*electronics*

MDPI

*Article*

# Decoding Maliciousness: LLM-Assisted Semantic Filtering for Scalable Malicious PyPI Package Detection

**Jiutian Wang *, Deming Mao, Ye Han, Zhaoheng Quan and Bingwen Wang**

The 30th Research Institute of China Electronics Technology Group Corporation, No. 6, Chuangye Road, Wuhou District, Chengdu 610041, China; mdm1201@126.com (D.M.)

* Correspondence: wwwtopz@163.com

**Abstract**

The Python Package Index (PyPI) has become a primary target for software supply chain attacks. Existing detection methods often rely on single-dimensional analysis, limiting their ability to support scalable detection in modern software ecosystems. This study presents a scalable framework for malicious PyPI package detection by combining graph centrality-based candidate identification with large language model semantic filtering. API interaction graphs are constructed from package source code, and four complementary centrality metrics are used to identify structurally important APIs. For each metric independently, a separate Top-*K* candidate pool is constructed and refined through semantic filtering with Qwen3.5-Plus, removing benign generic APIs while retaining security-sensitive APIs. The refined API features are further augmented with package-level metadata and used for package-level classification. Evaluations on a dataset of 20,743 packages (10,743 malicious and 10,000 benign) show that the proposed method achieves 96.3% precision, 95.9% recall, and a 96.1% F1-score. Ablation experiments indicate that semantic filtering provides the largest performance improvement, increasing F1-score by 4.2 percentage points over raw centrality features, while metadata augmentation provides a further improvement of 1.2 percentage points. The framework maintains robust detection performance under a highly imbalanced 1:50 class ratio and, after the one-time offline construction of the refined feature vocabulary, requires approximately 1 s per package for detection. A retrospective monitoring experiment on the PyPI release stream further demonstrates the practical applicability of the framework, while LIME provides interpretable evidence for the resulting predictions.

**Keywords:** software supply chain verification and validation; PyPI malicious package detection; LLM-assisted semantic filtering; graph centrality analysis; scalable malware detection

Check for updates

Academic Editor: Aryya Gangopadhyay

Received: 20 September 2026
Revised: 16 October 2026
Accepted: 19 October 2026
Published: 29 October 2026

**Copyright:** © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

## 1. Introduction

As the critical upstream node of the global software supply chain, open-source repositories like PyPI have evolved into the foundational infrastructure for modern software development. However, this interconnectedness implies that a single compromised package can propagate through automated dependency resolution, potentially contaminating thousands of downstream systems. Due to its openness and low entry threshold, PyPI has become a hotbed for malicious package attacks [1]. According to the Sonatype report [2], the number of malicious packages in third-party libraries increased by over 156% year-on-year in 2024, with a compound annual growth rate of 423% from 2019 to 2024;

Gartner previously projected [3] that by 2025, 45% of organizations worldwide would have experienced attacks on their software supply chains, and malicious packages have emerged as a core security threat to the open-source ecosystem. Therefore, establishing an efficient and accurate automated detection mechanism for malicious PyPI packages has become an urgent requirement for ensuring software supply chain integrity and security.

The attack mode of malicious PyPI packages is characterized by precise social engineering and covert execution. Typosquatting attacks are a common tactic: for example, the malicious package crytic-compilers-0.3.9 impersonates the well-known tool crytic-compile, luring downloads through version fraud [4], and the malicious package loadsh [5] mimics the popular lodash package, causing multiple projects to mistakenly depend on it unknowingly and thus exposing them to security risks. In addition, code injection attacks are another common method, such as the Event-Stream incident, where attackers stole cryptocurrency information by tampering with legitimate package dependencies [6]. Unlike traditional vulnerability exploitation, such attacks exhibit a "shift-left" trend—attackers actively infiltrate upstream development links rather than passively exploiting public vulnerabilities, making traditional detection methods difficult to adapt [7].

However, compared to fields such as traditional malicious code detection and software vulnerability detection, research on malicious package detection started relatively late: searching Google Scholar with keywords "malicious packages" and "PyPI" shows that relevant studies can be traced back to 2016 [8]. More importantly, this field differs significantly from related research, and traditional solutions cannot be directly applied. Existing detection methods face significant challenges in addressing increasingly complex attacks. Section 2 reviews related work in detail and summarizes the limitations of existing approaches. Based on that review, we identify two key challenges:

Structural Importance $\neq$ Security Relevance: Graph centrality metrics can effectively identify structurally important APIs in a package's call structure. However, high centrality does not necessarily indicate security sensitivity—benign generic utilities such as `print`, `len`, and `str.join` frequently appear as high-centrality nodes simply because they are called pervasively throughout the code. Using raw centrality features without semantic refinement introduces substantial noise into the detection model.

Code Evidence Alone Misses Package-Level Context: While API-level behavioral features capture "what is done internally" by the code, malicious packages also exhibit distinctive patterns at the metadata level—such as forged author information, version number fraud, and typosquatting names—that reflect "how the package is disguised externally." Relying solely on code-derived features discards this complementary contextual evidence.

To address these challenges, this paper proposes a detection framework based on centrality-guided candidate identification and LLM-assisted semantic filtering. The core idea is as follows: graph centrality metrics are first used to identify structurally important candidate APIs from the package's API interaction graph. These candidates are then passed to a Large Language Model (Qwen3.5-Plus) that evaluates each API's security relevance from a semantic perspective, removing benign generic APIs and retaining security-sensitive ones. The refined API features are subsequently combined with package-level metadata as auxiliary contextual information and fed into conventional machine learning classifiers for package-level malicious detection.

Importantly, in this work, the API interaction graph is used as a structural feature extraction mechanism rather than as a substrate for graph representation learning. The graph structure serves to compute centrality scores that guide candidate API selection; the subsequent semantic refinement is performed by the LLM, not by learning from the graph topology itself.

The main contributions of this paper are as follows:

(1) Centrality-Guided Candidate API Identification (C1): We apply four complementary graph centrality metrics (Degree, Closeness, Katz, and Harmonic) to the API interaction graph constructed from package source code. These metrics collectively rank structurally important APIs, providing a principled candidate pool for subsequent semantic evaluation.

(2) LLM-Assisted Semantic Refinement (C2): We introduce Qwen3.5-Plus as a semantic screening mechanism that evaluates Top-*K* candidate APIs and distinguishes security-relevant APIs from high-centrality but benign generic ones. This step separates structural importance from security sensitivity, yielding refined API features that better represent malicious behavioral patterns.

(3) Metadata-Augmented Package-Level Detection (C3): We incorporate package-level metadata features extracted from PKG-INFO files as auxiliary contextual information. Combined with the refined API features, this produces a fixed-dimensional package representation that conventional machine learning classifiers can effectively process.

(4) Scalable and Interpretable Detection Pipeline (C4): We demonstrate the practical utility of the proposed framework through LIME-based interpretability, real-world monitoring of the PyPI new-release stream, and efficient package-level processing (approximately 1 s per package on average after the one-time offline construction of the LLM-refined feature vocabulary), supporting potential integration into production environments.

To systematically evaluate the proposed framework, this study addresses five research questions: RQ1 examines detection effectiveness against representative baselines; RQ2 assesses the effectiveness of LLM-assisted semantic filtering; RQ3 quantifies the additional contribution of package-level metadata; RQ4 evaluates performance under highly imbalanced package distributions; and RQ5 analyzes the sensitivity to the Top-*K* candidate API pool size.

Ultimately, this study provides a scalable and interpretable solution for verifying and validating the integrity of the open-source software supply chain, directly addressing the practical needs of modern software quality assurance.

## 2. Related Work

In this section, we review existing malicious package detection techniques from three complementary perspectives—metadata-based, code-based, and machine learning-based—and summarize their limitations, which directly motivate the design of the proposed framework.

### *2.1. Metadata-Based Detection Techniques*

The core idea is to identify abnormal patterns using non-code features such as package names, versions, and author information. For example, Vu et al. [9] detected typosquatting packages through the package name edit distance, and Taylor et al. [5] added package name features and popularity evaluation mechanisms to reduce false positives. Such methods offer fast detection speed and low resource consumption but share a common flaw: lack of verification at the code behavior level—attackers can evade detection by forging homonymous repositories or infiltrating legitimate package maintainers' accounts to inject malicious code, resulting in high false positive rates that fail to meet the precise detection requirements of real production environments.

### *2.2. Code-Based Detection Techniques*

Focusing on code semantics and function call logic, these techniques capture malicious behaviors through static and dynamic analysis. For example, Vu et al. [10] identified code injection by comparing differences between packages and source code repositories, and the MalOSS framework proposed by Duan et al. [11] achieved comprehensive detection by combining static/dynamic analysis with heuristic rules. While these methods offer high precision, they suffer from significant practical bottlenecks: a heavy reliance on manual review or prohibitive computing resources (e.g., MalOSS requires 20 workstations). From a software engineering perspective, the core challenge lies in balancing detection depth with computational scalability. Such "performance taxes" in existing static and dynamic analyses often hinder the seamless integration of security auditing into continuous integration pipelines, making it difficult to keep pace with the real-time evolution of massive ecosystems like PyPI.

### *2.3. Machine Learning-Based Feature Analysis Techniques*

This approach extracts maliciously discriminative features, constructs feature vectors, and inputs them into classification models for training and detection. Early studies by Ohm et al. [12] verified the feasibility of supervised learning algorithms in malicious package detection. Recent learning-based approaches have explored semantic modeling of malicious behavior sequences, such as CEREBRO [13], and deep code representations based on pretrained programming-language models [14]. Sun et al. [15] further integrated deep code behaviors with metadata features for malicious PyPI package detection, demonstrating the complementarity of the two feature sources; Yu et al. [16] leveraged an LLM-enhanced dataset for fine-grained NPM malware tracking. MalGuard [17] employs graph centrality to construct sensitive-API features and further applies LLM-based refinement and LIME-based interpretation. However, such methods still face limitations: existing approaches that use graph centrality treat high structural importance as a proxy for security relevance, yet many benign generic APIs (e.g., `str.join`, `len`) also exhibit high centrality due to frequent calls, introducing noise into the feature space. Moreover, focusing solely on code-derived features without incorporating package-level metadata misses complementary contextual information that can help distinguish malicious from benign packages.

## 3. Preliminary Study on the Discriminative Features of PyPI Malware

To explore the essential differences in behavioral patterns between malicious and benign PyPI packages and motivate the design of the proposed detection framework, this section conducts an empirical analysis of API call structures and metadata distributions. The goal is to understand why centrality-based candidate identification alone is insufficient and how semantic refinement and metadata augmentation can address this gap.

### *3.1. Structural Differences in API Call Behaviors*

Drawing inspiration from Android malware detection [18], where graph centrality analysis has proven effective in identifying frequently called sensitive APIs, we apply this concept to PyPI packages. In the security domain, API call graphs have been widely used in malware detection [15,16], effectively identifying potential threats by analyzing abnormal call patterns and the frequent co-occurrence of malicious API sequences.

The Application Programming Interface (API) call graph is a common data structure used for the static or dynamic analysis of call relationships between API functions within a software system [19]. By constructing a directed graph $G = (V, E)$, where node $V$ represents an API and edge $E$ represents a call relationship, we can quantify the importance of APIs in the execution flow. To comprehensively characterize API topological features

from different dimensions, this empirical study selects four complementary centrality metrics for quantitative analysis:

- **Degree Centrality** [20]: Measures the number of nodes directly connected to an API, normalized by the maximum possible degree, reflecting its activity as a local core.
- **Closeness Centrality** [20]: Measures the reciprocal of the average shortest-path length from an API to the other reachable nodes, reflecting its efficiency in information propagation.
- **Katz Centrality** [21]: Considers not only direct connections but also captures potential long-path dependencies, reflecting the global influence of the API.
- **Harmonic Centrality** [22]: Suitable for analyzing disconnected graphs, ensuring that critical APIs in isolated or edge modules can also be accurately assessed.

Based on these metrics, we constructed API call graphs and calculated scores for malicious and benign packages in the dataset, respectively. The results show distinct structural differences in API usage patterns between the two types of packages (see Table 1).

**Table 1.** Representative high-centrality APIs identified by different centrality metrics in malicious and 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 |
| Benign | getattr | getattr | exist | Exec |
| | map | map | os.getenv | format |
| | os.getenv | Exec | replace | os.getenv |
| | install.run | os.getenv | b64decode | expanduser |
| | 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 |

- **"Short-Path" Attack Mode of Malicious Packages:** Malicious packages tend to directly call system-level or sensitive operation APIs (e.g., subprocess.Popen, os.getenv, base64.b64decode). These APIs often lie on the critical path of the execution flow, exhibiting extremely high centrality scores. Attackers typically use these functions to evade dynamic analysis, create malicious child processes, or perform data obfuscation.
- **"Instrumental" Characteristics of Benign Packages:** In contrast, high-centrality nodes in benign packages are mainly concentrated in basic data processing functions such as int, str, list, and print, reflecting conventional, transparent programming behavior.

This finding indicates that metrics based on graph centrality can effectively capture the "core nodes" of malicious behavior. However, the raw high-centrality list still contains

some benign generic APIs (e.g., join, open), suggesting the need for a more refined semantic filtering mechanism.

### *3.2. Anomalous Patterns in Metadata Distributions*

Beyond code behavior, malicious packages also exhibit distinct anomalous patterns at the metadata level. Through systematic analysis of known malicious packages, we found that while their overall format is generally fixed, they contain rich publisher-related information. Some earlier approaches primarily relied on package name and version information, while more recent studies have incorporated richer metadata alongside code-derived features; the PKG-INFO file, which contains complete metadata, is nonetheless often underexploited. In fact, attackers often expose their anomalous behavioral patterns in this file. We systematically analyzed key fields in the PKG-INFO file and found that malicious packages often possess the following characteristics (see Table 2):

- **Version Number Fraud:** Malicious packages often use non-semantic version numbers (e.g., 1.0.0.1) or abnormal version jumps to confuse users or induce updates.
- **Identity Forgery:** Attackers often fill the author field with random strings or irrelevant names or intentionally leave it blank. As shown in Figure 1, in the malicious package requstss-1.0.0, the description field is filled with meaningless random characters.
- **Typosquatting:** Malicious package names have extremely high similarity (>80%) to popular benign packages, attempting to exploit Typosquatting for attacks.

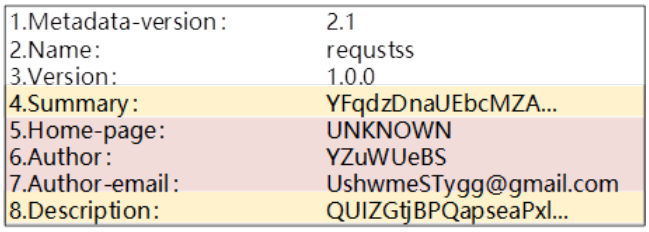



**Figure 1.** Metadata in PKG-INFO.

**Table 2.** Metadata Differences in Malicious and Benign Packages.

| Metadata Dimension | Benign Package Features | Malicious Package Features |
|---|---|---|
| Version Number | Semantic versioning (e.g., X.Y.Z) | Non-semantic (e.g., 1.0.0.1, a1b2)/abnormal jumps |
| Metadata Version | Valid Core Metadata-Version values | Outdated/non-standard values |
| Author & Contact Info | Real author name, complete and compliant email | Random/forged author name, abnormal or missing email |
| Similarity to Popular Packages | Unique name, low similarity (<30%) | Typosquatting popular names, high similarity (>80%) |
| Homepage & Project Links | Valid HTTP/HTTPS links, accessible | Invalid/unreachable links, or missing |
| Summary & Description | Clear semantics, containing function-related keywords | Random characters/gibberish, or missing |
| Package Size | Usually >1 MB, with complete functional modules | Usually <10 KB, containing only malicious scripts |

The above empirical analysis shows that code structural features and metadata features have strong complementarity in characterizing maliciousness. Code features reflect "what is done internally," while metadata features reflect "how it is disguised externally."

## 4. LLM-Assisted Centrality-Guided Detection Framework

Based on the empirical findings in Section 3, this paper proposes a detection framework that uses graph centrality to identify candidate APIs, applies LLM-based semantic filtering to refine the candidate set, and incorporates package-level metadata as auxiliary contextual information for classification.

### *4.1. Framework Overview*

As shown in Figure 2, the framework consists of three stages: (1) **Centrality-Guided API Candidate Construction**, which performs static analysis to extract the API interaction graph, computes centrality metrics, and selects a Top-$K$ candidate pool; (2) **LLM-Assisted Semantic Filtering**, which introduces a Large Language Model to evaluate each candidate API's security relevance, removing benign generic APIs and retaining security-sensitive ones; (3) **Metadata-Augmented Package-Level Classification**, which combines the refined API features with package-level metadata and feeds the resulting feature vector into conventional machine learning classifiers.

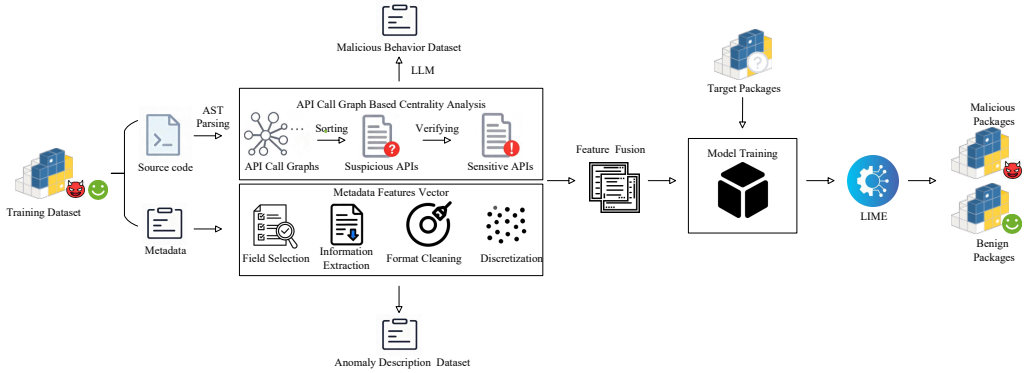



**Figure 2.** Architecture of the Proposed Detection Framework.

The detailed workflow of these three stages is as follows: Step 1: API candidate construction. Static analysis is performed on each package to extract its Abstract Syntax Tree (AST). Based on the AST, an API interaction graph is constructed and centrality values are computed for all nodes. For APIs with the same name, their centrality values are aggregated within each metric. For each centrality metric independently, APIs are ranked according to their aggregated centrality scores, and the top-$K$ ($K = 200, 300, 400, 500, 600$) APIs are selected as the candidate pool for that metric; the four metrics therefore yield four separate candidate pools rather than a single shared ranking. Step 2: LLM semantic filtering. Each metric-specific candidate pool is sent to the large language model (Qwen3.5-Plus) with specialized prompts (see Appendix A) for security sensitivity evaluation. APIs determined to be security-relevant are retained, and their scores under the corresponding metric form a metric-specific refined feature set. Step 3: Metadata-augmented classification. Package-level metadata features are extracted from the PKG-INFO file and discretized. The refined API features are combined with the metadata features to form a fixed-dimensional package representation, which is then fed into a conventional machine learning classifier for training and detection.

### *4.2. API Candidate Construction via Graph Centrality*

#### 4.2.1. Graph Centrality Metrics

As established in the empirical analysis (Section 3.1), graph centrality can identify structurally important APIs in a package's execution flow. This study computes four complementary graph statistics to rank candidate APIs from different structural perspectives. The specific definitions are as follows:

- Degree Centrality: Measures the number of nodes directly connected to an API, normalized by the maximum possible degree, reflecting the local activity of the API. It is used to identify the most active "hotspots" in the call graph, as malicious packages frequently invoke specific sensitive APIs (e.g., `os.system`):

$$C_{\mathrm{D}}(v) = \frac{\deg(v)}{N-1} \tag{1}$$

- Closeness Centrality: Measures the reciprocal of the average shortest-path length from a node to the other reachable nodes, reflecting the propagation efficiency of the API in the call chain. It characterizes the "short-path" attack mode; since attackers aim for efficient execution, malicious logic often maintains a minimal distance to critical system sinks:

$$C_{\mathrm{C}}(v) = \frac{N-1}{\sum_{\substack{u \in V \\ u \neq v}} d(v,u)} \tag{2}$$

- Katz Centrality: Measures the global influence of a node by weighting indirect neighbors on long paths via an attenuation factor. It addresses hidden dependencies and is essential for detecting obfuscated malicious chains where sensitive calls are buried under multiple indirect layers to evade detection:

$$C_{\mathrm{K}}(v) = \alpha \sum_{u \in V} A_{vu} C_{\mathrm{K}}(u) + \beta \tag{3}$$

- Harmonic Centrality: Sums reciprocal shortest-path distances to reachable nodes and is well suited to disconnected graphs, since unreachable node pairs contribute zero rather than an infinite distance. It is specifically integrated to handle the standalone malicious script fragments common in PyPI malware, ensuring that critical APIs in isolated modules are not overlooked:

$$C_{\mathrm{H}}(v) = \sum_{\substack{u \in V \\ u \neq v}} \frac{1}{d(v,u)} \tag{4}$$

Compared to Android malware [18,19], Python malicious packages are typically smaller in volume [15], and attackers often write malicious code directly in the global scope. Consequently, although attackers may call certain APIs, there are often no direct call relationships between these APIs, causing their raw centrality values to be zero. We note that adding a constant to all centrality values does not change the within-package ranking of APIs; the purpose of this adjustment is therefore not to alter the ranking but to disambiguate the feature-value encoding: an API that is called in the package but structurally isolated (raw centrality $= 0$) would otherwise be indistinguishable, in the feature vector, from an API that is entirely absent (missing-value encoding $= 0$, see Section 4.2). We therefore add a default value of 1 to the centrality score of every API that is present in the package, so that any called API receives a strictly positive feature value ($\geq 1$) regardless of its topological position, while absent APIs are encoded as 0.

Regarding the choice of the centrality metric set, the four metrics adopted here cover complementary structural perspectives: local connectivity (Degree), global reachability along short paths (Closeness), attenuated long-range influence (Katz), and distance-robust reachability on disconnected graphs (Harmonic). These perspectives jointly match the structural patterns observed in PyPI malware (Section 3.1). Additional centrality measures (e.g., Betweenness, Eigenvector, or PageRank centrality) could in principle be incorporated into the same pipeline; however, each additional metric multiplies the offline vocabulary-

construction cost, since one LLM-filtered feature vocabulary must be built per metric and per swept $K$ value (Section 4.2.2), and the four metrics already yield consistent, closely clustered results across all swept pool sizes (Section 5.5). We therefore consider the current set sufficient for capturing the relevant structural signals, and we leave the extension to additional centrality measures as future work.

#### 4.2.2. LLM-Based Sensitive API Semantic Filtering

After computing the centrality scores for all APIs, each centrality metric independently ranks the APIs by their aggregated scores, and the top $K$ ($K = 200, 300, 400, 500, 600$) APIs are selected as the candidate pool for that metric.

However, using these raw high-ranking centrality APIs directly as features has a fundamental limitation: **structural importance does not equal security sensitivity**. As demonstrated in Section 3.1, many benign generic APIs (e.g., `str.join`, `len`) exhibit high centrality simply because they are called pervasively, not because they are security-relevant. Including such APIs in the feature space introduces noise and degrades detection quality.

To address this, this study introduces a Large Language Model (Qwen3.5-Plus) as a **semantic screening mechanism**. We design specific prompt engineering (see Appendix A) to guide the LLM to act as a cybersecurity expert and evaluate each Top-$K$ candidate API's security relevance. Importantly, the LLM's role is not to perform end-to-end malware classification; rather, it serves as a **semantic refinement layer** that separates structural importance from security sensitivity. The LLM is prompted to eliminate high-centrality but harmless generic APIs (e.g., `print`, `len`, `isinstance`); meanwhile, for APIs with genuine high-risk potential such as `subprocess.Popen`, `os.system`, and `eval`, the LLM retains them based on semantic understanding even if their absolute call frequency is not the highest in certain obfuscated codes. Ultimately, only the APIs determined by the LLM to be security-sensitive are retained, and their scores under the corresponding centrality metric form a metric-specific refined feature set. The dimensions of the four metric-specific refined feature sets are summarized in Table 3.

The full processing rules of the API feature vector $V_{\text{API}}$ are as follows: (i) Per-metric independence: The four centrality metrics are evaluated independently—each metric yields its own refined feature set (Table 3) and its own detection model, and no cross-metric aggregation or concatenation of the four feature vectors is performed; this allows us to compare the discriminative power of different centrality measures. In the main experiments (Tables 4–6), the four metric-specific classifiers are trained independently under identical settings, and the best-performing metric-specific model—the Harmonic-centrality-based one—is adopted as the final detection model reported in the headline results. The ablation study in Table 5 (Models A–D) is conducted on this final Harmonic-based pipeline to isolate the contribution of each component (raw vs. LLM-refined API features, and metadata augmentation). (ii) Feature value type: The feature value of each refined API is the package-specific (adjusted) centrality score of that API under the corresponding metric, rather than a binary presence indicator, so that the model can exploit the graded structural importance of each security-sensitive API within a package. (iii) Merging of repeated APIs: If the same API (identified by its fully qualified name) appears multiple times in a package's API interaction graph, its centrality scores are aggregated by averaging. (iv) Missing-value encoding: If a refined API in the fixed vocabulary is absent from a package, its feature value is encoded as 0.

All LLM evaluations were performed through the official Qwen3.5-Plus API, accessed with an API key. We emphasize that Qwen3.5-Plus is used strictly as an off-the-shelf, pre-trained commercial model: we neither train nor fine-tune it, and all queries are executed remotely on the provider's cloud infrastructure—no local deployment or local inference

is involved, so our experiments cover the inference stage of the LLM only. Following the role-based prompt engineering commonly adopted for security-oriented API analysis [17], the prompts instruct the LLM to act as a cybersecurity expert (see Appendix A), and each candidate API was evaluated with an independent query under the default sampling configuration of the API. In the prompt template (Appendix A), the `<INSERT_CODE_HERE>` slot is filled with the fully qualified name and function signature of the candidate API under evaluation; no package-specific code context is inserted. Consequently, the LLM's judgment is a generic, package-independent assessment of each API's security sensitivity rather than a per-package verdict, which is precisely what enables the one-time, offline construction of the fixed vocabularies reused by all subsequent experiments. The sampling parameters (e.g., temperature and top-$p$) were left at the provider's default values and no random seed was fixed; since the filtering is a single offline pass, per-call statistics such as the exact model snapshot identifier and the failure and retry rates were not systematically logged. The prompt template (version v1.0) was used unchanged throughout all experiments. Because this filtering step is a one-time offline vocabulary-construction process rather than an online inference component, the returned judgment (security-sensitive or benign generic) was parsed to retain or discard the corresponding API; failed or malformed responses were retried, and APIs that could not be reliably evaluated were excluded from the corresponding refined feature set. The resulting metric-specific vocabularies were then fixed and reused unchanged in all subsequent experiments, so no LLM call is required at package-level detection time.

**Table 3.** Feature Set Dimensions.

| Centrality | Closeness | Degree | Katz | Harmonic |
|---|---|---|---|---|
| Dimensions | 265 | 255 | 294 | 135 |

**Table 4.** Effectiveness comparison with baselines.

| Approach | Precision (%) | Recall (%) | F1-Score (%) |
|---|---|---|---|
| VirusTotal | 92.8 | 81.1 | 86.6 |
| OSSGadget | 79.6 | 86.9 | 83.1 |
| Bandit4Mal | 84.7 | 95.7 | 89.9 |
| MalGuard [b] | 94.9 | 95.0 | 94.9 |
| Our Model | 96.3 | 95.9 | 96.1 |

[b] The MalGuard result is obtained from our reimplementation of the complete methodology described in the original study, evaluated under the dataset, preprocessing procedure, package-level split, and evaluation protocol adopted in this work. It is therefore not directly comparable with the performance reported in the original MalGuard study.

**Table 5.** Effectiveness comparison of variant models.

| Model | Feature Set | LLM | Precision (%) | Recall (%) | F1-Score (%) | Δ F1 |
|---|---|---|---|---|---|---|
| A | Top-500 Raw Harmonic | × | 88.5 | 93.1 | 90.7 | — |
| B | Metadata Only | — | 88.8 | 86.6 | 87.7 | — |
| C | Top-500 Refined Centrality | ✓ | 94.9 | 95.0 | 94.9 | +4.2 (vs. A) |
| D | Refined Centrality + Metadata | ✓ | 96.3 | 95.9 | 96.1 | +1.2 (vs. C) |

**Table 6.** Performance comparison under different dataset ratios.

| Dataset Ratio | Precision (%) | Recall (%) | F1-Score (%) |
|---|---|---|---|
| 1:1 | 96.3 | 95.9 | 96.1 |
| 1:50 | 91.7 | 94.3 | 93.0 |

### *4.3. Package-Level Metadata as Auxiliary Context*

Beyond code-level API features, package metadata provides complementary contextual information that helps distinguish malicious from benign packages at the package level. Based on the analysis in Section 3.2, we extract nine categories of key features from the PKG-INFO file: (1) the package name, (2) the package size, (3) the version number, (4) the metadata version, (5) the author name, (6) the author e-mail, (7) the homepage and project links, (8) the summary, and (9) the description. For presentation clarity, Table 2 groups these nine features into seven rows: the author name and author e-mail are merged into the "Author & Contact Info" row, and the summary and description are merged into the "Summary & Description" row. All nine features are discretized into numerical indicators according to the benign/malicious patterns identified in Section 3.2 and summarized in Table 2: (i) The package name is encoded by its similarity to popular packages, using the similarity thresholds reported in Table 2; the reference list of popular packages is derived from an external, publicly available PyPI download ranking and is constructed independently of the evaluation dataset, so no sample information—in particular nothing from the test partition—participates in its construction; names with intermediate similarity (30–80%) are conservatively treated as normal (encoded as 0), since only strong similarity (>80%) constitutes a reliable typosquatting signal while weak-to-moderate similarity is common among legitimately named packages. (ii) The package size is binned according to the size distribution observed in the collected dataset (below 10 KB, 10 KB–1 MB, 1–10 MB, and above 10 MB). (iii) The version number and metadata version are encoded as indicators of non-semantic or abnormal version values and outdated/non-standard metadata versions, respectively, where the "abnormal version jump" pattern is identified with reference to the publicly available release history of the corresponding legitimate packages on PyPI, which is likewise external to the evaluation dataset. (iv) The author name and e-mail are encoded as indicators of missing, random, or otherwise non-compliant values. (v) The homepage and project links are encoded as indicators of missing or unreachable links. (vi) The summary and description are encoded as indicators of gibberish content, missing text, or the absence of function-related keywords.

**File Hashes:** Regarding the SHA-256 digest of each release file, we deliberately do not use it as a model feature: a cryptographic hash is a unique content identifier, so any modification of a package—however small—yields a completely different digest, and a hash observed in the training data carries no discriminative signal for previously unseen packages. In our pipeline, digests serve only bookkeeping purposes, such as exact-duplicate removal during dataset construction and potential cross-checking against public malicious-hash blocklists. Hash-based exact matching is thus an orthogonal, blacklist-oriented complement to the generalizable feature-based detection pursued in this work.

**Package Name and Size:** Malicious package names are often deliberately similar to popular packages to mislead users and increase the likelihood of being downloaded. On the other hand, package size is also an important indicator: malicious packages are usually small in size because they generally only contain simple scripts for executing malicious behaviors. In the dataset we collected, approximately 83% of malicious packages are smaller than 10 KB, about 16.5% are between 10 KB and 1 MB, roughly 0.38% fall between 1 MB and 10 MB, and only a very small number (about 0.12%) exceed 10 MB.

**Version:** Packages that are frequently updated without substantial changes may be malicious, as attackers may use this method to evade detection. Malicious packages may use older or non-standard metadata formats to conceal their true intentions.

**Author Information:** Malicious attackers often use fake or anonymous author information, such as random strings or irrelevant names. Although PyPI provides a signing mechanism to verify the source and integrity of packages, malicious authors can still evade detection through various means.

**Homepage and Project Links:** Benign packages usually provide accessible homepage or project URLs pointing to their repositories or documentation, whereas malicious packages frequently omit these fields or fill them with invalid, unreachable addresses. The presence and reachability of these links therefore serve as an additional credibility signal.

**Description:** Malicious packages often use exaggerated or false statements in their summary and description fields to induce downloads, or deliberately make the description vague to conceal real behaviors.

### *4.4. Package-Level Feature Construction and Classification*

The refined API feature vector $V_{\text{API}}$ and the discretized metadata feature vector $V_{\text{Meta}}$ are concatenated to form a fixed-dimensional package representation $V_{\text{Final}} = V_{\text{API}} \oplus V_{\text{Meta}}$. This concatenation is a straightforward engineering-level integration step: since API behavioral features and metadata features characterize different aspects of a package (internal code behavior vs. external attribute patterns), direct concatenation preserves the complementary information from both sources without introducing additional model complexity. This design is consistent with recent work on fusing heterogeneous security data sources for classification, such as the optimized SVM-based fusion of multiple IoT security datasets for network security management proposed by Kumar et al. [23]; in our setting, however, the fusion operates on package-level API and metadata features rather than network-traffic data. For classifier selection, this framework supports various machine learning algorithms. Five mainstream classifiers—Random Forest (RF), Naive Bayes (NB), Extreme Gradient Boosting (XGBoost [24]), Multi-Layer Perceptron (MLP), and Support Vector Machine (SVM)—were considered during preliminary classifier selection. Random Forest was subsequently adopted as the default classifier for the main experiments. For completeness, we briefly note the main limitations of the considered algorithms: Naive Bayes relies on a strong conditional-independence assumption and degrades when features are strongly correlated; SVM scales poorly to large training sets and is sensitive to kernel and hyperparameter choices; MLP requires careful hyperparameter tuning and sufficient training data, and is prone to overfitting on tabular feature vectors; XGBoost performs well but involves more hyperparameters and can overfit small or noisy datasets; and Random Forest is robust to noise and requires little tuning, albeit at the cost of reduced interpretability and a higher memory footprint. All classifier training and testing were performed on the same machine used for the timing measurements reported in Section 5.6.3 (Intel Core i5-14400F CPU, 32 GB of RAM, 64-bit Windows 11).

### *4.5. Model Interpretability Analysis*

To address the "black-box" problem of malicious package detection models and improve the credibility and engineering practicality of detection results, this paper introduces the LIME (Local Interpretable Model-agnostic Explanations) [25] algorithm to perform local interpretable analysis on samples determined to be malicious by the model. The core objectives are to clarify which sensitive API features and abnormal metadata features dominate the model's malicious judgment, quantify the contribution of each feature, and explain the influence mechanism of feature value states on the judgment result.

We integrate the LIME algorithm into the machine learning model to enhance its interpretability by identifying the top 10 non-zero features that have the most significant impact on the model's prediction. For each of these features, we cross-reference the API and metadata names with the datasets generated in Steps 1 and 2 to retrieve the potential malicious behaviors associated with each feature. Finally, we output all suspicious features and their respective potential malicious behaviors, providing comprehensive and interpretable results for further investigation. We emphasize that these behavior descriptions are LLM-generated auxiliary annotations rather than analytic evidence: the attribution itself is provided solely by the LIME weights, and the textual annotations serve only as review cues to accelerate manual verification. Since LLM-generated descriptions may occasionally be inaccurate, behavior annotations are attached only to features with positive LIME weights (which support the malicious verdict), whereas features with negative weights—which correspond to benign utilities and argue against the malicious verdict—receive no malicious behavior annotation. The annotations for the top-ranked positive contributors should always be interpreted jointly with the corresponding LIME weights and manually verified before being relied upon in incident response.

## 5. Evaluation

This section evaluates the proposed framework through five research questions:

- **RQ1:** How does the proposed framework compare against representative baselines in detection effectiveness?
- **RQ2:** How effective is LLM-assisted semantic filtering in improving centrality-based features?
- **RQ3:** What additional contribution does package-level metadata provide?
- **RQ4:** How does the framework perform under highly imbalanced package distributions?
- **RQ5:** How sensitive is the framework to the choice of the Top-$K$ candidate API pool size?

**Baselines.** To evaluate the performance of the proposed framework against existing methods, we selected four representative baselines: VirusTotal [26], OSSGadget [27], Bandit4Mal [28], and MalGuard [17]. VirusTotal is an online detection platform that provides signature-based scanning of uploaded packages. OSSGadget detects potential backdoors and malicious code in software packages. Bandit4Mal constructs Abstract Syntax Trees and runs analysis plugins to discover common security issues in Python code. MalGuard uses graph centrality for malicious PyPI package detection and serves as the most directly comparable centrality-based baseline. For MalGuard, we reimplemented the complete methodology described in the original study and evaluated it under the experimental setting adopted in this work. The resulting performance is 94.9% F1-score. The difference from the performance reported in the original MalGuard study is expected because the two evaluations use different datasets and experimental protocols. Therefore, the original reported result and our reimplementation result should not be interpreted as directly comparable absolute performance figures.

**Evaluation Metrics.** This study adopts three commonly used binary classification evaluation metrics: Precision, Recall, and F1-score. Precision represents the proportion of samples detected as malicious packages that are actually malicious; Recall reflects the proportion of malicious packages successfully identified by the model among all malicious packages; F1-score is the harmonic mean of Precision and Recall. These three metrics were selected to remain directly comparable with the baselines (including MalGuard) and because they focus on the malicious class, which is the primary operational concern in this detection task; additional metrics such as Accuracy and ROC-AUC are acknowledged

as complementary perspectives, which we leave for extended reporting in future work. Equation (5) is defined as follows:

$$\text{F1-score} = 2 \times \frac{\text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}} \tag{5}$$

**Classifier.** Random Forest was selected as the default classifier based on its performance on the training-side validation procedure and was subsequently fixed for the main experiments and held-out test evaluation.

### *5.1. Dataset Construction and Preprocessing*

To establish a comprehensive and representative experimental benchmark, this study consolidates multiple authoritative open-source malicious package datasets. The malicious samples are primarily sourced from the repositories maintained by Duan et al. [11], Ohm et al. [12], and Guo et al. [29]. These datasets encompass a diverse spectrum of prevalent attack vectors within the PyPI ecosystem, including Typosquatting, Dependency Confusion, and Code Injection. After systematically removing duplicates and unparsable corrupted files, a total of 10,743 unique malicious package samples were retained.

Concurrently, we acquired 10,000 benign packages via the official PyPI API, adhering to the following inclusion criteria:

- **Persistence Verification:** The package must have been hosted on PyPI for a duration exceeding 90 days without official removal.
- **Popularity Verification:** The cumulative historical download count must surpass 1000.

To ensure a rigorous evaluation, we adopted a package-level random splitting strategy: since multiple versions of the same package often share significant code and metadata similarities, a sample-level split would likely cause data leakage. We randomly partitioned the unique package names into 80% for training and 20% for testing. All experiments, including feature construction, classifier training, and evaluation, were conducted on an Intel Core i5-14400F CPU (10 cores, 16 threads) with 32 GB of RAM, running 64-bit Windows 11.

To prevent information leakage, package-level splitting was performed before corpus-dependent feature construction. For each centrality metric, candidate API ranking, Top-$K$ selection, and LLM-based semantic filtering were performed using the training partition only. The resulting metric-specific feature vocabularies were then fixed and applied to the held-out test set.

**Distributional Analysis of the Dataset.** To characterize the composition of the consolidated dataset, we further analyzed the release-date and package-size distributions of both classes, as shown in Figure 3. (1) Temporal distributions: The two classes exhibit a noticeable temporal skew. The malicious packages are heavily concentrated in 2022–2023 (approximately 77% of the class; median year 2023), whereas the benign packages span 2005–2025 with the bulk released between 2015 and 2023 (median year 2020). We acknowledge this skew as largely inherent to the task: malicious packages are, by construction, disclosed only after their malicious behavior is identified, and public malicious-package datasets aggregate recently disclosed campaigns, while the benign inclusion criterion (hosted $>$ 90 days) excludes recently uploaded packages by design. Since both the training and test samples are drawn randomly from the same temporal pool, the reported within-distribution performance is not inflated by this skew; however, cross-temporal generalization (training on the past and detecting future threats) cannot be assessed under the current protocol, and we acknowledge it as a limitation and future work. (2) Package-size distributions: The two distributions overlap substantially in the lower range (median: 18 KB vs. 21 KB for

malicious vs. benign), while the benign distribution exhibits a heavier upper tail (90th percentile: 209 KB vs. 66 KB; two-sample Kolmogorov–Smirnov test: KS statistic = 0.339, $p < 0.001$), reflecting large legitimate projects that have no malicious counterpart. This asymmetry is unlikely to inflate detection performance, since it would if anything bias the classifier toward classifying large packages as benign, i.e., against the malicious class. Regarding popularity, the benign inclusion criteria (hosted $>$ 90 days, $>$1000 cumulative downloads) already exclude freshly uploaded or near-zero-download packages by design, mitigating the extreme popularity confound.

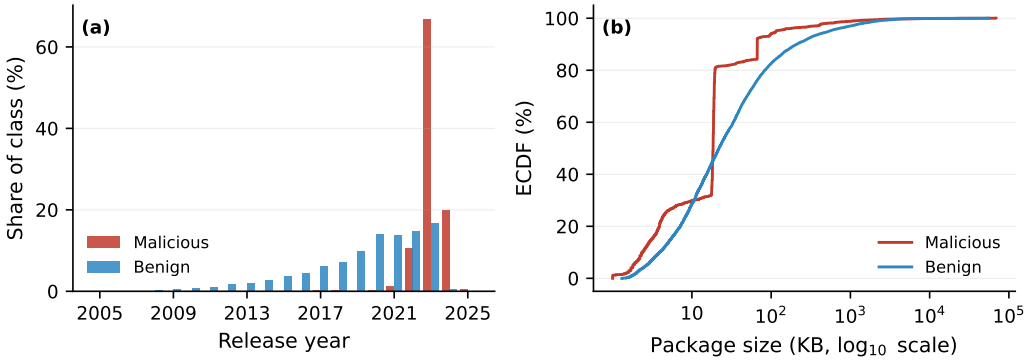



**Figure 3.** Distributional analysis of the dataset. (**a**) Release-year distributions of the malicious and benign packages (share of each class). (**b**) Empirical cumulative distributions of the package source-tree sizes (logarithmic scale).

### *5.2. RQ1: Detection Effectiveness Against Baselines*

**Experimental Setup.** We combined benign and malicious packages and divided the dataset into 80% training and 20% test sets using the package-level splitting strategy. We used the same dataset to evaluate all baselines.

**Results.** Table 4 shows the performance comparison. The proposed method achieves the highest precision (96.3%), recall (95.9%), and F1-score (96.1%) among all methods. Compared to MalGuard, the most competitive baseline, the proposed method improves precision by 1.4 percentage points and F1-score by 1.2 percentage points.

For MalGuard, we reimplemented the complete methodology described in the original study and evaluated it under the experimental setting adopted in this work. Under the same evaluation setting, our method improves the F1-score over the reimplemented MalGuard pipeline from 94.9% to 96.1%, corresponding to a gain of 1.2 percentage points. The original MalGuard study reported its performance under a different dataset and experimental protocol, so its published score is not directly compared with the results in Table 4. Therefore, the MalGuard result in Table 4 is included as a same-setting baseline rather than as the original published result.

**Analysis.** The performance advantage of the proposed framework can be attributed to its LLM-assisted semantic filtering mechanism. Rule-based and pattern-based detection methods (e.g., Bandit4Mal, OSSGadget) tend to misclassify normal behaviors as malicious operations due to their reliance on predefined heuristics. VirusTotal's signature-based approach lags behind in detecting emerging threats. MalGuard, the most competitive baseline, combines graph-centrality-based sensitive API extraction with LLM-based feature refinement and LIME-based interpretation. Unlike the original MalGuard pipeline, which aggregates and ranks centrality scores across malicious packages to construct a unified sensitive-API feature set, our framework performs metric-specific candidate ranking and Top-$K$ selection for each centrality measure, followed by LLM-based semantic filtering and metric-specific feature construction that explicitly separates structural importance from security sensitivity. We further incorporate package-level metadata as auxiliary contextual

information for package-level classification. This combination yields higher-quality features and improved detection performance under the same evaluation setting.

**Statistical Robustness.** Beyond Precision, Recall, and F1-score, we further characterize the detection errors and the statistical stability of the proposed method. Based on the reported Precision (96.3%) and Recall (95.9%), the model yields a false negative rate (FNR) of 4.1% and a false positive rate (FPR) of approximately 4.0%; the Matthews correlation coefficient (MCC) reaches 0.92 and the area under the ROC curve (ROC-AUC) reaches 0.99, indicating strong class separation beyond a single operating point. To quantify sampling uncertainty, we performed a bootstrap analysis (1000 resamples of the held-out test predictions), obtaining 95% confidence intervals of [95.8, 97.5] for Precision, [94.1, 96.0] for Recall, and [95.3, 96.5] for F1-score. Furthermore, re-training under five random seeds, each with a re-drawn shuffle and an 80/20 split, yields F1-scores within 95.9–96.8%, a variation of less than one percentage point, confirming that the reported performance is stable with respect to sampling variability. Finally, a paired McNemar test on the per-package predictions of our method and the reimplemented MalGuard baseline on the same held-out test set confirms that the performance improvement of our method over the most competitive baseline is statistically significant ($\chi^2 = 5.80$, $p = 0.016$).

### *5.3. RQ2 and RQ3: Contribution of LLM Semantic Filtering and Metadata*

To quantify the independent contributions of LLM-assisted semantic filtering and package-level metadata, this section designs four variant models. All models were trained and evaluated on the same training and test sets using Random Forest (RF) as the classifier. The definitions are as follows:

- **Model A (Raw Centrality):** Uses the top-500 APIs with the highest raw Harmonic centrality values as features (500 dimensions), without LLM screening. This serves as the baseline to evaluate the performance of pure structural features.
- **Model B (Metadata Only):** Uses only the nine categories of metadata features. This evaluates the standalone capability of package-level non-code features.
- **Model C (Centrality + LLM):** Introduces LLM-assisted semantic screening on top of Model A. The LLM removes high-centrality but benign generic APIs, retaining only security-sensitive APIs as features.
- **Model D (Refined Centrality + Metadata):** The complete proposed framework. Combines the LLM-refined API features from Model C with the metadata features from Model B.

Table 5 presents the results. We note that the dimension difference between Models A and C reflects exactly the LLM filtering step: Model A uses the full raw top-500 Harmonic candidate pool (500 dimensions), whereas Model C uses the LLM-refined vocabulary, which retains 135 of the 500 candidate APIs for the Harmonic metric (Table 3), corresponding to a filtering rate of 73%. The +4.2 percentage-point F1 gain from Model A to Model C is therefore directly attributable to the removal of high-centrality but generic APIs by LLM semantic filtering. **RQ2 (LLM Semantic Filtering):** Comparing Model A and Model C, LLM semantic filtering improves F1-score from 90.7% to 94.9%, a gain of +4.2 percentage points. This confirms that removing high-centrality but benign generic APIs through semantic screening is the primary mechanism for improving detection quality. **RQ3 (Metadata Contribution):** Comparing Model C and Model D, adding metadata features further improves F1-score from 94.9% to 96.1%, an additional gain of +1.2 percentage points. While metadata provides a meaningful improvement, the magnitude of this gain is smaller than that of LLM filtering, indicating that the main methodological contribution remains the centrality-guided LLM semantic refinement.

### *5.4. RQ4: Performance Under Class Imbalance*

To verify the adaptability of the proposed method in real malicious code detection scenarios, this section evaluates the framework under an extremely imbalanced malicious-to-benign ratio of 1:50, which resembles the distribution of newly uploaded packages in the wild.

**Experimental Setup.** To eliminate any risk of data leakage, the 1:50 imbalanced evaluation set was constructed exclusively from the held-out test partition of the main experiment, which neither the trained model nor the feature vocabulary observed: 35 malicious packages were randomly sampled from the test partition, and 1750 benign packages were drawn from the 2000 benign packages in the test partition, yielding a malicious-to-benign ratio of exactly 1:50 (35:1750, 1785 packages in total). The model trained in the main experiment was then directly applied to this evaluation set without any retraining. Precision, Recall, and F1-score were selected as core evaluation metrics.

Table 6 shows that under the 1:50 imbalanced dataset, although the model sacrifices a small amount of precision (4.6 percentage points) due to the increased complexity of the scenario, the recall only decreases slightly (1.6 percentage points). This indicates that the framework maintains reasonable adaptability under realistic class imbalance conditions. Given the limited number of malicious samples in this setting, we additionally report bootstrap 95% confidence intervals (1000 resamples over the 1785 evaluation packages): Precision = 91.7% [81.5, 100.0], Recall = 94.3% [84.2, 100.0], and F1-score = 93.0% [85.3, 98.4]. The intervals are wide but bounded, which transparently quantifies the statistical power of this extreme-imbalance evaluation; we acknowledge the limited statistical power as a limitation in the Conclusions.

### *5.5. RQ5: Sensitivity to Top-K Candidate Pool Size*

**Experimental Setup.** To study the impact of the Top-$K$ parameter on detection effectiveness, experiments were conducted with different $K$ values ($K = 200, 300, 400, 500, 600$). The parameter $K$ controls the breadth of the candidate API pool before LLM semantic screening. Since the candidate pool itself changes with $K$, the LLM-based semantic filtering was re-executed for every swept $K$ value, yielding a metric-specific refined feature vocabulary for each (metric, $K$) combination; Random Forest was used as the classifier.

**Results.** As shown in Figure 4, as $K$ increases, detection effectiveness improves substantially across all four centrality feature sets. When $K$ increases from 200 to 500, the F1-score increases by 3.9–6.3 percentage points across the four metrics (from 91.8% to 96.1% for Harmonic, from 91.8% to 95.7% for Degree, from 91.4% to 95.5% for Closeness, and from 89.1% to 95.4% for Katz), indicating that a broader candidate pool captures more suspicious APIs for LLM evaluation. When $K$ increases further from 500 to 600, the F1-score changes only marginally (at most +0.3 percentage points across all metrics). Notably, the Harmonic-centrality feature set achieves the highest F1-score among all metric–$K$ combinations (96.1% at $K = 500$, rising only to 96.2% at $K = 600$), which is why the Harmonic-based model is adopted as the final detection configuration. Since the additional candidate APIs gained beyond $K = 500$ yield a negligible improvement while enlarging the pool by 20%, $K = 500$ was adopted as the default setting in all subsequent experiments.

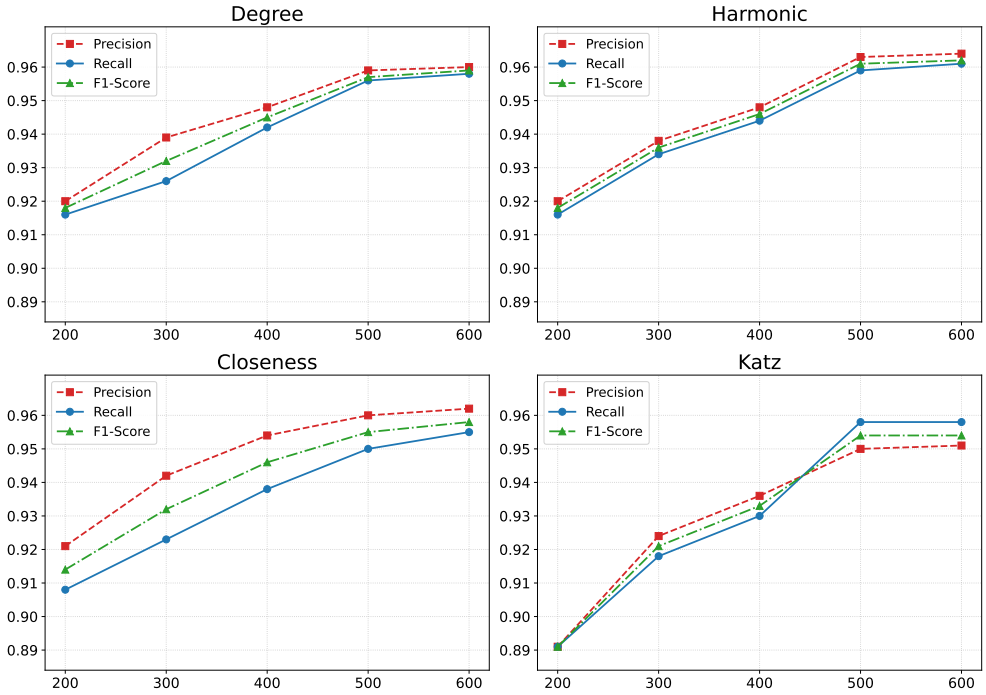



**Figure 4.** The effectiveness of the Random Forest (RF) model trained by top-*K* APIs in four different centrality feature sets.

### *5.6. Practical and Retrospective Analysis*

Beyond the formal research questions above, this section provides supporting empirical analysis of the proposed framework through a retrospective case analysis, a LIME-based interpretability analysis, and a practical deployment analysis. These analyses are intended to illustrate the behavior and operational utility of the detection pipeline rather than to constitute additional research questions.

#### 5.6.1. Retrospective Case Analysis

To further explore the detection ability of the proposed model in the face of covert attacks, this section selects a real malicious package sample from the test set for detailed retrospective analysis. The purpose is to examine whether the proposed detection framework produces meaningful API-level and metadata-level indicators for a package that was subsequently confirmed as malicious. This retrospective analysis is intended to illustrate the behavior of the proposed detection pipeline rather than to claim independent discovery or advance warning.

The malicious package crytic-compilers-0.3.11 is an impersonation of the well-known smart contract compilation tool crytic-compile [4]. The crytic-compilers typosquatting campaign spanned multiple malicious releases (e.g., 0.3.9 and 0.3.11) that exhibited different malicious behaviors [4]; the Introduction references the earlier 0.3.9 release, whereas this retrospective case study focuses specifically on version 0.3.11, which is the sample present in our test set. Taking advantage of developers' trust in version updates, the attacker deliberately set the version number of the malicious package to 0.3.11, slightly higher than the latest version 0.3.7 of the legitimate package at that time, inducing the package manager to automatically download and install the malicious update. It contains a Windows executable file named s.exe, which steals sensitive information once installed.

The proposed model correctly assigned a malicious verdict to this sample, mainly due to the synergistic effect of centrality-derived API features and metadata context. The specific analysis is as follows:

(1) Centrality-guided evidence: Although the attacker attempted to hide the malicious logic in the normal installation process, after constructing the API interaction graph, its behavior pattern exposed significant abnormalities. In the execution path of setup.py, the subprocess.Popen and platform.system nodes showed extremely high topological importance. Instead of performing conventional compilation configuration during the installation phase, the malicious package directly called platform.system to check the operating system type, and then executed the built-in s.exe through subprocess call. This "check-and-execute" structure led to a surge in the centrality values of related APIs, far exceeding those of similar APIs used for auxiliary construction in benign packages.

(2) Metadata anomaly: The model keenly captured flaws in the attacker's disguise at the package level:

- **Abnormal version strategy:** Although the 0.3.11 version number used by the attacker was format-compliant, its release time had a gap with historical versions, and the version number fell into the discrete interval of "abnormal jump" in the metadata feature vector.
- **Name similarity:** The package name crytic-compilers had an extremely high similarity (>0.9) with the popular benign package crytic-compile, which directly triggered the typosquatting detection threshold in the metadata module.

(3) Combined classification: Detection based on a single dimension has limitations: looking only at the code, its setup.py structure is simple and easily confused with a normal custom installation script; looking only at the metadata, the version number 0.3.11 itself complies with semantic specifications. However, combining the "high-centrality system execution behavior" from the LLM-refined API features with the "highly suspicious typosquatting name" from the metadata, the combination of these two sources supported the malicious prediction, and the package was ultimately determined to be malicious.

Through the analysis of crytic-compilers, it can be seen that the proposed method can effectively cope with complex attacks combining package impersonation (typosquatting) and malicious code hiding. Centrality-guided API analysis successfully revealed the abnormal execution flow during the installation phase, while the LLM's semantic evaluation confirmed the security relevance of the identified APIs. Metadata features provided complementary package-level evidence of typosquatting intent. The combination of these two sources enabled accurate detection of such threats.

#### 5.6.2. LIME-Based Interpretability Analysis

The LIME analysis further illustrates how the proposed classifier attributes an individual malicious verdict to sensitive API and metadata features. To validate the reliability of the model's decision-making process, we applied the LIME algorithm to the sample analyzed in the retrospective case analysis and report the feature attributions in Table 7. The interpretability analysis offers two distinct advantages over traditional black-box detection:

- **Precise Feature Attribution:** Unlike the manual analysis in the preceding case analysis, which requires inspecting the entire source code, LIME automatically highlights the top-ranked risk factors. The explanation identifies `subprocess.Popen` and `platform.system` as the two largest positive contributors in this case, indicating that the model prioritizes high-risk execution logic over benign code structures.
- **Semantic Explanation by LLM:** Beyond simple feature importance, the integrated LLM provides auxiliary semantic descriptions (e.g., interpreting `subprocess.Popen` as "Spawn a subprocess to execute system commands"). These descriptions are LLM-

generated review cues rather than analytic evidence, and they allow security analysts to grasp the likely threat type—command execution and typosquatting—at a glance, reducing the time required for manual verification; they should, however, always be validated against the LIME weights and, for consequential decisions, against the package source code itself.

**Table 7.** LIME-based feature attribution for the malicious package analyzed in the retrospective case study. Positive weights contribute toward the malicious classification, whereas negative weights argue against it. Behavior annotations are LLM-generated review cues rather than analytic evidence.

| Feature | LIME Weight | Direction | LLM-Assisted Behavior Annotation |
|---|---|---|---|
| subprocess.Popen | +0.4922 | Malicious | Execute system commands |
| platform.system | +0.4406 | Malicious | Gather information about the underlying operating system |
| find_packages | +0.1728 | Malicious | Enumerate available packages during installation (potential environment probing) |
| Metadata: NAME | – | Malicious | High similarity to popular PyPI packages (typosquatting suspicion) |
| system | −0.2923 | Benign | – † |
| mkdir | −0.2892 | Benign | – † |
| r.json | −0.2022 | Benign | – † |
| GetAge | −0.1707 | Benign | – † |
| subprocess.check_call | −0.1524 | Benign | – † |
| setuptools.setup | −0.1215 | Benign | – † |

† Negative-weight features correspond to benign utilities: they argue against the malicious classification, and no malicious behavior is inferred for them.

This case provides a qualitative example in which the dominant positive contributions are associated with security-relevant execution behavior and package-level metadata anomalies.

#### 5.6.3. Practical Deployment Analysis

The deployment analysis provides empirical evidence of the practical feasibility and operational utility of the proposed framework. To this end, we deployed the model to monitor the PyPI "new-releases" stream. Unlike static datasets, the live ecosystem presents a highly imbalanced and noisy environment, which is suitable for examining the framework's scalability and sensitivity.

##### Analysis of Anomalous Benign Placeholders (False Positives)

During a 2-day continuous monitoring period, the model flagged several packages, including pyreto and teledex. Post-hoc manual auditing revealed these were benign placeholder or name-reservation packages. From a software supply chain security perspective, these false positives offer useful insights:

- **Structural Sparsity:** These packages often contain only a minimal setup.py file and empty `__init__.py` files. This results in API call graphs with extreme structural sparsity, leading to high centrality scores for a few standard hooks (e.g., setup).
- **Metadata Anomalies:** Both packages exhibited "red-flag" metadata patterns, such as placeholder descriptions ("This name has been reserved") and non-compliant author information (e.g., test@test.com).
- **Manual-Review Cue:** Such packages may warrant manual review because their sparse structure and unusual metadata can produce false-positive signals.

Retrospective Analysis of a Confirmed Malicious Package

We retrospectively examined several high-risk samples that were subsequently removed by PyPI administrators. A notable instance was the litellm (version 1.82.7) incident in late March 2026, which was later documented in the official PyPI incident report [30]. This package was a sophisticated supply-chain attack that bypassed standard signature-based scanners by utilizing obfuscated payloads. In our retrospective analysis, the model assigned anomalously high centrality scores to sensitive execution APIs, such as subprocess.Popen, which were involved in establishing persistence via sysmon.py. In a package intended for LLM gateway functionality, such system-level execution patterns represented a significant deviation from expected behavioral norms.

**Validation.** The litellm malicious versions were subsequently quarantined and returned a 404 Not Found status on the PyPI repository. This retrospective analysis illustrates that the framework assigns strong anomaly signals to a package that was later officially addressed; it is intended to demonstrate the behavior of the detection pipeline rather than to claim independent discovery or advance warning.

Discussion on Engineering Utility

These in-the-wild results highlight three practical aspects of the proposed framework that support potential integration into production environments:

- **Automation:** The framework processed thousands of new releases without expert intervention, demonstrating its automated feature extraction capability.
- **Low Performance Tax:** After the one-time offline construction of the LLM-refined sensitive-API feature vocabulary, the package-level detection stage (AST parsing, API interaction graph construction, centrality calculation, feature extraction, and classifier inference) takes approximately 1 s per package on average. All timings were obtained by individually measuring the end-to-end processing time of each package through the full package-level stage on a machine equipped with an Intel Core i5-14400F CPU (10 cores, 16 threads) and 32 GB of RAM, running 64-bit Windows 11. The average per-package processing time was approximately 1.0 s; the per-package latency distribution was not recorded, which we acknowledge as a limitation, although the reported average is consistent with the throughput observed during the two-day live monitoring experiment described above. The offline LLM filtering stage and classifier training are excluded from this per-package measurement. This per-package cost is well within the requirements for integration into CI/CD pipelines.
- **Interpretability:** By providing interpretability outputs (via LIME and LLM semantic explanations), the system allows security teams to distinguish between benign placeholders and likely malicious packages, supporting triage between automated detection and incident response.

## 6. Conclusions

This paper presents a scalable detection framework for malicious PyPI packages based on centrality-guided candidate identification and LLM-assisted semantic filtering. The core contribution is the demonstration that graph centrality metrics can effectively identify structurally important APIs, but that structural importance alone does not equal security sensitivity. By introducing a Large Language Model (Qwen3.5-Plus) as a semantic screening mechanism, the proposed framework successfully separates security-relevant APIs from high-centrality benign generic ones, yielding refined API features that significantly improve detection quality.

Experimental results on a large-scale dataset of over 20,000 samples confirm the effectiveness of this approach. The proposed method achieves 96.3% precision and 96.1%

F1-score, surpassing existing industrial tools and academic methods including MalGuard. Ablation experiments demonstrate that LLM semantic filtering contributes the largest performance gain (+4.2 percentage points in F1-score over raw centrality features), while metadata augmentation provides an additional +1.2 percentage points. The framework exhibits strong robustness in highly imbalanced (1:50) scenarios and, after the one-time offline construction of the LLM-refined feature vocabulary, requires approximately 1 s per package at the package-level detection stage. A sensitivity analysis of the Top-$K$ candidate API pool further shows that detection effectiveness improves as the candidate pool broadens and then stabilizes, supporting the moderate pool size adopted in this study. Real-world monitoring of the PyPI new-release stream demonstrates practical engineering feasibility, and LIME-based interpretability provides supporting information for security triage.

Although the current results are encouraging, limitations remain. The method relies primarily on static features and may miss malicious behaviors that employ advanced dynamic obfuscation or trigger only in specific runtime environments. In addition, the detection pipeline relies on a sensitive-API vocabulary that is fixed after the one-time offline construction; as the PyPI ecosystem evolves, newly emerging APIs may not be covered by this vocabulary, so the framework may generalize poorly to previously unseen APIs unless the vocabulary is periodically refreshed. Furthermore, for APIs whose security risk can only be judged from the surrounding calling context, the LLM's one-time, per-API judgment is prone to misjudgement, and such cases would benefit from context-aware or lightweight dynamic analysis. Finally, the LLM-generated behavior descriptions attached to LIME outputs are auxiliary review cues rather than analytic evidence and may occasionally contain inaccurate descriptions; they should always be interpreted jointly with the LIME weights and manually verified for consequential decisions. Moreover, the malicious packages in our dataset do not carry family labels, so family-level splits and matched-sampling evaluations were not performed; the release-date distributions of the two classes exhibit a temporal skew (Section 5.1), so cross-temporal generalization to threats beyond the collected time span remains to be validated; and the 1:50 imbalanced evaluation involves only 35 malicious packages, which limits its statistical power. Future work will focus on the following: (1) incorporating lightweight dynamic sandbox behavioral features to complement static analysis; (2) improving the efficiency and accuracy of LLM inference for processing long code contexts; (3) evaluating cross-ecosystem generalization (e.g., npm, RubyGems); (4) developing online updating mechanisms for streaming detection as new packages are published; (5) integrating richer security knowledge bases to enhance the LLM's semantic evaluation capability; and (6) conducting family-level, matched-sampling, and temporally aware evaluations to further validate robustness.

**Author Contributions:** Conceptualization, J.W. and D.M.; methodology, J.W.; software, J.W.; validation, Y.H., Z.Q. and B.W.; formal analysis, J.W.; investigation, J.W.; resources, D.M.; data curation, J.W.; writing—original draft preparation, J.W.; writing—review and editing, D.M.; supervision, D.M.; project administration, D.M.; funding acquisition, D.M. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was funded by The 30th Research Institute of China Electronics Technology Group Corporation, grant number MWA25Y303.

**Data Availability Statement:** The data presented in this study are available on request from the corresponding author. The data are not publicly available due to security concerns related to malicious software samples.

**Conflicts of Interest:** All authors were employed by the company The 30th Research Institute of China Electronics Technology Group Corporation. The authors declare that this study received

funding from The 30th Research Institute of China Electronics Technology Group Corporation. The funder was not involved in the study design, collection, analysis, interpretation of data, the writing of this article or the decision to submit it for publication.

## Appendix A. LLM Prompts for Sensitive API Analysis

To automate the semantic filtering process, we designed the following prompt to guide the LLM (Qwen3.5-Plus) in evaluating the security sensitivity of Top-*K* high-centrality APIs from a cybersecurity expert's perspective. The detailed system prompt is shown in Figure A1. In the template, the `<INSERT_CODE_HERE>` placeholder denotes the fully qualified name and function signature of the candidate API under evaluation. No package-specific code context is provided to the LLM: its role is to assess the generic security sensitivity of each API in isolation, so the resulting vocabulary is package-independent and is constructed once and reused in an unchanged format across all experiments.

**System Role Prompt of Sensitive API Analysis**

Task:
You are a security API auditor. Your task is to deter mine 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 CODEHERE>
JSON Response:
{
"api_name": "The name of the API",
"is_potentially_malicious": true,
"malicious_usage": "Malicious purposes."
}

**Figure A1.** System Role Prompt of Sensitive API Analysis.

## References

1. Zimmermann, M.; Staicu, C.A.; Benedikt, T.; Pradel, M. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security 22)*; USENIX Association: Berkeley, CA, USA, 2022 ; pp. 2159–2176.

2. Sonatype. 10th Annual State of the Software Supply Chain. Technical Report, Sonatype, 2024. Available online: https://www.sonatype.com/state-of-the-software-supply-chain/2024/10-year-look (accessed on 18 October 2026).
3. Bhat, M.; Gardner, D.; Horvath, M. *How Software Engineering Leaders Can Mitigate Software Supply Chain Security Risks*; Gartner, Inc.: Stamford, CT, USA, 2021; Gartner Research, ID G00752454.
4. Sonatype. Russia-Linked 'Lumma' Crypto Stealer Now Targets Python Devs. 2024. Available online: https://www.sonatype.com/blog/crytic-compilers-typosquats-known-crypto-library-drops-windows-trojan (accessed on 18 October 2026).
5. Taylor, M.; Vaidya, R.; Davidson, D.; Doupé, A.; Ahn, G.J. Defending against package typosquatting. In *Proceedings of the 14th International Conference on Network and System Security (NSS)*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 112–131.
6. Ladisa, P.; Plate, H.; Martinez, M.; Barais, O. A Taxonomy of Attacks on Open-Source Software Supply Chains. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*; IEEE: Piscataway, NJ, USA, 2023; pp. 1509–1521.
7. Fitzgerald, B.; Stol, K.J. Continuous software engineering: A roadmap and agenda. *J. Syst. Softw.* **2017**, *123*, 176–189. [CrossRef]
8. Tschacher, N.P. Typosquatting in Programming Language Package Managers. Master's Thesis, Universität Hamburg, Eimsbüttel, Germany, 2016.
9. Vu, D.L.; Pashchenko, I.; Massacci, F.; Plate, H.; Sabetta, A. Typosquatting and combosquatting attacks on the Python ecosystem. In P*roceedings of the 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*; IEEE: Piscataway, NJ, USA, 2020; pp. 509–514.
10. Vu, D.L.; Pashchenko, I.; Massacci, F.; Plate, H.; Sabetta, A. Towards using source code repositories to identify software supply chain attacks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*; ACM: New York, NY, USA, 2020; pp. 2093–2095.
11. Duan, R.; Alrawi, O.; Kasturi, R.P.; Elder, R.; Saltaformaggio, B.; Lee, W. 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: Reston, VA, USA, 2021.
12. Ohm, M.; Boes, F.; Bungartz, C.; Pfretzschner, M. On the feasibility of supervised machine learning for the detection of malicious software packages. In *Proceedings of the 17th International Conference on Availability, Reliability and Security (ARES)*; ACM: New York, NY, USA, 2022; pp. 1–10.
13. Zhang, J.; Huang, K.; Huang, Y.; Chen, B.; Wang, R.; Wang, C.; Peng, X. 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. (TOSEM)* **2025**, *34*, 104 . [CrossRef]
14. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2020*; Association for Computational Linguistics: Kerrville, TX, USA, 2020; pp. 1536–1547.
15. Sun, X.; Gao, X.; Cao, S.; Bo, L.; Wu, X.; Huang, K. 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)*; ACM: New York, NY, USA, 2024; pp. 1159–1170.
16. Yu, Z.; Wen, M.; Guo, X.; Jin, H. Maltracker: A fine-grained NPM malware tracker copiloted by llm-enhanced dataset. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*; ACM: New York, NY, USA, 2024; pp. 1759–1771.
17. Gao, X.; Sun, X.; Cao, S.; Huang, K.; Wu, D.; Liu, X.; Lin, X.; Xiang, Y. MalGuard: Towards Real-Time, Accurate and Actionable Detection of Malicious Packages in PyPI Ecosystem. In P*roceedings of the 34th USENIX Security Symposium (USENIX Security)*; USENIX Association: Berkeley, CA, USA, 2025; pp. 4741–4758.
18. Taylor, M.; Anwar, N.; Xiao, X. MalScan: Android Malware Detection Based on Social-Network Centrality Analysis. *IEEE Trans. Dependable Secur. Comput.* **2025**, *22*, 5464–5479. [CrossRef]
19. Zou, D.; Wu, Y.; Yang, S.; Chauhan, A.; Yang, W.; Zhong, J.; Dou, S.; Jin, H. IntDroid: Android malware detection based on API intimacy analysis. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2021**, *30*, 39. [CrossRef]
20. Freeman, L.C. Centrality in social networks: Conceptual clarification. In *Social Network: Critical Concepts in Sociology*; Scott, J., Ed.; Routledge: London, UK, 2002; Volume 1, pp. 238–263.
21. Katz, L. A new status index derived from sociometric analysis. *Psychometrika* **1953**, *18*, 39–43. [CrossRef]
22. Marchiori, M.; Latora, V. Harmony in the small-world. *Phys. A Stat. Mech. Its Appl.* **2000**, *285*, 539–546. [CrossRef]
23. Kumar, D.; Pawar, P.P.; Ananthan, B.; Rajasekaran, S.; Prabhakaran, T.V. Optimized Support Vector Machine Based Fused IoT Network Security Management. In *Proceedings of the 2024 3rd International Conference on Artificial Intelligence For Internet of Things (AIIoT)*; IEEE: Piscataway, NJ, USA, 2024; pp. 1–5. [CrossRef]
24. Chen, T.; Guestrin, C. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*; ACM: New York, NY, USA, 2016; pp. 785–794.

25. Ribeiro, M.T.; Singh, S.; Guestrin, C. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*; ACM: New York, NY, USA, 2016; pp. 1135–1144. [CrossRef]
26. VirusTotal. VirusTotal—Analyze Suspicious Files, Domains, IPs and URLs. 2024. Available online: https://www.virustotal.com/ (accessed on 18 October 2026).
27. Bertus, J. OSSGadget: Collection of Tools for Analyzing Open Source Packages. 2020. Available online: https://github.com/microsoft/OSSGadget (accessed on 18 October 2026).
28. Vu, D.L. A Fork of Bandit Tool with Patterns to Identifying Malicious Python Code. 2020. Available online: https://github.com/lyvd/bandit4mal (accessed on 18 October 2026).
29. Guo, W.; Xu, Z.; Liu, C.; Huang, C.; Fang, Y.; Liu, Y. An empirical study of malicious code in PyPI ecosystem. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*; IEEE: Piscataway, NJ, USA, 2023; pp. 166–177.
30. Fiedler, M.; Larson, S. Incident Report: LiteLLM/Telnyx Supply-Chain Attacks, with Guidance. The Python Package Index Blog. 2026. Available online: https://blog.pypi.org/posts/2026-04-02-incident-report-litellm-telnyx-supply-chain-attack/ (accessed on 18 October 2026).


**Disclaimer/Publisher's Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.