MalGuard: Towards Real-Time, Accurate, and Actionable Detection of Malicious Packages in PyPI Ecosystem

Xingan Gao[1] Xiaobing Sun[1,†] Sicong Cao[1,†] Kaifeng Huang[2] Di Wu[3] Xiaolei Liu [4]
Xingwei Lin[5] Yang Xiang [6]
[1]*Yangzhou University* [2]*Tongji University* [3]*University of Southern Queensland*
[4]*China Academy of Engineering Physics* [5]*Zhejiang University* [6]*Swinburne University of Technology*

Abstract

Malicious package detection has become a critical task in ensuring the security and stability of the PyPI. Existing detection approaches have focused on advancing model selection, evolving from traditional machine learning (ML) models to large language models (LLMs). However, as the complexity of the model increases, the time consumption also increases, which raises the question of whether a lightweight model achieves effective detection. Through empirical research, we demonstrate that collecting a sufficiently comprehensive feature set enables even traditional ML models to achieve outstanding performance. However, with the continuous emergence of new malicious packages, considerable human and material resources are required for feature analysis. Also, traditional ML model-based approaches lack of explainability to malicious packages. Therefore, we propose a novel approach MalGuard based on graph centrality analysis and the LIME (Local Interpretable Model-agnostic Explanations) algorithm to detect malicious packages. To overcome the above two challenges, we leverage graph centrality analysis to extract sensitive APIs automatically to replace manual analysis. To understand the sensitive APIs, we further refine the feature set using LLM and integrate the LIME algorithm with ML models to provide explanations for malicious packages. We evaluated MalGuard against six SOTA baselines with the same settings. Experimental results show that our proposed MalGuard, improves *precision* by 0.5%-33.2% and recall by 1.8%-22.1%. With MalGuard, we successfully identified 113 previously unknown malicious packages from a pool of 64,348 newly-uploaded packages over a five-week period, and 109 out of them have been removed by the PyPI official.

1 Introduction

Python has emerged as the most popular programming language [4]. However, threats against Python language packages have been arising in recent years [33, 34]. According to a recent report [1] from Sonatype, a total of 704,102 malicious packages have been discovered in third-party registries by 2024, marking a year-on-year increase of more than 156%. As the official package registry for Python, the Python Package Index (PyPI) [2] hosts a vast array of rapidly evolving packages and their dependencies. Simultaneously, it has led to a growing number of security problems in the PyPI registry [12–14, 18, 23, 38]. For example, a Windows Trojan "lumma" [36] targeting cryptocurrency wallets and browser extensions has extended to the PyPI registry. Attackers dropped a malicious package named *crytic-compilers-0.3.9* onto the PyPI registry, intending to steal private information by typosquatting the well-known popular package *crytic-compile*.

To mitigate such security threats, an intuitive way is to regularly scan PyPI packages. Early studies primarily employed static analysis and/or Machine Learning (ML) [5, 15, 19, 28, 35, 39, 46] to identify malicious features. However, handcrafted expert rules are hard to obtain and difficult to keep pace with the rapidly evolving malicious behaviors [23]. To improve the usability of the existing approaches and avoid the intense labor of human experts on feature extraction, recent works investigate the potential of Large Language Models (LLMs) in a more automated way of malicious package detection [37, 45, 46]. For example, Zhang et al. [46] fine-tuned the BERT model [17] to understand the semantics of sequential malicious behaviors.

Despite their effectiveness, fine-tuning a dedicated LLM is accompanied by considerable computational overhead, while directly invoking commercial LLMs (e.g., ChatGPT) via API will entail significant deployment cost when dealing with massive packages within the PyPI ecosystem [50]. Therefore, it motivates us to explore a simple yet effective approach that can strike a balance between effectiveness and overhead, aligning with the requirements of real-time and accurate detection needed in industrial settings.

To this end, we first performed an empirical study that aims to investigate *whether simple ML models can achieve comparable effectiveness to LLMs in malicious package detection*.

[†]Xiaobing Sun and Sicong Cao are co-corresponding authors.

First, we constructed a feature set of 132 dimensions. Using this feature set, we performed feature extraction and trained five traditional ML models. i.e., Random Forest (RF), Naive Bayes (NB), Extreme Gradient Boosting (XGBoost) [11], Multilayer Perceptron (MLP), and Support Vector Machine (SVM). Our experimental results show that the RF model performs the best among all five ML models that we selected. While the *precision* is slightly lower than EA4MP, CEREBRO, and GPT-3.5-turbo [30], the difference is negligible. Importantly, the RF model exhibits significantly higher *recall* compared to these two state-of-the-art (SOTA) approaches. It indicates that with a comprehensive feature set, we can achieve comparable detection accuracy with a lightweight model. Furthermore, while pre-training or fine-tuning LLMs typically requires several hours to even tens of hours, it takes approximately 5 seconds to train ML models and less than 1 second to perform malicious detection per package. The high efficiency enables ML model-based malicious package detection to perform real-time detection on newly updated packages.

Nevertheless, we summarize two remaining challenges for ML models that can hinder the effectiveness of malicious package detection:

- **Challenge 1 (C1): Reducing Dependency on Manually Pre-defined Feature Sets.** Quality features lead to better performance, but existing approaches often rely on manual analysis of malicious packages to construct these feature sets. As time progresses, the dataset is continuously augmented with new malicious package samples, necessitating ongoing manual effort from security professionals to analyze their characteristics. This process is resource-intensive and may miss critical features.
- **Challenge 2 (C2): Generate Explainable Outputs of Suspicious Behaviors for Malicious Packages.** Current detection efforts tend to focus on binary classification tasks for identifying malicious packages. However, once a malicious package is identified, administrators must spend significant time manually verifying its behavior. Although GuardDog [16] provides descriptions of suspicious APIs found in malicious packages, its rule-based approach requires extensive manual effort to define specific rules and lacks comprehensive analysis of the APIs [10]. Further assistance is needed to help administrators understand which sensitive APIs are being used in malicious packages and to identify their potential malicious purposes [41], which has received little attention in prior work.

To address the two challenges, we propose a novel malicious package detection approach MALGUARD.

To address **C1**, we employ static analysis methods to construct API call graphs for each malicious package and compute the centrality value of individual APIs. We then average the centrality values of APIs with the same name across different samples. Based on these averaged scores, we rank the APIs and select the top-K as the sensitive API feature set. To enhance the *precision* of the identified sensitive APIs, we employ role-based prompt engineering and utilize LLM (GPT-3.5-turbo) to assess the top-K candidates. To address **C2**, we leverage LLM to analyze each sensitive API and infer its potential malicious purposes. This resulted in a ground truth dataset containing suspicious API names and their potential malicious behaviors. Notably, our approach requires only one query to the LLM per sensitive API. The feature set is incrementally updated as new sensitive APIs emerge.

Using this feature set, we extracted the feature vector from the dataset and trained ML models integrated with the LIME algorithm. The algorithm identifies the top 10 influential non-zero features in the model's decision-making process. These features are then cross-referenced with the *{API_name : API_malbehavior}* Ground_truth dataset to generate detailed explanation outputs for each malicious package. These outputs provide a clear explanation of the malicious behavior associated with the identified sensitive APIs for researchers to verify these packages.

Prior approaches rely on manually refined feature sets. In comparison, our approach leverages graph centrality analysis and automates feature extraction with the assistance of LLMs, eliminating the need for manually predefined feature sets. In addition, we incorporate the LIME algorithm to generate explanations for the malicious packages.

Moreover, our work involves more than just simple name-based retrieval operations. We also sorted the APIs by their invocation order within each file and systematically output the content of these explanations. This ordered presentation facilitates a better understanding of malicious packages for researchers, providing them with clearer insights into the sequence and context in which APIs are used.

We evaluated MALGUARD against six SOTA baselines on the new dataset, which consists of malicious packages collected from Guo et al. [23] and Sun et al. [37]. Experimental results show that MALGUARD improves *precision* by 0.5%-33.2% and *recall* by 1.8%-22.1%. We monitored 64,348 software packages uploaded on PyPI between December 21, 2024, and January 28, 2025, and successfully identified 113 previously unknown malicious packages. We reported these malicious packages to PyPI officials, and 109 of them have been removed.

The contributions of our paper are as follows:

- We propose MALGUARD, a novel approach based on Social-network Graph Centrality to detect malicious PyPI packages.
- We collected a feature set containing 132 features to conduct our empirical study and demonstrate that, with a sufficiently comprehensive feature set, lightweight models can achieve effectiveness comparable to LLMs.
- We use LLMs and ML models to achieve explainability in malicious package detection.

Table 1: Statistics of the constructed dataset.

Dataset	#Malicious	#Benign
Guo et al. [23]	9,148	-
Sun et al. [37]	516	-
Our work	-	10,000
Total	9,664	10,000

- MALGUARD has uncovered 113 previously unknown malicious packages. We reported these packages to PyPI officials, and 109 of them have been removed.

2 Empirical Study

Since attacks on the PyPI platform are ongoing and continuously evolving, it is necessary to update detection models with newly emerging malicious samples. Notably, existing SOTA approaches, such as CEREBRO and EA4MP, rely on fine-tuning large pre-trained models, and such fine-tuning incurs substantial time and computational costs. In this section, we conduct two empirical studies. First, we investigate whether lightweight ML models can achieve detection effectiveness comparable to SOTA approaches. Second, we examine how temporal differences (i.e., the time gap between training and test samples) affect models' effectiveness.

2.1 Dataset

Malicious Sample. Since the detection capability of learning-based approaches benefits from large-scale and high-quality datasets, we built our evaluation benchmark by merging two reliable human-labeled datasets collected from real-world Python packages, including Guo et al. [23] and Sun et al. [37]. Detailed statistics for the two datasets are shown in Table 1. In total, our merged dataset contains 9,664 malicious PyPI packages.
Benign Sample. We randomly sampled 10,000 popular packages from PyPI. Following [28, 37], a package will be considered as benign if it has been (❶) hosted on PyPI for more than 90 days and (❷) downloaded over 1,000 times.

2.2 Baseline

To evaluate the differences between LLMs and ML models in malicious package detection, we compared our approach with two of the latest SOTA approaches: Zhang et al. [46] proposed CEREBRO, an approach for extracting code behavior sequences using abstract syntax trees (AST). By analyzing the AST, they extracted available APIs to form code sequences that describe malicious behaviors. These sequences were then used as input for fine-tuning a BERT model. Similarly, Sun et al. [37] introduced an integrated detection approach based on deep code behavior sequences and metadata. Their approach used static analysis tools to extract control-flow graphs (CFGs) and call graphs (CGs) to generate code behavior sequences, which were subsequently fine-tuned with the BERT model. We leveraged the one-shot technique along with role-based promoting to analyze the test dataset[1] using GPT-3.5-turbo [30]. All these approaches have demonstrated exceptional effectiveness in their respective datasets, so we chose them as the benchmarks for comparison.

Table 2: The categories of 132 different APIs in Feature Set.

Categories	API example

File-system access	os.mkdir()
	os.remove
	shutil.copy()
	write()
	...

Process creation	subprocess.Popen multiprocessing.Process threading.Thread ...

Network access	socket.socket()
	requests
	request.urlopen()
	...

Data encode & decode	base64.b64encode()
	base64.b64decode()
	...

Package install	install.run()
	pip.main()
	...

System access	os.getenv()
	os.getcwd()
	...

2.3 Study 1: Effectiveness Comparison of ML Models and LLM-based Approaches

Experiment Setup.

To evaluate whether traditional ML models can achieve competitive effectiveness compared to LLM-based approaches in malicious package detection, we conducted experiments using five widely adopted ML classifiers: Extreme Gradient Boosting (XGBoost), Random Forest (RF), Naive Bayes (NB), Support Vector Machine (SVM), and Multi-Layer Perceptron (MLP). To ensure the quality of the feature set, we manually analyzed 9,664 malicious packages and incorpo-

[1] For cost considerations, we randomly selected 300 benign packages and 300 malicious packages from the dataset for testing when conducting experiments with ChatGPT.

Table 3: Effectiveness comparison of five different ML models and LLM-based approaches on the same dataset.

Group	Model	Precision (%)	Recall (%)	F1 score (%)	Time Consumption	
					Pre-process (s/package)	Train (s)
ML	NB	55.2	98.4	70.7	**0.8457**	0.19467
	XGBoost	98.1	98.4	98.2		4.79
	RF	98.5	98	98.2		1.0126
	SVM	89.2	94.7	91.9		0.097
	MLP	98.1	98.2	98.1		22.85157
PTM	EA4MP [37]	**99.1**	95.4	97.2	6.28	30,741.67
	CEREBRO [46]	98.6	85.7	91.7	12.489	2,439
LLM	GPT-3.5-turbo [30]	99.0	**99.3**	**99.1**	-	-

Table 4: Temporal partitioning of the dataset.

Year	2021	2022	2023	2024
Counts	227	1,054	6,412	1,971

rated features identified in previous studies. This process yielded a comprehensive feature set with 132 dimensions. These features were further organized into six distinct categories, as summarized in Table 2.

Result. The experimental results are shown in Table 3. Four ML (i.e., XGBoost, RF, SVM, and MLP) models exhibited effectiveness comparable to LLM-based approaches. Among them, Random Forest (RF) and XGBoost models performed particularly well, achieving *precision* and *recall* rates of 98.0% and 98.2%, respectively. The Naive Bayes (NB) model yielded a lower *precision* of 55.2%, underperforming than LLM-based methods. GPT-3.5-turbo achieved the highest *recall* and *F1 score* among all approaches, highlighting the inherent advantage of large language models in understanding code semantics. Nonetheless, the effectiveness gap is not substantial. For instance, the Random Forest model achieved an accuracy only 0.5% lower and a recall rate just 1.3% lower than that of GPT-3.5-turbo. Compared to EA4MP and CEREBRO, while their *precision* (slightly below EA4MP's 99.1% and CEREBRO's 98.6%) is marginally lower, ML models' recall and *F1 scores* are significantly higher.

Moreover, in terms of time efficiency, we found that for ML models, feature vector extraction can be achieved using simple static analysis tools or even regular expression matching, requiring an average of only 0.0035 seconds per package. Training an ML model takes 0.097 to 22.85 seconds, which is significantly faster compared to EA4MP's 6.28 seconds and CEREBRO's 12.489 seconds per package and time required to fine-tune a BERT model. This highlights the significant time advantage of ML models.

2.4 Study 2: Robustness Against PyPI Malicious Packages

Experiment Setup. To evaluate the resilience of existing approaches and ML models in malicious package detection, we partitioned the malicious package dataset chronologically, as detailed in Table 4. To ensure the validity of the evaluation, we selected an equal number of popular benign packages from each year, maintaining a 1:1 ratio between benign and malicious samples. Given that the number of malicious packages in 2021 was relatively limited and significantly fewer than in subsequent years, we merged the samples from 2021 and 2022 to form the training set. The models were then evaluated separately on samples from 2023 and 2024. Based on the newly defined training set, we reanalyzed the feature distributions and reconstructed the feature set. For ML models, we selected four classifiers that demonstrated strong effectiveness in Study 1: XGBoost, RF, SVM, and MLP. Since we are unable to locally fine-tune or deploy GPT-3.5-turbo, we chose EA4MP, which achieves effectiveness second only to GPT-3.5-turbo, as the baseline for comparison.

Result. The experimental results, as shown in Table 5, demonstrate that with the addition of new samples, models trained solely on outdated data experience a notable effectiveness decline. For instance, in the case of ML models, the XGBoost model shows only a modest drop in *precision* (from 88.2% to 81.5%), yet its recall plummets significantly from 80.3% to 53.4%, indicating that a large portion of newly uploaded malicious samples cannot be effectively detected by the old model. Similarly, the *F1 score* of EA4MP decreases from 92.7% to 71.6%, reflecting a degradation in its detection capability.

Combined with the findings from **Study 1**, it is evident that both ML-based and LLM-based approaches require continu-

Table 5: Effectiveness comparison of different ML models and LLM-based approaches on newer samples by training an old dataset.

	XGBoost			RF			SVM			MLP			EA4MP		
Metrics (%)	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
2021&2022	88.2	80.3	84.1	97.1	82.0	88.9	88.6	80.3	84.2	95.3	80.6	87.3	94.7	90.7	92.7
2023	86.4	59.0	70.1	90.1	59.3	71.5	83.3	49.2	61.9	87.3	62.1	72.6	81.6	84.3	82.9
2024	81.5	53.4	64.5	72.6	52.1	60.7	75.4	51.0	60.8	79.6	57.1	66.5	72.7	70.5	71.6

ous updates with new data to maintain detection effectiveness. Compared to LLM-based approaches, ML models offer the advantage of faster iteration. Moreover, it is important to emphasize that while ML models may achieve detection effectiveness comparable to LLMs, constructing a comprehensive and high-quality feature set remains a major challenge. In our study, three master's students spent over a week analyzing 9,664 malicious packages to develop the final feature set consisting of 132 dimensions, amounting to 21 person-hours. Therefore, it is essential to develop a solution that supports efficient and automated feature set updates to enable timely and scalable model iteration.

3 API Call Graph Based Centrality Analysis

To enable automated extraction and rapid iteration of the feature set, we draw inspiration from Android malware detection, where graph centrality analysis has proven to be effective in identifying frequently invoked sensitive APIs. In this section, we provide a detailed description of the API Call Graph and the application of centrality analysis. Furthermore, we investigate whether there exist significant differences in API invocation patterns between benign and malicious packages within the PyPI ecosystem.

3.1 API Call Graph and Centrality Analysis

API Call Graph. The Application Programming Interface (API) Call Graph is a commonly used data structure in static or dynamic program analysis [6], designed to abstract the calling relationships among various API functions within a software system. In such a graph, each node represents an API function, method, or module, while edges denote the direction of invocation (i.e., the source node calls the target node). Compared to traditional structures such as the Control Flow Graph (CFG), the API Call Graph focuses more on high-level semantic invocation logic. API call graphs have been extensively employed in the security domain for malware detection [37, 44, 51], where they facilitate the identification of potential threats through the analysis of abnormal invocation patterns and frequent co-occurrence of malicious API sequences.

Centrality Analysis. Centrality metrics commonly used in social network analysis, such as degree centrality [21], Katz centrality [24], closeness centrality [21], and harmonic centrality [29], have been widely applied in Android malware detection tasks [42,51]. Studies have shown that these metrics effectively capture the behavioral characteristics of critical nodes within malicious code, thereby enhancing the discriminative power of detection models. Structurally, API call graphs share fundamental modeling similarities with social network graphs in both form and graph-theoretic properties. Both can be formalized as directed graphs, where nodes represent individual entities (typically users or actors in social networks, and functions, methods, or class modules in API call graphs), while edges denote interactions, such as social connections in the former and invocation or execution dependencies in the latter. These graph types also exhibit similar topological properties. For example, their node degree distributions often follow a power-law pattern, where a small number of nodes (e.g., frequently invoked functions or influential users) possess disproportionately high connectivity, while most nodes remain peripheral. Moreover, both types of graphs tend to form local clusters or community structures—subgraphs where node connectivity is significantly denser than the global average. These structures often correspond to functionally cohesive modules (such as attack chains) or social communities.

3.2 Difference between Malicious and Benign Packages

To verify whether malicious PyPI packages share similar traits with Android malware, we constructed API call graphs for both benign and malicious packages in the dataset and calculated the centrality values of APIs to compare their differences. The Experiment results highlight significant differences in API usage tendencies between benign and malicious packages. For instance, in the malicious package dataset, APIs such as *exists*, *subprocess.Popen*, *os.getenv*, *install.run*, *b64decode*, and *encode* are frequently invoked. Attackers often use *os.getenv* to evade dynamic analysis tools, *subprocess.Popen* to create malicious processes, and advanced attackers may employ APIs like *b64decode* to obfuscate data, making detection more challenging. In contrast, benign packages tend to favor simpler APIs for data processing, such as *int*, *str*, *list*, and *print*. Due to the limitation of the length of the article, we only present the top 10 APIs ranked by centrality scores calculated for both benign and malicious package datasets using four different centrality metrics in Table 6.

Table 6: The top 10 APIs calculated with different centrality in malicious&benign packages.

	Closeness	Degree	Harmonic	Katz
Malicious	setup	setup	join	setup
	exists	exists	open	exists
	subprocess.Popen	subprocess.Popen	decode	subprocess.Popen
	open	join	getattr	open
	join	open	encode	join
	range	range	map	install.run
	getattr	aetattr	exists	exec
	map	map	os.getenv	format
	os.getenv	exec	replace	os.getenv
	install.run	os.getenv	b64decode	expanduser
Benign	open	open	len	setup
	len	len	join	open
	setup	setup	str	len
	print	join	isinstance	join
	str	print	open	print
	isinstance	str	int	str
	int	isinstance	list	isinstance
	format	int	print	int
	list	range	append	range
	super	format	super	list

To address these discrepancies, we propose a detection tool based on API call graph centrality and ML models. For each malicious package, we generate an API call graph and compute the centrality value for its nodes. By aggregating and averaging the centrality scores of nodes with the same API name across all malicious packages, we rank the APIs based on their averaged centrality scores. We then select the top K ($K = 200, 300, 400, 500$) APIs as the feature set for feature vector extraction. Using these extracted feature vectors, we train an ML model to detect malicious packages effectively.

4 MalGuard: Graph Centrality and ML-Based Malicious Package Detection

To automate the extraction of sensitive API feature sets and train ML models for malicious package detection, and improve the explainability of ML models, we propose a novel approach: MalGuard. The workflow of MalGuard, illustrated in Figure 1, comprises Four main steps: **Step 1: API Call Graph Generation.** We perform static analysis on each malicious package to extract its AST. Based on the AST, we generate an API call graph and calculate the centrality values of its nodes. **Step 2: Sensitive API Feature Set Extraction and Filter.** We compute the centrality values of all nodes across the malicious packages, summing, averaging, and ranking the values for APIs with the same name. The top 500 APIs with the highest centrality values are selected as the sensitive API feature set. Based on the feature set that was extracted, we leveraged the one-shot technique along with role-based promoting to send the extracted APIs to an LLM (*GPT-3.5-turbo*) for analysis. If the API could potentially be used for malicious purposes, we retained it in the feature set; otherwise, it was removed. Additionally, for the retained APIs, we leveraged the language model to perform an analysis of possible malicious behaviors. The analysis results were saved in the format *api_name: malicious_behavior*, creating a Ground_Truth dataset for further reference. **Step 3: Model Training.** Using the feature set obtained in the second step, we extract feature vectors and train an ML model for malicious package detection. **Step 4: The Explanation Output based on LIME Algorithm.** Based on the ML model trained in Step 3, we integrated the LIME algorithm into the model. By identifying the top 10 most influential non-zero features for the model's decision, we matched these feature names against the Ground_Truth dataset. For each matched feature, all its associated potential malicious behaviors were retrieved and compiled to produce an explanation output, providing insights into the reasoning behind the model's detection decision.

4.1 API Call Graph

Generation Since social network graphs have been proven effective in Android malware detection, and our experiments in Section 3.2 further demonstrate that sensitive APIs in PyPI malicious packages often exhibit significantly higher and more anomalous centrality values in API call graphs compared to those in benign software, we focus our analysis on malicious packages when extracting API call graphs.

First, for each malicious package, we construct an AST. Using the relationships between nodes and edges in the AST, we generate a corresponding API call graph for each package. In the API call graph, each node represents an API, and the edges capture the invocation relationships between APIs. Based on these relationships, we calculate the centrality values for all nodes (APIs).

Given the variety of centrality measures available, we selected four of the most widely used centrality metrics to ensure comprehensive analysis: **Closeness centrality**, **Degree centrality**, **Katz centrality**, and **Harmonic centrality.** These measures provide a framework for identifying APIs with anomalous behaviors that may indicate malicious activity.

Closeness centrality. Closeness centrality is based on the reciprocal of the sum of the shortest path distances from a node to all other nodes, emphasizing the node's efficiency in spreading information or influence.

$$C_C(v) = \frac{N-1}{\sum_{u \in V, u \neq v} d(v,u)}$$

Where N is the total number of nodes in the graph, $d(v,u)$ represents the shortest path distance between node v and u,V is the set of all nodes in the graph.

Degree centrality. Degree centrality evaluates a node's importance based on the number of direct connections (edges) it has with other nodes.

$$C_D(v) = \frac{\deg(v)}{N-1}$$

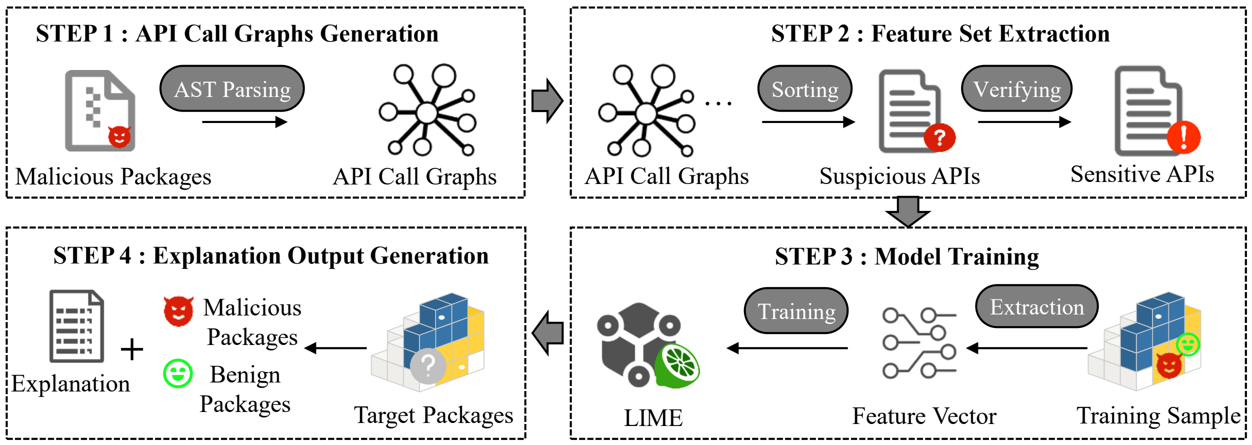


Figure 1: MalGuard architecture.

Where N is the total number of nodes in the graph, $\deg(v)$ represents the degree of the node v (*i.e.*, the number of edges connected to node v) and N is the total number of nodes in the graph.

Katz centrality. Katz centrality measures a node's influence by considering both the immediate neighbors and the neighbors further away, applying a weighting factor to penalize more distant connections.

$$C_K(v) = \alpha \sum_{u \in V} A_{vu} C_K(u) + \beta$$

Where α is the damping factor, typically a small value less than 1, which controls the influence of distant nodes, A_{vu} is the entry in the adjacency matrix V ($A_{vu} = 1$ if there is an edge from node v to u, otherwise ($A_{vu} = 0$), β is a constant that can be adjusted to represent the baseline centrality of each node.

Harmonic centrality. Harmonic centrality accounts for disconnected nodes by summing the reciprocal of distances rather than taking their total sum.

$$C_H(v) = \sum_{u \in V, u \neq v} \frac{1}{d(v,u)}$$

Where $d(v,u)$ represents the shortest path distance between node v and u,V is the set of all nodes in the graph.

We observed that, unlike Android malware [42,51], Python malicious packages are often smaller in size [37], and attackers tend to write malicious code directly in the global scope. As a result, even though the attackers use certain APIs, there are often no direct invocation relationships between them. This leads to a centrality value of zero for these APIs, This problem could result in a scenario where, although malicious packages invoke certain APIs, these invocation relationships are not accurately reflected in the centrality values. Consequently, important patterns of API usage may be overlooked.

Table 7: The feature set dimension after pre-processing by general-purpose LLM.

Centrality	Closeness	Degree	Katz	Harmonic
Total Dimensions	265	255	294	135

To address this issue, we adjust the calculation of centrality values by adding a default value of 1 to all centrality scores. This ensures no API's centrality value can be zero, effectively mitigating the problem and allowing us to capture the significance of APIs even in the absence of direct invocation relationships.

4.2 Sensitive API Extraction and Filter

After generating the API call graph and calculating node centrality values for each malicious package, the next step is to identify the sensitive API feature set. To achieve this, we aggregate the centrality values of APIs with the same name across all malicious packages. Specifically, we sum the centrality values of each API and then divide the total by the number of malicious packages to obtain an averaged centrality value for each API. This results in a comprehensive list of APIs with their corresponding averaged centrality values. We then rank the APIs based on their averaged centrality values and select the top K ($K = 200, 300, 400, 500$) as the sensitive API feature set. Although ML models can achieve effectiveness in malicious package detection comparable to that of LLMs, they have an inherent limitation: their inability to explain malicious behaviors. To address the need for a reliable dataset to explain malicious behaviors, we utilized an LLM (GPT-3.5-turbo) to construct a ground truth dataset. Specifically, the sensitive APIs extracted in Step 2 were analyzed using prompt engineering.

To ensure the feature set included as many suspicious APIs as possible while reducing the influence of irrelevant APIs, we selected the top 500 APIs for analysis. Each API was evaluated by the LLM to determine whether it could be used for malicious purposes: If potentially malicious, the API was retained in the feature set. Otherwise, it was removed. This process resulted in a refined, accurate, and relatively comprehensive feature set of suspicious APIs. The specific feature dimensions are summarized in Table 7.

In addition to fully leveraging the code understanding capabilities of the LLM, the filtered APIs were analyzed further. We instructed the model to generate outputs in a JSON format, similar to the gray-highlighted section in Figure 1, where each API was mapped to its potential malicious behaviors. After manual verification and the removal of unreasonable or inaccurate entries from the generated analysis, we finalized a ground-truth dataset that links API names to their potential malicious behaviors.

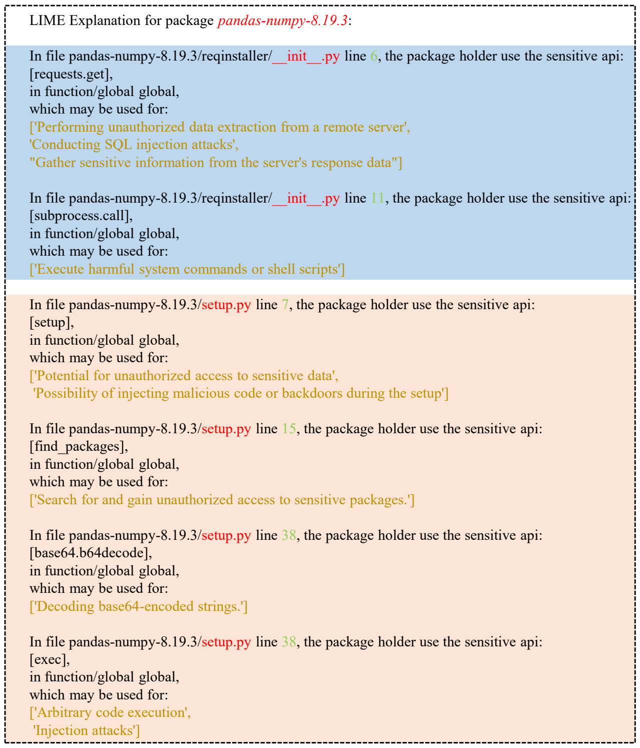
It is important to highlight that, unlike other approaches utilizing LLMs for malicious package detection, which require repeated invocations of the LLMs while our approach only necessitates a single invocation to construct the feature set. Additionally, the number of APIs that need to be analyzed is limited to 500. This results in significant advantages in terms of both time and economic costs.

4.3 Malicious Package Detection

To achieve efficient and precise detection of malicious packages, we utilized the feature set generated in Step 2 and applied methods such as regular expression matching to extract relevant features from the dataset's packages. For each API in the sensitive API feature set, we computed its corresponding centrality value within the context of a given package. This centrality value was then used as the feature value, reflecting the importance of the API within the package's structure. The extracted values were compiled into comprehensive feature vectors, which were subsequently used to train ML models.

4.4 Explanation output based on LIME

Local Interpretable Model-agnostic Explanations (LIME). LIME is a widely used algorithm designed to enhance the explainability of ML models. It explains the predictions of any black-box model by approximating the model's behavior locally around a specific instance [9]. We integrated the LIME algorithm into our ML model to enhance its explainability by identifying the top 10 non-zero features that had the most significant impact on the model's predictions. For each of these features, we cross-referenced the API names with the ground truth dataset generated in Step 2 to retrieve all potential malicious behaviors associated with each API. Simultaneously, we located the specific lines of code where these APIs appeared within the analyzed package, allowing us to understand their exact usage. To provide a clearer view of the API interactions,



```
LIME Explanation for package pandas-numpy-8.19.3:

In file pandas-numpy-8.19.3/reqinstaller/__init__.py line 6, the package holder use the sensitive api:
[requests.get],
in function/global global,
which may be used for:
['Performing unauthorized data extraction from a remote server',
'Conducting SQL injection attacks',
"Gather sensitive information from the server's response data"]

In file pandas-numpy-8.19.3/reqinstaller/__init__.py line 11, the package holder use the sensitive api:
[subprocess.call],
in function/global global,
which may be used for:
['Execute harmful system commands or shell scripts']

In file pandas-numpy-8.19.3/setup.py line 7, the package holder use the sensitive api:
[setup],
in function/global global,
which may be used for:
['Potential for unauthorized access to sensitive data',
 'Possibility of injecting malicious code or backdoors during the setup']

In file pandas-numpy-8.19.3/setup.py line 15, the package holder use the sensitive api:
[find_packages],
in function/global global,
which may be used for:
['Search for and gain unauthorized access to sensitive packages.']

In file pandas-numpy-8.19.3/setup.py line 38, the package holder use the sensitive api:
[base64.b64decode],
in function/global global,
which may be used for:
['Decoding base64-encoded strings.']

In file pandas-numpy-8.19.3/setup.py line 38, the package holder use the sensitive api:
[exec],
in function/global global,
which may be used for:
['Arbitrary code execution',
 'Injection attacks']
```

Figure 2: The explanation output result of malicious package *pandas-numpy-8.19.3*.

we sorted the identified APIs based on the order in which they appeared in the code, creating a sequential representation of their calling order and relationships across different Python script files in the package. Finally, we output all suspicious APIs along with their respective potential malicious behaviors, providing a comprehensive and interpretable result for further investigation. The output results are shown in Figure 2.

Each explanation output includes the following details: **Sensitive API Name:** The name of the sensitive API. **File Name and Code Line:** The name of the file in which the sensitive API is located, along with the specific line(s) of code where it is used. **Usage Context:** Whether the sensitive API is used in the global scope or within a specific function. **Potential Malicious Behaviors:** A list of all possible malicious behaviors associated with the sensitive API.

5 Evaluation

In this section, we evaluate MalGuard from different perspectives. First, we compare the effectiveness of MalGuard with SOTA detection approaches (Section §5.1). Second, we evaluate whether the feature set filtered by general-purpose LLMs can really affect the effectiveness of our approach (Section §5.2). Third, we evaluate if the explanation outputs generated by MalGuard can truly help researchers capture the malicious intent of attackers (Section §5.3). Then we measure what is the optimal value for the parameter top K in selecting sensitive APIs to achieve the best effectiveness of the model (Section §5.4). Besides, we evaluate the robustness of MALGUARD against the adversarial attack (Section §5.5). Finally, we show that MALGUARD can identify malicious packages that exist in the wild (Section §5.6).

Datasets and Models. To better validate the effectiveness of MALGUARD, we constructed a model training and testing dataset. As shown in Table 1, we collected a dataset containing 9,664 malicious packages and 10,000 benign packages. The benign packages dataset was the same dataset mentioned in Section §2.1. The entire dataset was randomly divided into training and testing sets in an 8:2 ratio, with the former used for model training and the latter for testing.

Baselines. To evaluate the effectiveness of MALGUARD against existing approaches, we selected six SOTA approaches as our baselines for comparison. These six approaches are VIRUSTOTAL [15], OSSGADGET [5], BANDIT4MAL [39], EA4MP [37], CEREBRO [46] and GUARDDOG [16]. VIRUSTOTAL [15] provides an online detection platform where packages can be uploaded directly for analysis. It automatically detects whether the software package contains suspicious files, IPs, URLs, *etc.* OSSGADGET [5] can identify potential backdoors and malicious code within a package. BANDIT4MAL [39] is an approach to finding common security issues in Python code. It processes each file, builds an AST from it, and runs appropriate plugins against the AST nodes. EA4MP [37] is an integrated detection approach based on deep code behavior sequences and metadata. Their approach used static analysis tools to extract CFGs and CGs to generate code behavior sequences, which were subsequently fine-tuned with the BERT model. CEREBRO is an approach by extracts a code sequence that can describe the malicious behaviors of attackers, and then uses this sequence to fine-tune the BERT model. GUARDDOG leverages Semgrep's semantic analysis capabilities and YARA's pattern-matching features to perform static code analysis and metadata scanning on packages from PyPI, NPM, and Go.

Implementation. We implement MALGUARD in Python using PyTorch [32]. Our experiments are performed on a Linux workstation with an AMD RYZEN 7735HS CPU, 32GB RAM, and an NVIDIA V100 GPU with 32GB memory, running Ubuntu 22.04 with CUDA 12.1. We implemented the ML models using the scikit-learn library (sklearn) in Python. Specifically, we utilized five widely used classifiers: NB, XGBoost, RF, SVM, and MLP.

Evaluation Metrics. We utilize three widely-used binary classification metrics for evaluation: *Precision*, *Recall*, and *F1 score*. *Precision* is the ratio of correctly identified malicious samples to all samples classified as malicious, representing the model's ability to avoid false positives. *Recall* measures the proportion of correctly identified malicious samples out of all actual malicious samples, reflecting the model's ability

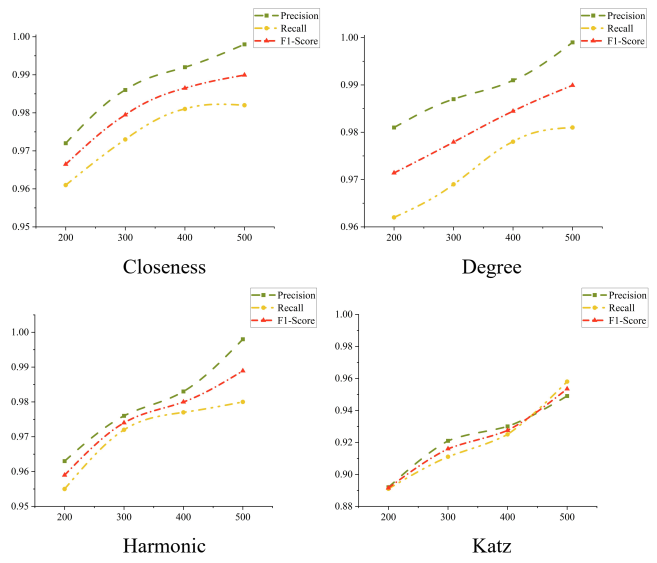


Figure 3: The effectiveness of the Random Forest (RF) model trained by top n APIs in four different centrality feature sets.

to detect true positives. *F1 score* is the harmonic mean of *Precision* and *Recall*, providing a balanced measure of a model's effectiveness. It is calculated as follows: $2 \times \frac{Recall \times Precision}{Recall + Precision}$.

5.1 Effectiveness Evaluation

Experimental Setup. To evaluate the effectiveness of MALGUARD, we combined benign and malicious packages, then randomly split the dataset into two parts: 80% was used as the training set, while the remaining 20% served as the test set. The same dataset was used to evaluate OSSGADGET, BAND4MAL, EA4MP,CEREBRO and ourapproach MALGUARD. Since VIRUSTOTAL provides an online detection platform. Similarly, GUARDDOG relies on static analysis and heuristic rules without requiring model training, so we only use the test set to verify their effectiveness. The experimental results are shown in Table 8.

Result. Table 8 shows the effectiveness of MALGUARD compared to the five baseline approaches on the same dataset. Comparing the data in the table, it is evident that MALGUARD improves precision by 0.5% to 33.2% over the other approaches, achieving the highest precision among all. Regarding recall, MALGUARD improved by 1.8% to 22.1% compared to all other approaches. The experimental results demonstrate that our approach achieves optimal effectiveness in terms of *precision*, *recall*, and *F1 scores*. In contrast, other approaches exhibit inherent limitations that affect their effectiveness, making them less effective compared to MALGUARD. **First,** Approaches like Bandit4Mal and OSSGadget often misclassify benign behaviors such as network connections and file operations in normal packages as malicious. This misclassification arises from their inability to distinguish between malicious and benign behaviors effectively. **Second,** Signature-based detection methods, such as VirusTotal, struggle to keep pace with the diversity and constant evolution of malicious code. The difficulty in maintaining consistent fingerprints for new malicious code makes it challenging for signature databases to detect the latest malicious packages. **Third,** Although LLMs often demonstrate excellent effectiveness in malicious package detection, as seen in approaches like EA4MP and CEREBRO, they tend to be overly sensitive to certain sensitive operations. This heightened sensitivity can lead to false positives, where benign packages are incorrectly flagged as malicious.

Case Study. To understand why certain benign packages that also invoke sensitive APIs can be more effectively distinguished from malicious ones by our approach compared to existing methods, we analyzed benign samples that were misclassified by other approaches but correctly identified by ours. Since most existing methods function as black-box models, it is difficult to determine the rationale behind their decisions. In contrast, GUARDDOG provides a list of suspicious APIs contained in each flagged package, which offers partial insight into its classification process. Therefore, we selected benign samples misclassified by GuardDog for further analysis.

Our analysis revealed that GUARDDOG tends to exhibit strong sensitivity toward specific categories of APIs. Once such APIs are invoked in a package, GUARDDOG is more likely to classify it as malicious. For example, in the case of *GACF-1.0.1*, the package invoked APIs such as *os.environ.copy()*, *subprocess*, and *os.makedirs()*, which fall under the category of system-level operations that interact with the environment or the file system. These triggered a false positive detection by GUARDDOG. In contrast, our approach not only considers the presence of sensitive API calls but also incorporates their centrality values within the API call graph as feature indicators. We found that, except for the relatively high centrality of the subprocess API, the other sensitive APIs in this benign package exhibited no significant centrality anomalies. This suggests that the benign package did not rely heavily on a small set of sensitive APIs, as reflected in their low centrality values. This result demonstrates that incorporating API centrality can help reduce false positives and more accurately distinguish benign packages from malicious ones, outperforming existing approaches.

5.2 Ablation Study

Experimental Setup. To validate the effectiveness of using LLMs for feature selection and analysis, we designed a comparative experiment to assess the impact of two different feature sets on the experimental results. For the initial feature set, we selected the top 500 APIs ranked by centrality values as sensitive APIs and used this feature set for feature extraction from the dataset. For the second feature set, we refined the initial set by utilizing the *GPT-3.5-turbo* model for further filtering and analysis. To comprehensively evaluate the classification effectiveness of the model on both benign and malicious packages, we separately calculated the *Precision*, *Recall*, and *F1 score* for benign and malicious samples.

Table 8: Effectiveness comparison with the SOTA baselines.

Approach	Precision (%)	Recall (%)	F1 score (%)
VIRUSTOTAL [15]	95.2	80.6	87.3
OSSGADGET [5]	74.8	85.0	79.6
BAND4MAL [39]	84.8	96.7	90.4
EA4MP [37]	99.1	95.4	97.2
CEREBRO [46]	98.6	85.7	91.7
GUARDDOG [16]	95.6	82.6	88.6
MALGUARD	**99.6**	**98.4**	**99.0**

Result. Through the analysis of APIs filtered out by the large language model, we observed that APIs like *print*, *range*, and *int*, despite being ranked within the top 500, are primarily used for basic data processing or output operations. These APIs are seldom, if at all, employed as vehicles for malicious activities. The experimental results, shown in Table 9, reveal that for the NB model, simply selecting the top 500 APIs as the feature set causes the model to overly favor classifying packages as either entirely benign or entirely malicious, rendering it almost ineffective in practical applications. However, the feature set refined using the LLM significantly improved the NB model's effectiveness. While its effectiveness still lags behind that of other models, this is primarily due to the inherent limitations of the NB model itself. The experimental results show that the remaining four ML models performed similarly across both feature sets. These findings support two key conclusions: **First**, our approach of using API call graph centrality for automated feature extraction is effective. This indicates that centrality measures can successfully capture relevant features for malicious package detection. **Second**, leveraging a general large language model can effectively help in filtering out irrelevant APIs from the feature set.

5.3 Explainability Evaluation

Explanation outputs verification dataset. To validate the accuracy of the explanation outputs, we randomly selected 100 malicious packages and 100 benign packages from the dataset for analysis. First, we employed prompt engineering to query the *GPT-3.5-turbo* model, instructing it to generate malicious behavior analyses in a specified format for the selected packages. To mitigate the potential impact of hallucinations in the LLM on the experimental results, we further conducted manual verification of the model's outputs. This process resulted in the creation of an explanation output verification dataset that contains 100 malicious packages.

To validate the accuracy of the model's final explanation outputs, we used the **Explanation Outputs Verifica-**

Table 9: Effectiveness of models trained by different centrality feature sets.

		with Feature Filtering				w/o Feature Filtering			
Metrics (%)		Closeness	Harmonic	Degree	Katz	Closeness	Harmonic	Degree	Katz
RF	Precision	99.4	92.5	99.3	99.6	99.9	99.9	99.9	94.9
	Recall	97.0	97.1	97.3	98.4	98.1	98.0	98.2	95.8
	F-1	98.2	94.8	98.3	99.0	99.0	99.0	99.1	95.3
XGBoost	Precision	99.4	99.2	92.5	99.3	99.2	99.3	99.2	93.0
	Recall	96.5	96.3	95.5	96.9	98.5	98.7	98.6	94.5
	F-1	97.9	97.7	94.0	98.1	98.8	99.0	98.9	93.7
SVM	Precision	97.9	87.1	97.6	97.9	82.8	86.6	72.6	71.8
	Recall	96.5	91.2	96.2	96.3	80.9	83.5	96.1	95.9
	F-1	97.2	89.1	96.9	97.1	81.8	85.0	82.7	82.1
MLP	Precision	98.5	92.0	98.2	98.4	99.0	99.1	98.3	89.8
	Recall	97.8	97.0	98.0	98.1	98.9	95.5	98.6	92.5
	F-1	98.1	94.4	98.1	98.2	99.0	97.3	98.4	91.1

tion Dataset as a benchmark. Two evaluation criteria were adopted: the number of sensitive APIs included in the output and their precise localization. If an explanation output contained at least 80% of the sensitive APIs and correctly identified their locations, it was deemed accurate.

The experimental results, shown in Table 10, all four ML models successfully identified the vast majority of malicious packages and generated accurate explanation outputs. Specifically, for degree centrality, 96 malicious packages were detected by at least three models, and 90 were identified by all four models. Even for the feature set based on harmonic centrality, which performed less effectively, there were 79 malicious packages detected by the four models.

To further assess whether the explanation output could help researchers analyze malicious behaviors, we randomly invited *n=24* volunteers (each with at least two years of experience in software engineering or software security) to rate the explanation results. Ratings ranged from 1 to 5, with higher scores indicating better quality. If a model failed to detect a malicious package or did not generate any explanation output, a score of 0 was assigned. The final scores were averaged and shown in Figure 4.

The experimental results demonstrate that the explainability content generated by our approach achieved an average score of 3.5 or higher, indicating that the explanation outputs are effective and useful for aiding in malicious behavior analysis.

False Positive Analysis. To investigate why the model misclassified certain benign samples as malicious, we conducted a detailed analysis of three false positive cases identified by

Table 10: Effectiveness of different ML Models in **Explanation Outputs Verification Dataset** (The Third Column shows the number of malicious packages that every model can detect and explain while the Fourth and Fifth Columns show the number of malicious packages that can be detected and accurately explained by more than 3 or 4 different models.).

Centrality	Model	Total detected	r>=3	r=4
Closeness	XGBoost	96	95	93
	RF	95		
	SVM	94		
	MLP	98		
Degree	XGBoost	97	96	90
	RF	96		
	SVM	91		
	MLP	97		
Katz	XGBoost	95	93	86
	RF	93		
	SVM	88		
	MLP	96		
Harmonic	XGBoost	95	92	79
	RF	93		
	SVM	82		
	MLP	95		

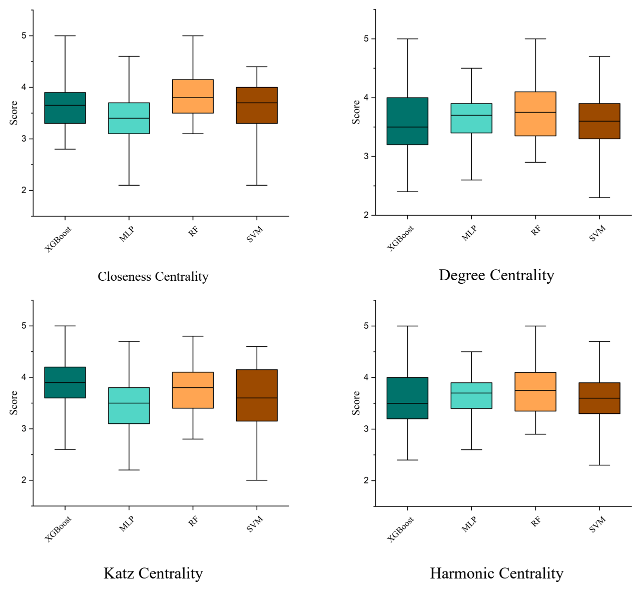


Figure 4: Box plot analysis of the average score of 100 malicious packages' explanation output.

the Random Forest model (e.g., *eazure-0.1.1*, *gitlab-clone-0.1.1*, *py-sourcemap-.1.14*) and examined the corresponding explainability outputs. We found that these benign packages all invoked multiple high-ranking sensitive APIs from the feature set. For example, *py-sourcemap-0.1.14* performed file operations (e.g., *read*, *open*), issued network requests (e.g., *urlopen*), and executed installation-related commands (e.g., *install.run*). These APIs also showed significantly higher centrality values within the package's API call graph. These findings suggest that the frequent use of diverse high centrality sensitive APIs likely led to the model's misclassification.

5.4 Hyperparameter Sensitivity Analysis

Experiment Setup. To systematically examine the impact of varying the top K parameter on the final model's effectiveness, we performed experiments with different ($K = 200, 300, 400, 500$). The lower bound of $K = 200$ was established based on the dimensionality of a manually curated feature set, which consisted of 132 features, ensuring that the automatically extracted feature set would not be less informative. The upper bound of $K = 500$ was derived from manual analysis, which demonstrated that APIs ranked beyond this threshold seldom exhibited suspicious or malicious behavior. We selected the Random Forest (RF) model that demonstrated the best effectiveness in our previous experiments to evaluate the effect of these parameter choices on model effectiveness. This approach facilitates a thorough evaluation of how the selection of K influences both the effectiveness and reliability of the detection system, thereby providing insights into optimal parameter configuration for malicious package detection.

Result. The experimental results, illustrated in Figure 2, show that as K increases, the model's effectiveness consistently improves across feature sets derived using four different centrality metrics. For instance, the *F1 scores* increase by 2%–7% when K is raised from 200 to 500, indicating that higher K values include more suspicious APIs in the feature set. These findings suggest that setting $K = 500$ allows the feature set to capture the most comprehensive set of suspicious APIs. To ensure optimal model effectiveness, all subsequent experiments adopt $K = 500$ as the default parameter setting.

5.5 Robustness against Adversarial Attack

Experiment Set. To evaluate the robustness of MALGUARD against adversarial attacks, we selected two representative and widely adopted categories of attack strategies [48]. **Category 1: Feature Space Attacks.** This category targets the feature vectors that are used by our detection model. Two distinct attack methods were applied: The first method introduces random noise into the feature vectors, following the randomization-based attack [43]. The second method transforms all non-zero feature values in both the training and test sets to 1, instead of using their original centrality scores of API nodes. **Category 2: Source Code Level Adversarial Attack.** Several adversarial attack strategies against software packages have been proposed in prior works. For example, Kreuk [26] injects adversarial byte sequences into binary files; IPR [25] obfuscates code by inserting randomized, semantically ineffective instructions, and DISP [31] introduces code randomization using equivalent instruction replacement. However, these techniques target executable binaries (e.g., *.exe*, *.apk*), whereas most malicious PyPI packages are distributed as source archives (e.g., *.tar.gz*). Moreover, to ensure the effectiveness of static analysis tools used to construct API call graphs, the injected content must preserve source code validity. Therefore, we designed a **source code level attack** inspired by IPR [25], adapted to Python packages. Specifically, for each malicious package, we randomly selected α benign packages ($1 <= \alpha <= 3$). From these, we randomly chose β Python source code files to inject. After analyzing the dataset of malicious packages, we found that each package contains an average of 4.18 *.py* files. To avoid injecting an excessive amount of dead code that could distort the package structure, we rounded the value β up to a maximum of 5. To maintain randomness in the poisoning process, we allowed β to vary within the range of 1 to 5. These benign files were then injected into the malicious package without modifying any original code. Then we reconstructed the API call graph, re-extracted the feature set and feature vectors based on closeness centrality, and retrained the detection models.

Result. Figure 5 presents the *F1 score* of the four ML models used in our approach (RF, XGBoost, SVM, and MLP) when subjected to adversarial attacks. **Under the first cate-**

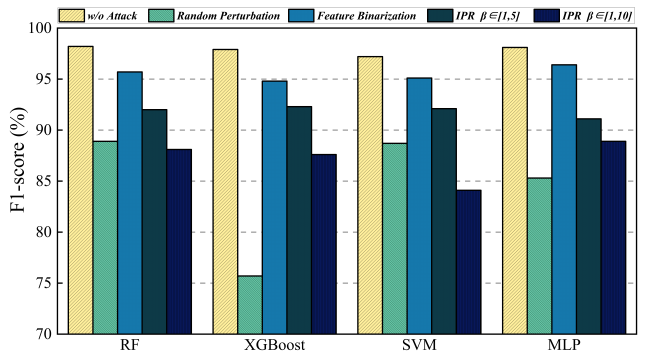


Figure 5: Model robustness against adversarial attacks.

gory of attacks, we observed that the strategy of replacing all non-zero feature values with *1* led to only a slight drop in effectiveness. This is expected, as the transformation does not alter the underlying API invocation patterns of the malicious packages. In contrast, the randomization-based strategy introduces noise to the feature vectors, effectively disrupting the API call patterns. As a result, the *F1 scores* dropped more substantially. Nevertheless, even under this more aggressive attack, the worst-performing model (XGBoost) still achieved an *F1 score* of 75.7%, while the remaining models maintained *F1 scores* above 85%. **Under the second category of attacks**, our approach also experienced a effectiveness decline. However, even for the worst-performing model (MLP), the *F1 scores* remained as high as 91.1%. Even when we increased the parameter β from 1 to 10, our approach was still able to maintain an *F1 score* above 84%. Additionally, the effectiveness gap among the four models under this IPR-inspired source-level attack was relatively small.

To understand this result, we analyzed the post-attack feature sets and found that although the injected benign code (i.e., dead code) somewhat reduced the centrality values of certain sensitive APIs, these added components did not form direct invocation relationships with the sensitive APIs. As a result, within the local subgraphs containing sensitive APIs, their centrality remained significantly higher than that of unrelated APIs. These findings demonstrate that although our approach suffers some degradation under adversarial conditions, it maintains effectiveness at an acceptable level, highlighting its robustness against such attacks.

5.6 Practicality

Wild truth. To verify if MALGUARD can identify malicious packages that exist in the wild, we crawled *all* packages uploaded to PyPI between December 21, 2024, and January 28, 2025, from the official website [3]. In total, we collected 64,348 packages for real-world validation. Two authors separately review the reported malicious packages. All suspicious packages (including samples that did not reach a consensus) would be forwarded to a security expert from a prominent IT enterprise with at least five years of experience in software supply security to conduct a secondary review.

In total, MALGUARD discovered 144 suspicious packages. After manual review, 113 out of them were confirmed malicious. We reported these packages to the PyPI official. As of January 21, 2025, 109 of them have been removed.

False Positives. Upon analyzing 31 packages that were incorrectly flagged as malicious, we discovered that 25 of them were user-uploaded test demos. Although these demo packages utilized numerous suspicious APIs, such as *request.get* and *subprocess.run*, they did not engage in any malicious behavior. The remaining 6 packages were identified as prank packages, such as the *"crazy-thursday"* package, which creates local processes to display "Crazy Thursday" messages to users but does not perform any harmful actions. We have reported these prank packages to the PyPI official too.

6 Discussion

Code Obfuscation. Code obfuscation is a common technique used to evade existing detection approaches. Currently, most approaches primarily analyze software packages based on their source code files. Some attackers circumvent detection by packaging their code into binary executable files. Detecting such packages requires software security professionals to have reverse engineering skills and to continuously monitor the resource usage of the software package during execution. Fortunately, code obfuscation also requires some technical expertise from the attackers. Currently, the majority of malicious software packages mainly obfuscate the parameter values of functions. This means that MALGUARD can still accurately detect them. However, how we deobfuscate more complex forms of code obfuscation remains a challenge that we need to address.

Cross Platforms Detection. Although existing approaches for detecting malicious packages demonstrate strong effectiveness, most of these methods are tailored to one or two specific platforms. In the broader context of the open-source software ecosystem, each programming language has its own maintained open-source community. This implies that for different open-source communities, maintainers need to employ distinct analysis methods and train separate detection models. Such a requirement undoubtedly increases the complexity of maintaining the stability of these communities. Zhang et al. [46] proposed a dual-platform (NPM&PyPI) detection tool based on the BERT model and code behavior sequences. Their experiments demonstrated that although API names vary across platforms, malicious actors must invoke APIs with specific functionalities. Given this, the intrinsic relationships among these APIs may potentially be leveraged to develop cross-platform detection approaches for malicious packages.

7 Related Work

Malicious Package Detection. Detecting malicious software packages within open-source software registries is a complex challenge. Liang et al. [27] introduced PPD, a third-party malware identification framework employing anomaly detection. This approach forms a comprehensive code package by importing required packages, uses AST (Abstract Syntax Tree) and RegExp (Regular Expressions) to extract code features (e.g., IP addresses, dangerous functions), and incorporates the Levenshtein distance of package names into the feature set. It then applies anomaly detection algorithms to identify malicious packages. Given that developers often host open-source code on platforms like GitHub, inconsistencies between the code released on registries such as PyPI and the corresponding GitHub repositories may signal malicious injection. To address this issue, Vu et al. [40] proposed LASTPYMILE, a framework designed to identify disparities between software package construction artifacts and their source repositories. This approach enhances monitoring of registry security, helping mitigate risks. Zhang et al. [46] proposed CEREBRO, which extracts code behavior sequences based on abstract syntax trees. By identifying available APIs in the AST, they construct a sequence describing the attacker's malicious behavior and use this sequence to fine-tune a BERT model. However, while Zhang's method leverages code behavior sequences to enhance model understanding of attack patterns, it remains limited by the reliance on manual feature recognition. Liang et al. [28] introduced MPHUNTER, which identifies malicious packages by extracting code behavior sequences, converting them into vectors, and employing clustering to detect anomalies. However, MPHUNTER is constrained to analyzing only the ‘setup.py‘ script in packages. As noted by Guo et al. [23], attackers often distribute malicious code across multiple scripts, complicating detection and circumventing such approaches.

Package Registry Security. There are numerous repositories exploited as platforms for distributing malicious code and software libraries. GURADDOG [16] is an open-source command-line tool (CLI) developed by Datadog, designed to identify malicious packages in PyPI, npm, and Go through static code analysis and metadata scanning. It combines the semantic analysis capabilities of Semgrep with the pattern-matching power of YARA to detect suspicious behaviors. ANOMALICIOUS [22] addresses this issue by leveraging commit logs and repository metadata to automatically identify anomalies and potentially malicious commits. Attackers often misuse GitHub's fork functionality to store and distribute malware [22]. To counter this threat, Zhang et al. [47] proposed an enhanced deep neural network (DNN) [7, 8] for analyzing the code content of GitHub repositories. Their approach employed a heterogeneous information network (HIN) to model neighborhood relationships, thereby improving recognition accuracy. Malicious actors frequently embed harmful shell commands within Python scripts to achieve illicit objectives. Traditional static analysis methods often struggle to detect such sophisticated attacks. To address this gap, Zhou et al. [49] introduced PYCOMM, ML model specifically designed to detect malicious commands in Python scripts. PYCOMM evaluates multidimensional features, simultaneously analyzing 12 statistical characteristics of Python source code and string sequences. In addition, Fang et al. [20] employed ML techniques to identify Python backdoors. Their method represented text using statistical features derived from obfuscation and opcode sequence characteristics during compilation. By matching suspicious modules and functions within the code, their approach effectively detected embedded backdoors.

8 Conclusion

In this paper, we demonstrate that with a sufficiently comprehensive feature set, ML models can achieve effectiveness comparable to that of LLMs. We also proposed a novel approach MALGUARD, leveraging graph centrality to extract the sensitive APIs automatically, eliminating reliance on manually predefined feature sets. Moreover, we employ GPT-3.5-turbo to refine and analyze the feature set, and by integrating the LIME algorithm, we achieved explainable outputs for malicious package detection, enhancing both explainability and effectiveness. We evaluate MALGUARD against the state-of-the-art approaches on a newly-constructed dataset. MALGUARD performs better on all metrics. Moreover, compared to existing works, our approach supports much faster iteration. The computation of centrality values for all malicious packages can be completed within a few hours, while feature extraction and model training require only a few minutes. This efficiency enables our approach can be updated on a daily basis using newly emerging packages, ensuring timely adaptation to evolving threats. We also applied MALGUARD to real-world detection and found 113 malicious packages by detecting 64,348 newly uploaded PyPI packages, 109 of which have been removed by PyPI officials. This indicates that MALGUARD is a practical approach that can be adopted by the Python community to detect malicious packages.

Acknowledgments

We thank the reviewers for their valuable comments. This work was supported in part by the National Natural Science Foundation of China (Grant No. 62402342), the Jiangsu “333” Project, Postgraduate Research & Practice Innovation Program of Jiangsu Province (KYCX24_3747), and Shanghai Sailing Program (No. 24YF2749500).

Ethics Considerations

Throughout the study, we ensured that no personally identifiable information (PII), sensitive user data was involved at any stage of data collection, analysis, or validation. All datasets used for training and testing, including benign and malicious packages, were obtained from publicly available repositories or previous peer-reviewed studies (*e.g.*, [20, 22, 23, 27, 28, 46]).

To ensure ethical integrity, we followed a transparent and responsible disclosure procedure for all identified malicious packages. Specifically, when our tool, MALGUARD, detected previously unknown malicious packages among newly uploaded PyPI packages, we reported these findings to the PyPI security team prior to any public disclosure. Out of 113 packages flagged as malicious, 109 were subsequently reviewed and removed by PyPI, indicating the practical value and responsible execution of our methodology.

Open Science

This work aligns with the principles of Open Science and aims to facilitate transparency, reproducibility, and community collaboration. All resources are available via our project repository at: https://doi.org/10.5281/zenodo.15545824. We provide full documentation and guidelines to replicate our experiments.

References

[1] 10th annual state of the software supply chain, 2024. https://www.sonatype.com/state-of-the-software-supply-chain/2024/scale.

[2] Pypi index, 2024. https://pypi.org/.

[3] Pypi simple, 2024. https://pypi.org/simple/.

[4] Tiobe index, 2024. https://www.tiobe.com/tiobe-index/.

[5] Bertus. Oss gadget: Collection of tools for analyzing open source packages., 2020. https://github.com/microsoft/OSSGadget.

[6] Sicong Cao, Biao He, Xiaobing Sun, Yu Ouyang, Chao Zhang, Xiaoxue Wu, Ting Su, Lili Bo, Bin Li, Chunlei Ma, Jiajia Li, and Tao Wei. Oddfuzz: Discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (SP)*, pages 2726–2743. IEEE, 2023.

[7] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. *BGNN4VD*: Constructing bidirectional graph neural-network for vulnerability detection. *Inf. Softw. Technol.*, 136:106576, 2021.

[8] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 1456–1468. ACM, 2022.

[9] Sicong Cao, Xiaobing Sun, Ratnadira Widyasari, David Lo, Xiaoxue Wu, Lili Bo, Jiale Zhang, Bin Li, Wei Liu, Di Wu, and Yixin Chen. A systematic literature review on explainability for machine/deep learning-based software engineering research. *arXiv preprint arXiv: 2401.14617*, 2024.

[10] Sicong Cao, Xiaobing Sun, Xiaoxue Wu, David Lo, Lili Bo, Bin Li, and Wei Liu. Coca: Improving and explaining graph neural network-based vulnerability detection systems. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 155:1–155:13. ACM, 2024.

[11] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 785–794. ACM, 2016.

[12] Alibaba company. Pypi mirror of alibaba company, 2024. https://mirrors.aliyun.com/pypi/simple/.

[13] Huawei company. Pypi mirror of huawei company, 2024. https://mirrors.huaweicloud.com/repository/pypi/simple/.

[14] Tencent company. Pypi mirror of tencent company, 2024. https://mirrors.cloud.tencent.com/pypi/simple.

[15] VirusTOTAL company. Analyse suspicious files, domains, ips and urls to detect malware and other breaches, automatically share them with the security community, 2024. https://www.virustotal.com/gui/home/upload.

[16] DataDog. Guarddog, 2024. https://github.com/DataDog/guarddog.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill

Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[18] douban. Pypi mirror of douban company, 2024. http://pypi.doubanio.com/simple/.

[19] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. In *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2021.

[20] Yong Fang, Mingyu Xie, and Cheng Huang. PBDT: python backdoor detection model based on combined features. *Secur. Commun. Networks*, 2021:9923234:1–9923234:13, 2021.

[21] Linton C Freeman et al. Centrality in social networks: Conceptual clarification. *Social network: critical concepts in sociology. Londres: Routledge*, 1:238–263, 2002.

[22] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schaefer. Anomalicious: Automated detection of anomalous and potentially malicious commits on github. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*, pages 258–267. IEEE, 2021.

[23] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. An empirical study of malicious code in pypi ecosystem. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 166–177. IEEE, 2023.

[24] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.

[25] Hyungjoon Koo and Michalis Polychronakis. Juggling the gadgets: Binary-level code randomization using instruction displacement. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, page 23–34, New York, NY, USA, 2016. Association for Computing Machinery.

[26] Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. Adversarial examples on discrete sequences for beating whole-binary malware detection. *CoRR*, abs/1802.04528, 2018.

[27] Genpei Liang, Xiangyu Zhou, Qingyu Wang, Yutong Du, and Cheng Huang. Malicious packages lurking in user-friendly python package index. In *20th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2021, Shenyang, China, October 20-22, 2021*, pages 606–613. IEEE, 2021.

[28] Wentao Liang, Xiang Ling, Jingzheng Wu, Tianyue Luo, and Yanjun Wu. A needle is an outlier in a haystack: Hunting malicious pypi packages with code clustering. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 307–318. IEEE, 2023.

[29] Massimo Marchiori and Vito Latora. Harmony in the small-world. *Physica A: Statistical Mechanics and its Applications*, 285(3-4):539–546, 2000.

[30] OpenAI. Chatgpt., 2024. https://chatgpt.com/.

[31] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 601–615. IEEE Computer Society, 2012.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Proceedings of the 33rd Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 8024–8035, 2019.

[33] Qi'anxin. 2023 china software supply chain security analysis report, 2023. https://www.qianxin.com/threat/reportdetail?report_id=297.

[34] Qi'anxin. Pypi massive forged packet name attack, 2024. https://mp.weixin.qq.com/s/VIThE0I5BkQBW6hIOubnkQ.

[35] Adriana Sejfia and Max Schäfer. Practical automated detection of malicious npm packages. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1681–1692. ACM, 2022.

[36] Sonatype. Russia-linked 'lumma' crypto stealer now targets python devs, 2024. https://www.sonatype.com/blog/crytic-compilers-typosquats-known-crypto-library-drops-windows-trojan.

[37] Xiaobing Sun, Xingan Gao, Sicong Cao, Lili Bo, Xiaoxue Wu, and Kaifeng Huang. 1+1>2: Integrating deep code behaviors with metadata features for malicious pypi package detection. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1159–1170. ACM, 2024.

[38] Tsinghua university. Pypi mirror of tsinghua university, 2024. https://pypi.tuna.tsinghua.edu.cn/simple/.

[39] D.-L. Vu. A fork of bandit tool with patterns to identifying malicious python code., 2020. https://github.com/lyvd/bandit4mal.

[40] Duc Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. Lastpymile: identifying the discrepancy between sources and packages. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 780–792. ACM, 2021.

[41] Bozhi Wu, Sen Chen, Cuiyun Gao, Lingling Fan, Yang Liu, Weiping Wen, and Michael R. Lyu. Why an android app is classified as malware: Toward malware classification interpretation. *ACM Trans. Softw. Eng. Methodol.*, 30(2):21:1–21:29, 2021.

[42] Yueming Wu, Xiaodi Li, Deqing Zou, Wei Yang, Xin Zhang, and Hai Jin. Malscan: Fast market-wide mobile malware scanning by social-network centrality analysis. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 139–150. IEEE, 2019.

[43] Cihang Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, and Alan Yuille. Mitigating adversarial effects through randomization. *arXiv preprint arXiv:1711.01991*, 2017.

[44] Zeliang Yu, Ming Wen, Xiaochen Guo, and Hai Jin. Maltracker: A fine-grained NPM malware tracker copiloted by llm-enhanced dataset. In Maria Christakis and Michael Pradel, editors, *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, pages 1759–1771. ACM, 2024.

[45] Nusrat Zahan, Philipp Burckhardt, Mikola Lysenko, Feross Aboukhadijeh, and Laurie Williams. Leveraging large language models to detect npm malicious packages. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 683–683. IEEE, 2025.

[46] Junnan Zhang, Kaifeng Huang, Yiheng Huang, Bihuan Chen, Ruisi Wang, Chong Wang, and Xin Peng. Killing two birds with one stone: Malicious package detection in npm and pypi using a single model of malicious behavior sequence. *ACM Trans. Softw. Eng. Methodol.*, 34(4):104:1–104:28, 2025.

[47] Yiming Zhang, Yujie Fan, Shifu Hou, Yanfang Ye, Xusheng Xiao, Pan Li, Chuan Shi, Liang Zhao, and Shouhuai Xu. Cyber-guided deep neural network for malicious repository detection in github. In Enhong Chen and Grigoris Antoniou, editors, *2020 IEEE International Conference on Knowledge Graph, ICKG 2020, Online, August 9-11, 2020*, pages 458–465. IEEE, 2020.

[48] Yinyuan Zhang, Cuiying Gao, Yueming Wu, Shihan Dou, Cong Wu, Ying Zhang, Wei Yuan, and Yang Liu. Fighting fire with fire: Continuous attack for adversarial android malware detection.

[49] Anmin Zhou, Tianyi Huang, Cheng Huang, Dunhan Li, and Chuangchuang Song. Pycomm: Malicious commands detection model for python scripts. *J. Intell. Fuzzy Syst.*, 42(3):2261–2273, 2022.

[50] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Trans. Softw. Eng. Methodol.*, 34(5):145:1–145:31, 2025.

[51] Deqing Zou, Yueming Wu, Siru Yang, Anki Chauhan, Wei Yang, Jiangying Zhong, Shihan Dou, and Hai Jin. Intdroid: Android malware detection based on API intimacy analysis. *ACM Trans. Softw. Eng. Methodol.*, 30(3):39:1–39:32, 2021.

Appendix

A LLM Prompts for Malicious Analysis

Here are three prompts that we used to query LLM for malicious analysis. The first one is used to analyze whether the package that we send is malicious, illustrated in *System Role Prompt of Malicious Packages Detection.* The second one is used to analyze weather the API we send can be used for malicious purposes, if so, analyze the malicious purposes, illustrated in *System Role Prompt of Sensitive API Analysis.* The second one is used to analyze which APIs the malicious package we send has used, and describe the malicious behavior it contains, illustrated in *System Role Prompt of Malicious Packages Analysis.*

System Role Prompt of Malicious Packages Detection

Task:
You are a cybersecurity expert. Your task is to analyze a given Python code snippet and determine whether it contains malicious behavior.
Guidelines:
-Examine the code for patterns commonly used in malware.
-Consider whether the code could be harmful when executed, even if it appears simple.
-Be objective. If malicious behavior is suspected, clearly explain why.
-Output must be in valid JSON format.
Code:
<INSERT CODE HERE>
JSON Response:

```
{
  "is_malicious": true or false,
  "reason": "The malicious behaviors the
             code cotained.",
  "confidence": 0-1,
  "indicators": ["", ""]
}
```

System Role Prompt of Malicious Packages Analysis

Task:
You are a behavioral malware analyst. Your task is to extract and classify all malicious behaviors present in a given Python code snippet.
Guidelines:
-Identify and label all suspicious or clearly malicious behaviors in the code.
-For each behavior, provide a category, a short description, and a relevant code snippet.
-If no malicious behaviors are found, return an empty array.
-Output must be in valid JSON format.
Code:
<INSERT CODE HERE>
JSON Response:

```
{
  "malicious_behaviors": [
  {
    "type": "Malicious type n",
    "description": "Malicious behavior
                    description.",
    "code_snippet": "The code that contain
                      malicious APIs."
  },
  ...]
}
```

System Role Prompt of Sensitive API Analysis

Task:
You are a security API auditor. Your task is to determine whether a given Python API can potentially be used for malicious purposes.
Guidelines:
-Consider common attack techniques such as command execution, code obfuscation, data exfiltration, privilege escalation, etc.
-If the API is not typically used in a malicious context, return a neutral evaluation.
-Output must follow the required JSON format.
Code:
<INSERT CODE HERE>
JSON Response:

```
{
  "api_name": "The name of the API",
  "is_potentially_malicious": true,
  "malicious_usage": "Malicious purposes."
}
```