# Cutting the Gordian Knot: Detecting Malicious PyPI Packages via a Knowledge-Mining Framework

Wenbo Guo[1], Chengwei Liu[2]*, Ming Kang[3], Yiran Zhang[1], Jiahui Wu[1],
Zhengzi Xu[4], Vinay Sachidananda[1], Yang Liu[1]
[1]*Nanyang Technological University,* [2]*Nankai University,*
[3]*Sichuan University,* [4]*Imperial Global Singapore*

## Abstract

The Python Package Index (PyPI) has become a target for malicious actors, yet existing detection tools generate false positive rates of 15-30%, incorrectly flagging one-third of legitimate packages as malicious. This problem arises because current tools rely on simple syntactic rules rather than semantic understanding, failing to distinguish between identical API calls serving legitimate versus malicious purposes. To address this challenge, we propose PyGuard, a knowledge-driven framework that converts detection failures into useful behavioral knowledge by extracting patterns from existing tools' false positives and negatives. Our method utilizes hierarchical pattern mining to identify behavioral sequences that distinguish malicious from benign code, employs Large Language Models to create semantic abstractions beyond syntactic variations, and combines this knowledge into a detection system that integrates exact pattern matching with contextual reasoning. PyGuard achieves 99.50% accuracy with only 2 false positives versus 1,927-2,117 in existing tools, maintains 98.28% accuracy on obfuscated code, and identified 219 previously unknown malicious packages in real-world deployment. The behavioral patterns show cross-ecosystem applicability with 98.07% accuracy on NPM packages, demonstrating that semantic understanding enables knowledge transfer across programming languages.

## 1 Introduction

The Python Package Index (PyPI) serves as the central repository for Python packages, hosting over 400,000 packages with billions of downloads annually [1]. However, its open nature has made it an attractive target for malicious actors seeking to compromise software supply chains through typosquatting, dependency confusion, and code injection attacks [2, 3]. Recent studies document a substantial increase in malicious package uploads, with researchers identifying 116 malicious packages downloaded more than 10,000 times in 2023 alone [4], including sophisticated attacks by groups like Lazarus that deployed typosquatting packages downloaded 300-1,200 times each [5], demonstrating how compromised packages can cascade through the entire software supply chain, affecting millions of downstream applications.

To combat malicious packages, detection approaches span three primary categories: static analysis tools (like Bandit [6] and GuardDog [7]) that scan source code patterns with minimal overhead, dynamic analysis systems (like DySec [8]) that achieve 95.99% accuracy by monitoring runtime behaviors, and machine learning methods (like PypiGuard [9]) that combine metadata with API behaviors to reach 98.43% accuracy. However, these tools suffer from prohibitively high false positive rates, with current PyPI detection systems flagging approximately one-third of legitimate packages as malicious and generating false positive rates of 15-30% [10, 11], forcing analysts to manually inspect over 4,000 false alerts weekly [10] and leading to alert fatigue that drives organizations to turn off security tools or configure them permissively, creating exploitable security blind spots.

The fundamental limitation of existing detection tools lies in their reliance on coarse-grained, semantically unclear rules that fail to distinguish between benign and malicious behavioral boundaries, where rule-based static analysis methods employ simple pattern matching that captures broad suspicious behaviors without contextual nuances [7, 12]. For example, detecting network requests or file operations may flag legitimate packages performing similar actions for benign purposes [10], while existing approaches lack fine-grained knowledge about how identical operation sequences can serve entirely different purposes depending on execution context and intent [13]. This knowledge gap is particularly problematic as modern malware increasingly employs legitimate-looking API sequences and obfuscation techniques that mimic benign behaviors [4], making detection systems unable to achieve the semantic-level precision required for practical deployment without a comprehensive understanding of contextual distinctions between legitimate and illegitimate behavioral patterns.

Our key insight is that detection failures themselves contain

*Corresponding author

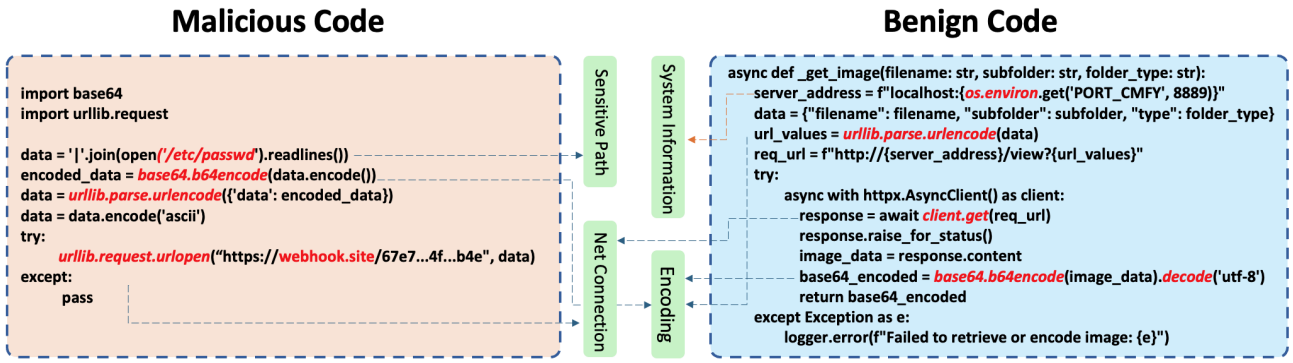


Figure 1: API-level similarity between malicious and benign packages leading to false positive.

valuable knowledge that can be systematically extracted and leveraged to enhance malicious package identification, transforming the traditional reactive approach of simply reducing false positives into a proactive knowledge-driven detection enhancement framework. We propose a three-stage methodology: first, a comprehensive study phase that analyzes existing tools' false positives and false negatives on a curated dataset to extract fine-grained execution sequences and contextual patterns; second, a knowledge extraction phase that employs Large Language Models (LLMs) to dynamically generate semantic descriptions of program operations and systematically mine sequential patterns using gradient support thresholds to distinguish malicious from benign behaviors; and third, a RAG-enhanced detection phase that integrates extracted knowledge into a retrieval-augmented generation framework to provide context-aware reasoning for improved detection accuracy. The unique advantage of LLMs in this approach lies in their ability to understand semantic relationships between code contexts and their behaviors, enabling automated generation of fine-grained taxonomies and contextual knowledge that traditional rule-based systems cannot capture, thereby bridging the gap between low-level operation patterns and high-level semantic understanding required for precise malicious package detection.

This work makes four key contributions to malicious PyPI package detection: **(1)** We systematically analyze detection failures across multiple state-of-the-art tools on 18,137 PyPI packages, extracting 304 discriminative behavioral patterns that distinguish malicious from benign code through hierarchical pattern mining; **(2)** We develop an LLM-driven methodology that transforms concrete program operation sequences into semantic behavioral abstractions, enabling cross-implementation pattern recognition that transcends syntactic limitations; **(3)** We design a RAG-enhanced detection framework that achieves 99.50% accuracy with only 2 false positives compared to 1,927-2,117 in existing tools, while maintaining 98.28% accuracy on obfuscated code; **(4)** We demonstrate practical impact through real-world deployment, identifying 219 previously undetected malicious packages confirmed by PyPI officials and achieving effective cross-ecosystem generalization with 98.07% accuracy on NPM packages.

## 2 Preliminaries

We first introduce the problem definition and a motivating example of this work to demonstrate the key challenges in current malicious package detection.

**Problem Definition.** Current malicious package detection relies on static analysis to identify known threat signatures and behavioral patterns, particularly API calls, using either rule-based or learning-based pattern matching techniques. Formally expressed as:

$$is\_malicious(p) = \bigvee_{i=1}^{k} match(r_i, p) \tag{1}$$

where $r \in R$ and $R$ is the set of matching rules, $p$ represents a given suspicious package, if any rule $r_i$ matches the corresponding content, i.e., API calls, resource manipulations, or even suspicious names, then $p$ is considered malicious.

However, such designs make current malicious package detection techniques struggle with high false positive rates, mainly because of two key limitations: (1) they fail to achieve fine-grained analysis, often relying on coarse patterns like package names, metadata, or high-level code features that overlook subtle malicious behaviors; and (2) the patterns they use frequently lack discriminative power, making it difficult to distinguish malicious packages from benign ones that share similar structures or dependencies.

**Motivating Example.** Consider two representative cases: the benign package `gen_wrappers` (version 0.7.1), which performs legitimate image processing operations, and the malicious package `gmgeoip` (version 0.0.2), which exfiltrates sensitive system information to remote servers. Figure 1 demonstrates that at the API level, both pack-

ages exhibit strikingly similar patterns: they both utilize `urllib.parse.urlencode` for URL parameter encoding and `base64.b64encode/decode` for data transformation. However, all leading state-of-the-art detection tools (i.e., Bandit4Mal, OSSGadget, and GuardDog) incorrectly flagged the benign package as malicious. After inspecting the detection logic of these tools, we found that they all classified `gen_wrappers` (version 0.7.1) as malicious mainly because they simply classify "shady-links" as malicious.

This API-level similarity coupled with contradictory intentions demonstrates that, although current detection rules capture features of malicious code, they still fail to be determinative in distinguishing malicious intentions from benign ones in different contexts. To this end, in this paper, we aim to propose a detection framework that reduces false positives by incorporating a novel strategy to mine determinative pattern rules for malicious code detection.

## 3 Behavior Pattern Mining

To this end, we propose a four-stage behavior pattern mining approach that systematically extracts behavioral patterns to distinguish between benign and malicious packages at the semantic level. Specifically, we ① first collect suspicious codes that are either real malicious code or benign ones that can easily be classified as malicious by existing tools, then, based on ② a well-constructed behavioral taxonomy, ③ abstract their behaviors by corresponding actions and intentions behind, and finally, ④ extract the representative subsequences that are discriminative and highly related to the classification of benign and malicious codes as behavioral patterns for further detection. Figure 2 illustrates the overview of this approach.

We define the following key terms used throughout this paper:

- **Action**: A semantic atomic behavioral operation representing a single meaningful step in code execution (e.g., `create_socket`, `send_http_post`).
- **Action Sequence**: An ordered list of actions representing the execution flow of a code snippet.
- **Pattern**: Common subsequences shared across multiple action sequences that reveal fundamental differences between malicious and benign behaviors.

### 3.1 Suspicious Code Collection

Considering that identifying determinative patterns should be highly sensitive to the difference between benign code and malicious code, we first construct a dataset of suspicious code that is easily misclassified by existing malicious code detection tools.

To this end, we employ four representative static analysis tools, GuardDog, Bandit4Mal, OSSGadget, and PyPI Malware Checks, as filters to identify potential malicious code. These tools are selected because they are open-source and static analysis-based tools, and they can produce detailed results containing the exact position of malicious code in user projects. Based on them, for the well-selected 18K packages (half benign and half malicious, as detailed in Section 4.1), we 1) employ these tools to scan each of them to identify code snippets that are considered malicious by these tools, and then 2) extract the related code context for each identified code snippet for further analysis.

**Suspicious Code Identification.** We employ these four tools to identify potential malicious code across the 18K packages. Each tool processes both benign and malicious packages, generating structured detection reports that label potential malicious code with their locations, line numbers, matched pattern types, and confidence scores (if any). Specifically, we parse each tool's scanning report for every package using regular expressions. We extract the complete file path information, line numbers, and code fragments identified by the tools to locate each suspicious code segment within the source files precisely. For duplicate detections where multiple tools flag the same file location, we record only one instance since the surrounding code context remains identical. Based on this, we can collect all true positives and false positives using these tools. After this, we also merged these results with all malicious code snippets from the ground truth (i.e., well-labeled malicious code snippets for true malicious packages) to avoid missing false negatives, so that we can obtain a comprehensive dataset of suspicious code snippets that includes all true malicious cases and false positive cases.

**Code Context Extraction.** Based on the collected suspicious codes, we extract the complete code context for each of them to understand their behavioral environment. For each identified code snippet, we read the entire Python source file and use the detection information (line numbers and code fragments) as clues to guide LLMs to extract their execution context, i.e., their data dependencies (variable definitions, assignments, and data flow leading to the flagged APIs) and control dependencies (conditional statements, loops, and function call chains that determine when and how the flagged code executes). This process ensures the extracted code segments are syntactically complete and preserve the logical structure necessary for understanding the operational environment surrounding each potential malicious code location. Note that the LLM in this phase serves purely as a context extraction tool, while the classification labels (malicious or benign) are determined by the ground truth dataset rather than LLM judgment. The extraction prompt is available on our website [14].

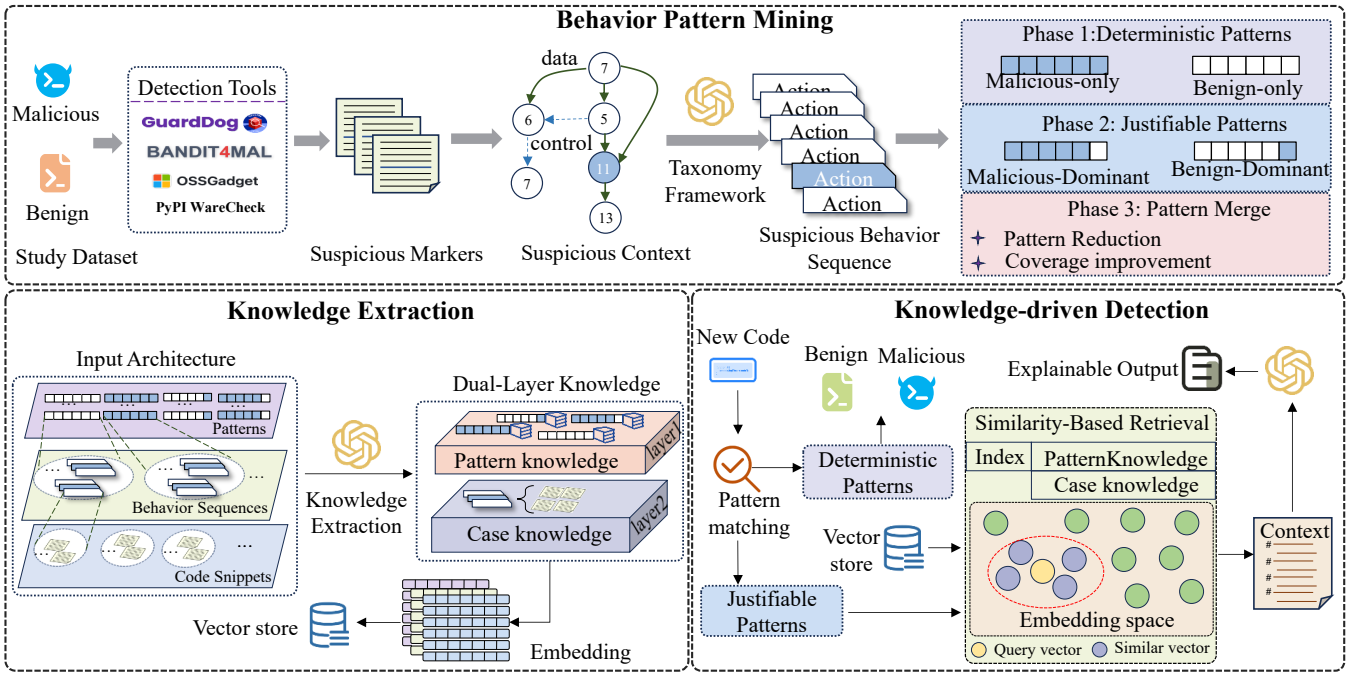


Figure 2: Overview of Hierarchical Behavior Pattern Mining and Detection Framework

Table 1: Taxonomy Categories and Subset of Behaviors

| Category | Representative Behaviors | Category | Representative Behaviors |
|---|---|---|---|
| File Operations | basic_file_reading<br>create_directory<br>copy_file | Basic Network Ops | create_http_connection<br>send_http_post<br>establish_tcp_connection |
| Network File Transfer | download_file_url<br>upload_file_transfer_sh<br>exfiltrate_folder | Command & Control | execute_shell_command<br>spawn_process_shell<br>decrypt_fernet_data |
| Third-party Platform Abuse | create_discord_bot<br>send_discord_message<br>check_roblox_cookie | Data Exfiltration | init_evil_class<br>init_grabber_class<br>init_sender_class |
| Code Execution | exec_python_code<br>import_dynamic<br>compile_code_object | Info Gathering | get_chrome_passwords<br>capture_screen_region<br>get_os_info |
| Encryption/Hashing | generate_aes_cipher<br>decrypt_aes_data<br>create_md5_hash | System Operations | create_child_process<br>create_thread<br>set_registry_value |
| Data Transformation | decode_base64_to_bytes<br>serialize_to_json<br>convert_int_to_char | Persistence/Stealth | check_persistence_entry<br>open_sqlite_db<br>disable_ssl_warnings |

1. This table presents a subset of behavioral categories.

## 3.2 Context-aware Behavioral Modeling

Next, we inspect the behavioral semantics of each identified code snippet to explore why it is considered malicious using existing tools, based on the extracted execution context.

Before summarizing the behavioral semantics, we first develop a behavioral taxonomy to describe the execution behaviors of code (i.e., intentions of code snippets) for better understanding and presentation. Specifically, for the identified context of each suspicious code snippet, we leverage the LLMs to incorporate the Card Sorting strategy to generate summary text (i.e., cards) to summarize the behaviors and potential intentions in their context for each identified action (i.e., operations, such as declarations, assignments, and function calls). In detail, there are three major steps:

**Behavioral Summarization.** We first summarize execution behavior for each identified action. Considering that obfuscation techniques, such as alias imports, dynamic imports, and nested function calls, are often employed in malicious packages to evade detection, we first incorporate LLMs to identify and resolve these obfuscation patterns while preserving actual execution order to de-obfuscate code snippets. Next, we ask LLMs to generate descriptions of corresponding behaviors within 20 words for each identified action, during which LLMs are required to neutrally summarize the behavior of the specific action in the context of a given code snippet. Based on this, we can tag each identified action with descriptive text under its corresponding context.

**Card Sorting.** Considering that LLMs could use different descriptive text to summarize similar actions and intentions, we employ the hybrid Card Sorting strategy during the summarization. Specifically, for the set of actions we identified, we randomize them into three different ordered lists, and we ask the LLMs to process them in sequence. For each action during the summarization of a sequence, apart from requiring the LLMs to do the summarization (generate a card), we also provide the LLMs with a collected set of cards (generated summary texts of previous actions), and ask the LLMs to choose these existing ones if suitable, before generating a new one. Based on this, we are able to obtain three generated card sets after the automated summarization.

**Behavioral Taxonomy Construction.** Since the LLMs could

introduce uncertainty and hallucination during the summarization, we also conduct human-guided consolidation using three domain experts (two doctoral candidates and one postdoc, with at least three years of experience in malicious code detection) to independently review all generated cards, exclude nonsensical text, and merge semantically similar descriptions into unified categories. When at least two experts approve a decision, we proceed with the corresponding action to refine the generated cards. Based on this, we can generate a set of summary texts for the behaviors, actions, and their intentions in identified suspicious code snippets.

Our results showed that, from the 8,543 malicious packages, we extracted 2,283 unique malicious code snippets after deduplication. This reduction occurs because malicious packages are often released in batches with identical malicious code embedded. From these snippets, we identified 13,834 actions in total, which were reduced to 837 distinct ones after deduplication. The LLM-driven semantic analysis generated 495 initial descriptions, which were consolidated through expert review into a refined taxonomy of 327 unique API behavioral categories. A subset of categories is shown in Table 1.

## 3.3 Context-to-Action Sequence Mapping

With the code context extracted from Section 3.1 and the behavioral taxonomy constructed in Section 3.2, we then abstract the specific code implementations into behavioral sequences that capture the essential execution logic. This abstraction process converts specific operations in code into generalized behavioral sequences, where each item in the sequence represents a discrete step toward achieving a broader behavioral objective. For instance, code implementations may use different APIs like `urllib.request` or `requests.get`, both of which map to the abstract action "network_communication" within data leak patterns.

## 3.4 Hierarchical Pattern Discovery

After the abstraction of behavioral sequences for each code snippet, we then identified the major behavioral patterns that can reliably distinguish malicious code behaviors from benign ones. To this end, we designed a systematic process to hierarchically discover two specific types of patterns, ① deterministic patterns that provide definitive classification signals, and ② justifiable patterns that require contextual reasoning for accurate assessment.

Our pattern discovery approach employs a hierarchical mining strategy using the PrefixSpan algorithm to systematically identify behavioral subsequences (i.e., identified patterns) that are recognizable with the classification of benign and malicious packages. The core methodology involves iterative identification of such subsequence patterns, where each iteration focuses on discovering subsequences that were not captured in previous rounds with progressively decreasing support thresholds, which ensures that both common and rare but significant behavioral patterns can be systematically identified. The detailed process is presented in Algorithm 1.

**Phase 1: Deterministic Pattern Mining.** We first identify patterns with perfect classification reliability by extracting patterns that appear exclusively in either benign or malicious action sequences. Let $R_B$ and $R_M$ denote the sets of action sequences extracted from benign and malicious code snippets, respectively. For a given predefined support value $s \in S$, we apply the PrefixSpan algorithm on the combined dataset $R_B \cup R_M$ to discover frequent subsequences. For each discovered pattern $p$, we check whether it covers only benign action sequences or only malicious action sequences. Patterns that exclusively cover benign sequences are collected into $det_B = \{p \mid p \in patterns \wedge \text{CoverOnly}(p, R_B)\}$, while patterns that exclusively cover malicious sequences form $det_M = \{p \mid p \in patterns \wedge \text{CoverOnly}(p, R_M)\}$. We combine these exclusive patterns into our deterministic pattern set using $P_{det} \leftarrow P_{det} \cup det_B \cup det_M$. After each round of mining, we remove all sequences covered by the newly discovered deterministic patterns from our working datasets to ensure only uncovered action sequences are mined in subsequent rounds: $R_B \leftarrow R_B \setminus \text{CoveredBy}(det_B \cup det_M)$ and $R_M \leftarrow R_M \setminus \text{CoveredBy}(det_B \cup det_M)$. This process repeats across all predefined support values, ensuring that each deterministic pattern provides 100% classification confidence.

**Phase 2: Justifiable Pattern Mining.** Next, we target the remaining uncovered sequences in $R_B$ and $R_M$ that could not be perfectly classified by these identified deterministic patterns, where we aim to further identify patterns with strong but imperfect discriminative power. In this phase, we still apply the PrefixSpan algorithm on the residual datasets $R_B \cup R_M$ at each support level $s \in S$ to discover patterns from the remaining behavioral sequences that escaped deterministic classification. For each discovered pattern $p$, we calculate its classification bias using $MaxCoverageRatio(p, R_B, R_M)$, which computes the percentage of action sequences covered by pattern $p$ that belong to the dominant class. For example, if pattern $p$ covers 10 action sequences where 9 are malicious and 1 is benign, the ratio is 0.9, indicating 90% bias toward malicious behavior. We select patterns with strong bias by checking if the ratio meets our threshold: when $ratio \geq \tau$ (set to 0.9 empirically), we add the pattern to our justification set using $P_{just} \leftarrow P_{just} \cup \{p\}$. This process identifies patterns that, while not providing perfect separation, still offer valuable probabilistic classification signals for action sequences that resist deterministic classification. This also follows the iterative mining process by the predefined order of support value $s \in S$.

**Phase 3: Pattern Merge.** After the identification, there is a pattern explosion problem that PrefixSpan could generate numerous overlapping sub-patterns from each action sequence, making the pattern set impractical for deployment due to excessive redundancy. Therefore, we employ a greedy algorithm to select the minimal pattern subset from $P_{det} \cup P_{just}$

**Algorithm 1:** Hierarchical Sequence Pattern Mining

**input :** Benign samples $B$, Malware samples $M$, Support levels $S = \{s_1, s_k\}$, Distinction threshold $\tau$
**output :** Merged pattern set $P_{opt}$

$B_{seq} \leftarrow$ `ExtractSequences`$(B)$;
$M_{seq} \leftarrow$ `ExtractSequences`$(M)$;
$R_B \leftarrow B_{seq}$; $R_M \leftarrow M_{seq}$; $P_{det} \leftarrow \emptyset$;
`// Phase 1: Deterministic pattern mining`
**foreach** $s \in S$ **do**
  $patterns \leftarrow$ `PrefixSpan`$(R_B \cup R_M, s)$;
  $det_B \leftarrow \{p \mid p \in patterns \wedge$ `CoverOnly`$(p, R_B)\}$;
  $det_M \leftarrow \{p \mid p \in patterns \wedge$ `CoverOnly`$(p, R_M)\}$;
  $P_{det} \leftarrow P_{det} \cup det_B \cup det_M$;
  $R_B \leftarrow R_B \setminus$ `CoveredBy`$(det_B \cup det_M)$;
  $R_M \leftarrow R_M \setminus$ `CoveredBy`$(det_B \cup det_M)$;
**end**
`// Phase 2: Justification pattern mining`
$P_{just} \leftarrow \emptyset$;
**foreach** $s \in S$ **do**
  $patterns \leftarrow$ `PrefixSpan`$(R_B \cup R_M, s)$;
  **foreach** $p \in patterns$ **do**
    $ratio \leftarrow$ `MaxCoverageRatio`$(p, R_B, R_M)$;
    **if** $ratio \geq \tau$ **then**
      $P_{just} \leftarrow P_{just} \cup \{p\}$;
    **end**
  **end**
**end**
`// Phase 3: Pattern Merge`
$P_{opt} \leftarrow \emptyset$; $covered \leftarrow \emptyset$;
$all\_indices \leftarrow$ `AllIndices`$(B_{seq}, M_{seq})$;
**while** $covered \neq all\_indices$ **do**
  $best\_p \leftarrow \arg\max_{p \in (P_{det} \cup P_{just})} |$`CoveredBy`$(p) \setminus covered|$;
  **if** $|$`CoveredBy`$(best\_p) \setminus covered| > 0$ **then**
    $P_{opt} \leftarrow P_{opt} \cup \{best\_p\}$;
    $covered \leftarrow covered \cup$ `CoveredBy`$(best\_p)$;
  **end**
  **else**
    **break**;
  **end**
**end**
**return** $P_{opt}$;

that achieves complete action sequence coverage. Starting with an empty result set $P_{opt}$ and an empty *covered* set to track which action sequences have been covered, the algorithm performs iterative selection. In each iteration, we evaluate all remaining patterns in $P_{det} \cup P_{just}$ to find the one that covers the maximum number of previously uncovered action sequences, calculated as $|\text{CoveredBy}(p) \setminus covered|$ for each pattern $p$. We select the pattern with the highest coverage using $best\_p \leftarrow \arg\max_{p \in (P_{det} \cup P_{just})} |\text{CoveredBy}(p) \setminus covered|$, add this selected pattern to our final set $P_{opt}$, and update our tracking set by adding all action sequences covered by this pattern: $covered \leftarrow covered \cup \text{CoveredBy}(best\_p)$. This selection process repeats until either all action sequences are covered or no remaining pattern can increase coverage, ensuring we obtain the minimum number of patterns required for complete behavioral coverage.

After these three steps of actively mining deterministic and justifiable patterns, we identified the corresponding patterns that can be easily used to classify malicious and benign code snippets. These patterns serve as the knowledge base for downstream malicious code detection.

Table 2: Detection statistics for static analysis tools

| Tool | TP | FP | FN | Detection Positions |
|---|---|---|---|---|
| Bandit4Mal | 3,899 | 7,699 | 4,633 | 695,974 |
| GuardDog | 6,640 | 447 | 757 | 11,420 |
| OSSGadget | 7,529 | 8,113 | 1,001 | 4,955,383 |
| PyPI Warehouse | 8,332 | 8,422 | 208 | 248,737 |

# 4 Study

We first conducted an empirical investigation to understand the distribution, coverage, and potential applicability of these extracted behavioral patterns on malicious code detection by answering the following research question:

**RQ1. Distribution Analysis:** What behavioral patterns emerge from hierarchical mining and how suitable are they for detection rules?

## 4.1 Dataset Construction

We constructed a large-scale dataset of PyPI packages to support systematic analysis of malicious behavior patterns. The dataset consists of 9,552 unique packages across 10,906 versions, sourced from the curated collection by Guo et al. [15], which provides confirmed ground-truth labels for supply chain attacks. For benign packages, we selected 11,988 packages based on PyPI download statistics from the preceding month. We prioritized highly downloaded packages because prior studies have shown that popular packages trigger significantly higher false positive rates than randomly sampled packages [11, 16], providing a more rigorous test for detection precision. Additionally, popular packages are less likely to contain malicious code, as any malicious behavior would be quickly discovered given their large user base. These packages also represent real-world scenarios with higher code complexity and functional diversity. We verified with mainstream vulnerability databases (OSV [17] and Snyk) that no known malicious packages were present in our benign dataset. For each package, we selected one random version to avoid redundancy, as different versions of the same package exhibit minimal code differences. We added a 5-second delay between downloads to minimize server load. Our one-time collection of approximately 12,000 packages represents less than 0.0006% of PyPI's 2.1 billion [1] daily downloads. We did not collect any personally identifiable information during this process. We partitioned the complete dataset using an 80-20 split, allocating 18,137 packages for pattern analysis and reserving 4,757 for unbiased evaluation.

## 4.2 Experimental Setup

We applied four static analysis tools to all 18,137 packages in the study partition to identify potential locations of malicious code. These tools exhibit low false negative rates, with only 64 out of 8,543 malicious packages being missed by all

four tools simultaneously. Our taxonomy captures behavioral intentions rather than specific code implementations. Different implementations often share the same malicious intent, so these missed packages likely do not introduce new behavioral patterns. As shown in Table 2, the tools exhibit varying detection characteristics: GuardDog identified 11,420 positions, while OSSGadget flagged 4,955,383 positions, demonstrating different sensitivity thresholds. After deduplication of overlapping detections at identical file locations, we obtained 956,131 unique detection positions for subsequent context extraction.

For each Python file containing detection positions, we consolidate all flagged locations within the file and employ GPT-4.1 to extract comprehensive code snippets in a single analysis pass. Through this process, we successfully collected 5,813 false positive contexts (benign code incorrectly flagged as malicious) and 7,458 malicious code snippets from actual malicious and benign packages.

From these 13,271 code snippets, we identified 13,834 individual API calls. Using card sorting with GPT-4.1, 495 initial behavioral descriptions were generated at a cost of 126 USD. Three domain experts then reviewed these descriptions and merged semantically similar ones into 327 unique behavioral categories, as shown in Table 1. This taxonomy construction is a one-time effort.

We mapped the extracted code snippets to abstract action sequences using the constructed API taxonomy, generating 2,431 action sequences with an average length of 4.2 actions per sequence. The action sequences comprise 1,350 benign sequences and 1,081 malicious sequences. For the three-phase hierarchical mining process, we configure support thresholds as $S = \{30, 25, 20, 15, 10, 7, 5, 3, 2\}$ and distinction threshold $\tau = 0.9$. Phase 1 deterministic pattern mining discovered 115,960 pure patterns achieving 72.44% overall coverage. Phase 2 justification pattern mining processes remaining uncovered sequences and identified 47 patterns with $\geq 90\%$ class bias, contributing additional 23.41% coverage. The combined phases produced 116,007 patterns, covering 95.85% of action sequences. Phase 3 pattern merge applied greedy set cover optimization, reducing the pattern set from 116,007 to 304 patterns (a 99.74% reduction) while maintaining 92.60% sequence coverage. The final optimized pattern set contains 278 deterministic patterns and 26 justifiable patterns.

## 4.3 Pattern Extraction Evaluation

To validate the quality of action sequence extraction, we randomly selected 500 code snippets covering both malicious and benign packages. Three independent security experts were provided with the original code snippets and the taxonomy of actions, and we asked the experts to manually label (1) the malicious behavior related actions implemented in the code snippets, and (2) the action sequence that constitutes the complete malicious behavior process. For both of them, we adopted majority voting after they discuss when inconsistencies occur.

As for the results, at the action level, we retrieved 3,608 individual actions from the 500 code snippets. Among these, 3,432 actions (95.12%) were correctly mapped to their semantic labels. The errors consisted primarily of hallucinated actions (174, 4.82%) where the model generated actions not present in the code, and wrong-type mappings (2, 0.06%) where similar operations were mislabeled. Notably, no missing actions were reported, indicating the extraction tends toward over-generation rather than under-generation.

At the sequence level, 461 out of 500 snippets (92.2%) were correctly extracted with all actions accurately identified. The remaining 39 snippets contained errors, primarily including hallucinated actions, over-abstraction, and incorrect semantic mapping. The evaluation achieved expert consensus on 97.2% of samples with a Fleiss' Kappa [18] coefficient of 0.8778, indicating almost perfect inter-rater agreement, which validates the reliability of our evaluation protocol.

## 4.4 RQ1. Distribution Analysis

To understand the practical value of our extracted patterns, we first investigated the distribution, coverage, and complexity of these identified patterns to evaluate their applicability in real-world malicious code detection.

**Pattern Distribution.** The final pattern set contains 278 deterministic patterns (91.4%) and 26 justifiable patterns (8.6%), indicating that the majority of extracted behavioral signatures provide definitive classification signals. Among the deterministic patterns, 192 patterns (69.1%) exclusively identify benign behaviors, while 86 patterns (30.9%) exclusively cover malicious behaviors. This 2.2:1 ratio reflects the diversity of legitimate functionalities compared to concentrated malicious attack vectors. The 26 justifiable patterns exhibit varying degrees of class bias, with 12 patterns showing strong benign dominance ($> 80\%$ benign coverage), 11 patterns demonstrating mixed characteristics (50-80% bias), and 3 patterns exhibiting malicious dominance ($> 80\%$ malicious coverage).

**Coverage and Discriminative Power.** Deterministic patterns demonstrate superior coverage capabilities, with malicious-only patterns achieving 87.7% coverage of malicious action sequences (948/1,081) while benign-only patterns cover 60.2% of benign sequences (813/1,350). This asymmetric coverage indicates that malicious behaviors exhibit more concentrated and predictable patterns compared to the diverse legitimate functionalities in benign packages. Justifiable patterns provide crucial coverage for boundary cases, covering an additional 200 action sequences that could not be definitively classified, representing 8.2% of the total sequence space.

**Pattern Complexity and Behavioral Characteristics.** Analysis of pattern lengths reveals that deterministic patterns tend to be more specific, with malicious-only patterns averaging 2.8 actions per pattern, while benign-only patterns

average 3.2 actions. The most frequent malicious patterns include complex attack sequences such as [`create_socket`, `establish_tcp_connection`, `dup_socket_stdin`, `dup_socket_stdout`, `dup_socket_stderr`] covering 29 malicious sequences for reverse shell establishment, and [`get_env_var`, `get_clipboard_text`, `copy_to_clipboard`] appearing in 43 sequences for information harvesting. In contrast, benign patterns demonstrate functional diversity with legitimate sequences like [`get_env_var`, `spawn_process_no_shell`, `read_process_stdout`] for system administration tasks and [`path_string_operations`, `execute_shell_command`, `exit_program`] for package installation processes. Justifiable patterns exhibit intermediate complexity, typically involving 2-4 actions that require contextual analysis to determine intent, such as patterns combining file operations with network communications that could serve either legitimate or malicious purposes.

Response to RQ1: Our hierarchical mining approach successfully extracts 304 discriminative behavioral patterns, with 278 deterministic patterns providing definitive classification signals and 26 justifiable patterns handling boundary cases. The patterns demonstrate clear behavioral distinctions, achieving 92.60% overall sequence coverage.

## 5 PyGuard

We transform the behavioral patterns from our empirical analysis into an operational detection system. Based on the identified behavioral patterns that can be informative to classify benign and malicious code, we further implement a classification system (PyGuard) to demonstrate their efficacy. The system bridges static pattern discoveries (i.e., the 278 deterministic patterns enabling direct classification and the 26 justification patterns that provide strong indications) with contextual reasoning, by incorporating RAG-enhanced analysis for greater generalizability. In this section, we introduce the design of PyGuard, including ① the construction of knowledge base for RAG, and ② the working pipeline of PyGuard to detect malicious packages. Figure 2 illustrates the overview of PyGuard.

### 5.1 Knowledge Base Construction

The knowledge base transforms behavioral patterns from the mining phase into structured detection knowledge through systematic extraction and indexing. Let $\mathcal{P} = P_{det} \cup P_{just}$ denote patterns from Section 3, where $P_{det}$ contains 278 deterministic patterns and $P_{just}$ contains 26 justification patterns.

**Dual-Layer Knowledge Storage.** The pattern knowledge layer stores each pattern $p \in \mathcal{P}$ with its semantic interpretation and classification properties. For deterministic patterns where $p \in P_{det}$, we annotate consistent behavioral characteristics: attack vectors for malicious-only patterns and legitimate use cases for benign-only patterns. For justification patterns where $p \in P_{just}$, we extract distinction rules that capture contextual differences determining classification. The case knowledge layer preserves the mapping between patterns and their original implementations. Each pattern $p$ links to its covered action sequences $S_p = \{s \mid s \text{ is covered by } p\}$ and corresponding code snippets. We pre-compute embedding vectors for all sequences and contexts using text-embedding-3-large [19], producing $e(s) \in \mathbb{R}^{3072}$ for each sequence, enabling efficient similarity search during detection.

**Knowledge Extraction Process.** For each pattern $p \in \mathcal{P}$, we employ LLM to analyze its constituent sequences and extract detection-relevant insights. For deterministic patterns where $p \in P_{det}$, the LLM synthesizes common characteristics across all implementations in $S_p$. For justification patterns where $p \in P_{just}$, we partition the covered sequences into $S_p^B$ (benign instances) and $S_p^M$ (malicious instances). The LLM conducts comparative analysis between $S_p^B$ and $S_p^M$, identifying contextual factors that differentiate them, such as data flow destinations (user files versus system credentials), network endpoints (local versus external), and execution triggers (user-initiated versus automated). These insights produce distinction rules that guide classification when the same behavioral pattern appears in different contexts.

**Knowledge Indexing.** The extracted knowledge undergoes multi-modal indexing to support different retrieval strategies. We implement hash-based indexing for exact pattern matching, subsequence indexing for partial pattern detection, and FAISS vector indexing for similarity search across all case embeddings. This ensures efficient retrieval whether input sequences match patterns exactly, partially, or semantically.

### 5.2 Detection Pipeline

The detection pipeline applies the constructed knowledge base to classify unknown packages through behavioral analysis and contextual reasoning. Given an unknown package, we extract behavioral sequences using the same methodology from Section 3, then leverage the knowledge base to determine classification.

**1) Behavioral Extraction and Pattern Matching.** For each Python file in an unknown package, we identify sensitive APIs from our category taxonomy and extract their execution contexts using the same LLM-based approach from training. The APIs are mapped to behavioral categories, generating action sequence $s_{new}$. We first attempt exact and subsequence matching against patterns in $\mathcal{P}$. If $s_{new}$ matches a deterministic pattern $p \in P_{det}$, classification is immediate: malicious-only patterns flag the package as malicious while benign-only patterns indicate legitimate functionality. If $s_{new}$ matches both benign and malicious deterministic patterns, the file is classified as malicious since any malicious pattern indicates a

potential threat.

**2) Similarity-Based Retrieval.** When $s_{new}$ does not match any deterministic pattern, or when it yields only justification patterns $p \in P_{just}$, we employ similarity-based retrieval to find relevant cases. We compute embedding $e(s_{new})$ for the input sequence and separately calculate behavioral similarity $\text{sim}_s(e(s_{new}), e(s_i))$ and code context similarity $\text{sim}_c(c_{new}, c_i)$ with all stored sequences in $S_p$. We retrieve the top-k (k=5) most similar cases based on behavioral similarity and the top-k most similar cases based on context similarity from both $S_p^B$ and $S_p^M$, combining these retrieved sets to provide comprehensive similarity coverage for subsequent RAG-enhanced analysis.

**3) Knowledge-driven Detection.** For justification patterns, classification requires contextual reasoning. The LLM receives: (1) the target code $c_{new}$ and its sequence $s_{new}$, (2) the matched pattern $p$ with its distinction rules, (3) top-k similar benign cases from $S_p^B$ with similarity scores, and (4) top-k similar malicious cases from $S_p^M$ with similarity scores. The LLM analyzes whether $c_{new}$ satisfies the distinction rules, comparing against retrieved examples to determine behavioral intent. For instance, if the distinction rule indicates that "base64 encoding of system files suggests malicious intent while encoding user documents is benign," the model examines the data sources in $c_{new}$. The output provides classification $f(s_{new}) \in \{\text{benign}, \text{malicious}\}$, confidence score based on evidence strength, and explanatory reasoning tracing specific rules and examples that influenced the decision.

# 6 Evaluation

To comprehensively validate our approach, we conducted systematic experiments addressing four interconnected research questions. Our evaluation covers detection accuracy comparison with state-of-the-art tools, ablation analysis of knowledge components, real-world deployment effectiveness, and cross-ecosystem generalizability.

**RQ2. Accuracy**: How does our knowledge-driven framework compare to SOTA tools?

**RQ3. Ablation**: What is the impact of knowledge integration on framework performance?

**RQ4. Usability**: How effective is the PyGuard against emerging threats in real-world scenarios?

**RQ5. Cross-ecosystem**: How effective is the extracted knowledge when applied to other package ecosystems?

Table 3: Evaluation Dataset Characteristics

| Dataset | Benign | Malicious | Total | Ecosystem |
|---|---|---|---|---|
| Original Test Packages | 2,394 | 2,363 | 4,757 | PyPI |
| Latest Packages | 1,001 | 1,097 | 2,098 | PyPI |
| Obfuscated Packages | 1,019 | 1,050 | 2,069 | PyPI |
| NPM Packages | 953 | 809 | 1,762 | NPM |
| **Total** | **4,414** | **4,510** | **8,924** | / |

## 6.1 Dataset

We constructed four evaluation datasets to assess our knowledge-driven detection framework against different operational challenges, as detailed in Table 3. The Original Test Dataset comprises our reserved 20% holdout set with 2,394 benign and 2,363 malicious packages. The Latest Packages Dataset contains 1,001 benign packages randomly sampled from recent PyPI uploads and 1,097 malicious packages from the Open Source Vulnerability Database (OSV) [17], collected through July 2024, which is after the knowledge cutoff date of June 1, 2024 for both GPT-4.1 and GPT-4.1-mini used in our evaluation, ensuring no data leakage in our LLM-based experiments. The Obfuscated Dataset was created using Intensio Obfuscator [33] to transform 1,019 benign and 1,050 malicious packages from our test set, evaluating detection resilience when syntactic signatures are deliberately obscured through variable renaming, control flow flattening, and string encoding. The NPM Dataset includes 809 malicious JavaScript packages from Ohm et al. [2] and 953 benign packages randomly sampled from highly downloaded NPM packages, testing whether behavioral knowledge extracted from Python packages transfers effectively to different programming ecosystems.

## 6.2 Baseline Selection

As shown in Table 4, various tools have been developed for detecting malicious PyPI packages using rule-based static analysis, dynamic analysis, and ML-based approaches. While numerous tools exist, many lack public availability or source code access, preventing reproducible evaluation. Therefore, we selected baseline tools for comparison based on two criteria: open-source implementation and active maintenance.

Based on these criteria, we selected eight baseline tools for evaluation. Bandit4Mal [20], OSSGadget [21], and GuardDog [7] employ rule-based static analysis with predefined patterns for suspicious operations. Bandit4Mal flags security-sensitive APIs, OSSGadget uses regular expressions for pattern matching, and GuardDog implements heuristics for detection. Hercule [31] performs static analysis using CodeQL to detect malicious behaviors. PyPI WareCheck [23] represents PyPI's official malware checks. Machine learning approaches include SAP, which applies Random Forest (SAP-RF), Decision Tree (SAP-DT), and XGBoost (SAP-XGB) classifiers to features extracted from package metadata and code structure. Cerebro [13] uses behavior abstraction and BERT models for cross-ecosystem detection on both NPM and PyPI packages. SocketAI [22] is a purely LLM-based detection method that employs two models in cooperation. In our configuration, we use GPT-4.1 and GPT-4.1-nano for detection. Unlike other tools that output package-level results, SocketAI examines individual Python files and marks the entire package as malicious if any single file is flagged as malicious. These tools

Table 4: Comparison of Baseline Tools for Malicious Package Detection

| Baseline | Tech. | Output | | | | Avail. | Date | Baseline | Tech. | Output | | | | Avail. | Date |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | MB | BP | CL | KC | | | | | MB | BP | CL | KC | | |
| Bandit4Mal [20] | Rule | ✓ | ✓ | ✓ | × | ✓ | 2022 | OSSGadget [21] | Rule | ✓ | ✓ | ✓ | × | ✓ | 2025 |
| SocketAI [22] | LLM | ✓ | ✓ | ✓ | × | ✓ | 2025 | PyPI WareCheck [23] | Rule | ✓ | ✓ | ✓ | × | ✓ | 2024 |
| MalOSS [24] | Dynamic | ✓ | × | ✓ | × | × | 2021 | Guarddog [25] | Rule | ✓ | ✓ | ✓ | × | ✓ | 2025 |
| Cerebro [13] | ML | ✓ | ✓ | × | × | ✓ | 2025 | SAP [26] | ML | × | × | × | × | ✓ | 2024 |
| ClamAV [27] | ML | × | × | × | × | × | 2024 | MalWuKong [28] | Rule | ✓ | ✓ | ✓ | × | × | 2023 |
| OSCAR [29] | Dynamic | ✓ | ✓ | ✓ | × | × | 2024 | MPHunter [30] | ML | × | × | × | × | × | 2023 |
| Hercule [31] | Dynamic | ✓ | ✓ | ✓ | × | ✓ | 2025 | MalGuard [32] | ML | ✓ | × | × | × | ✓ | 2025 |

***Tech.*** refers to the technology used by the baseline. In **Output**: ***MB*** (Malicious Behavior), ***BP*** (Behavior Pattern), ***CL*** (Code Location), ***KC*** (Key Context). ***Avail.*** indicates the availability of the baseline, where ✓ denotes availability and × indicates unavailability. ***Date*** is the most recent update time.

Table 5: Performance of PyGuard and Baseline Tools

| Dataset | Tool | Accuracy | Precision | Recall | F1-Score | FP | FN |
|---|---|---|---|---|---|---|---|
| Evaluation Dataset | Bandit4Mal | 33.37% | 34.72% | 48.69% | 40.54% | 1,927 | 1,080 |
| | GuardDog | 94.12% | 92.38% | 94.42% | 93.39% | 141 | 101 |
| | OSSGadget | 50.01% | 48.13% | 88.38% | 62.32% | 2,000 | 244 |
| | PyPI WareCheck | 52.99% | 49.66% | 99.19% | 66.18% | 2,117 | 17 |
| | SAP-DT | 59.99% | 68.68% | 25.42% | 37.10% | 244 | 1,570 |
| | SAP-RF | 86.74% | 92.20% | 78.05% | 84.54% | 139 | 462 |
| | SAP-XGB | 65.55% | 80.00% | 34.39% | 48.11% | 181 | 1,381 |
| | Cerebro | 89.12% | 90.21% | 85.88% | 88.00% | 196 | 297 |
| | Hercule | 88.76% | 90.67% | 84.47% | 87.46% | 183 | 327 |
| | SocketAI | 90.95% | 92.01% | 88.12% | 90.03% | 161 | 250 |
| | **PyGuard** | **99.50%** | **99.90%** | **99.08%** | **99.49%** | **2** | **19** |
| Latest Dataset | Bandit4Mal | 58.45% | 49.60% | 94.40% | 65.03% | 822 | 48 |
| | GuardDog | 83.84% | 89.06% | 68.12% | 77.20% | 69 | 263 |
| | OSSGadget | 52.63% | 45.79% | 86.33% | 59.84% | 875 | 117 |
| | PyPI WareCheck | 59.87% | 50.54% | 88.00% | 64.20% | 739 | 103 |
| | SAP-DT | 63.01% | 58.33% | 33.45% | 42.52% | 205 | 571 |
| | SAP-RF | 62.44% | 60.17% | 24.13% | 34.44% | 137 | 651 |
| | SAP-XGB | 71.97% | 73.36% | 49.42% | 59.05% | 154 | 434 |
| | Cerebro | 77.98% | 83.22% | 57.81% | 68.23% | 100 | 362 |
| | Hercule | 83.03% | 71.08% | 81.35% | 75.87% | 284 | 160 |
| | SocketAI | 84.80% | 81.89% | 80.65% | 81.27% | 153 | 166 |
| | **PyGuard** | **97.37%** | **99.37%** | **94.14%** | **96.68%** | **5** | **49** |
| Obfuscation Dataset | Bandit4Mal | 54.84% | 54.31% | 64.75% | 59.07% | 567 | 367 |
| | GuardDog | 89.60% | 96.96% | 81.49% | 88.56% | 25 | 181 |
| | OSSGadget | 50.95% | 50.89% | 85.37% | 63.77% | 856 | 152 |
| | PyPI WareCheck | 63.51% | 58.25% | 97.02% | 72.79% | 724 | 31 |
| | SAP-DT | 52.88% | 56.16% | 28.91% | 38.17% | 235 | 740 |
| | SAP-RF | 65.83% | 76.01% | 46.88% | 57.99% | 154 | 553 |
| | SAP-XGB | 60.51% | 70.44% | 37.08% | 48.58% | 162 | 655 |
| | Cerebro | 78.00% | 78.93% | 76.73% | 77.82% | 213 | 242 |
| | Hercule | 84.34% | 85.64% | 83.41% | 84.51% | 145 | 172 |
| | SocketAI | 84.10% | 82.13% | 87.42% | 84.69% | 198 | 131 |
| | **PyGuard** | **98.28%** | **97.58%** | **99.02%** | **98.29%** | **25** | **10** |

provide comprehensive coverage of current detection methodologies, enabling systematic comparison with our knowledge-driven approach.

## 6.3 RQ2: Accuracy

Table 5 presents detection performance across three evaluation datasets. PyGuard analyzes individual Python files within each package, flagging a package as malicious if any file is detected as malicious. PyGuard achieves 99.50% accuracy on the original dataset, substantially outperforming GuardDog's 94.12% accuracy. Bandit4Mal achieved only 33.37% accuracy with 1,927 false positives and 1,080 false negatives.

On the evaluation dataset, traditional tools exhibit context-blind detection that generates excessive false positives. GuardDog achieves the lowest baseline false positive rate at 4.30%, yet 58.02% of its errors stem from code-execution flags on legitimate operations such as `exec(compile(open('version.py').read(), 'version.py', 'exec'))` used for version checking. Bandit4Mal demonstrates a catastrophic 71.01% false positive rate, with 1,682 files (87.97%) flagged for url_found alerts that include project homepages and documentation links, alongside 56.28% and 39.75% false positive rates for basic read and write operations, respectively. The tool's overly broad rules generate an average of 5.37 detection types per file, exemplified by `os_sys-2.1.4`, which triggers 48 distinct alerts. PyPI WareCheck employs coarse-grained YARA rules that flag 97.00% of packages for metaprogramming_in_setup merely for utilizing standard Python imports, with one file generating 31,242 individual alerts. OSSGadget's 83.23% false positive rate results from broad LOLBAS [34] rules that flag 96.97% of packages for Linux commands, 87.13% for Windows utilities, and 71.85% for data exfiltration when packages execute routine file operations. In contrast, PyGuard achieves only 2 false positives and 19 false negatives. Among the 19 false negatives, 8 packages involve dependency confusion where the analyzed package does not contain malicious code but imports a malicious dependency, 6 packages install code from remote repositories at runtime, and 5 packages embed malicious payloads in non-Python files. These cases fall outside PyGuard's design scope of Python code behavioral analysis.

Hercule employs CodeQL-driven static analysis to detect five predefined behavior categories: exfiltration, file operations, network activities, obfuscation, and process manipulation. While it achieves 88.76% accuracy on the evaluation dataset, its analysis can take a long time for complex packages. We set a one-hour timeout per package, after which packages are classified as benign by default. Only a small number of packages exceeded this timeout: 112 on evaluation dataset (109 benign, 3 malicious), 10 on latest dataset (all benign), and none on obfuscation dataset. Since timed-out benign packages are correctly classified by default, the

timeout has minimal impact on the reported metrics. Its rigid rule definitions also limit adaptability. When applied to newer packages, accuracy drops to 83.03% as novel attack patterns fall outside its predefined rules. Cerebro constructs behavior sequences and fine-tunes a BERT model as a binary classifier. It achieves 89.12% accuracy on the evaluation dataset but suffers significant degradation on obfuscated code (78.00%), as syntactic transformations disrupt the token-level features that BERT relies on for classification.

The latest package dataset reveals adaptation limitations in rule-based systems. GuardDog's recall drops from 94.42% to 68.12% when encountering newer packages, with false positives shifting to concentrate on shady-links detection rather than code-execution patterns. PyGuard maintains 94.14% recall with only 49 false negatives, demonstrating resilience to evolving attack techniques. Obfuscation resistance varies dramatically across detection approaches. Machine learning models (SAP-DT, SAP-RF, SAP-XGB) exhibit 20-30% accuracy degradation when code transformations alter feature distributions. SAP-RF's recall drops from 78.05% to 46.88%, missing 553 malicious packages after obfuscation. Rule-based tools show mixed results: GuardDog maintains 89.60% accuracy through semantic-aware heuristics, while Bandit4Mal and OSSGadget hover around 50% accuracy. Our framework achieves 98.28% accuracy on obfuscated packages by analyzing behavioral sequences rather than syntactic patterns since variable renaming and control flow obfuscation do not alter the underlying API execution order that our pattern mining captures.

SocketAI, as an LLM-based approach, achieves 90.95% accuracy on the evaluation dataset, which is the highest among traditional baselines. Yet, it still generates 161 false positives and demonstrates performance degradation on newer packages (84.80%) and obfuscated code (84.10%). This decline reveals the limitations of relying solely on LLM general knowledge without domain-specific behavioral understanding. In contrast, our approach systematically extracts and summarizes knowledge from both malicious and false positive detection patterns, creating a specialized knowledge base that enables more precise contextual reasoning and maintains robust performance across all evaluation scenarios.

The performance differential stems from architectural differences in detection logic. Traditional tools scan code statements or extract statistical features, losing behavioral context during analysis. Our approach preserves execution sequences: a reverse shell pattern remains detectable as $[create\_socket, establish\_tcp\_connection, dup\_socket\_stdin, dup\_socket\_stdout, dup\_socket\_stderr]$ regardless of implementation variations. This sequence is inherently malicious because it initiates an outbound connection to a remote server and redirects standard I/O streams (stdin, stdout, stderr) to the socket, enabling attackers to interactively execute commands on the compromised machine. On our evaluation dataset, deterministic pattern matching identified 1,763 malicious packages and flagged 873 benign packages with justifiable patterns. The knowledge-driven LLM successfully filtered these false positives through contextual reasoning, achieving detection accuracy that exceeds the best baseline by 5.38% while reducing false positives by 98.58%.

Response to RQ2: Our knowledge-driven method achieves 99.50% accuracy with only 2 false positives compared to baselines. The behavioral knowledge enables consistent performance across emerging threats (97.37%) and obfuscated code (98.28%), where traditional tools drop to about 50% accuracy due to syntactic pattern disappearance.

## 6.4 RQ3: Ablation

We designed comparative experiments to evaluate RAG knowledge enhancement effects. Table 6 compares original detection results from static analysis tools with performance after RAG framework integration. The experimental workflow: static tools (OSSGadget, Bandit4Mal) detect and mark suspicious locations, then we extract corresponding code snippets and utilize the RAG framework to retrieve relevant behavioral patterns for secondary analysis and reclassification. Table 7 compares different LLM configurations: GPT-4.1 alone relies solely on pre-trained knowledge for direct code analysis, GPT-4.1-Mini+Knowledge combines a lightweight model with our extracted pattern and sequence knowledge, and GPT-4.1+Knowledge represents the complete solution. This design isolates and quantifies the independent contribution of RAG knowledge enhancement.

Knowledge enhancement dramatically improves static analysis tool performance across different limitation profiles. OSSGadget's accuracy increases from 50.01% to 89.91% while false positives drop from 2,000 to 192. The tool maintains high recall (88.38%) but generates excessive false positives through overly broad pattern matching. Adding behavioral knowledge preserves recall at 87.57% while increasing precision from 48.13% to 90.55%. Bandit4Mal exhibits the opposite problem: low recall (48.69%) due to conservative rule definitions. RAG enhancement maintains similar recall but increases precision from 34.72% to 81.28% and overall accuracy from 33.37% to 70.53%. These improvements stem from behavioral patterns providing semantic context that distinguishes legitimate API usage from malicious intent based on execution sequences rather than individual function calls.

Language model comparisons reveal that knowledge quality trumps model capacity. GPT-4.1 alone achieves 96.72% accuracy on normal packages but degrades to 68.58% on obfuscated code, generating 637 false positives when syntactic cues disappear. GPT-4.1-Mini with Knowledge reaches 99.51% accuracy, which surpasses the larger model without Knowledge by 2.79%, demonstrating that 304 behavioral patterns provide detection signals that raw language understanding cannot extract. This obfuscation resistance extends to enhanced static

Table 6: Knowledge Enhancement on Malicious Package Detection against Existing Tools

| Dataset | Method | Accuracy | Precision | Recall | F1-Score | FP | FN |
|---|---|---|---|---|---|---|---|
| Evaluation Dataset | OSSGadget (Only) | 50.01% | 48.13% | 88.38% | 62.32% | 2,000 | 244 |
| | OSSGadget + Knowledge | ↑39.90% 89.91% | ↑42.42% 90.55% | ↓0.81% 87.57% | ↑26.71% 89.03% | ↑1808 192 | ↓17 261 |
| | Bandit4Mal (Only) | 33.37% | 34.72% | 48.69% | 40.54% | 1,927 | 1,080 |
| | Bandit4Mal + Knowledge | ↑37.16% 70.53% | ↑46.56% 81.28% | ↓0.85% 47.84% | ↑19.69% 60.23% | ↑1695 232 | ↓18 1,098 |
| Latest Dataset | OSSGadget (Only) | 52.63% | 45.79% | 86.33% | 59.84% | 875 | 117 |
| | OSSGadget + Knowledge | ↑38.82% 91.45% | ↑51.16% 96.95% | ↓4.67% 81.66% | ↑28.81% 88.65% | ↑853 22 | ↓40 157 |
| | Bandit4Mal (Only) | 58.45% | 49.60% | 94.40% | 65.03% | 822 | 48 |
| | Bandit4Mal + Knowledge | ↑36.01% 94.46% | ↑47.24% 96.84% | ↓5.02% 89.38% | ↑27.93% 92.96% | ↑797 25 | ↓43 91 |
| Obfuscation Dataset | OSSGadget (Only) | 50.95% | 50.89% | 85.37% | 63.77% | 856 | 152 |
| | OSSGadget + Knowledge | ↑39.56% 90.51% | ↑45.48% 96.37% | ↓0.96% 84.41% | ↑26.22% 89.99% | ↑823 33 | ↓10 162 |
| | Bandit4Mal (Only) | 54.84% | 54.31% | 64.75% | 59.07% | 567 | 367 |
| | Bandit4Mal + Knowledge | ↑25.48% 80.32% | ↑41.24% 95.55% | ↓0.87% 63.88% | ↑16.50% 76.57% | ↑536 31 | ↓9 376 |

Table 7: Knowledge Enhancement on Malicious Package Detection against LLMs

| Dataset | Method | Accuracy | Precision | Recall | F1-Score | FP | FN |
|---|---|---|---|---|---|---|---|
| Evaluation Dataset | GPT-4.1 | ↓2.78% 96.72% | ↓5.16% 94.74% | ↓0.70% 98.38% | ↓2.96% 96.53% | ↓113 115 | ↓15 34 |
| | GPT-4.1-Mini + Knowledge | ↑0.01% 99.51% | ↓0.33% 99.57% | ↑0.29% 99.37% | ↓0.02% 99.47% | ↓7 9 | ↑6 13 |
| | **PyGuard (GPT-4.1 + Knowledge)** | **99.50%** | **99.90%** | **99.08%** | **99.49%** | **2** | **19** |
| Latest Dataset | GPT-4.1 | ↓9.91% 87.46% | ↓2.73% 96.64% | ↓15.38% 78.76% | ↓9.89% 86.79% | ↓25 30 | ↓184 233 |
| | GPT-4.1-Mini + Knowledge | ↑0.23% 97.60% | ↓2.22% 97.15% | ↑2.78% 96.92% | ↑0.35% 97.03% | ↓19 24 | ↓23 26 |
| | **PyGuard (GPT-4.1 + Knowledge)** | **97.37%** | **99.37%** | **94.14%** | **96.68%** | **5** | **49** |
| Obfuscation Dataset | GPT-4.1 | ↓29.70% 68.58% | ↓35.84% 61.74% | ↓0.27% 98.75% | ↓22.31% 75.98% | ↓612 637 | ↓3 13 |
| | GPT-4.1-Mini + Knowledge | ↓0.63% 97.65% | ↓1.03% 96.55% | ↓0.20% 98.82% | ↓0.62% 97.67% | ↓11 36 | ↓2 12 |
| | **PyGuard (GPT-4.1 + Knowledge)** | **98.28%** | **97.58%** | **99.02%** | **98.29%** | **25** | **10** |

1. PyGuard (GPT-4.1 + Knowledge): Full GPT-4.1 model enhanced with our Knowledge framework.

Table 8: Performance with Different LLM Backends on the Latest Dataset

| Method | Accuracy | Precision | Recall | F1-Score |
|---|---|---|---|---|
| PyGuard (GPT-4.1) | 97.37% | 99.37% | 94.14% | 96.68% |
| PyGuard (Qwen3-8B) | 95.24% | 96.86% | 92.55% | 94.65% |
| PyGuard (DeepSeek-V3) | **97.74%** | **98.30%** | **96.77%** | **97.53%** |

tools: OSSGadget+Knowledge maintains 90.51% accuracy on obfuscated code versus 50.95% baseline performance. The improvement stems from sequence-level patterns that persist through variable renaming and control flow transformations, remaining detectable regardless of syntactic obfuscation.

To evaluate reproducibility and reduce dependence on closed-source models, we tested PyGuard with open-source LLMs on the Latest Dataset. As shown in Table 8, DeepSeek-V3 achieves 97.74% accuracy, slightly outperforming GPT-4.1 (97.37%). Qwen3-8B achieves 95.24% accuracy with acceptable performance degradation. These results demonstrate that PyGuard generalizes across different LLM backends and does not require proprietary APIs.

Response to RQ3: RAG knowledge enhancement increases OSSGadget's accuracy by 39.90% and Bandit4Mal's by 37.16%, while reducing their false positives by 90.4% and 88.0% respectively. The behavioral knowledge enables GPT-4.1-Mini to achieve 99.51% accuracy, demonstrating that extracted sequences provide critical detection signals independent of model capacity.

## 6.5 RQ4: Usability

To evaluate PyGuard's real-world effectiveness, we deployed PyGuard on *PyPI.org* from March to June 2024. During this three-month period, 8,249 new packages were released on *PyPI.org*. Our system detected 233 packages as malicious, identifying 219 previously unknown malicious packages encompassing 287 versions that were not listed in the *OSV* or *Snyk* databases. Following responsible disclosure practices, each detected package underwent manual verification by our research team before submission to the PyPI security team via their official platform (https://pypi.org/security/). All 219 reported malicious packages were confirmed and subsequently removed by official PyPI maintainers, with 253 official acknowledgment letters received. We did not disclose any package information publicly before removal. We notified downstream mirror maintainers only after PyPI had confirmed and removed the packages. In contrast, baseline tools generated over 7,342 false positives on legitimate packages during the same period, highlighting the practical superiority of our approach in production environments. Table 9 shows a subset of the new malicious packages we detected. By analyzing Google Cloud's PyPI package download data, we found that these malicious packages were downloaded 39,420 times in three months. Approximately

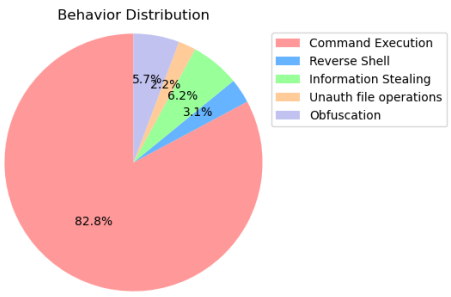


Figure 3: Distribution of Malicious Package Behaviors

37,584 downloads (95.1%) utilized the *sdist* (source code distribution) method, while the remaining downloads were conducted via *bdist_wheel* (binary distribution). Regarding geographical distribution, we observed that 66.5% of these downloads originated from the United States. Furthermore, China, Germany, and Singapore accounted for 8.4%, 7.4%, and 3.2% of the downloads, respectively. Source code analysis showed that 94.4% of malicious packages used ***install-time*** attacks. These attacks embed malicious code in the *setup.py* file or overwrite *CustomInstallCommand* class function, and trigger when the package is installed. Another 4.2% of malicious packages employed the ***import-time*** attack, inserting malicious code into *__init__.py* file, which activates when a user imports the package. The remaining malicious packages utilized the ***run-time*** attack method. Figure 3 shows the distribution of malicious behaviors in the newly detected packages. *Command execution* was the most prominent malicious behavior, occurring in 188 packages, accounting for 82.8% of the total. Additionally, information theft and code obfuscation were observed in 14 and 13 packages, respectively. Reverse shell behavior had a lower incidence, appearing in 7 packages, making up 3.1%. In the package metadata analysis, the majority of malicious packages employed typosquatting attacks, misleading developers by mimicking the names of legitimate packages.

Simultaneously, we conducted an analysis of the major downstream mirrors of PyPI and identified over 4,500 malicious packages across 7,200 versions. Among these, the Tsinghua, Tencent, Douban, and BFSU mirrors contained the highest number of malicious packages, each exceeding 1,000, as shown in Table 10. Notably, the Tsinghua mirror had more than 2,010 affected versions. In contrast, the Aliyun and Huawei mirrors were relatively secure, with only a small number of malicious packages detected. These residual malicious packages have been downloaded more than 200,000 times, posing a significant threat to user security. Using PyGuard, we detected and identified these malicious packages and informed the maintainers of the downstream mirrors. Both Tsinghua and Tencent have confirmed the issue and have removed the malicious packages.

PyGuard is fully automated. Package files without sensitive APIs need no analysis. Files with sensitive APIs require 1-3 LLM queries depending on pattern matching results: context extraction, action mapping, and RAG-based analysis if deterministic patterns are not matched. On average, PyGuard takes 34.09 seconds per package (8.8 seconds for malicious packages, while benign packages are usually larger and require more time). This is slower than static analysis tools such as Bandit4Mal (1.50s), OSSGadget (4.13s), and GuardDog (25.10s), but significantly faster than LLM-based SocketAI (164.67s) and the dynamic analysis tool Hercule (578.69s). The efficiency gain over SocketAI comes from our hybrid strategy where RAG-based analysis runs only when deterministic patterns are not matched. Regarding cost, PyGuard requires $0.023 per package compared to SocketAI's $0.156. When using open-source models such as DeepSeek-V3, the inference cost can be eliminated entirely.

Response to RQ4: PyGuard demonstrated strong practical utility in real-world application, successfully identifying 219 newly discovered malicious packages (confirmed by PyPI officials) and over 4,500 malicious packages in major downstream mirrors.

## 6.6 RQ5: Cross-ecosystem

To evaluate cross-ecosystem generalization, we assessed our approach on 1,762 NPM packages (953 benign, 809 malicious) using established baselines for JavaScript malware detection. We compared against GuardDog and OSSGadget, the most widely adopted static analysis tools for NPM malicious package detection, Cerebro [13], which performs cross-platform detection using behavior abstraction and BERT models, and GPT-4.1 as a foundational baseline. GuardDog and OSSGadget apply their native rule-based detection directly to NPM packages, while both GPT-4.1 and our knowledge-driven method analyze individual JavaScript files within each package. A package receives a malicious classification if any constituent file is flagged as malicious. We directly apply PyPI-derived patterns to NPM packages without any adaptation or ecosystem-specific modifications. The detection workflow remains identical to PyPI detection.

Table 11 demonstrates that PyGuard achieves superior cross-language performance with 98.07% accuracy, substantially outperforming all baselines. PyGuard significantly outperforms rule-based approaches, achieving 3.97% higher accuracy than GuardDog and 44.27% higher than OSSGadget. Compared to Cerebro, PyGuard achieves 10.75% higher accuracy and 25.42% higher recall, reducing false negatives from 226 to 20. The GPT-4.1 foundation model reaches 96.71% accuracy, while our approach delivers a 1.36% improvement and reduces false negatives from 43 to 20. The RAG enhancement maintains high precision at 98.26% and strong recall at 97.53%, with 14 false positives compared to Cerebro's 2 and OSSGadget's 734, demonstrating that behav-

Table 9: Some New Malicious Packages Detected from PyPI.org by PyGuard

| Package Name | Versions | Location | Command Execution | Information Stealing | Reverse Shell | Obfuscation | File Operation | Downloads |
|---|---|---|---|---|---|---|---|---|
| bussardweg4av3 | 1.0.0 | setup.py | ✓ | - | - | - | - | 106 |
| pytyon | 1.0.0 | setup.py | ✓ | - | - | - | - | 102 |
| eutherium | 1.0.0 | setup.py | ✓ | - | - | - | - | 199 |
| openeaa | 1.0.0 | setup.py | ✓ | - | - | - | - | 245 |
| class-py | 1.0.0 | setup.py | - | - | ✓ | - | - | 107 |
| quickwebbasicauth | 2.3.2 | setup.py | - | - | ✓ | - | ✓ | 307 |
| artifact-lab-3-package-e90915e1 | 0.1.1, 0.1.3, 0.1.4 | setup.py | ✓ | - | - | - | ✓ | 466 |
| testkaralpoc45654 | 1.0.0 | setup.py | - | ✓ | - | - | - | 176 |
| google-requests | 99.3.9 | pre_install.py | ✓ | - | - | - | - | 164 |
| lyft-service | 9.99.1 | pre_install.py | ✓ | - | - | - | - | 205 |
| jupyter-calendar-extension | 0.1 | pre_install.py | ✓ | - | - | - | - | 88 |
| networkx-match-algr | 0.1.1 | core.py | - | - | - | - | - | 353 |
| pyjous | 1.0.2 | setup.py | ✓ | - | - | ✓ | - | 113 |
| pyzelf | 2.0.1 | setup.py | ✓ | - | - | ✓ | - | 123 |
| thesis-package | 1.0.0 | setup.py | - | - | - | - | ✓ | 84 |
| thesis-uniud-package | 1.0.0 | setup.py | - | - | - | - | ✓ | 118 |
| builderknower2 | 0.1.12 ~0.1.30 | setup.py | ✓ | - | - | - | - | 3,137 |
| booto3 | 0.0.1 | setup.py | ✓ | - | - | - | - | 108 |
| utilitytools | 0.0.2 ~0.0.9 | __init__.py | - | - | - | - | ✓ | 1,152 |
| utilitytool | 0.0.2 | __init__.py | - | - | - | - | ✓ | 95 |
| nt4padyp3 | 0.0.2 | setup.py | ✓ | - | - | - | - | 340 |
| importlib-metadate | 99.99 | setup.py | ✓ | - | - | - | - | 229 |
| reqwestss | 0.1.0 | index.py | - | - | ✓ | - | - | 242 |
| numberpy | 0.1.0 | index.py | - | - | ✓ | - | - | 249 |
| defca | 3.0.0 | test.py | ✓ | - | - | ✓ | - | 444 |
| builderknower | 0.1.8 ~0.1.12 | setup.py | ✓ | - | - | - | - | 1,409 |
| rev0001q1 | 2.0.0 | setup.py | - | - | ✓ | - | - | 268 |
| revabc01q1 | 0.0.2 | setup.py | - | - | ✓ | - | - | 196 |

1. The symbol ✓indicates the presence of such information, while the symbol '-' indicates its absence. Location: the file where the malicious code is located. Downloads: the number of times the malicious package has been downloaded.

Table 10: Number of Malicious Packages in Mirrors

| Mirror Source | Package Nums | Version Nums |
|---|---|---|
| Tencent Mirror | 1,225 | 1,650 |
| Huawei Mirror | 2 | 5 |
| Douban Mirror | 1,219 | 1,649 |
| Aliyun Mirror | 1 | 3 |
| Tsinghua Mirror | 1,039 | 2,010 |
| BFSU Mirror | 1,034 | 2,005 |

ioral pattern abstractions facilitate effective cross-language transfer beyond pure language model capabilities.

This performance differential stems from fundamental limitations in rule-based detection approaches when applied across ecosystems. GuardDog's JavaScript rules rely on specific syntactic constructs such as `child_process.exec` calls and obfuscation patterns like `while (!![])` loops, but struggle with semantically equivalent malicious behaviors that use different implementation approaches, resulting in 3.97% lower accuracy compared to PyGuard. OSSGadget's NPM detection depends on literal string matching and basic regular expressions that capture surface-level indicators but lack contextual understanding, achieving 44.27% lower accuracy than our approach when legitimate packages employ similar constructs for benign purposes. These approaches fundamentally operate on language-specific syntactic signatures that cannot capture underlying behavioral semantics. Conversely, our behavioral abstraction approach captures semantic intent rather than syntactic patterns. Concepts like "process manipulation," "network communication," and "data exfiltration" manifest consistently across programming languages through different but functionally equivalent API calls. The PyPI-derived behavioral knowledge successfully transfers to NPM detection because malicious behaviors follow similar logical patterns regardless of implementation language: establishing network connections, executing system commands, and collecting sensitive information represent universal attack primitives. PyGuard has higher false positives than Cerebro because NPM has language-specific code semantics like npm script hooks that are not covered by PyPI-derived patterns. This semantic-level understanding enables effective cross-ecosystem knowledge transfer, demonstrating that behavioral patterns extracted from one package ecosystem can enhance malicious package detection across different programming environments.

Response to RQ5: Our pattern knowledge demonstrates effective cross-ecosystem generalization, achieving 98.07% accuracy on NPM packages and outperforming GuardDog by 3.97%, Cerebro by 10.75%, and OSSGadget by 44.27%. PyPI-derived behavioral knowledge successfully transfers to JavaScript malware detection, demonstrating effective pattern generalization across package ecosystems.

Table 11: Cross-language applicability: Performance evaluation on different programming languages

| Method | Accuracy | Precision | Recall | F1-Score | FP | FN |
|---|---|---|---|---|---|---|
| GuardDog | ↓3.97% 94.10% | ↓0.82% 97.44% | ↓8.04% 89.49% | ↓4.59% 93.30% | ↓5 19 | ↓1 85 |
| OSSGadget | ↓44.27% 53.80% | ↓48.43% 49.83% | ↓7.42% 90.11% | ↓33.72% 64.17% | ↓41 734 | ↓1 80 |
| Cerebro | ↓10.75% 87.32% | ↑1.40% 99.66% | ↓25.42% 72.11% | ↓14.21% 83.68% | ↑12 2 | ↓206 226 |
| GPT-4.1 | ↓1.36% 96.71% | ↓0.18% 98.08% | ↓2.85% 94.68% | ↓1.54% 96.35% | ↓1 15 | ↓23 43 |
| **PyGuard (RAG + GPT-4.1)** | **98.07%** | **98.26%** | **97.53%** | **97.89%** | **14** | **20** |

## 7 Limitation and Threats to Validity

**Knowledge Base Update.** PyGuard supports incremental updates. When novel attack techniques emerge, new patterns can be added to the existing knowledge base without rebuilding from scratch. In practice, such updates are rare. Most emerging malware reuses behavioral patterns that our knowledge base already covers.

**Pattern-based Limitations.** PyGuard performs code-level detection and cannot identify dependency confusion [15] attacks where malicious code resides in referenced packages rather than the analyzed package. It also cannot detect threats in remote repository installations or non-Python files embedded in packages. Attackers might try inserting benign sequences to evade detection. However, this evasion strategy is ineffective because our matching operates on minimum subsequences that do not need to appear consecutively. Our behavioral abstraction approach analyzes execution semantics rather than code syntax, which provides resilience against obfuscation techniques.

**LLM Dependency.** LLM hallucination is a potential concern. We mitigate this risk through two strategies. First, we use deterministic pattern matching as the primary detection method. Second, when patterns do not match, we retrieve similar cases from the knowledge base as few-shot examples to guide the LLM's reasoning. This grounds the LLM output in real-world examples rather than relying solely on its internal knowledge. We tested PyGuard with both closed-source models (GPT-4.1, GPT-4.1-mini) and open-source models (DeepSeek-V3, Qwen3-8B). All models achieved over 95% accuracy on the Latest dataset. As LLM capabilities improve, we expect PyGuard's performance to improve as well.

**Potential Attacks on Detection.** Adversaries may attempt to evade or attack PyGuard. Very long contexts may present challenges for LLM-based analysis, but segmented analysis can address this. Prompt injection is another concern. Our slicing approach extracts only relevant code contexts instead of entire files. This reduces the attack surface and limits opportunities for injected prompts to affect detection. Existing LLM defense techniques can also be integrated to further mitigate these risks [35, 36].

## 8 Related Works

In this section, we introduce the related works on malicious code detection and other security issues for PyPI packages.

**Malicious Code Detection.** Malicious code detection represents a critical research direction in cybersecurity, with researchers achieving significant progress in static analysis, dynamic monitoring, and deep learning methods. In the Python security domain, PyXhon [37] integrates static analysis and dynamic monitoring techniques to achieve precise function call tracing and security risk assessment. PyComm [38] constructs a machine learning model based on statistical attributes and string sequences. PBDT [39] combines call attributes, text statistics, and opcode sequence features for backdoor detection. The application of deep learning methods has further extended the boundaries of detection techniques. MSDT [40] proposes a static analysis method based on vector space representation to identify injection-type malicious code through anomaly detection strategies. ScriptNet [41] utilizes deep learning models [42] to process JavaScript files as byte sequences, enabling multi-level detection and classification. Additionally, researchers have developed specialized detection frameworks for specific scripting languages such as PowerShell [43] and JavaScript [44]. Given the concise syntax characteristics of scripting languages like Python [45], these methods demonstrate significant value in overcoming the limitations of traditional analysis techniques, providing new technical approaches for malicious code detection.

**Other Security Issues of PyPI Packages.** In the domain of software package ecosystem security, PyPI faces diverse security threats. Typosquatting [46–48] emerges as a primary attack vector, where attackers create malicious packages with names similar to popular ones. To counter these threats, the TypoGard [49] tool integrates lexical similarity models with package popularity metrics. Source code consistency verification [50] represents another key research direction, where py2src [51] establishes automatic mapping between PyPI packages and GitHub repositories, while LastPyMile [52] identifies potential malicious code injections through comparative analysis. In cross-ecosystem detection research, Cerebro [13] builds a detection framework based on high-level behavior abstraction and BERT models, while Ruian [24] achieves multi-ecosystem threat detection through registry metadata and system call analysis. Machine learning approaches demonstrate notable advantages, with Amalfi [53]

and PPD [54] enhancing malicious package detection through feature engineering and anomaly detection algorithms, respectively. Ea4mp [55] combines metadata and behavior to detect malicious PyPI packages [32]. However, existing methods rely on simple rules and black-box machine learning models, failing to effectively handle obfuscation and false positives that challenge practical deployment.

## 9 Conclusion

We present a knowledge-driven framework that transforms detection failures into behavioral knowledge for malicious package identification. By mining 304 discriminative patterns from 18,137 PyPI packages and integrating knowledge into our method, we achieve 99.50% accuracy with only 2 false positives compared to 1,927-2,117 in existing tools. The framework maintains robust performance on obfuscated code with 98.28% accuracy and demonstrates cross-ecosystem generalizability with 98.07% accuracy on NPM packages. This semantic approach reduces false positives by 99% while maintaining high performance against evolving threats.

## A Ethical Considerations

We conducted a stakeholder-based ethical analysis following the USENIX Security guidelines.

**Stakeholder Identification.** We identified the following stakeholders in our research: (1) *Package registry maintainers* (PyPI, NPM, and mirror sites such as Tsinghua and Tencent), who host and distribute packages; (2) *Developers and end users*, who may unknowingly install malicious packages; (3) *Malicious package authors*, who create and distribute malicious packages; and (4) *Security researchers*, who may build upon our work for future malware detection research.

**Impact Analysis and Mitigations.** For *(1) package registry maintainers*, our data collection could potentially increase server load. To mitigate this, we implemented request throttling with 5-second intervals between consecutive downloads. Given PyPI's approximately 2.1 billion daily downloads [1], our one-time collection of approximately 12,000 benign packages represents negligible load (<0.0006% of daily traffic). Upon detecting potentially malicious packages during our real-world deployment, we followed PyPI's official security reporting guidelines. Each detected package underwent manual verification by our research team before submission to the PyPI security team via their official platform (https://pypi.org/security/). All 219 reported malicious packages encompassing 287 versions were confirmed and subsequently removed by PyPI maintainers, with 253 official acknowledgment letters received. We notified downstream mirror maintainers only after PyPI had confirmed and removed the malicious packages from the main repository.

For *(2) developers and end users*, our research aims to protect them by detecting malicious packages before they cause harm. No malicious package information was publicly disclosed through any channels prior to removal, preventing potential exploitation during the disclosure period.

For *(3) malicious package authors*, while our detection techniques could potentially inform evasion strategies, we believe the benefit of protecting the broader developer community outweighs this risk. PyGuard focuses on semantic-level detection rather than signature-based methods, making evasion more challenging.

For *(4) security researchers*, we release our dataset and tools to facilitate future research in malicious package detection.

**Data Collection.** All malicious packages in our study were obtained from publicly available datasets, including Guo et al. [15] and Backstabbers-Knife-Collection [2], which are continuously maintained repositories of confirmed malicious packages. Benign packages were downloaded directly from PyPI.org and npmjs.com public registries. No personally identifiable information or sensitive user data was accessed or collected at any stage of this research. Additional details on dataset construction are provided in Section 4.1 and Section 6.1.

**Publication Decision.** We decided to publish this research because the benefits of enabling better malicious package detection outweigh the potential risks. Software supply chain attacks pose significant threats to developers and organizations worldwide, and our work provides practical tools and insights to defend against such attacks. The behavioral patterns we identified can help the security community develop more robust detection mechanisms.

## B Open Science

Our paper fully adheres to the open science policy introduced by USENIX Security. All code and experimental datasets are available at https://doi.org/10.5281/zenodo.17929520.

## C Acknowledgments

This research is supported by the National Research Foundation, Singapore, and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG4-GC-2023-008-1B); by the National Research Foundation Singapore and the Cyber Security Agency under the National Cybersecurity R&D Programme (NCRP25-P04-TAICeN); and by the Prime Minister's Office, Singapore under the Campus for Research Excellence and Technological Enterprise (CREATE) Programme. Any opinions, findings and conclusions, or recommendations expressed in these materials are those of the author(s) and do not reflect the views of the National

Research Foundation, Singapore, Cyber Security Agency of Singapore, Singapore.

## References


[1] Python Software Foundation, "Python package index statistics," https://pypi.org/stats/, 2024, accessed: May 24, 2025.

[2] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2020, pp. 23–43.

[3] A. Sejfia and J. Schöning, "Towards measuring supply chain attacks on package managers for interpreted languages," in *Network and Distributed System Security Symposium*, 2020.

[4] M.-E. M. Léveillé and R. Holt, "116 malware packages found on pypi repository infecting windows and linux systems," https://www.welivesecurity.com/2023/12/14/malicious-pypi-packages-stealing-sensitive-data/, 2023, accessed: May 24, 2025.

[5] S. Tomonaga, "New malicious pypi packages used by lazarus," https://blogs.jpcert.or.jp/en/2024/02/lazarus_pypi.html, 2024, accessed: May 24, 2025.

[6] PyCQA, "Bandit: Security linter for python," https://github.com/PyCQA/bandit, 2024, accessed: May 24, 2025.

[7] P. Kirhmajer, "Finding malicious pypi packages through static code analysis: Meet guarddog," https://securitylabs.datadoghq.com/articles/guarddog-identify-malicious-pypi-packages/, 2022, accessed: May 24, 2025.

[8] S. T. Mehedi *et al.*, "Dysec: A machine learning-based dynamic analysis for detecting malicious packages in pypi ecosystem," *arXiv preprint arXiv:2503.00324*, 2025.

[9] T. Iqbal, G. Wu, and Z. Iqbal, "Pypiguard: A novel meta-learning approach for enhanced malicious package detection in pypi through static-dynamic feature fusion," *Journal of Systems and Software*, 2025.

[10] L. Newman *et al.*, "Taming bad python packages: Assessing python malware detectors with a benchmark dataset," https://www.chainguard.dev/unchained/taming-bad-python-packages-assessing-python-malwar\e-detectors-with-a-benchmark-dataset, 2023, accessed: May 24, 2025.

[11] M. J. Schwartz, "One-third of popular pypi packages mistakenly flagged as malicious," https://www.darkreading.com/application-security/one-third-pypi-packages-mistakenly-flagged-malicious, 2023, accessed: May 24, 2025.

[12] L. Newman *et al.*, "Hunting malware on package repositories," https://www.chainguard.dev/unchained/hunting-malware-on-package-repositories, 2023, accessed: May 24, 2025.

[13] J. Zhang, K. Huang, Y. Huang, B. Chen, R. Wang, C. Wang, and X. Peng, "Killing two birds with one stone: Malicious package detection in npm and pypi using a single model of malicious behavior sequence," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 4, pp. 1–28, 2025.

[14] "PyGuard: Code and data for malicious pypi package detection," https://doi.org/10.5281/zenodo.17929520, 2025.

[15] W. Guo, Z. Xu, C. Liu, C. Huang, Y. Fang, and Y. Liu, "An empirical study of malicious code in pypi ecosystem," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 166–177.

[16] D.-L. Vu, Z. Newman, and J. S. Meyers, "Bad snakes: Understanding and improving python package index malware scanning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 499–511.

[17] Open Source Vulnerabilities Database, "Osv: Open source vulnerabilities," https://osv.dev/, 2024, accessed: May 24, 2025.

[18] J. L. Fleiss, "Measuring nominal scale agreement among many raters." *Psychological bulletin*, vol. 76, no. 5, p. 378, 1971.

[19] OpenAI, "Text embedding model that turns text into a numerical form," https://platform.openai.com/docs/guides/embeddings, 2024, accessed: May 24, 2025.

[20] D.-L. Vu, "Bandit4mal: A fork of bandit tool with patterns for identifying malicious python code," https://github.com/lyvd/bandit4mal, 2024, accessed: May 24, 2025.

[21] Microsoft, "Ossgadget: Collection of tools for analyzing open source packages," https://github.com/microsoft/OSSGadget, 2025, accessed: May 24, 2025.

[22] N. Zahan, P. Burckhardt, M. Lysenko, F. Aboukhadijeh, and L. Williams, "Leveraging large language models to detect npm malicious packages," in *2025 IEEE/ACM*

*47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2025, pp. 683–683.

[23] Warehouse, "Pypi malware checks," https://warehouse.readthedocs.io/development/malware-checks/#malware-checks, 2020, accessed: May 24, 2025.

[24] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *28th Annual Network and Distributed System Security Symposium, NDSS*, Feb. 2021. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/ndss2021_1B-1_23055_paper.pdf

[25] Guarddog, "Guarddog is a cli tool to identify malicious pypi and npm packages," https://github.com/DataDog/guarddog/, 2023, accessed: May 24, 2025.

[26] P. Ladisa, S. E. Ponta, N. Ronzoni, M. Martinez, and O. Barais, "On the feasibility of cross-language detection of malicious packages in npm and pypi," in *Proceedings of the 39th Annual Computer Security Applications Conference*, 2023, pp. 71–82.

[27] ClamAV, "Clamav - open source antivirus engine for detecting trojans, viruses, malware & other malicious threats," https://www.clamav.net/, 2024, accessed: May 24, 2025.

[28] N. Li, S. Wang, M. Feng, K. Wang, M. Wang, and H. Wang, "Malwukong: Towards fast, accurate, and multilingual detection of malicious code poisoning in oss supply chains," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1993–2005.

[29] X. Zheng, C. Wei, S. Wang, Y. Zhao, P. Gao, Y. Zhang, K. Wang, and H. Wang, "Towards robust detection of open source software supply chain poisoning attacks in industry environments," in *Proceedings of the 39th IEEE/ACM international conference on automated software engineering*, 2024, pp. 1990–2001.

[30] W. Liang, X. Ling, J. Wu, T. Luo, and Y. Wu, "A needle is an outlier in a haystack: Hunting malicious pypi packages with code clustering," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 307–318.

[31] R. Shariffdeen, B. Hassanshahi, M. Mirchev, A. El Husseini, and A. Roychoudhury, "Detecting python malware in the software supply chain with program analysis," in *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2025, pp. 203–214.

[32] X. Gao, X. Sun, S. Cao, K. Huang, D. Wu, X. Liu, X. Lin, and Y. Xiang, "Malguard: Towards real-time, accurate, and actionable detection of malicious packages in pypi ecosystem," in *Proceedings of the 34th USENIX Security Symposium*, Seattle, WA, USA, August 2025, pp. 4741–4758.

[33] Hnfull, "Intensio-obfuscator: Python source code obfuscator," https://github.com/Hnfull/Intensio-Obfuscator, 2024, accessed: May 24, 2025.

[34] Fortinet, "Living off the land (lotl) attacks and techniques," https://www.fortinet.com/resources/cyberglossary/living-off-the-land-lotl, 2025, accessed: May 24, 2025.

[35] B. C. Das, M. H. Amini, and Y. Wu, "Security and privacy challenges of large language models: A survey," *ACM Comput. Surv.*, vol. 57, no. 6, Feb. 2025. [Online]. Available: https://doi.org/10.1145/3712001

[36] Y. Liu, Y. Jia, R. Geng, J. Jia, and N. Z. Gong, "Formalizing and benchmarking prompt injection attacks and defenses," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 1831–1847. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/liu-yupei

[37] M. Sun, D. Gu, J. Li, and B. Li, "Pyxhon: Dynamic detection of security vulnerabilities in python extensions," in *2012 IEEE International Conference on Information Science and Technology*. IEEE, 2012, pp. 461–466.

[38] A. Zhou, T. Huang, C. Huang, D. Li, and C. Song, "Pycomm: Malicious commands detection model for python scripts," *Journal of Intelligent & Fuzzy Systems*, no. Preprint, pp. 1–13, 2022.

[39] Y. Fang, M. Xie, and C. Huang, "Pbdt: Python backdoor detection model based on combined features," *Security and Communication Networks*, vol. 2021, 2021.

[40] C. Tsfaty and M. Fire, "Malicious source code detection using transformer," *arXiv preprint arXiv:2209.07957*, 2022.

[41] J. W. Stokes, R. Agrawal, G. McDonald, and M. Hausknecht, "Scriptnet: Neural static analysis for malicious javascript detection," in *MILCOM 2019-2019 IEEE Military Communications Conference (MILCOM)*. IEEE, 2019, pp. 1–8.

[42] D. Gonzalez, T. Zimmermann, P. Godefroid, and M. Schäfer, "Anomalicious: Automated detection of anomalous and potentially malicious commits on github," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 258–267.

[43] D. Hendler, S. Kels, and A. Rubin, "Amsi-based detection of malicious powershell code using contextual embeddings," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 679–693.

[44] W. Xu, F. Zhang, and S. Zhu, "Jstill: mostly static detection of obfuscated malicious javascript code," in *Proceedings of the third ACM conference on Data and application security and privacy*, 2013, pp. 117–128.

[45] M. Ohm, F. Boes, C. Bungartz, and M. Meier, "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*, 2022, pp. 1–10.

[46] M. Taylor, R. Vaidya, D. Davidson, L. De Carli, and V. Rastogi, "Defending against package typosquatting," in *Network and System Security: 14th International Conference, NSS 2020, Melbourne, VIC, Australia, November 25–27, 2020, Proceedings 14*. Springer, 2020, pp. 112–131.

[47] G. Liu, X. Gao, H. Wang, and K. Sun, "Exploring the unchartered space of container registry typosquatting," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 35–51.

[48] B. Kaplan and J. Qian, "A survey on common threats in npm and pypi registries," in *Deployable Machine Learning for Security Defense: Second International Workshop, MLHat 2021, Virtual Event, August 15, 2021, Proceedings 2*. Springer, 2021, pp. 132–156.

[49] D.-L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Typosquatting and combosquatting attacks on the python ecosystem," in *2020 ieee european symposium on security and privacy workshops (euros&pw)*. IEEE, 2020, pp. 509–514.

[50] S. Scalco, R. Paramitha, D.-L. Vu, and F. Massacci, "On the feasibility of detecting injections in malicious npm packages," in *Proceedings of the 17th International Conference on Availability, Reliability and Security*, 2022, pp. 1–8.

[51] D.-L. Vu, "Py2src: Towards the automatic (and reliable) identification of sources for pypi package," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1394–1396.

[52] D.-L. Vu, F. Massacci, I. Pashchenko, H. Plate, and A. Sabetta, "Lastpymile: identifying the discrepancy between sources and packages," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 780–792.

[53] A. Sejfia and M. Schäfer, "Practical automated detection of malicious npm packages," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1681–1692.

[54] G. Liang, X. Zhou, Q. Wang, Y. Du, and C. Huang, "Malicious packages lurking in user-friendly python package index," in *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2021, pp. 606–613.

[55] X. Sun, X. Gao, S. Cao, L. Bo, X. Wu, and K. 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*, 2024, pp. 1159–1170.