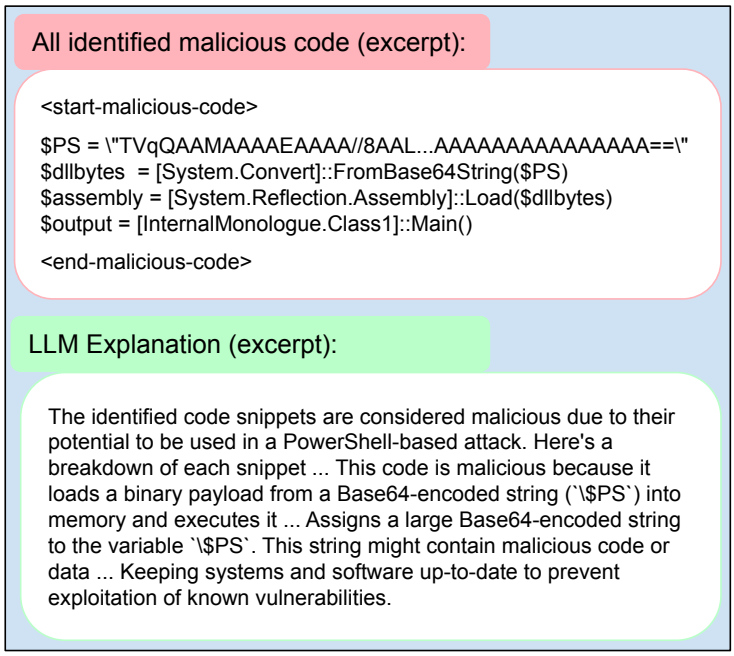

Fig. 4. An example of malicious code segment identified by the LLM.

4 Experiments

This section presents our experimental setup, evaluation metrics, and results. We evaluated five LLMs installed locally on servers at Binghamton University: GPT-OSS-20B, Llama 3.3, Qwen-2.5-Coder-32B, Mixtral-8×22B, and Gemma 3-27B. The APIs for these models are available at no cost, allowing unlimited runs during our evaluation. Each model follows the same multi-step instruction described in Section 3.2. To assess the consistency of LLM outputs, each experiment was repeated three times, and the results were averaged.

All models were accessed via open-source inference frameworks. Wall-clock inference time was recorded for each model to evaluate runtime efficiency. All experiments were conducted without fine-tuning, using the default model weights and inference settings provided by the API service. Default decoding parameters include a temperature of 0.8, top-p of 0.9, and top-k of 40.

The evaluated LLMs are detailed as follows:

- GPT-OSS-20B: An open-weight LLM released by OpenAI and is available for local deployment.
- Llama 3.3 (70B): Developed by Meta AI, this model is intended for general-purpose reasoning tasks.

- Qwen-2.5-Coder-32B: A code-specialized LLM released by Alibaba DAMO Academy. We use the 32B instruction-tuned variant designed for code understanding and generation tasks.
- Mixtral-8×22B: A mixture-of-experts model from Mistral AI, which consists of multiple expert networks that are selectively activated during inference and is designed to achieve high performance with efficient inference.
- Gemma 3-27B: An open-source model released by Google, designed for general-purpose language understanding and reasoning.

To automate and standardize the process, we developed a suite of Python scripts to perform the following tasks: [4]

- API Client: Sends the generated prompts to LLMs' APIs and collects their responses.
- Malicious Code Masker: Parses LLM responses to identify malicious code segments and removes these segments from the scripts.
- VirusTotal Client: Submits the original and masked scripts to VirusTotal and retrieves their community scores.

4.1 Evaluation Metrics

We define the following metrics to quantitatively assess the performance of LLMs.

- **Detection accuracy:** This metric assesses script-level detection performance by measuring the proportion of benign and malicious scripts correctly classified by LLMs. We compute true positives, true negatives, false positives, and false negatives, and report the overall detection accuracy and F1 scores.
- **Community score reduction:** Measures the change in VirusTotal's community score before and after masking malicious code segments identified by LLMs. A substantial score reduction indicates that the LLM likely identifies the malicious portions of the script. For example, if the community score decreases from 80% to 40%, this means that 40% fewer vendors classify the script as malicious after the identified segments are removed. Note that computing a relative reduction rate (e.g., $(80-40)/80$) is not meaningful in this context, because the VirusTotal community score represents the proportion of vendors labeling the file as malicious rather than a continuous measure of maliciousness. Therefore, the absolute change in percentage more accurately reflects how vendor classifications shift after the identified code segments are removed.
- **Benign conversion rate:** A masked script is considered benign once its community score reaches zero. This metric reflects the percentage of scripts that are successfully "converted" from malicious to benign through masking.
- **Percentage of masked lines:** Represents the proportion of lines removed (masked) in the script. Lower percentages, combined with substantial score reductions, indicate more precise localization of malicious behavior.

[4] All datasets and code used for experiments, including prompt construction and parsing procedures, are available at: https://github.com/mwangz/LLMs-XAI

Table 1. Comparison with graph-based baselines.

Method	Accuracy (%)	F1 (%)
Graphite (4-gram)	87.7	86.3
FEATHER-GRAPH	64.8	61.9
Graph2Vec	60.0	58.0
GAT	60.07	53.39
GIN	60.4	56.7
GPT-OSS-20B	88.64	90.16
Llama 3.3-70B	91.67	92.93
Mixtral-8×22B	90.15	91.53
Qwen-2.5-Coder-32B	91.16	91.85
Gemma 3-27B	84.09	86.17

4.2 Evaluation Results

We used 200 benign and 196 malicious PowerShell scripts from the test dataset in [35] for our evaluation. These scripts were collected from publicly available GitHub repositories. Although [35] contains a total of 949 malicious PowerShell scripts, SHAP values are available only for the test dataset.

Detection accuracy: Table 1 compares the accuracy and F1 score of LLM-based detection against graph-based methods [35], including Graphite (4-gram), FEATHER-GRAPH, Graph2Vec, GAT, and GIN. As shown in Table 1, most LLM-based models outperform the graph-based methods in both metrics. Llama 3.3-70B achieves the best overall performance, with 91.67% accuracy and a 92.93% F1 score. Mixtral-8×22B (90.15% accuracy) and Qwen-2.5-Coder-32B (91.6% accuracy) also demonstrate strong performance.

In terms of error rates, Mixtral-8×22B achieves the lowest false-positive rate (FPR) at 1.52%. This is followed by Llama 3.3-70B and Qwen-2.5-Coder-32B at 3.03% , GPT-OSS-20B at 6.06%, and Gemma 3-27B at 28.14%. Regarding the false-negative rate (FNR), Gemma 3-27B achieves the lowest value at 2.56%. Llama 3.3-70B, Qwen-2.5-Coder-32B, GPT-OSS-20B, and Mixtral-8×22B obtain FNRs of 10.47%, 12.44%, 12.70%, and 14.29%, respectively. These results highlight a trade-off between false positives and false negatives across models. For example, Mixtral-8×22B achieves the lowest FPR, indicating stronger robustness against false alarms. Conversely, Gemma 3-27B produces the lowest FNR but at the cost of a substantially higher FPR, suggesting more aggressive detection behavior.

We also analyzed the scripts on which most LLMs did not perform well and identified several contributing factors. First, large script size can negatively affect model performance because large inputs increase reasoning complexity and may dilute the model's attention across many benign operations. As a result, the model may fail to accurately localize the malicious segments. Second, in several scripts, the malicious activities occurred near the end of the code, which compli-

Table 2. VirusTotal results for masked PowerShell scripts, with highlights indicating the best performance.

Inputs	Models	Mask-all-malicious-code			Mask-high-confidence-code		
		Community Score Reduction	Benign Conversion Rate	Percentage of Masked Lines	Community Score Reduction	Benign Conversion Rate	Percentage of Masked Lines
Code-only	**GPT-OSS**	27.6%	26.2%	29.4%	25.5%	20.4%	23.5%
	Llama 3.3	26.2%	13.1%	14.7%	23.5%	7.7%	7.3%
	Qwen-2.5	25.0%	15.8%	15.8%	21.2%	7.1%	6.6%
	Mixtral	25.8%	15.1%	20.8%	22.4%	8.3%	11.2%
	Gemma 3	23.3%	15.1%	15.9%	21.2%	7.7%	8.1%
Code+SHAP	**GPT-OSS**	24.1%	19.7%	19.9%	22.6%	17.5%	17.1%
	Llama 3.3	25.9%	13.3%	17.1%	23.1%	8.2%	7.9%
	Qwen-2.5	24.0%	13.3%	15.4%	20.5%	6.3%	7.1%
	Mixtral	23.0%	12.2%	15.0%	21.1%	8.8%	9.7%
	Gemma 3	23.7%	11.1%	16.6%	20.7%	6.0%	7.5%

cates the detection as the model has to process long sequences of benign operations before encountering the malicious behavior. In addition, 4 scripts are misclassified as benign by all evaluated LLMs. These samples have an average original community score of 18.1%, compared to 40.3% over the full dataset, which shows that significantly fewer venders consider them as malicious. Inspection of these scripts indicates that some perform information-gathering operations, including retrieving user account and privilege details (e.g., Get-UserInfo) and network configuration data (e.g., Get-SystemDNSServer). Other scripts modify file metadata such as creation, access, and modification timestamps (e.g., Set-MacAttribute), or capture screenshots (e.g., `Get-TimedScreenshot`). While such functionalities are commonly associated with anti-forensic techniques, they can also serve legitimate purposes such as file synchronization or system maintenance.

Community score reduction: Table 2 shows the reduction in community scores after masking the malicious code segments identified by the five LLMs. As a baseline, the average VirusTotal Community Score of the original malicious scripts is 40.3%. For each LLM, the experiment was repeated three times, and the standard deviation across runs was computed for all evaluation metrics. In the table, "Code-only" refers to prompts containing only the PowerShell scripts, while "Code+SHAP" refers to prompts that include the scripts, and SHAP-based explanations. Two prompt engineering strategies described in Section 3.2 are evaluated: Mask-all-malicious-code and Mask-high-confidence-code.

Table 2 shows that, without SHAP explanations, masking malicious segments identified by all LLMs leads to a substantial reduction in community scores, ranging from 23.3% to 27.6%. Among the models, Llama 3.3 performed best overall, achieving the second-highest reduction in community score (1.4% lower than GPT-OSS-20B) while masking the smallest proportion of code. Although GPT-OSS-20B achieved the highest score reduction, it did so by masking a significantly larger portion of the code (14.7% more lines than Llama 3.3).

In terms of Benign Conversion Rate, GPT-OSS-20B achieved the highest value at 26.2%. This indicates that GPT-OSS-20B successfully identified all malicious code segments in 26.2% of malicious scripts. In addition, we observed that for more than half of these scripts, over 20 vendors had flagged them as malicious before GPT-OSS-20B masked the identified portions.

When prompted to identify only high-confidence malicious code segments, all LLMs identified a smaller portion of the code with higher community scores. In particular, Mixtral-8×22B shows the largest decrease in the percentage of code identified (a 9.6% reduction), accompanied by a 3.4% drop in the community score and a 6.9% drop in benign conversion rate. These results suggest that the smaller malicious segments identified with high confidence may not fully capture the complete malicious behavior of the code. In other words, high-confidence predictions may capture the most obvious indicators, while additional supporting behaviors may remain in the script, which enables some security vendors to continue classifying it as malicious.

Table 2 shows that combining SHAP explanations slightly improves community score reduction for Gemma 3-27B but reduces performance for the other models. In terms of benign conversion rate, Llama 3.3-70B shows improvement, whereas other models show decreased performance. A potential reason is that SHAP features are derived from a different machine learning model trained on event-based features (4-gram ETW events), whereas the LLMs primarily reason about source code semantics. This mismatch between event-level features and code-level reasoning may introduce noise into the prompt. Another possible reason is that the additional SHAP information increases prompt complexity, which may dilute the model's attention from the most relevant parts of the script. Prompt optimization techniques such as DsPy [15] may help improve the robustness of the reasoning process.

Consistency across runs: Each experiment was repeated three times, and the results were analyzed for consistency. We computed per-sample standard deviations across runs (i.e., the standard deviation of each sample across three runs for each configuration is computed and then averaged). The standard deviations are very small: 0.059 for GPT-OSS-20B, 0.031 for Llama 3.3, 0.042 for Mixtral-8×22B, and 0.035 for both Qwen-2.5-Coder-32B and Gemma 3-27B. This indicates that the models produce highly consistent results on individual samples despite the inherent stochasticity of LLM inference, though occasional deviations were observed in a small subset of samples.

For each LLM, we also calculated the average community score within a single run, computed the standard deviation of this average across the three runs,

and reported the average of this standard deviation over all configurations. The results show very low variability across runs, with average standard deviations of 0.007 for GPT-OSS-20B, 0.0029 for Llama 3.3, 0.0038 for Qwen-2.5-Coder-32B, 0.0036 for Mixtral-8×22B, and 0.0046 for Gemma 3-27B. This further confirms that the models produce highly consistent overall results across runs.

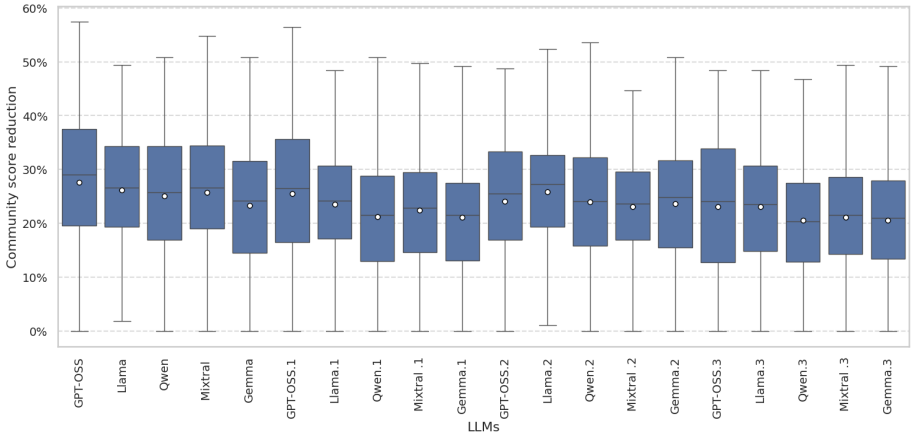

Fig. 5. Distribution of community score reduction. `LLM`: code-only with Mask-all-malicious-code strategy. `LLM.1`: code-only with Mask-high-confidence-code strategy. `LLM.2`: code+SHAP with Mask-all-malicious-code strategy. `LLM.3`: code+SHAP with Mask-high-confidence-code strategy.

Distribution of community score reduction: Figure 5 shows the distribution of community score reduction for each LLM using boxplots. The central line within each box indicates the 50th percentile (median) of the score reduction. A high median reduction means that in most cases, masking the LLM-identified segments reduces the VirusTotal detection score. A small white dot in the box indicates the mean. The top and bottom edges of the box correspond to the 75th percentile (Q_3) and the 25th percentile (Q_1), respectively. This means that the box spans the interquartile range IQR $= Q_3 - Q_1$, which captures the middle 50% of the values. The whiskers extend to the most extreme data points within $1.5 \times$ IQR above Q_3 and below Q_1.

The x-axis in the figure indicates both the LLM used and the corresponding prompt strategy and masking configuration. The figure shows that GPT-OSS-20B and Llama 3.3 exhibit higher median reductions, indicating that they generally achieve larger community score reductions across scripts. GPT-OSS-20B has a slightly higher median than Llama 3.3, but also a larger interquartile range (IQR), suggesting that its performance varies more across scripts. Gemma 3-27B has the lowest median reduction among the evaluated models. In most configurations, the mean is slightly lower than the median, indicating that the LLMs

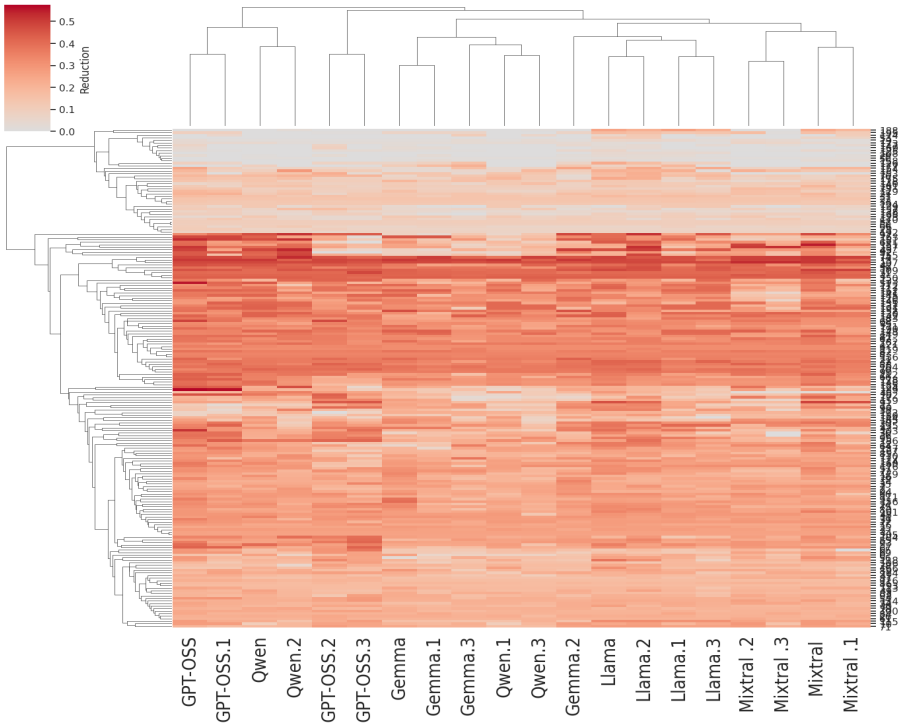

Fig. 6. Clustered heatmap of community score reduction. `LLM`: code-only with Mask-all-malicious-code strategy. `LLM.1`: code-only with Mask-high-confidence-code strategy. `LLM.2`: code+SHAP with Mask-all-malicious-code strategy. `LLM.3`: code+SHAP with Mask-high-confidence-code strategy.

achieve weaker reductions on a small number of scripts. In addition, the lower whiskers of the boxplots extend to nearly 0% for most models. Our analysis shows that, for each model, there is at most one script for which masking the LLM-identified code does not reduce the VirusTotal community score. Moreover, this script differs across models, suggesting that individual LLMs may occasionally fail to identify certain malicious components.

Figure 6 presents a clustered heatmap of community score reduction for five LLMs under two masking strategies, evaluated with and without SHAP. Hierarchical clustering is applied to both axes - scripts (rows) and model-prompt combinations (columns), to reveal performance patterns. Models exhibiting darker color intensities across a large fraction of scripts indicate stronger community score reduction, whereas lighter colors indicate weaker reduction. GPT-OSS-20B and Llama 3.3 show darker columns overall, indicating stronger reductions, while Gemma 3 exhibits lighter colors, reflecting weaker performance. Mixtral-8×22B shows mixed color intensity, suggesting more variable performance across scripts.

Table 3. Average BERTScore-F1 for each LLM.

Model	Code-only	Code+SHAP
GPT-OSS	82.5%	83.4%
Llama 3.3	84.1%	84.5%
Qwen-2.5	83.9%	83.5%
Mixtral	84.4%	83.9%
Gemma 3	83.5%	84.1%

The heatmap shows that all Llama 3.3 configurations cluster closely, regardless of SHAP explanations or masking strategy. This suggests that Llama 3.3 exhibits similar success and failure patterns across scripts, indicating that the model itself has a stronger impact on community score reduction than SHAP or masking. Similar within-model clustering is observed for Mixtral. In contrast, GPT-OSS-20B's configurations form separate clusters depending on the use of SHAP, indicating that these additions can change reduction in different directions rather than uniformly. A similar split is observed for Qwen-2.5-Coder-32B under different masking strategies.

LLM Explanation BERTScore: We compute BERTScore [39] to evaluate how closely LLM-generated explanations align with VirusTotal reference explanations. BERTScore is a semantic similarity metric ranging from 0 to 1, which measures the similarity between two texts based on contextualized token embeddings, with higher scores indicating greater similarity between the generated explanations and the reference descriptions.

As shown in Table 3, all evaluated LLMs achieve consistently high BERTScore-F1 values, ranging from 82.5% to 84.5%. These results suggest that the generated explanations are semantically similar to the VirusTotal reference explanations, indicating a high degree of alignment between the model outputs and the reference descriptions.

LLM inference time and scalability: LLM inference time is influenced by multiple factors, including server location, available computational resources, network latency, model size, system load, and prompt length. As a result, response times can vary significantly across different deployment environments. In this work, the average input size per sample is approximately $15,000$ tokens, while the output size is typically under $40,000$ tokens. All models in our experiments were run locally on university servers. In our experiments, the average inference time for all LLMs was under 94 seconds per sample. Among them, the Mixtral-8×22B model had the shortest average inference time, approximately 63 seconds per sample.

In terms of scalability, since multiple scripts can run in parallel, our approach can easily scale, provided the server hosting the LLMs has sufficient resources.

Paired t-test results: We performed paired t-tests to evaluate the statistical significance of changes in community scores following the removal of malicious

Table 4. Paired t-test results for community score reduction.

Inputs	Models	Mask-all		Mask-high	
		t-stat	p-value	t-stat	p-value
Code-only	**GPT-OSS**	-27.73	1.40×10^{-70}	-26.18	1.13×10^{-66}
	Llama 3.3	-29.85	1.01×10^{-75}	-28.13	1.43×10^{-71}
	Qwen-2.5	-27.04	7.44×10^{-69}	-24.41	4.75×10^{-62}
	Mixtral	-27.44	7.36×10^{-70}	-26.61	8.84×10^{-68}
	Gemma 3	-25.65	2.68×10^{-65}	-24.36	6.54×10^{-62}
Code+ SHAP	**GPT-OSS**	-26.55	1.25×10^{-67}	-24.13	2.54×10^{-61}
	Llama 3.3	-29.79	1.43×10^{-75}	-25.79	1.14×10^{-65}
	Qwen-2.5	-25.35	1.56×10^{-64}	-24.53	2.24×10^{-62}
	Mixtral	-25.31	2.01×10^{-64}	-23.82	1.75×10^{-60}
	Gemma 3	-26.02	2.96×10^{-66}	-24.88	2.71×10^{-63}

code identified by the LLMs. Table 4 summarizes the results. In the table, the t-statistic (t-stat) measures both the magnitude and direction of the observed score changes, while the associated p-value indicates the likelihood that these differences could have occurred by random chance. As shown in Table 4, all evaluated models produced statistically significant reductions in community scores ($p \ll 0.0001$), indicating that the observed reductions are unlikely to be random and instead reflect the impact of removing the code flagged by the models.

5 Related work

This section reviews related work in the literature.

Malicious PowerShell detection: Mimura et al. proposed a static machine learning based detection method for malicious PowerShell based on word embeddings [23]. Tajiri et al. [32] proposed a method using word-level language models to classify unknown PowerShell scripts without dynamic analysis. Li et al. proposed PowerPeeler, a dynamic analysis approach that uses Abstract Syntax Trees (ASTs) to identify and reconstruct obfuscated script segments via instruction-level execution tracking [17]. Rubin et al. leveraged deep learning with contextual embeddings from unlabeled PowerShell scripts, fine-tuned on AMSI data, to improve detection performance [29]. Choi applied attention-based LSTM models to restore obfuscated PowerShell data, addressing the issue of malicious actors using generative adversarial networks to avoid AI-based detection [4]. Hendler et al. developed malicious PowerShell command detectors based on NLP and convolutional neural networks, and concluded that a hybrid approach combining both yielded the best results [8]. Fang et al. proposed a model that embeds PowerShell scripts with FastText and classifies them using a Random Forest [6]. Tsai et al. introduced PowerDP, a framework to predict obfuscated malicious code using character distribution and profile code behaviors via extracted features from the

AST for multi-label classification [33]. However, none of them leveraged LLMs for malicious PowerShell script detection. While these works show that machine learning and deep learning are effective for PowerShell malware detection, they focus solely on detection and classification, without addressing malicious code localization or interpretability. In contrast, our work leverages LLMs to detect malicious scripts, localize code segments, and provide interpretable, natural language explanations.

LLM for code understanding and security: Prior studies have shown that LLMs can generate code [36], correct programming errors [12], and produce accurate explanations of code behavior [16]. They have also been found to outperform traditional search engines when answering developer questions about unfamiliar code [26], and can refactor code syntax while preserving its functionality [25]. Building on this foundation, recent research has explored applying LLMs to a range of security tasks, including static code analysis [18,19], anomaly detection [14], threat intelligence [10] and malware detection [27]. Some approaches integrate LLMs into broader analysis pipelines to classify Android applications [40], while others use LLMs alone to classify Java code blocks [9]. To the best of our knowledge, we are the first to leverage LLMs and ablation-based evaluation for malicious PowerShell script detection and explanation.

6 Conclusions and Future Work

This paper investigates the use of LLMs to detect malicious PowerShell scripts, identify code segments responsible for malicious behavior, and generate human-readable explanations to support analyst interpretation. We used an ablation-based evaluation approach to assess the accuracy of LLMs in identifying malicious code segments by measuring changes in VirusTotal community scores. Our results show that LLMs can identify and explain malicious components, but the performance varies across models, and the addition of external knowledge does not necessarily improve results.

Our preliminary analysis indicates that while some scripts in our dataset are obfuscated, LLMs exhibit a degree of capability in analyzing such code. As future work, we plan to conduct a comprehensive study of LLM robustness to intentionally obfuscated code and prompt injection attacks. Furthermore, as discussed in Section 4.2, we identified two key factors that degrade model performance: (1) large script size and (2) malicious activities appearing near the end of scripts. One potential solution is to partition large scripts into smaller, logical segments for iterative analysis, or to employ a sliding window strategy that enables the model to process manageable portions of code while maintaining contextual information across segments.

Another limitation of our current approach is the use of fixed prompt templates for all samples. This static prompting strategy prevents the model from adapting its input based on intermediate reasoning steps or its own confidence levels. Future work could explore adaptive prompting techniques and Retrieval-Augmented Generation (RAG) to dynamically adjust both the amount and type

of contextual information provided. In addition, BERTScore evaluates semantic similarity but does not fully capture factual correctness. Future work will incorporate human expert evaluation to assess the quality of explanations produced by LLMs.

Acknowledgement: This work is supported in part by a SUNY-IBM AI Research Alliance grant, and NSF grants 2146212 and 2153056. We thank the anonymous reviewers for their constructive comments.

References

1. Ali, T., Kostakos, P.: HuntGPT: Integrating machine learning-based anomaly detection and explainable AI with large language models (LLMs). arXiv preprint arXiv:2309.16021 (2023)
2. Cakir, B., Dogdu, E.: Malware classification using deep learning methods. In: Proceedings of the 2018 ACM Southeast Conference. pp. 1–5 (2018)
3. Carvalho, J., Zhang, M., Geyer, R., Cotrini, C., Buhmann, J.M.: Invariant anomaly detection under distribution shifts: a causal perspective. Advances in Neural Information Processing Systems **36**, 56310–56337 (2023)
4. Choi, S.: Malicious PowerShell detection using attention against adversarial attacks. Electronics **9**, 1817 (11 2020). https://doi.org/10.3390/electronics9111817
5. Deng, G., Liu, Y., Mayoral-Vilches, V., Liu, P., Li, Y., Xu, Y., Zhang, T., Liu, Y., Pinzger, M., Rass, S.: {PentestGPT}: Evaluating and harnessing large language models for automated penetration testing. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 847–864 (2024)
6. Fang, Y., Zhou, X., Huang, C.: Effective method for detecting malicious powershell scripts based on hybrid features. Neurocomputing **448**, 30–39 (2021). https://doi.org/https://doi.org/10.1016/j.neucom.2021.03.117, https://www.sciencedirect.com/science/article/pii/S0925231221005099
7. Gwak, J.Y., Wakodikar, P., Wang, M., Yan, G., Shu, X., Stoller, S.D., Yang, P.: Debugging malware classification models based on event logs with explainable ai. In: 2023 IEEE International Conference on Data Mining Workshops (ICDMW). pp. 939–948. IEEE (2023)
8. Hendler, D., Kels, S., Rubin, A.: Detecting malicious powershell commands using deep neural networks (2018), https://arxiv.org/abs/1804.04177
9. Hossain, A.A., PK, M.K., Zhang, J., Amsaad, F.: Malicious code detection using LLM. In: NAECON 2024 - IEEE National Aerospace and Electronics Conference. pp. 414–416 (2024). https://doi.org/10.1109/NAECON61878.2024.10670668
10. Hu, Y., Zou, F., Han, J., Sun, X., Wang, Y.: Llm-tikg: Threat intelligence knowledge graph construction utilizing large language model. Computers & Security **145**, 103999 (2024)
11. Jiang, M., Han, S., Huang, H.: Anomaly detection with score distribution discrimination. In: Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. pp. 984–996 (2023)
12. Jiang, N., Li, X., Wang, S., Zhou, Q., Hossain, S.B., Ray, B., Kumar, V., Ma, X., Deoras, A.: LeDex: Training LLMs to better self-debug and explain code. In: Globerson, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J., Zhang, C. (eds.) Advances in Neural Information Processing Systems. vol. 37, pp. 35517–35543. Curran Associates, Inc. (2024), https://proceedings.neurips.cc/paper_files/paper/2024/file/3ea832724870c700f0a03c665572e2a9-Paper-Conference.pdf

13. Kalash, M., Rochan, M., Mohammed, N., Bruce, N.D., Wang, Y., Iqbal, F.: Malware classification with deep convolutional neural networks. In: 2018 9th IFIP international conference on new technologies, mobility and security (NTMS). pp. 1–5. IEEE (2018)
14. Karlsen, E., Luo, X., Zincir-Heywood, N., Heywood, M.: Benchmarking large language models for log analysis, security, and interpretation. J. Netw. Syst. Manag. **32**(3) (Jul 2024)
15. Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., Haq, S., Sharma, A., Joshi, T.T., Miller, H., Zaharia, M., Potts, C.: Dspy: Compiling declarative language model calls into self-improving pipelines. In: International Conference on Learning Representations (ICLR) (2024), https://arxiv.org/abs/2310.03714
16. Leinonen, J., Denny, P., MacNeil, S., Sarsa, S., Bernstein, S., Kim, J., Tran, A., Hellas, A.: Comparing code explanations created by students and large language models. In: Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1. p. 124–130. ITiCSE 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3587102.3588785, https://doi.org/10.1145/3587102.3588785
17. Li, R., Zhang, C., Chai, H., Ying, L., Duan, H., Tao, J.: Powerpeeler: A precise and general dynamic deobfuscation method for powershell scripts (2024), https://arxiv.org/abs/2406.04027
18. Li, Z., Dutta, S., Naik, M.: Iris: Llm-assisted static analysis for detecting security vulnerabilities (2025), https://arxiv.org/abs/2405.17238
19. Liu, P., Liu, J., Fu, L., Lu, K., Xia, Y., Zhang, X., Chen, W., Weng, H., Ji, S., Wang, W.: Exploring ChatGPT's capabilities on vulnerability management. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 811–828 (2024)
20. Lundberg, S.M., Lee, S.I.: A unified approach to interpreting model predictions. Advances in neural information processing systems **30** (2017)
21. Masum, M., Faruk, M.J.H., Shahriar, H., Qian, K., Lo, D., Adnan, M.I.: Ransomware classification and detection with machine learning algorithms. In: 2022 IEEE 12th annual computing and communication workshop and conference (CCWC). pp. 0316–0322. IEEE (2022)
22. Microsoft: PowerShell Documentation (2025), https://learn.microsoft.com/powershell/, [Online; accessed 10-May-2025]
23. Mimura, M., Tajiri, Y.: Static detection of malicious PowerShell based on word embeddings. Internet of Things **15**, 100404 (2021). https://doi.org/https://doi.org/10.1016/j.iot.2021.100404, https://www.sciencedirect.com/science/article/pii/S2542660521000482
24. Mirsky, Y., Doitshman, T., Elovici, Y., Shabtai, A.: Kitsune: an ensemble of autoencoders for online network intrusion detection. arXiv preprint arXiv:1802.09089 (2018)
25. Mohseni, S., Mohammadi, S., Tilwani, D., Saxena, Y., Ndawula, G.K., Vema, S., Raff, E., Gaur, M.: Can LLMs obfuscate code? a systematic analysis of large language models into assembly code obfuscation. Proceedings of the AAAI Conference on Artificial Intelligence **39**(23), 24893–24901 (Apr 2025). https://doi.org/10.1609/aaai.v39i23.34672, https://ojs.aaai.org/index.php/AAAI/article/view/34672
26. Nam, D., Macvean, A., Hellendoorn, V., Vasilescu, B., Myers, B.: Using an LLM to help with code understanding. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3597503.3639187, https://doi.org/10.1145/3597503.3639187

27. Patsakis, C., Casino, F., Lykousas, N.: Assessing llms in malicious code deobfuscation of real-world malware campaigns. Expert Systems with Applications **256**, 124912 (2024)
28. Perini, L., Davis, J.: Unsupervised anomaly detection with rejection. Advances in Neural Information Processing Systems **36**, 69673–69691 (2023)
29. Rubin, A., Kels, S., Hendler, D.: Amsi-based detection of malicious powershell code using contextual embeddings (2019), https://arxiv.org/abs/1905.09538
30. Shone, N., Ngoc, T.N., Phai, V.D., Shi, Q.: A deep learning approach to network intrusion detection. IEEE transactions on emerging topics in computational intelligence **2**(1), 41–50 (2018)
31. Sudhakar, Kumar, S.: An emerging threat fileless malware: a survey and research challenges. Cybersecurity **3**(1), 1 (2020)
32. Tajiri, Y., Mimura, M.: Detection of malicious powershell using word-level language models. In: International Workshop on Security. pp. 39–56. Springer (2020)
33. Tsai, M.H., Lin, C.C., He, Z.G., Yang, W.C., Lei, C.L.: Powerdp: De-obfuscating and profiling malicious powershell commands with multi-label classifiers. IEEE Access **PP**, 1–1 (01 2023). https://doi.org/10.1109/ACCESS.2022.3232505
34. VirusTotal: VirusTotal — Free Malware Analysis Service (2004), https://www.virustotal.com/, [Online; accessed 10-May-2025]
35. Wakodikar, P., Gwak, J.Y., Wang, M., Yan, G., Shu, X., Stoller, S., Yang, P.: Graphite: Real-time graph-based detection of windows fileless malware attacks. In: 20th EAI International Conference on Security and Privacy in Communication Networks (SecureComm 2024), Proceedings, Part III. LNICST, vol. 629, pp. 154–178. Springer (2025). https://doi.org/https://doi.org/10.1007/978-3-031-94455-0_8
36. Wang, J., Chen, Y.: A review on code generation with LLMs: Application and evaluation. In: 2023 IEEE International Conference on Medical Artificial Intelligence (MedAI). pp. 284–289 (2023). https://doi.org/10.1109/MedAI59581.2023.00044
37. Yin, C., Zhu, Y., Fei, J., He, X.: A deep learning approach for intrusion detection using recurrent neural networks. IEEE Access **5**, 21954–21961 (2017)
38. Zhang, H., Sediq, A.B., Afana, A., Erol-Kantarci, M.: Large language models in wireless application design: In-context learning-enhanced automatic network intrusion detection. arXiv preprint arXiv:2405.11002 (2024)
39. Zhang, T., Kishore, V., Wu, F., Weinberger, K.Q., Artzi, Y.: Bertscore: Evaluating text generation with bert. arXiv preprint arXiv:1904.09675 (2019)
40. Zhao, W., Wu, J., Meng, Z.: AppPoet: Large language model based Android malware detection via multi-view prompt engineering. Expert Syst. Appl. **262**(C) (Mar 2025). https://doi.org/10.1016/j.eswa.2024.125546, https://doi.org/10.1016/j.eswa.2024.125546