MalTotal: Cost-Effective and Language-Agnostic Malicious Code Poisoning Detection for Millions of Repositories

JIAN ZHAO*, Huazhong University of Science and Technology, China
SHENAO WANG*, Huazhong University of Science and Technology, China
QINGYANG WU, Huazhong University of Science and Technology, China
YANJIE ZHAO†, Huazhong University of Science and Technology, China
XIAO CHENG, Macquarie University, Australia
HAOYU WANG, Huazhong University of Science and Technology, China

The widespread adoption of open source software (OSS) has introduced significant security risks, with malicious code poisoning attacks increasingly targeting public package registries and open-source platforms. Existing detection approaches, including heuristic-, learning-, and LLM-based methods, suffer from language-specific designs, limited generalization, and high analysis costs, making them unsuitable for large-scale multi-language analysis. To address these challenges, we propose MalTotal, a scalable and cost-effective framework for language-agnostic malicious code detection. MalTotal leverages LLM-assisted semantic reasoning to identify sensitive APIs, perform hybrid semantic slicing, and reconstruct malicious behavior contexts while reducing analysis overhead. Our evaluations show that MalTotal outperforms 8 state-of-the-art baselines, achieving an average F1-score of 93.1% across 5 mainstream languages. Its hybrid slicing reduces LLM token consumption by 94.0%, lowering the analysis cost from $86.25 to $5.19 on 2,168 repositories. In a large-scale study of 120K GitHub repositories containing over 7.3 million files, MalTotal discovered 564 previously unknown malicious repositories across multiple languages at a total cost of $338. These results demonstrate the effectiveness, scalability, and cost-efficiency of MalTotal in mitigating large-scale code poisoning attacks.

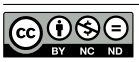
CCS Concepts: • **Security and privacy → Malware and its mitigation**.

Additional Key Words and Phrases: Open Source Software, Software Supply Chain, Code Poisoning

ACM Reference Format:
Jian Zhao, Shenao Wang, Qingyang Wu, Yanjie Zhao, Xiao Cheng, and Haoyu Wang. 2026. MalTotal: Cost-Effective and Language-Agnostic Malicious Code Poisoning Detection for Millions of Repositories. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA137 (October 2026), 24 pages. https://doi.org/10.1145/3832228

*Jian Zhao and Shenao Wang are the co-first authors.
†Yanjie Zhao (yanjie_zhao@hust.edu.cn) is the corresponding author.

Authors' Contact Information: Jian Zhao, jian_zhao_@hust.edu.cn, Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China; Shenao Wang, shenaowang@hust.edu.cn, Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China; Qingyang Wu, wuqingyang040802@gmail.com, Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China; Yanjie Zhao, Yanjie_Zhao@hust.edu.cn, Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China; Xiao Cheng, xiao.cheng@mq.edu.au, Macquarie University, Sydney, Australia; Haoyu Wang, haoyuwang@hust.edu.cn, Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China.



ACM 2994-970X/2026/10-ARTISSTA137
https://doi.org/10.1145/3832228

1 Introduction

The software supply chain has undergone a profound transformation over the past decade, with open source software (OSS) becoming a cornerstone of modern application development. OSS components now constitute up to 90% of typical software applications [54], and ecosystems such as Node Package Manager (NPM) [40] and Python Package Index (PyPI) [8] have experienced exponential growth. However, this widespread adoption has also introduced significant security risks [65, 76]. In particular, malicious code poisoning attacks [6, 41] targeting public package registries have become a critical threat, leveraging techniques such as typosquatting [6, 27], dependency confusion [2, 57], and vulnerability exploitation [16, 27, 70] to compromise downstream users. Recent reports [59] identified 17,954 new open source malware packages in Q1 2025 alone, with more than 828,925 malicious packages discovered since 2019. These threats have further expanded beyond package managers to a broader range of OSS components, including VSCode extensions [7, 32], pre-trained models [73, 75], MCP servers [11, 20], and agent skills [9, 49, 58]. Among them, GitHub has become a major attack surface due to its massive scale and central role in modern software development. Attackers exploit GitHub repositories through stealthy techniques such as repo confusion and fork bombs [52, 56], while real-world incidents such as the XZ Utils backdoor [1, 66] demonstrate the severe impact of repository-level supply chain attacks. Recent studies further reveal the prevalence of this issue, with over 100,000 GitHub repositories affected in a large-scale repo confusion campaign [56] and 9,294 out of 35.2K educational repositories containing malicious content [34]. Given that GitHub hosts over 518 million repositories [12], detecting malicious code poisoning in large-scale, heterogeneous, and multi-language codebases remains a critical challenge.

Research Gaps. To address these security concerns, researchers have proposed various detection techniques [42], which can be broadly classified into heuristic-based [6, 21, 30, 74], learning-based [28, 50], and large language model (LLM)-based approaches [62, 67, 69]. Heuristic-based methods rely on predefined rules or patterns, such as metadata [6, 30] or malicious behavior patterns [15, 21], and often use static [6, 15, 30] or dynamic analysis [35, 74] to detect threats [6, 15]. Learning-based approaches improve detection by learning malicious behavior patterns from labeled datasets, using features extracted from code [28, 48, 50], metadata [18, 47], or behavior sequences [22, 55, 72]. More recently, LLM-based methods have emerged as a promising direction, leveraging the reasoning capabilities of LLMs to analyze entire code files [67, 69] or package behaviors [22, 62]. Despite these advancements, existing methods face significant challenges when applied to malicious code detection across millions of language-heterogeneous repositories:

L1: Limited Scope and Language-Specific Design. Most methods are designed to work exclusively with specific programming languages (e.g., Python in PyPI [10, 47, 55] or JavaScript in NPM [22, 62, 68]), relying on ecosystem-specific behavior patterns [21, 30, 31] and manually curated sensitive API specifications [22]. Extending these tools to support additional languages often requires substantial re-engineering [6, 28, 30, 72], which makes large-scale, multi-language detection across millions of repositories infeasible. Furthermore, attackers can bypass detection by leveraging sensitive APIs in third-party libraries [17, 22], resulting in inadequate coverage of malicious behaviors.

L2: Limited Generalization to Malicious Semantics. Existing tools often abstract the rich and complex semantics of code into predefined behavioral patterns [6, 30] or statistical features [18, 28], relying on pattern-matching against known malicious behaviors [21, 30] or classification through ML/DL models [28, 50]. While these methods perform well for detecting known threats, they struggle with 0-day attack patterns or variations in malicious logic [61]. Additionally, learning-based methods typically yield binary classification results but offer little interpretable information to assist manual review [28, 50, 61], thus burdening security teams.

L3: High Costs and Limited Scalability. Emerging LLM-based approaches have yet to fully exploit LLMs' capabilities in analyzing malicious code semantics. Most methods either use LLMs for feature extraction [62], sensitive API analysis [10, 22], or data augmentation [68]. Some approaches attempt to use LLMs as direct judges of malicious code, but they suffer from severe hallucination issues [23, 71], context window limitations [67, 69], and expensive token processing costs [67, 69]. These limitations make it impractical to scale such methods across millions of repositories.

Motivation & Insights. Motivated by the limitations outlined in *L1* and *L2*, we propose LLMs as a silver bullet for language-agnostic malicious code semantic analysis. Our key insight is that LLMs, pre-trained on diverse code, possess domain knowledge on malicious patterns, and their alignment training enables them to distinguish malicious patterns from benign ones, making them a free lunch for language-agnostic malicious code detection. To validate this assumption, we conducted a pilot study where we used a few-shot approach to provide LLMs with repository-level, file-level, and minimal contextual code slices for evaluation. Our findings reveal that, with sufficiently simplified code contexts, LLMs can accurately reason about malicious code semantics. However, applying the LLM-as-a-Judge paradigm to analyze millions of repositories introduces significant challenges stemming from *L3*, specifically the context window limitations of LLMs and the prohibitive token costs associated with large-scale analysis. To address these issues, our research aims to design a scalable and cost-effective malicious semantic context slicing strategy for LLM-assisted repository-scale analysis. Inspired by the divide-and-conquer paradigm, we decompose this objective into two tasks: (1) parsing and identifying sensitive APIs across multi-language and third-party library environments, and (2) performing code slicing from sensitive API call sites while preserving malicious semantic contexts as much as possible.

Our Work. In this paper, we propose MalTotal, which uses LLMs for language-agnostic and cost-effective malicious code poisoning detection across millions of repositories. Specifically, MalTotal first identifies sensitive APIs across multi-language and third-party libraries through LLM-assisted semantic summarization. This enables the discovery of derived sensitive operations in third-party libraries, with the summarization cached for efficient reuse. Building on this, MalTotal performs backward code slicing for each identified sensitive operation, combining control dependency, data dependency, and call chain to construct a logically complete malicious behavior context. Finally, the sliced contexts are submitted to the LLM-as-a-Judge paradigm, which leverages its semantic reasoning capabilities to assess the intent of the code, classify potentially malicious behaviors, and provide interpretable reasons. By integrating these techniques, MalTotal systematically addresses the scalability and token cost challenges while ensuring precise, language-agnostic malicious code detection across large-scale codebases.

Contributions. To summarize, this paper makes the following contributions:

- **Language-Agnostic Analysis.** We propose MalTotal, a novel framework that leverages LLMs to enable language-agnostic malicious code detection. By identifying sensitive APIs across multi-language and third-party libraries, MalTotal generalizes beyond language-specific patterns and ecosystem-specific designs.
- **Cost-Effective Slicing.** To address the token costs and scalability challenges of LLMs, MalTotal introduces a hybrid semantic context slicing strategy, reducing token consumption by 94.0% while maintaining detection performance. This enables practical, cost-efficient analysis of large-scale repositories.
- **Scalability to Millions of Repositories.** We demonstrate the scalability and practical impact of MalTotal by analyzing 120K GitHub repositories spanning over 7.3 million files. With a total cost of just $338, MalTotal identified 564 previously unknown malicious repositories, showcasing its ability to handle real-world, large-scale codebases.

2 A Pilot Study

To validate the feasibility of leveraging LLMs for malicious code detection, we conducted a pilot study to explore the capabilities and limitations of LLMs in analyzing malicious code semantics. This study aimed to explore two key questions: (1) Can LLMs accurately reason about malicious intent in code across diverse programming languages? (2) What are the practical challenges, such as context window limitations and token consumption, when scaling to millions of repositories?

2.1 Study Setup

Dataset. To evaluate the feasibility of LLM-assisted malicious code detection, we curated malicious code samples from MalwareBench [29] (Python and JavaScript) and SourceFinder [45] (Java, Go, and PHP). To ensure diversity, we stratified projects by size (LoC and file counts) into five tiers and randomly sampled 10 valid projects from each tier for each language, resulting in 50 samples per language. Each sample was manually reviewed and annotated by three independent reviewers to verify its malicious behavior, with invalid or duplicate samples replaced by new samples from the same tier. The resulting dataset covers diverse malicious behaviors, including backdoors, spyware, sniffers, and cryptomining. We focus only on malicious samples in this pilot study, as our goal is to evaluate whether LLMs can effectively understand malicious code semantics rather than distinguish malicious code from benign code.

Code Context Configurations. To investigate the influence of different levels of contextual information on LLM-assisted malicious code detection, we prepared three granular levels of input contexts for each malicious sample.

- **Repository-Level (Repo):** The full source code of the repository, concatenated into a single input context. This strategy is similar to SocketAI [69], which provides the most complete context but often exceeds the model's context window and incurs high token costs.
- **File-Level (File):** All files within the repository that contain sensitive API calls, concatenated into a single input context. This configuration balances context richness and token cost.
- **Slice-Level (Slice):** A manually curated, minimal code snippet that represents the complete malicious logic extracted from the repository. By eliminating irrelevant or unrelated code, this configuration represents the idealized minimum context required for analyzing malicious logic.

LLM and Prompting Strategy. We used DeepSeek-V3 [5] as the foundational LLM in our pilot study. DeepSeek-V3 was selected due to its reasoning capabilities and relatively lower computational cost, making it a practical choice for this study. While it is not the current SOTA model for code reasoning, its performance is representative of trends expected from other advanced LLMs under similar conditions. The detailed comparison of different LLMs is provided in § 4.4. For the prompting strategy, we employed commonly used techniques such as role assignment, Chain-of-Thought (CoT) guidance, few-shot learning, and structured output to guide the model's reasoning process. The specific design of these prompts is described in § 3.3.

2.2 General Findings

Our experimental results, as summarized in Table 1, provide a comprehensive evaluation of the performance, cost, and scalability of LLMs for malicious code detection. To address the two key questions outlined in our study, we summarize the findings as follows:

Accuracy Across Context Levels. Our results demonstrate that LLMs can effectively identify malicious intent across programming languages when provided with optimized context granularity. The *Slice Context*, which focuses on minimal, directly relevant code snippets, achieved the lowest false negative rate (FNR) across all languages (e.g., 2.0% for JavaScript and 0.0% for Python), indicating that LLMs can accurately reason about malicious behaviors when unnecessary noise

Table 1. Performance comparison of different context levels across languages.

Language	Context	#All	Avg. LoC	Avg. #Files	OOC* (%)	FNR (%)	Tokens	Cost†
	Repo				14.0	16.3	1,502,941	$0.41
JavaScript	File	50	1085	9.5	4.0	14.6	900,943	$0.24
	Slice				**0.0**	**2.0**	**86,432**	**$0.02**
	Repo				40.0	0.0	2,141,938	$0.58
Python	File	50	1772	6.3	32.0	0.0	865,686	$0.23
	Slice				**0.0**	**0.0**	**133,212**	**$0.04**
	Repo				14.0	0.0	5,353,978	$1.45
Go	File	50	6428	28.3	0.0	0.0	338,074	$0.09
	Slice				**0.0**	**0.0**	**128,943**	**$0.03**
	Repo				12.0	2.3	2,515,472	$0.68
Java	File	50	3888	23.2	0.0	2.0	309,808	$0.08
	Slice				**0.0**	**0.0**	**69,991**	**$0.02**
	Repo				46.0	3.7	15,633,522	$4.22
PHP	File	50	7704	12.1	12.0	2.3	2,855,182	$0.77
	Slice				**0.0**	**0.0**	**107,396**	**$0.03**
	Repo				**25.2**	**4.1**	**39,703,851**	**$10.72**
Total	**File**	250	4176	15.9	**9.6**	**3.4**	**8,182,693**	**$2.21**
	Slice				**0.0**	**0.5**	**939,916**	**$0.25**

∗ **OOC (Out-of-Context)(%):** The percentage of samples where the input exceeded the maximum context, causing analysis failures.
† **Cost:** The cost is calculated based on the official DeepSeek pricing of $0.27 per million tokens.

is removed. However, the *Repository Context*, while comprehensive, suffers from higher FNRs in some cases (e.g., 16.3% in JavaScript) due to irrelevant or excessive information. The *File Context*, serving as a middle ground, provides reasonable accuracy while reducing token consumption. These findings confirm that LLMs are promising for malicious code detection across diverse languages, especially when the input context is carefully curated.

Scalability and Cost Efficiency. The *Repository* and *File Contexts* face significant challenges due to context window limitations, which restrict the ability of LLMs to process large codebases effectively. As shown in Table 1, 46.0% of PHP samples and 40.0% of Python samples could not be analyzed at the *Repository Context*, resulting in a high percentage of analysis failures. Although the *File Context* mitigates this issue to some extent, it still struggles with larger files. In contrast, the *Slice Context* completely eliminates token window constraints, achieving 0.0% N.A. samples across all languages by focusing on minimal, relevant code snippets. In addition to these technical constraints, the choice of context granularity has a profound impact on analysis cost. Analyzing 409 samples at the *Repository Context* consumed approximately 39.7 million tokens, costing $10.72, while the *File Context* required 8.2 million tokens ($2.21). By comparison, the *Slice Context* consumed only 0.94 million tokens, costing just $0.25, which reaches a 97.6% reduction in both token consumption and cost relative to the *Repository Context*. When scaled to millions of repositories, these differences become even more pronounced: repository-level analysis is estimated to cost over $26K for one million repositories, compared to just $625 for the *Slice Context*. While this estimation is approximate, it provides a general indication of the significant computational and cost burden for large-scale deployment. These results underscore the critical importance of optimizing context granularity to ensure cost-effective and scalable large-scale deployment.

Motivation and Insights. The results of this pilot study provide valuable insights that motivate the design of MalTotal. While LLMs demonstrate great promise as language-agnostic detectors

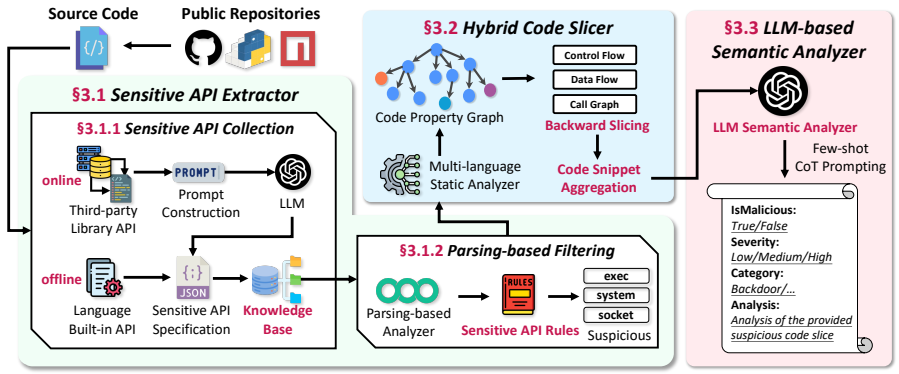


Fig. 1. The workflow of MalTotal

for identifying malicious code, directly applying them to vast raw codebases presents significant scalability and cost challenges. Our findings highlight that the core issue lies in providing LLMs with minimal yet precise malicious code contexts, as excessive or irrelevant information can lead to inefficiencies and reduced accuracy. These insights underscore the need for an automated framework that can extract precise malicious code contexts efficiently and at scale.

3 Design of MalTotal

To address the challenges of language-agnostic, scalable, and cost-effective malicious code detection, we present MalTotal, as shown in Figure 1. The methodology consists of three key components: *Sensitive API Extractor*, *Hybrid Code Slicer*, and *LLM-based Semantic Analyzer*.

3.1 Sensitive API Extractor

Drawn from existing studies [21, 22, 30], almost all malicious behaviors across programming languages are typically implemented through a series of sensitive APIs and their combined sequences of operations, such as those enabling code execution, cryptographic operations, and file operations. Rule-based methods and learning-based methods have widely utilized these sensitive API sequences to model malicious behaviors [21, 22, 30]. Following this insight, we also take the identification of sensitive APIs as the first step in pinpointing potential malicious behaviors within large codebases. However, prior works have primarily defined sensitive APIs based on heuristic knowledge, and they are often limited to built-in functions in Python and JavaScript [6, 10, 30, 74]. The most relevant work, SpiderScan [22], extends the sensitive API specification by considering TPLs within the NPM ecosystem. However, it is constrained to JavaScript and only reports the number of sensitive TPL APIs identified, without providing a usable or comprehensive list. To address these limitations, MalTotal systematically identifies sensitive APIs through two steps. First, MalTotal collects and organizes sensitive APIs, including both built-in functions and TPL functions, by leveraging LLM-assisted semantic analysis. Next, it employs parsing-based static analysis to preliminarily locate and identify sensitive operations within large codebases.

3.1.1 Sensitive API Collection. To comprehensively identify sensitive behaviors within codebases, MalTotal systematically collects sensitive APIs by targeting two main sources: *language-specific built-in APIs* and *third-party APIs* dynamically identified from external libraries. The collection of sensitive APIs is divided into two distinct phases: *language-specific built-in APIs* are collected through an *offline* process, where a static knowledge base is constructed prior to the analysis

Table 2. Taxonomy of sensitive APIs and representative examples across languages.

Category	Subcategory	Representative API Examples
Payload Execution	System Command Execution	`subprocess.run` (L1), `exec` (L1, L5)
	Dynamic Code Execution	`eval`, `exec` (L1, L5), `javax.script.ScriptEngineManager` (L4)
	External File Execution	`subprocess.run` (L1), `proc_open` (L5)
Network Access	Create Connections/Servers	`socket.bind/listen` (L1, L5), `net.Listen` (L3)
	Resolve DNS	`dns.lookup` (L2), `dns_get_record` (L5)
	Send Data	`socket.send` (L1, L5), `conn.Write` (L3)
	Receive Data	`socket.recv` (L1, L5), `conn.Read` (L3)
File Operation	Create or Delete Files	`os.mkdir` (L1), `mkdir` (L5), `os.Mkdir` (L3)
	Read Files	`open` (L1, L5), `file_get_contents` (L5), `os.Open` (L3)
	Write to Files	`os.write` (L1), `file_put_contents` (L5), `os.Create` (L3)
	Modify File Permissions	`os.chmod` (L1, L5), `os.Chmod` (L3)
	Link Operations	`os.symlink` (L1), `symlink` (L5), `os.Symlink` (L3)
	Search Files or Directories	`os.walk` (L1), `scandir` (L5), `filepath.Walk` (L3)
Sensitive Data Access	System Information	`os.uname` (L1), `php_uname` (L5), `runtime.GOOS` (L3)
	Env/Process Information	`os.getenv` (L1, L5), `process.env` (L2), `os.Getenv` (L3)
	User Information	`getpass.getuser` (L1), `get_current_user` (L5)
Cryptography	Create Cipher Objects	`rsa.generate_private_key` (L1)
	Encryption/Decryption	`crypto.createCipheriv` (L4), `openssl_encrypt` (L5)
	Encoding/Decoding	`base64.b64encode` (L1), `base64_encode` (L5)

∗ **L1**: Python, **L2**: JavaScript, **L3**: Go, **L4**: Java, **L5**: PHP.

phase; in contrast, *third-party APIs* are identified and analyzed in an *online* manner, where LLMs are employed during runtime to analyze and summarize the semantics of third-party APIs. As summarized in Table 2, these APIs are categorized based on their potential risk and malicious usage patterns, using a heuristically constructed taxonomy derived from previous rule-based and learning-based methods [21, 22, 30]. Specifically, the taxonomy consists of five major categories: payload execution, network access, file operations, sensitive data access, and cryptography.

Language-specific Built-in APIs. To construct a comprehensive knowledge base of built-in sensitive APIs, we integrated three primary sources: MalOSS, MalTracker, and official language documentation. First, we consolidated sensitive API specifications from MalOSS [6], which provides a detailed list of 1,257 sensitive APIs across Python, JavaScript, Java, and PHP. To further expand the knowledge base, we integrated 293 additional sensitive APIs for JavaScript extracted from MalTracker [68], ensuring no overlap with MalOSS. Next, we manually curated sensitive APIs from official language documentation [14, 24, 39, 43, 44], especially focusing on Go that no prior specifications were available. This process contributed 597 new sensitive APIs across all supported languages, including augmentations for Python, JavaScript, Go, Java, and PHP. As shown in Table 3, after combining all three sources and removing duplicates, we offline constructed a comprehensive sensitive API knowledge base comprising 2,147 built-in APIs[1].

Third-party Library APIs. The analysis of TPLs is conducted as an *online process*, tailored to the dependencies of the target project. During analysis, MalTotal first parses the dependency configuration files of the project (e.g., `pyproject.toml` for Python, `package.json` for JavaScript, etc.) to enumerate all declared TPLs and their corresponding versions. These libraries are then downloaded from their respective official repositories for analysis. To identify third-party APIs, MalTotal statically analyzes the project's source code to locate all imported functions and resolve their origins by inspecting their namespaces. If a function is determined to originate from a TPL, MalTotal retrieves its implementation by resolving the library structure and locating the corresponding source files. To maintain cost-effectiveness and prevent token bloat, MalTotal

[1]The complete list of these 2,147 sensitive built-in APIs is publicly available in our Anonymous Artifact.

Table 3. Statistics of collected built-in and TPL sensitive APIs across languages.

Language	Execution		Network		File Operation		Sensitive Data		Cryptography		Total	
	Built-in	TPL	Built-in	TPL	Built-in	TPL	Built-in	TPL	Built-in	TPL	Built-in	TPL
Python	127	52	153	97	141	58	14	4	23	0	458	211
JavaScript	79	31	149	41	194	59	20	14	82	2	524	147
Go	22	17	45	63	58	30	34	42	55	5	214	157
Java	80	1	225	26	172	7	19	1	175	0	671	35
PHP	69	26	63	37	84	26	3	20	61	35	280	144
Total	**377**	**101**	**635**	**227**	**649**	**154**	**90**	**61**	**396**	**7**	**2147**	**694**

System Prompt

You are a security expert. Your task is to analyze the provided {language} third-party package API for potential malicious usage.
TASK: Analyze the API and determine whether it involves any of the following sensitive behaviors: 1. Payload execution: ... 2. Network access: ... 3. File operations: ... 4. Sensitive data access: ... 5. Cryptography: ...
INPUT FORMAT: I will provide the following information: 1. Package name: {package_name} 2. Version: {package_version} 3. API path: {api_path} 4. Object type: {object_type} 5. Source code: {source_code}
OUTPUT FORMAT: Return the analysis in JSON format with the following fields:

```
{
    "is_risky": <Boolean>,
    "risk_types": [<String array of sensitive behaviors>],
    "description": "<Detailed explanation>", ...
}
```

SPECIAL INSTRUCTIONS: 1. Identify specific code snippets or function calls that exhibit sensitive behaviors. 2. Use the five categories of sensitive behaviors to classify risks...

Fig. 2. The partial prompt template used for LLM-based third-party API analysis across languages.

strictly restricts the extraction scope to the API's immediate function body, without recursively tracing its transitive callees. In practice, the function signature, its immediate implementation, and any available inline documentation provide sufficient semantic cues for intent inference, a design consistent with prior scalable analyses like SpiderScan[22]. The extracted code for each function is then provided to an LLM, which performs semantic reasoning to analyze its behavior and determine whether it corresponds to a sensitive operation, such as payload execution, network access, file operation, sensitive data access, and cryptography. Specifically, the semantic reasoning process is guided by a prompt template, which instructs the LLM to generate structured JSON outputs with risk assessments and specific reasons. A partial version of this prompt template is shown in Figure 2. The structured semantic summary generated for each third-party library API, regardless of whether it is identified as sensitive, is cached along with the (library, version, function) tuple. By caching these summaries, MalTotal avoids redundant reanalysis of the same APIs across different projects, improving efficiency and scalability. As part of our evaluation on the *Multi-Lang-Bench* (see § 4.1), MalTotal has analyzed and cached a total of 27,746 third-party APIs, among which 694 are sensitive APIs[2]. These sensitive third-party APIs are categorized and summarized in Table 3, showcasing the diversity of sensitive operations introduced by TPLs across different ecosystems.

3.1.2 Parsing-based Filtering. To efficiently filter suspicious projects and avoid unnecessary heavy analysis, we employ a parsing-based multi-language analyzer, Semgrep [51], as a lightweight pre-processing step. This process aims to quickly determine whether a target project contains any

[2]The complete list of these 694 sensitive third-party APIs is publicly available in our Anonymous Artifact.

sensitive API usages. If no sensitive APIs are identified, subsequent slicing and analysis are skipped, reducing the computational overhead while analyzing large-scale repositories.

Direct Calls. Direct calls refer to explicit invocations of sensitive APIs, such as `os.system` in Python or `exec` in JavaScript. For each sensitive API in the knowledge base, we compile specific rules to match these direct patterns, allowing us to efficiently identify sensitive API usage.

Indirect Calls and Aliases. Indirect calls occur when sensitive functions are invoked through intermediate references or aliases, such as `import os.system as executor` or `executor = os.system` in Python. For these cases, we intentionally avoid using alias analysis due to the high computational cost of tracking the point-to relationships at the pre-processing stage. Instead, we focus on identifying potential suspicious projects by matching sensitive library imports and alias assignment statements, which ensures that our pre-processing remains scalable and efficient.

By scanning the codebase, we identify all occurrences of sensitive APIs or libraries, either through direct calls or indirect references. Each matched instance is recorded with its corresponding location, API name, category, and source code for further processing. Projects without any matched sensitive APIs are filtered out, allowing us to focus subsequent slicing and deeper analysis on projects that are more likely to exhibit sensitive behaviors.

3.2 Hybrid Code Slicing

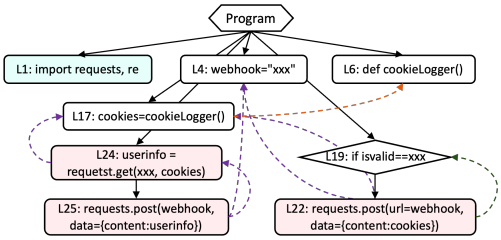
The hybrid code slicer extends the sensitive APIs identified during the parsing stage into semantically rich and logically complete *Suspicious Behavior Contexts* for further analysis. MalTotal's slicer is implemented using *Joern* [26], a fully open-source static analyzer that supports Code Property Graph (CPG) [25]. While tools such as CodeQL [13] are also capable of performing code slicing, we choose Joern for its fully open-source nature and ease of extension, which make it well-suited for our workflow. Nevertheless, MalTotal is generalizable and can be extended to other multi-language static analyzers.

CPG Generation and Pre-processing. The slicing process begins with the generation of a Code Property Graph (CPG), a unified intermediate representation that combines the Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG) into a language-agnostic graph structure. Using Joern, we convert the source code of each target project into a corresponding CPG, which serves as the foundation for subsequent slicing operations. For sensitive APIs identified during the filtering stage, we distinguish between two scenarios: direct calls and alias-based references. For direct calls, we initiate a backward slicing from the CPG node of the API call to identify all relevant ancestor code that could influence its execution. For alias-based references, where sensitive APIs are accessed via intermediate aliases, we perform a forward data-flow analysis starting from the alias definition node, such as an alias assignment or import statement. This forward trace identifies the actual call sites where the alias invokes the sensitive API. Once these call sites are located, we apply backward slicing from these positions to extract their complete contexts of execution.

Backward Hybrid Slicing. Following prior taint-based malicious package detection, we observe that malicious behaviors typically culminate in sensitive operations, i.e., security-relevant sinks such as data exfiltration or command execution. Based on this observation, MalTotal adopts a sink-driven backward slicing approach. This design mitigates the path explosion often caused by forward slicing from data sources, because a source value may propagate through numerous aliases, branches, function calls, and intermediate variables, most of which do not eventually reach a security-relevant sink. By starting from sinks, MalTotal focuses only on the control and data dependencies that directly affect sensitive operations, thereby reducing irrelevant paths and code contexts. Specifically, for each sensitive API call, whether identified directly or resolved through alias tracing, we recursively traverse the CPG to extract all ancestor code that could potentially influence its

Code Snippet from xoloqvqexetcqo-0.0.0/__init.py:

```
1  import requests,re
2  from discord_webhook import *
3  import browser_cookie3
4  webhook="discord.com/api/webhooks/xxx"
5
6  def cookieLogger():
7
8      data = []
9      cookies = browser_cookie3.firefox(domain_name="xxx.com")
10     for cookie in cookies:
11         if cookie.name == ".ROBLOSECURITY":
12             data.append(cookies)
13             data.append(cookie.value)
14             return data
15     cookies = browser_cookie3.chromium(domain_name="xxx.com")
16     ...
17 cookies = cookieLogger()
18  ...
19 if isvalid == "xxx":
20     pass
21 else:
22     requests.post(url=webhook,data={"content":cookies})
23     exit()
24 userinfo = requests.get("https://www.xxx.com/userinfo", cookies)
25 requests.post(webhook, data={"content":userinfo})
```



API (Line)	Dependency	Slice
Browser_cookie3.firefox (L9)	**CD:** N/A **DD:** L9 **CC:** L6-L15, L18	L6-L15, L17
Requests.post (L22)	**CD:** L19-L22 **DF:** L4, L6-L15, L17, L22 **CC:** L22	L4, L6-L15, L17, L19-L22
Requests.post(L25)	**CD:** N/A **DD:** L4, L6-L15, L17, L24-L25 **CC:** L24-L25	L4, L6-L15, L17, L24-L25
Aggregation	L4, L6-15, L17, L19-L22, L24-L25	

Fig. 3. An example of malicious behavior context extraction using hybrid slicing on the CPG. It includes (1) the input code snippet, (2) the CPG with call, control, and data dependencies, and (3) the aggregated slice that captures cookie collection and data exfiltration behavior.

behavior. This slicing process combines three types of dependencies: call-chain, control dependency, and data dependency slicing, allowing us to capture a complete representation of the suspicious behavior context. To prevent context bloat during this call-chain expansion, MalTotal enforces a maximum call depth constraint (default $k = 3$) and includes the whole callee function body. While strict intra-procedural slicing extracts fewer lines, preserving the complete structural context of the callee is crucial for LLMs to accurately comprehend the semantics without losing critical contextual cues, as empirically validated in § 4.3. For example, as shown in Figure 3, the `cookieLogger` function (L6) collects browser cookies using the `browser_cookie3` library (L9 and L15), and the collected cookies are subsequently exfiltrated to a hardcoded webhook URL via two separate `requests.post` calls (L22 and L25). When analyzing the API call `browser_cookie3.firefox` within the `cookieLogger` function (L9), call-chain slicing is used to expand the context to include the entire `cookieLogger` function and its call site (L6-L15 and L17). For the `requests.post` call located in the `else` branch of the conditional block (L22), control dependency slicing captures the enclosing `if-else` block (L19-L22) that dictates its execution conditions. Finally, data dependency slicing on both `requests.post` calls identifies their shared data origins, such as the `webhook` variable (L4) and the `cookieLogger` function, providing a complete context for the suspicious behavior. For example, the second `requests.post` call (L25) depends on user information fetched via `requests.get` (L24) and shares the same webhook destination. The raw output of this stage includes slices for each sensitive API call, capturing the relevant lines of code and their dependencies.

Context Refinement and Aggregation. The raw slices generated often contain significant redundancies and overlaps, making them inefficient for direct LLM analysis. To address this, we perform a two-step refinement process. First, we apply maximal content filtering, where any slice that is a proper subset of another is discarded. For instance, in Figure 3, the raw slice generated for `browser_cookie3.firefox` is entirely contained within the larger slices generated for the `requests.post` calls and is therefore discarded. Next, we perform similarity-based iterative aggregation, where slices whose line-level Jaccard similarity (the ratio of intersecting lines to the union of lines) exceeds a predefined empirical threshold ($\tau = 0.3$ in our implementation) are merged into

System Prompt

You are a security expert. I will provide a code slice and you must determine if it is malicious.
TASK: Your goal is to analyze the provided code snippets and determine if they contain malicious behavior. To do this, follow these steps in your reasoning process: 1. Identify sensitive operations: Pinpoint any code that performs actions like ... 2. Trace data flow: Trace the origin and flow of data to these sensitive operations ... 3. Assess intent: Based on the data flow and the context, determine... 4. Synthesize Findings: Consolidate your findings into the final JSON output.
INPUT FORMAT: I will provide code snippets in the following format:...
OUTPUT FORMAT: Return your analysis ONLY in the following JSON format:

```
{"is_malicious": <Boolean>,
"category":<Backdoor/Ransomeware/...>,
"analysis": "<Your brief reasoning>",...}
```

###EXAMPLES: ...
###SPECIAL INSTRUCTIONS: ...

Fig. 4. The prompt template for LLM-based analysis.

a single, more holistic context. Crucially, this empirical threshold prevents the aggregation from degenerating into a naive "merge-by-line" union. Semantically distinct slices that merely share trivial statements (e.g., common `import` declarations or global variables) will yield a low Jaccard similarity and remain separate. For example, the two slices for the `requests.post` calls, while not identical, share substantial code related to cookie theft and data exfiltration. These slices are aggregated into a unified context that encapsulates the entire data exfiltration routine (L4, L6-L15, L17, L19-L22, and L24-L25). This merging process continues iteratively until no more slices satisfy the similarity condition, resulting in a minimal set of highly condensed, logically distinct contexts. These refined slices are optimized for the final stage of semantic analysis, ensuring both efficiency and completeness in analyzing malicious behaviors.

3.3 LLM-based Semantic Analysis

The final stage of the MalTotal involves semantic analysis on high-fidelity contexts to determine malicious intent. This allows the system to distinguish between functionally similar but intentionally different behaviors, such as a legitimate update check versus a malicious data exfiltration. To guide the LLM in this complex task, we engineer a curated prompt template, partially illustrated in Figure 4, which systematically integrates four key prompting skills. 1) *Role Assignment*: We instruct the LLM to assume the role of a "Security Expert" to activate its specialized knowledge and reasoning patterns acquired during pre-training. 2) *CoT Guidance*: We prompt the LLM to follow an explicit four-step reasoning process to structure its analysis. It must first identify the sensitive operations within the slice, then trace the data flow to these operations, subsequently assess the likely intent based on this flow, and finally synthesize these findings into a conclusive judgment. This structured approach makes the LLM's reasoning process more transparent and enhances its analytical robustness. 3) *In-Context Learning*: We provide several few-shot examples of benign, suspicious, and malicious code snippets along with their expected analyses. These examples provide a clear reference for the model to adhere to our required output format, and they offer concrete demonstrations of the CoT process in various scenarios. 4) *Standardized JSON Output*: We require the LLM to return its findings exclusively in a uniform JSON format for machine-parsable results, including well-defined fields like `is_malicious` and `severity`. By combining these techniques, this LLM-based semantic analysis process becomes efficient, precise, and interpretable, forming the decision-making core of the MalTotal framework.

4 Evalution

To demonstrate the effectiveness, generalization, and scalability of MalTotal, we conducted extensive experiments addressing four RQs.

- **RQ1: Effectiveness & Generalization.** How effective is MalTotal in detecting malicious code poisoning across multiple programming languages compared to SOTA tools? Beyond Python and JavaScript, how well does MalTotal generalize to other languages?
  RQ2: Ablation Study & Sensitivity Analysis. What are the contributions of MalTotal's key components, and how sensitive is its performance to key slicing design choices?
- **RQ3: Impact of LLMs.** How does the choice of different LLMs affect MalTotal 's performance? And, to what extent does potential data contamination influence the results?
- **RQ4: Practicality & Cost-effectiveness.** How practical and cost-effective is MalTotal for large-scale analysis, including its end-to-end cost (e.g., time and token usage), false positive rates, and its ability to identify previously undetected malicious repositories in the wild?

4.1 Evaluation Setup

Implementation. We have implemented a prototype of MalTotal using over 5K lines of code (LoC) with Python and 3K LoC with Scala, excluding any third-party libraries or open-source tools. So far, MalTotal supports 5 mainstream programming languages: Python, JavaScript, Java, Go, and PHP. In our analysis workflow, we first employ Semgrep (v1.102.0) for a rapid, initial scan of the source code to identify calls to sensitive APIs. To enable the deeper, structural analysis required for our slicing algorithm, we then leverage Joern (v4.0.303) to uniformly convert the multi-language source code into Code Property Graphs. In the final semantic analysis, we selected the open-source DeepSeek-V3 model as our LLM adjudicator, which demonstrated powerful code comprehension capabilities and contributed to the cost-effectiveness of our solution.
Running Environment. All experiments were conducted on a server running Ubuntu Linux 22.04, equipped with two AMD EPYC Milan 7713 CPUs (2.0 GHz, 64 cores, 128 threads each), 512 GB RAM (8 x 64 GB modules), two NVIDIA A100 GPUs with 80 GB memory each, and four 7.68 TB NVMe SSDs (Western Digital SN640), providing a total storage capacity of 30.72 TB.
Dataset. To comprehensively evaluate MalTotal, we curated two datasets.

(1) Multi-Lang-Bench. To evaluate the effectiveness of MalTotal across multiple programming languages, we constructed a comprehensive dataset called *Multi-Lang-Bench* by combining two sources: the public MalwareBench [29] and a multi-language dataset derived from SourceFinder [45]. For Python and JavaScript, we adopted the widely recognized MalwareBench [29], which aggregates multiple public malicious package databases and supplements them with internal threat intelligence from Socket Security [53]. We consider MalwareBench relatively reliable for our study as it has undergone peer review and includes a large-scale pre-curated dataset. Initially, this dataset contained 4,430 malicious and 11,829 neutral NPM packages, alongside 3,770 malicious and 5,495 neutral PyPI packages. To prevent evaluation bias caused by repetitive malicious templates, we performed rigorous deduplication. After removing unreviewed samples and near-duplicates, the final retained set consists of 6,444 PyPI packages (3,188 malicious and 3,256 benign) and 9,545 NPM packages (3,438 malicious and 6,107 benign). For other languages, including Go, Java, and PHP, we constructed a dataset based on SourceFinder [45], which initially identified 870 potentially malicious repositories from GitHub (422 for Java, 274 for PHP, and 174 for Go). To ensure dataset quality, we first excluded repositories that were no longer accessible or labeled as "unknown" family in SourceFinder, resulting in 746 valid repositories (265 for Java, 169 for PHP, and 119 for Go). We then conducted a thorough manual review, where three reviewers independently cross-verified metadata (e.g., README files, descriptions, commit messages) and source code to confirm the presence of malicious logic. During

Table 4. Summary of the Multi-Lang-Bench used for evaluation.

Language	Source	Original	Valid	Reviewed	Malicious	Benign	Avg. LoC	Avg. #Files
Python	MalwareBench [29]	10,143	-	6,444	3,188	3.256	4,089	16.6
JavaScript	MalwareBench [29]	12,102	-	9,545	3,438	6,107	6,118	28.8
Java	SourceFinder [45]	422	265	75	75	75	7,560	79.2
PHP	SourceFinder [45]	274	169	143	143	143	6,330	41.6
Go	SourceFinder [45]	174	119	80	80	80	9,332	51.4

this process, we filtered out repositories misidentified as malicious by SourceFinder (e.g., due to heuristic errors), Android malware (unsupported by MalTotal now), and low-quality repositories such as spoofing or spam that lacked actual malicious code. After this process, we retained 75 malicious samples for Java, 143 for PHP, and 80 for Go. To construct a balanced dataset, we randomly selected an equal number of benign repositories with high stars from GitHub. These repositories were manually verified to minimize label noise. This resulted in final datasets of 150 samples for Java, 286 for PHP, and 160 for Go. Table 4 summarizes the data construction process, including the original dataset size, filtered valid data, and the final number of all samples for each language.

(2) GitHub-120K[3]. For the large-scale study in RQ4, our dataset was derived exclusively from the open-source data provided by StarScout [19], which has identified repositories exhibiting anomalous popularity signals, making them high-probability candidates for hosting malicious or scam-related content. From the initial list of 186K unique repositories, we successfully downloaded and prepared over 120K for further analysis. This strategy allowed us to focus our large-scale evaluation on a high-risk population of repositories to discover in-the-wild threats.

Baseline. To ensure a rigorous and comprehensive comparison, we systematically reviewed recent research and selected open-source baselines that are widely used in prior works [10, 22, 74]. Specifically, we select 8 representative SOTA techniques targeting the Python and JavaScript ecosystems. The rule-based detectors included Application Inspector [36], OSSGadget [37], Guarddog [3], and Bandit4mal [33] (Python only). For learning-based approaches, we selected the SAP [28], Amalfi [50] (JavaScript only), MalTracker [68] (JavaScrip only), and MalGuard [10] (Python only). To ensure a fair comparison, we established clear and consistent criteria for determining malicious behavior. For tools like Guarddog and OSSGadget, any reported alert was sufficient to classify a package as malicious. For ApplicationInspector and Bandit4mal, we adopted a stricter and widely-used criterion, where a package was deemed malicious only if the severity and confidence were both rated as "High". For other languages (Java, Go, PHP), we only compared MalTotal with ApplicationInspector and OSSGadget due to language support limitations, and applied the same criteria as mentioned before.

4.2 RQ1: Effectiveness & Generalization

To address RQ1, we evaluated MalTotal and 8 baseline tools on the Multi-Lang-Bench dataset. As summarized in Table 5, these results demonstrate the consistent superiority of MalTotal across all 5 programming languages and ecosystems.

Performance on Python & JavaScript. As summarized in Table 5, MalTotal demonstrated overall superior performance compared to the baseline tools across both ecosystems. On the Python dataset, MalTotal achieved an F1-score of 96.30%, marking a significant improvement of 0.80 to 74.90 percentage points over competing baselines, with MalGuard achieving the second-best F1-score of 95.50%. Similarly, on the JavaScript dataset, MalTotal led with an F1-score of 91.04%, surpassing the baselines by margins ranging from 5.63 to 35.43 percentage points. While MalTotal

[3]Although MalTotal is designed to scale to millions of repositories, we focused on this subset due to the substantial storage, download, and processing costs associated with the full StarScout dataset.

Table 5. Performance comparison of MalTotal and 8 baseline tools across different programming languages. The best performance is highlighted with **red**, and the second-best results are highlighted with **blue**.

Tool	Python			JavaScript			Go			Java			PHP		
	Pre.	Rec.	F1	Pre.	Rec.	F1	Pre.	Rec.	F1	Pre.	Rec.	F1	Pre.	Rec.	F1
AppInspector [36]	28.2	17.2	21.4	48.8	64.7	55.6	**35.1**	**43.0**	**38.6**	42.7	**54.7**	**48.0**	**50.6**	**63.6**	**56.4**
OSSGadget [37]	48.9	9.0	15.2	68.9	66.8	67.8	34.6	22.8	27.5	**44.1**	20.0	27.5	48.2	28.7	36.0
Guarddog [3]	92.4	92.9	92.6	75.9	**84.3**	79.9	-	-	-	-	-	-	-	-	-
Bandit4mal [33]	38.8	18.6	25.1	-	-	-	-	-	-	-	-	-	-	-	-
SAP [28]	86.2	53.3	65.8	**98.5**	46.6	63.3	-	-	-	-	-	-	-	-	-
Amalfi [50]	-	-	-	91.5	84.3	**85.4**	-	-	-	-	-	-	-	-	-
MalTracker [68]	-	-	-	76.6	66.7	71.3	-	-	-	-	-	-	-	-	-
MalGuard [10]	**97.1**	**94.0**	**95.5**	-	-	-	-	-	-	-	-	-	-	-	-
MalTotal	**95.7**	**96.9**	**96.3**	**93.3**	**88.9**	**91.0**	**95.9**	**87.5**	**91.5**	**97.1**	**88.0**	**92.3**	**98.5**	**90.2**	**94.2**

achieved the best overall performance on the JavaScript dataset, it showed lower precision (89.00%) compared to SAP, which achieved the highest precision of 98.50%. This discrepancy can be attributed to SAP's tendency to prioritize precision at the expense of recall, as evidenced by its significantly lower recall (46.60%) compared to MalTotal (89.00%). In contrast, MalTotal demonstrated a more balanced approach, excelling in recall and F1-score. These results highlight the robustness and high detection efficacy of MalTotal, validating its ability to effectively identify malicious packages in diverse ecosystems.

Multi-Language Generalization. The results in Table 5 highlight MalTotal's exceptional cross-language generalization capabilities. On the multi-language dataset, MalTotal achieved an impressive F1-score of 92.51%, significantly outperforming the baselines, ApplicationInspector and OSSGadget, which recorded F1-scores of 47.82% and 32.37%, respectively. Notably, this superior performance was consistent across all 5 tested languages, with MalTotal maintaining F1-scores above 85% in every case. In contrast, the baseline tools not only underperformed but also exhibited high variability in their effectiveness across different languages. These results validate that MalTotal effectively abstracts away language-specific syntax, enabling its detection engine to focus on the underlying semantic behavior of the code. This capability underscores MalTotal's ability to achieve language-agnostic generalization, making it a robust solution for detecting malicious code across diverse programming ecosystems.

False Positive Analysis. Our analysis of false positives (FPs) highlights the inherent limitations of static detection based on code semantics. In certain corner cases, developers of benign software, particularly in domains such as system administration or automated testing, may employ programming paradigms that closely resemble known malicious behaviors (e.g., dynamically constructing and executing system commands). These scenarios, where the code exhibits genuine ambiguity, present challenges for static detection. In such instances, our LLM-based analyzer may misclassify this benign yet high-risk code as malicious.

False Negative Analysis. The analysis of our false negatives (FNs) reveals two key areas for future improvement. First, MalTotal's current design focuses exclusively on source-level analysis, rendering it unable to detect attacks leveraging malicious dependencies. For example, a package with benign source code might introduce malicious behavior by depending on a compromised or maliciously-named package—an attack vector that lies outside MalTotal's current analysis scope. Second, MalTotal demonstrates limitations in detecting threats involving complex, indirect execution chains. In one observed case, malicious code used a command execution call to invoke the `powershell.exe`, which then executed a `.ps1` script file containing the actual malicious payload. Here, the critical malicious logic resides outside the immediate source code being analyzed, making it challenging for the LLM to confidently classify the package as malicious. Although employing

Table 6. Ablation study results on the impact of different configurations across 5 programming languages. The best results for each language are highlighted in bold.

Configuration	Python			JavaScript			Go			Java			PHP		
	Pre.	**Rec**	**F1**	**Pre.**	**Rec**	**F1**	**Pre.**	**Rec**	**F1**	**Pre.**	**Rec**	**F1**	**Pre.**	**Rec**	**F1**
w/o TPL API	**96.6**	94.5	95.6	**93.9**	86.5	90.1	93.2	85.0	88.9	97.0	86.7	91.6	97.7	88.8	93.0
w/o slicing	96.6	91.2	93.8	93.1	86.9	89.9	95.8	85.0	90.1	95.5	84.0	89.4	98.4	88.1	93.0
MalTotal	95.7	**96.9**	**96.3**	93.3	**88.9**	**91.0**	**95.9**	**87.5**	**91.5**	**97.1**	**88.0**	**92.3**	**98.5**	**90.2**	**94.2**

stricter prompts could potentially address such threats, this approach would likely increase false positives in legitimate use cases, highlighting the trade-off between sensitivity and precision in detection strategies.

4.3 RQ2: Ablation Study & Sensitivity Analysis

Ablation Study. To understand the contributions of individual components in MalTotal, we conducted an ablation study across 5 programming languages. In this study, we evaluated two variants of MalTotal:

(1) w/o TPL API identification: This variant disables the third-party API identification mechanism, which is responsible for detecting evasive threats leveraging obscure TPLs.

(2) w/o slicing: This variant removes the slicer and processes the full file as input to the LLM, providing the model with additional benign context but without filtering irrelevant code. Unlike the file-level configuration in the pilot study (§ 2), which concatenates multiple files and suffers from context overflow, this variant analyzes suspicious files individually.

The results of the ablation study are shown in Table 6. Removing either of the core components led to a noticeable decline in detection performance. Disabling the third-party API identification mechanism significantly reduced recall in all tested languages, highlighting its critical role in identifying evasive threats. These threats include corner-case attacks that exploit obscure TPLs to bypass scanners reliant on a predefined set of sensitive APIs. Without this mechanism, the system struggled to identify such threats, resulting in false negatives and a lower overall F1-score. The impact of removing the slicer was even more pronounced. Using the full file as input slightly improved precision, as the additional benign context reduced misclassifications. However, this came at the cost of a substantial drop in recall, which led to a much lower F1-score overall. This demonstrates that the slicer is not merely a cost-saving measure but an essential component for reducing noise in the input. By filtering out irrelevant code, the slicer ensures that the LLM focuses on the core malicious logic, thus improving performance and achieving a superior F1-score. It is also worth noting that because this variant processes files individually, it bypasses the severe out-of-context failures observed in the pilot study's baseline. Consequently, the performance drop here is more moderate.

Sensitivity of Slicing Strategy. To further validate our design choices for context extraction, we conducted an extended sensitivity study on 500 samples to evaluate the call-chain depth (k), component combinations, and slicing granularity. First, we evaluated the impact of the call-chain depth constraint, as shown in Table 7. The results demonstrate a clear trade-off between semantic completeness and context bloat. Shallow expansion ($k = 1$) is highly token-efficient (averaging 7,443.8 tokens) but fails to capture deeply nested cross-function behaviors, yielding a lower F1-score of 91.04%. As the depth increases to $k = 3$, the F1-score peaks at 94.18%. However, overly deep expansion ($k = 4, 5$) continuously inflates token usage (up to 11,157.6 tokens) while introducing excessive benign noise, which distracts the LLM and degrades the F1-score to 91.98%. Thus, $k = 3$ is selected as the default configuration for an optimal effectiveness-cost balance. Second, we conducted

Table 7. Call-Depth Sensitivity Analysis ($k = 1$ to 5)

Depth (k)	Precision	Recall	F1-Score	Avg. Tokens
$k = 1$	0.9348	0.8872	0.9104	7,443.8
$k = 2$	0.9402	0.9023	0.9209	9,182.9
$k = 3$ (MalTotal)	**0.9521**	**0.9318**	**0.9418**	**9,613.4**
$k = 4$	0.9517	0.9167	0.9339	10,227.4
$k = 5$	0.9504	0.9012	0.9198	11,157.6

Table 8. Slicing Strategy Analysis. CC: Call-Chain, DD: Data Dependency, CD: Control Dependency.

Variant	Components	Granularity	Precision	Recall	F1-Score	Avg. Tokens
V1	CC only	Whole Function	0.9365	0.8872	0.9112	8,201.6
V2	CC+DD+CD	Sliced Lines	0.8889	0.8421	0.8649	4,031.7
V3	CC+DD	Whole Function	0.9449	0.9091	0.9266	9,422.7
V4	CC+CD	Whole Function	0.9426	0.8647	0.9020	8,394.6
V5	DD+CD	Whole Function	0.9268	0.8636	0.8941	4,747.1
V6 (Full)	**CC+DD+CD**	**Whole Function**	**0.9521**	**0.9318**	**0.9418**	**9,613.4**

a fine-grained ablation on the dependency components and granularity at $k = 3$, as shown in Table 8. Comparing the full model (V6) with its sub-variants (V3, V4, V5) reveals the necessity of each dependency type. For instance, disabling Call-Chain (CC) expansion (V5) drastically reduces token usage but severely limits the context scope, dropping the F1-score to 89.41%. Disabling Data Dependency (DD) tracing (V4) hurts recall (86.47%), as the LLM loses visibility into the origins of sensitive variables. Finally, we assessed the call-chain granularity by comparing V2 (*Sliced Lines*) with V6 (*Whole Function*). While extracting only the strict execution lines (V2) further reduces average token consumption to 4,031.7, it severely deteriorates both precision and recall, plunging the F1-score to 86.49%. This suggests that removing the surrounding structural contexts of a callee, including variable initializations, loop conditions, and developer comments, can undermine the semantic integrity required by LLMs for accurate reasoning.

4.4 RQ3: Impact of LLMs

Impact of Different LLMs. To evaluate the impact of different LLMs on the performance of MalTotal, we conducted experiments using several well-known general-purpose LLMs, including GPT-4o, GPT-3.5 Turbo, Claude 4 Sonnet, Gemini 3 Pro, and DeepSeek-V3. The results, summarized in Table 9, show that the performance of different LLMs varies across programming languages, with no single model consistently dominating in all cases. For instance, GPT-4o demonstrates some advantages in Python, JavaScript, and Go, achieving the highest F1 scores in these languages. However, the improvements over other models, such as DeepSeek-V3 or Claude 4 Sonnet, are not always substantial. On the other hand, DeepSeek V3 achieves competitive or even superior results in PHP and Go, while maintaining a balanced performance across all languages. Considering both performance and inference costs, DeepSeek V3 emerges as a practical choice for large-scale analysis. While high-performing models like GPT-4o offer marginally better results in certain cases, their significantly higher computational costs may limit their feasibility in large-scale analysis.

Data Contamination Analysis. A potential concern in using LLMs for malicious package detection is the possibility of data contamination, where some malicious packages in the evaluation set may already exist in the LLM's training data. To address this issue, we leveraged the Multi-Lang-Bench dataset, whose packages were all released before May 2024, as the subset of data published *before*

Table 9. Comparison of MalTotal's performance using different LLMs across languages. The best results for each metric are highlighted with **red**, and the second-best results are highlighted with **blue**.

LLM	Python			JavaScript			Go			Java			PHP		
	Pre.	Rec.	F1	Pre.	Rec.	F1	Pre.	Rec.	F1	Pre.	Rec.	F1	Pre.	Rec.	F1
GPT-3.5 Turbo	**95.7**	95.7	95.7	**97.1**	84.2	90.2	95.2	**90.9**	**93.0**	88.2	**93.8**	90.9	**97.9**	82.1	89.3
GPT-4o	**98.6**	**97.0**	**97.8**	**94.3**	**91.9**	**93.1**	**95.5**	**91.3**	**93.3**	95.2	93.0	**94.1**	96.0	**87.3**	**91.4**
Claude 4 Sonnet	95.2	95.2	95.2	94.1	86.8	90.3	92.9	89.7	91.2	**97.6**	90.9	**94.1**	94.0	83.9	88.7
Gemini 3 Pro	86.9	95.2	90.9	78.7	**91.3**	84.5	82.0	89.3	85.5	83.9	**94.0**	88.7	85.1	87.0	86.0
DeepSeek-V3	**95.7**	**96.9**	**96.3**	93.3	88.9	**91.0**	**95.9**	87.5	91.5	**97.1**	88.0	**92.3**	**98.5**	**90.2**	**94.2**

Table 10. Performance comparison on packages released before and after the LLM training cut-off date for different LLMs. The increase is highlighted with **red**, and the decrease is highlighted with **blue**.

LLM	Subset	Python			JavaScript		
		Pre.	Rec.	F1	Pre.	Rec.	F1
DeepSeek V3	Before Cut-off	95.7	96.9	96.3	93.3	88.9	91.0
	After Cut-off	95.7	92.6	94.1	89.2	92.5	90.8
	Change	**0.0%**	**↓ 4.3%**	**↓ 2.2%**	**↓ 4.1%**	**↑ 3.6%**	**↓ 0.2%**
GPT-4o	Before Cut-off	98.6	97.0	97.8	94.3	91.9	93.1
	After Cut-off	95.3	91.0	93.1	88.0	90.4	89.2
	Change	**↓ 3.3%**	**↓ 6.0%**	**↓ 4.7%**	**↓ 6.3%**	**↓ 1.5%**	**↓ 3.9%**

the LLM training cut-off date. This ensures that these packages were released before the publication dates of both DeepSeek-V3 and GPT-4o. Additionally, we randomly collected 260 malicious packages released after January 2025 from the dataset maintained by DataDog [4] (130 for Python and JavaScript each). This specific date ensures that these packages were published after the release of both DeepSeek-V3 and GPT-4o, guaranteeing that none of these malicious packages appear in the training data of either of the two LLMs. The performance comparison, shown in Table 10, reveals that while there are slight performance differences between the two subsets, the results remain robust overall. For packages released *before* the cut-off date, there is a marginal improvement in precision and recall, which could suggest potential effects of data contamination, as these packages might have been seen during the LLM's training phase. However, for packages released *after* the cut-off date, the system still demonstrates strong performance, indicating that MalTotal effectively generalizes to unseen malicious packages. These findings suggest that while data contamination may introduce a slight advantage, its impact on the evaluation results is relatively acceptable and does not undermine the broader conclusions about MalTotal's generalization capabilities.

Reliability of LLM-based TPL Identification. A potential concern in LLM-assisted analysis is hallucination, particularly during the identification of sensitive third-party APIs. To systematically evaluate the output quality of our LLM-based TPL extractor, we conducted a manual verification study. Specifically, we randomly sampled and manually reviewed 50% (347 out of 694) of the TPL APIs flagged as sensitive by the LLM during our multi-language evaluation. Our manual inspection revealed only 4 false positives within this sample. This low error rate suggests that, for this well-defined semantic extraction task, the LLM can produce accurate and stable outputs when guided by explicit function signatures, immediate implementation bodies, and structured prompt engineering.

4.5 RQ4: Practicality and Cost-effectiveness

To evaluate the cost-effectiveness of MalTotal, we randomly selected 2,168 repositories from the GitHub-120K dataset for analysis. These repositories contain a total of 118,200 source files, amounting to over 300 million tokens. A naive file-by-file LLM analysis of such a dataset would

Table 11. Token cost reduction, scalability, and end-to-end time analysis across 2168 projects.

Context	#Projects	#Files	Total Tokens	Cost ($)	Savings	End-to-End Time (s)		
						Parsing	Slicing	LLM
All Source Files	2168	118,200	319,459,902	86.25	-	-	-	252.4
Suspicious Files		19,477	118,034,161	31.87	↓ 63.0%	40.6	-	83.7
Suspicious Slices		N/A	19,221,564	5.19	↓ 94.0%	40.6	53.3	46.2

be prohibitively expensive. However, as shown in Table 11, MalTotal achieved a substantial reduction in workload. The initial sensitive API identification effectively filtered out 83.5% of the files (from 118,200 to 19,477), significantly narrowing the analysis scope. The remaining suspicious files were then refined using the hybrid slicing technique, which further reduced the content to be analyzed. Ultimately, the token cost of the LLM-based semantic analysis was reduced to just 19.2 million, which means a 94.0% reduction compared to file-by-file analysis. In terms of time cost, incorporating parsing and slicing techniques did not significantly increase the overall analysis time. As shown in Table 11, the parsing step took only 40.6 seconds per project, while the slicing stage added another 53.3 seconds. These steps accounted for less than 100 seconds in total, an acceptable overhead given the cost reduction in the subsequent LLM-based analysis stage. These results demonstrate that MalTotal effectively narrows the scope of analysis without imposing substantial overhead, making MalTotal a practical solution for large-scale repo-level analysis.

To evaluate the real-world scalability and effectiveness of MalTotal, we deployed it on the GitHub-120K dataset, comprising 7.3 million files. The entire analysis pipeline consumed 1,255 million tokens at a total cost of approximately $338, demonstrating the economic feasibility of our approach at scale. MalTotal initially reported 1,553 repositories as potentially malicious, which were subsequently subjected to a rigorous manual review. This review was independently conducted by two security researchers with over 5 years of experience, with any disagreements resolved by a senior expert possessing 7 years of experience. To ensure precision, repositories were only confirmed as malicious if they exhibited clear, undeniable malicious intent. This review process filtered out 502 false positives. Although this yields a False Discovery Rate (FDR) of 32.2%, the overall False Positive Rate (FPR) across the 120K analyzed repositories remains remarkably low. Further investigation into these false positives revealed that they primarily stem from "gray-ware" or benign utilities whose behaviors closely mimic malicious actions (e.g., legitimate remote administration tools or system monitors executing shell commands). We also excluded 487 repositories that explicitly self-identified in their README files as being for penetration testing, security research, or malware collection. This process yielded a final set of 564 confirmed, previously unknown malicious repositories. Specifically, we identified 200 repositories in C/C++, 174 in Python, 56 in JavaScript, 31 in Java, and 21 in Go, with the remainder in other environments. The dominant malicious patterns included credential exfiltration (e.g., token grabbers and injectors, 73 cases) and direct payload execution (71 cases), alongside covert cryptocurrency miners and deceptive malicious forks. Following responsible disclosure guidelines, all 564 confirmed repositories were reported to the GitHub security team. This large-scale study demonstrates that MalTotal is not only capable of scaling cost-effectively to analyze massive codebases but is also highly effective at uncovering multi-language threats that evade traditional security measures.

5 Discussion

Limitations. MalTotal has several limitations arising from practical and methodological constraints. First, the current implementation of MalTotal is affected by the maturity of the underlying static analysis toolchain. Although its LLM-based semantic analysis is language-agnostic in

principle, the slicing stage requires language-aware static-analysis support. Therefore, extending MalTotal to a new language mainly requires integrating a suitable parsing or slicing backend and curating an initial set of built-in sensitive APIs. While our prototype uses a Joern-based CPG backend, other mature multi-language analysis frameworks, such as CodeQL [13] and YASA [63], could also be incorporated to provide parsing, dependency analysis, or slicing capabilities. Second, MalTotal is designed for multi-language but not cross-language attacks. It cannot trace data or control flow in language boundaries—e.g., when a Python script invokes a C library, which creates a significant blind spot for advanced attacks where malicious payloads are concealed in native extensions. Third, to ensure scalability across millions of repositories, our pre-filtering stage deliberately omits complex intra- and inter-procedural alias resolution. While our high overall recall (averaging over 90%) empirically demonstrates that this trade-off is acceptable for current real-world threats, it inherently creates a blind spot for sophisticated attacks that exclusively hide sensitive operations behind deep, multi-hop alias chains. Finally, our reliance on LLMs introduces a new attack surface. Finally, our reliance on LLMs introduces a new attack surface. The LLM-based analyzer is susceptible to prompt injection attacks, where embedded instructions in code could manipulate the model's reasoning. For example, attackers could embed deceptive comments or misleading strings to trick the model into classifying malicious code as benign.

Future Work. The modular architecture of MalTotal opens up opportunities for future enhancements and applications. One key direction is to expand its applicability beyond the current programming languages and repositories to encompass other ecosystems, such as VSCode extensions, MCP servers, agent skills, and other domain-specific software ecosystems. Additionally, the extensibility of MalTotal enables support for more programming languages and addressing more sophisticated threats, such as C/C++, Ruby, etc. Another important area of improvement is incremental analysis. For large-scale codebases, analyzing every file during each scan can be computationally expensive and time-consuming. Instead, MalTotal can focus on analyzing changes introduced in pull requests (PRs) or commits, significantly reducing the overhead while maintaining security coverage. Finally, to maximize the real-world impact of MalTotal, we are actively collaborating with the VirusTotal team to integrate MalTotal into their platform.

6 Related Work

Software supply chain security has become a critical research focus due to the increasing prevalence of attacks targeting package registries and open-source repositories.

Attack Vectors in Software Supply Chain Recent research has extensively examined the diverse and evolving attack vectors in software supply chains. Gu et al. [16] systematically investigated six major software registry ecosystems, identifying twelve potential attack vectors, including six novel ones. Their work highlights how adversaries exploit inconsistencies in registries, mirrors, and clients to distribute malicious code. Similarly, Zahan et al. [70] analyzed metadata from over 1.63 million npm packages, proposing six signals of security weaknesses, such as install scripts and expired maintainer email domains. To structure this complex threat landscape, Ladisa et al. [27] proposed a comprehensive taxonomy of 107 unique attack vectors, covering all stages of the supply chain and linking them to real-world incidents and safeguards. Neupane et al. [38] further expanded on specific threats, categorizing 13 distinct mechanisms of package confusion attacks that adversaries use to mislead developers, moving beyond traditional typosquatting. Finally, Wermke et al. [64] explored the challenges companies face when integrating open-source components, such as the difficulty of auditing dependencies and the lack of dedicated resources for managing supply chain risks. Their findings emphasize the importance of improving security practices and fostering a healthier open-source ecosystem.

Mitigating Malicious Package in PyPI/NPM Ecosystems. In response to these threats, a diverse array of detection techniques has been developed. Established approaches leverage *rule-based program analysis*[6, 30], using either static analysis engines like CodeQL for semantic signature matching[15] or dynamic analysis in sandboxes to monitor runtime behavior [21, 35, 74]. However, a critical barrier for these tools is their often unacceptably high false positive rate in real-world deployments [61]. To improve generalization, *learning-based methods* learn patterns from features extracted from package metadata [18, 47] or source code [28, 48, 50]. While more flexible, their performance is fundamentally tied to hand-crafted features that may fail to capture the semantic nuances of novel attacks. The advent of *LLMs* has opened a new frontier. However, their application has often been limited to auxiliary tasks like sensitive API identification [10, 22], feature generation [62], or cross-language data augmentation [68]. Studies that do use LLMs as direct judges have highlighted their potential but also underscored the prohibitive costs and context window limitations [67, 69]. While these works provide valuable insights into malicious package detection, MalTotal addresses multi-language generalization challenges and context limitations.

Mitigating Malicious Behavior in GitHub Repositories. Beyond package registries, GitHub has been identified as a significant reservoir for malicious source code [34, 45, 60]. However, existing detection methods for this domain predominantly operate at a coarse-grained, non-code level. These approaches typically rely on identifying malicious projects through repository similarity and clustering based on metadata and file structure [46], or by training models on project-level metadata such as README content or anomalous social signals like fake stars [19]. While computationally efficient for flagging certain types of malicious activity, these methods fundamentally bypass deep, semantic inspection of the source code itself. This leaves a critical blind spot for novel or stealthy malware that does not exhibit obvious structural or social anomalies. In contrast, MalTotal addresses their limitations by enabling fine-grained semantic analysis of source code, leveraging LLMs to detect malicious behaviors in language-heterogeneous Github codebases.

7 Conclusion

In this paper, we propose MalTotal, a novel framework for scalable and language-agnostic malicious code detection in large-scale codebases. By integrating LLM-assisted semantic reasoning with a hybrid semantic slicing strategy, MalTotal effectively identifies and analyzes malicious behaviors across diverse programming languages while reducing token consumption by 94.0%. Extensive evaluations demonstrate its superiority over 8 state-of-the-art baselines, achieving an average F1-score of 93.1% across 5 programming languages. Furthermore, MalTotal successfully scaled to analyze 120K GitHub repositories, identifying 564 previously unknown malicious repositories with a total cost of just $338. These results validate MalTotal's effectiveness, cost-efficiency, and scalability, highlighting its practical potential to mitigate large-scale code poisoning attacks.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China (grants No. 62502168, 62572209), and by the Hubei Provincial Key Research and Development Program (grant No. 2025BAB057).

Statement on the Use of AI Tools

We confirm that this work involved the use of generative AI tools for polishing the language of the paper as well as for LLM-assisted coding in non-critical tasks. No AI-generated content contributed to the scientific or technical originality of the paper. All use of AI tools complies with ACM's policies on authorship and the use of generative AI technologies.

Data Availability Statement

We have made the implementations of MalTotal and experimental data publicly accessible at https://github.com/security-pride/MalTotal.

References

[1] Akamai Security Intelligence Group. 2024. XZ Utils Backdoor: Everything You Need to Know, and What You Can Do. https://www.akamai.com/blog/security-research/critical-linux-backdoor-xz-utils-discovered-what-to-know Accessed: 2025-07-12.

[2] Alex Birsan. 2021. Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610 Accessed: 2026-01-30.

[3] DataDog. 2024. GuardDog. https://github.com/DataDog/guarddog. Accessed: 2026-01-30.

[4] DataDog. 2026. malicious-software-packages-dataset. https://github.com/DataDog/malicious-software-packages-dataset. Accessed: 2026-01-30.

[5] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[6] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *28th Annual Network and Distributed System Security Symposium, NDSS*. https://www.ndss-symposium.org/wp-content/uploads/ndss2021_1B-1_23055_paper.pdf

[7] Shehan Edirimannage, Charitha Elvitigala, Asitha Kottahachchi Kankanamge Don, Wathsara Daluwatta, Primal Wijesekara, and Ibrahim Khalil. 2024. Developers Are Victims Too : A Comprehensive Analysis of The VS Code Extension Ecosystem. *CoRR* abs/2411.07479 (2024). arXiv:2411.07479 doi:10.48550/ARXIV.2411.07479

[8] Python Software Foundation. 2025. PyPI: The Python Package Index. https://pypi.org/ Accessed: 2026-01-30.

[9] Laura French. 2025. Claude Agent Skills could be used to deploy malware, researchers say. https://www.scworld.com/news/claude-agent-skills-could-be-used-to-deploy-malware-researchers-say. Accessed: 2026-01-30.

[10] Xingan Gao, Xiaobing Sun, Sicong Cao, Kaifeng Huang, Di Wu, Xiaolei Liu, Xingwei Lin, and Yang Xiang. 2025. MalGuard: Towards Real-Time, Accurate, and Actionable Detection of Malicious Packages in PyPI Ecosystem. *CoRR* abs/2506.14466 (2025). arXiv:2506.14466 doi:10.48550/ARXIV.2506.14466

[11] Mohamed Ghobashy. 2025. Shiny tools, shallow checks: how the AI hype opens the door to malicious MCP servers. https://securelist.com/model-context-protocol-for-ai-integration-abused-in-supply-chain-attacks/117473/. Accessed: 2026-01-30.

[12] GitHub. 2024. Octoverse: AI leads Python to top language as the number of global developers surges. https://github.blog/news-insights/octoverse/octoverse-2024/ Accessed: 2025-07-12.

[13] GitHub. 2025. CodeQL: Semantic Code Analysis Engine. https://codeql.github.com/. Accessed: 2026-01-30.

[14] Go. 2025. Go Standard Library. https://pkg.go.dev/std. Accessed: 2026-01-30.

[15] Matías F. Gobbi and Johannes Kinder. 2023. Poster: Using CodeQL to Detect Malware in npm. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 3519–3521. doi:10.1145/3576915.3624401

[16] Yacong Gu, Lingyun Ying, Yingyuan Pu, Xiao Hu, Huajun Chai, Ruimin Wang, Xing Gao, and Haixin Duan. 2023. Investigating Package Related Security Threats in Software Registries. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 1578–1595. doi:10.1109/SP46215.2023.10179332

[17] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. 2023. An Empirical Study of Malicious Code In PyPI Ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 166–177. doi:10.1109/ASE56229.2023.00135

[18] Sajal Halder, Michael Bewong, Arash Mahboubi, Yinhao Jiang, Md. Rafiqul Islam, Md Zahidul Islam, Ryan H. L. Ip, Muhammad Ejaz Ahmed, Gowri Sankar Ramachandran, and Muhammad Ali Babar. 2024. Malicious Package Detection using Metadata Information. In *Proceedings of the ACM on Web Conference 2024, WWW 2024, Singapore, May 13-17, 2024*, Tat-Seng Chua, Chong-Wah Ngo, Ravi Kumar, Hady W. Lauw, and Roy Ka-Wei Lee (Eds.). ACM, 1779–1789. doi:10.1145/3589334.3645543

[19] Hao He, Haoqin Yang, Philipp Burckhardt, Alexandros Kapravelos, Bogdan Vasilescu, and Christian Kästner. 2026. Six Million (Suspected) Fake Stars on GitHub: A Growing Spiral of Popularity Contests, Spam, and Malware. In *International Conference on Software Engineering (ICSE)*. ACM. doi:10.1145/3744916.3764531

[20] Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. 2025. Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions. *CoRR* abs/2503.23278 (2025). arXiv:2503.23278 doi:10.48550/ARXIV.2503.23278

[21] Cheng Huang, Nannan Wang, Ziyan Wang, Siqi Sun, Lingzi Li, Junren Chen, Qianchong Zhao, Jiaxuan Han, Zhen Yang, and Lei Shi. 2024. DONAPI: Malicious NPM Packages Detector using Behavior Sequence Knowledge Mapping. In *33rd*

USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/huang-cheng

[22] Yiheng Huang, Ruisi Wang, Wen Zheng, Zhuotong Zhou, Susheng Wu, Shulin Ke, Bihuan Chen, Shan Gao, and Xin Peng. 2024. SpiderScan: Practical Detection of Malicious NPM Packages Based on Graph-Based Behavior Modeling and Matching. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 1146–1158. doi:10.1145/3691620.3695492

[23] Motunrayo Ibiyo, Thinakone Louangdy, Phuong T. Nguyen, Claudio Di Sipio, and Davide Di Ruscio. 2025. Detecting Malicious Source Code in PyPI Packages with LLMs: Does RAG Come in Handy? *CoRR* abs/2504.13769 (2025). arXiv:2504.13769 doi:10.48550/ARXIV.2504.13769

[24] Java. 2025. Java API Specification. https://docs.oracle.com/javase/8/docs/api/overview-summary.html. Accessed: 2026-01-30.

[25] Joern. 2025. Code Property Graph Specification. https://cpg.joern.io/. Accessed: 2026-01-30.

[26] Joern. 2025. Joern: Open-source code analysis platform based on code property graphs. https://github.com/joernio/joern. Accessed: 2026-01-30.

[27] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2023. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In *2023 IEEE Symposium on Security and Privacy (SP)*. 1509–1526. doi:10.1109/SP46215.2023.10179304

[28] Piergiorgio Ladisa, Serena Elisa Ponta, Nicola Ronzoni, Matias Martinez, and Olivier Barais. 2023. On the Feasibility of Cross-Language Detection of Malicious Packages in npm and PyPI. In *Annual Computer Security Applications Conference, ACSAC 2023, Austin, TX, USA, December 4-8, 2023*. ACM, 71–82. doi:10.1145/3627106.3627138

[29] Haoyang Li, Huan Gao, Zhiyuan Zhao, Zhiyu Lin, Junyu Gao, and Xuelong Li. 2025. LLMs Caught in the Crossfire: Malware Requests and Jailbreak Challenges. *CoRR* abs/2506.10022 (2025). arXiv:2506.10022 doi:10.48550/ARXIV.2506.10022

[30] Ningke Li, Shenao Wang, Mingxi Feng, Kailong Wang, Meizhen Wang, and Haoyu Wang. 2023. MalWuKong: Towards Fast, Accurate, and Multilingual Detection of Malicious Code Poisoning in OSS Supply Chains. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 1993–2005. doi:10.1109/ASE56229.2023.00073

[31] Wentao Liang, Xiang Ling, Jingzheng Wu, Tianyue Luo, and Yanjun Wu. 2023. A Needle is an Outlier in a Haystack: Hunting Malicious PyPI Packages with Code Clustering. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 307–318. doi:10.1109/ASE56229.2023.00085

[32] Yue Liu, Chakkrit Tantithamthavorn, and Li Li. 2025. Protect Your Secrets: Understanding and Measuring Data Exposure in VSCode Extensions. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 551–562. doi:10.1109/SANER64311.2025.00058

[33] lyvd. 2022. bandit4mal. https://github.com/lyvd/bandit4mal. Accessed: 2026-01-30.

[34] Md Rayhanul Masud and Michalis Faloutsos. 2024. Unveiling A Hidden Risk: Exposing Educational but Malicious Repositories in GitHub. *CoRR* abs/2403.04419 (2024). arXiv:2403.04419 doi:10.48550/ARXIV.2403.04419

[35] Sk. Tanzir Mehedi, Chadni Islam, Gowri Sankar Ramachandran, and Raja Jurdak. 2025. DySec: A Machine Learning-based Dynamic Analysis for Detecting Malicious Packages in PyPI Ecosystem. *CoRR* abs/2503.00324 (2025). arXiv:2503.00324 doi:10.48550/ARXIV.2503.00324

[36] Microsoft. 2024. ApplicationInspector. https://github.com/microsoft/ApplicationInspector. Accessed: 2026-01-30.

[37] Microsoft. 2024. OSSGadget. https://github.com/microsoft/OSSGadget. Accessed: 2026-01-30.

[38] Shradha Neupane, Grant Holmes, Elizabeth Wyss, Drew Davidson, and Lorenzo De Carli. 2023. Beyond Typosquatting: An In-depth Look at Package Confusion. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 3439–3456. https://www.usenix.org/conference/usenixsecurity23/presentation/neupane

[39] Node.js. 2025. Node.js Documentation. https://nodejs.org/docs/latest/api/. Accessed: 2026-01-30.

[40] NPM. 2025. NPM: The NodeJS Package Manager. https://www.npmjs.com/ Accessed: 2026-01-30.

[41] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24-26, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12223)*, Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves (Eds.). Springer, 23–43. doi:10.1007/978-3-030-52683-2_2

[42] Marc Ohm and Charlene Stuke. 2023. SoK: Practical Detection of Software Supply Chain Attacks. In *Proceedings of the 18th International Conference on Availability, Reliability and Security* (Benevento, Italy) *(ARES '23)*. Association for Computing Machinery, New York, NY, USA, Article 33, 11 pages. doi:10.1145/3600160.3600162

[43] PHP. 2025. PHP Function Reference. https://www.php.net/manual/en/funcref.php. Accessed: 2026-01-30.

[44] Python. 2025. Python Module Index. https://docs.python.org/3/py-modindex.html. Accessed: 2026-01-30.
[45] Md Omar Faruk Rokon, Risul Islam, Ahmad Darki, Evangelos E. Papalexakis, and Michalis Faloutsos. 2020. SourceFinder: Finding Malware Source-Code from Publicly Available Repositories in GitHub. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 149–163. https://www.usenix.org/conference/raid2020/presentation/omar
[46] Md Omar Faruk Rokon, Pei Yan, Risul Islam, and Michalis Faloutsos. 2021. Repo2Vec: A Comprehensive Embedding Approach for Determining Repository Similarity. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, Luxembourg, September 27 - October 1, 2021*. IEEE, 355–365. doi:10.1109/ICSME52107.2021.00038
[47] Haya Samaana, Diego Elias Costa, Emad Shihab, and Ahmad Abdellatif. 2025. A Machine Learning-Based Approach For Detecting Malicious PyPI Packages. In *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing, SAC 2025, Catania International Airport, Catania, Italy, 31 March 2025 - 4 April 2025*, Jiman Hong, Sebastiano Battiato, Christian Esposito, Juw Won Park, and Adam Przybylek (Eds.). ACM, 1617–1626. doi:10.1145/3672608.3707756
[48] Simone Scalco, Ranindya Paramitha, Duc-Ly Vu, and Fabio Massacci. 2022. On the feasibility of detecting injections in malicious npm packages. In *ARES 2022: The 17th International Conference on Availability, Reliability and Security, Vienna,Austria, August 23 - 26, 2022*. ACM, 115:1–115:8. doi:10.1145/3538969.3543815
[49] David Schmotz, Sahar Abdelnabi, and Maksym Andriushchenko. 2025. Agent Skills Enable a New Class of Realistic and Trivially Simple Prompt Injections. *CoRR* abs/2510.26328 (2025). arXiv:2510.26328 doi:10.48550/ARXIV.2510.26328
[50] Adriana Sejfia and Max Schäfer. 2022. Practical Automated Detection of Malicious npm Packages. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1681–1692. doi:10.1145/3510003.3510104
[51] Semgrep. 2025. Semgrep Registry. https://semgrep.dev/r. Accessed: 2026-01-30.
[52] Robert Simmons. 2025. Threat Actor Banana Squad Exploits GitHub Repos in New Campaign. https://www.reversinglabs.com/blog/threat-actor-banana-squad-exploits-github-repos-in-new-campaign Accessed: 2026-01-30.
[53] Socket Inc. [n. d.]. Socket: Secure your supply chain with zero trust open source. https://socket.dev/. Accessed: 2026-01-30.
[54] Sonatype. 2024. 2024 State of the Software Supply Chain. https://www.sonatype.com/state-of-the-software-supply-chain/Introduction Accessed: 2026-01-30.
[55] Xiaobing Sun, Xingan Gao, Sicong Cao, Lili Bo, Xiaoxue Wu, and Kaifeng Huang. 2024. 1+1>2: Integrating Deep Code Behaviors with Metadata Features for Malicious PyPI Package Detection. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 1159–1170. doi:10.1145/3691620.3695493
[56] Apiiro Team. 2025. Malicious Code Campaign: GitHub Repo Confusion Attack. https://apiiro.com/blog/malicious-code-campaign-github-repo-confusion-attack/ Accessed: 2026-01-30.
[57] FOSSA Team. 2022. Understanding and Preventing Dependency Confusion Attacks. https://fossa.com/blog/dependency-confusion-understanding-preventing-attacks/ Accessed: 2026-01-30.
[58] SafeDep Team. 2026. Agent Skills Threat Model. https://safedep.io/agent-skills-threat-model/. Accessed: 2026-01-30.
[59] Sonatype Security Research Team. 2025. Open Source Malware Index Q1 2025: Data exfil threats rising sharply. https://www.sonatype.com/blog/open-source-malware-index-q1-2025 Accessed: 2026-01-30.
[60] Michal Tereszkowski-Kaminski, Santanu Kumar Dash, and Guillermo Suarez-Tangil. 2024. A Study of Malicious Source Code Reuse Among GitHub, StackOverflow and Underground Forums. In *Computer Security - ESORICS 2024 - 29th European Symposium on Research in Computer Security, Bydgoszcz, Poland, September 16-20, 2024, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 14984)*, Joaquín García-Alfaro, Rafal Kozik, Michal Choras, and Sokratis K. Katsikas (Eds.). Springer, 45–66. doi:10.1007/978-3-031-70896-1_3
[61] Duc-Ly Vu, Zachary Newman, and John Speed Meyers. 2023. Bad Snakes: Understanding and Improving Python Package Index Malware Scanning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 499–511. doi:10.1109/ICSE48619.2023.00052
[62] Jian Wang, Zhen Li, Jixiang Qu, Deqing Zou, Shouhuai Xu, Ziteng Xu, Zhenwei Wang, and Hai Jin. 2025. MalPacDetector: An LLM-Based Malicious NPM Package Detector. *IEEE Trans. Inf. Forensics Secur.* 20 (2025), 6279–6291. doi:10.1109/TIFS.2025.3580336
[63] Yayi Wang, Shenao Wang, Jian Zhao, Shaosen Shi, Ting Li, Yan Cheng, Lizhong Bian, Kan Yu, Yanjie Zhao, and Haoyu Wang. 2026. YASA: Scalable Multi-Language Taint Analysis on the Unified AST at Ant Group. *CoRR* abs/2601.17390 (2026). arXiv:2601.17390 doi:10.48550/ARXIV.2601.17390
[64] Dominik Wermke, Jan H. Klemmer, Noah Wöhler, Juliane Schmüser, Harshini Sri Ramulu, Yasemin Acar, and Sascha Fahl. 2023. "Always Contribute Back": A Qualitative Study on Security Challenges of the Open Source Supply Chain. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 1545–1560. doi:10.1109/SP46215.2023.10179378

[65] Dominik Wermke, Jan H. Klemmer, Noah Wöhler, Juliane Schmüser, Harshini Sri Ramulu, Yasemin Acar, and Sascha Fahl. 2023. "Always Contribute Back": A Qualitative Study on Security Challenges of the Open Source Supply Chain. In *2023 IEEE Symposium on Security and Privacy (SP)*. 1545–1560. doi:10.1109/SP46215.2023.10179378

[66] Wikipedia. 2025. XZ Utils backdoor. https://en.wikipedia.org/wiki/XZ_Utils_backdoor Accessed: 2025-07-12.

[67] Di Xue, Gang Zhao, Zhongqi Fan, Wei Li, Yahong Xu, Zhen Liu, Yin Liu, and Zhongliang Yuan. 2024. Poster: An Exploration of Large Language Models in Malicious Source Code Detection. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie (Eds.). ACM, 4940–4942. doi:10.1145/3658644.3691374

[68] Zeliang Yu, Ming Wen, Xiaochen Guo, and Hai Jin. 2024. Maltracker: A Fine-Grained NPM Malware Tracker Copiloted by LLM-Enhanced Dataset. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 1759–1771. doi:10.1145/3650212.3680397

[69] Nusrat Zahan, Philipp Burckhardt, Mikola Lysenko, Feross Aboukhadijeh, and Laurie A. Williams. 2025. Leveraging Large Language Models to Detect NPM Malicious Packages. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 2625–2637. doi:10.1109/ICSE55347.2025.00146

[70] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. 2022. What are weak links in the npm supply chain?. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) *(ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 331–340. doi:10.1145/3510457.3513044

[71] Muhammad Umar Zeshan, Phuong T. Nguyen, and Davide Di Ruscio. 2025. Are They a Silver Bullet? On the Ability of LLMs to Detect Malicious Code in PyPI Packages. *SSRN* (2025). https://ssrn.com/abstract=5220789 Accessed: 2025-07-12.

[72] Junan Zhang, Kaifeng Huang, Yiheng Huang, Bihuan Chen, Ruisi Wang, Chong Wang, and Xin Peng. 2025. Killing Two Birds with One Stone: Malicious Package Detection in NPM and PyPI using a Single Model of Malicious Behavior Sequence. *ACM Trans. Softw. Eng. Methodol.* 34, 4, Article 104 (April 2025), 28 pages. doi:10.1145/3705304

[73] Jian Zhao, Shenao Wang, Yanjie Zhao, Xinyi Hou, Kailong Wang, Peiming Gao, Yuanchao Zhang, Chen Wei, and Haoyu Wang. 2024. Models Are Codes: Towards Measuring Malicious Code Poisoning Attacks on Pre-trained Model Hubs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 2087–2098. doi:10.1145/3691620.3695271

[74] Xinyi Zheng, Chen Wei, Shenao Wang, Yanjie Zhao, Peiming Gao, Yuanchao Zhang, Kailong Wang, and Haoyu Wang. 2024. 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, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 1990–2001. doi:10.1145/3691620.3695262

[75] Ruofan Zhu, Ganhao Chen, Wenbo Shen, Xiaofei Xie, and Rui Chang. 2025. My Model is Malware to You: Transforming AI Models into Malware by Abusing TensorFlow APIs. In *2025 IEEE Symposium on Security and Privacy (SP)*. 486–503. doi:10.1109/SP61157.2025.00012

[76] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 995–1010. https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman