DONAPI: Malicious NPM Packages Detector using Behavior Sequence Knowledge Mapping

Cheng Huang, Nannan Wang, Ziyan Wang, Siqi Sun, Lingzi Li, Junren Chen, Qianchong Zhao, Jiaxuan Han, and Zhen Yang, *Sichuan University;* Lei Shi, *Huawei Technologies*

https://www.usenix.org/conference/usenixsecurity24/presentation/huang-cheng

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

DONAPI: Malicious NPM Packages Detector using Behavior Sequence Knowledge Mapping

Cheng Huang[1], Nannan Wang[1†], Ziyan Wang[1], Siqi Sun[1], Lingzi Li[1],
Junren Chen[1], Qianchong Zhao[1], Jiaxuan Han[1], Zhen Yang[1], and Lei Shi[2]

[1]*Sichuan University*
[2]*Huawei Technologies*

Abstract

With the growing popularity of modularity in software development comes the rise of package managers and language ecosystems. Among them, npm stands out as the most extensive package manager, hosting more than 2 million third-party open-source packages that greatly simplify the process of building code. However, this openness also brings security risks, as evidenced by numerous package poisoning incidents.

In this paper, we synchronize a local package cache containing more than 3.4 million packages in near real-time to give us access to more package code details. Further, we perform manual inspection and API call sequence analysis on packages collected from public datasets and security reports to build a hierarchical classification framework and behavioral knowledge base covering different sensitive behaviors. In addition, we propose the DONAPI, an automatic malicious npm packages detector that combines static and dynamic analysis. It makes preliminary judgments on the degree of maliciousness of packages by code reconstruction techniques and static analysis, extracts dynamic API call sequences to confirm and identify obfuscated content that static analysis can not handle alone, and finally tags malicious software packages based on the constructed behavior knowledge base. To date, we have identified and manually confirmed 325 malicious samples and discovered 2 unusual API calls and 246 API call sequences that have not appeared in known samples.

1 Introduction

JavaScript is increasingly popular as the demand for web applications and web-based applets grows. It has become one of the most widely used programming languages. Many developers use third-party open-source libraries [55, 72] to ensure development efficiency and reduce costs by leveraging existing solutions. Open-source development models help avoid redundant development efforts and lower expenses. Additionally, statistics [32, 80] show that almost every JavaScript package relies on dependencies, and a project (e.g., next.js[1]) may depend on hundreds of third-party packages.

Npm (*node package manager*) is the default package manager for the JavaScript runtime Node.js, consisting of an online repository and a CLI (command-line interface). Since its first introduction in 2009, it has grown in popularity due to its ease of use and substantial package repository, with over 1.3 million packages and 100 billion downloads per month [46], surpassing other package managers such as *Maven* and *pip* as the most extensive package manager. While some companies and organizations maintain their package registries for security reasons, npm remains the most widely recognized and used package registry in the JavaScript community.

As npm gains popularity among developers, it also attracts the attention of attackers. In 2022, an incident [65] occurred where RIAEvangelist, the maintainer of the *node-ipc*[2] package, introduced malicious code into the repository and the affected versions *@10.1.1* and *@10.1.2* contained code that overwrote disk files. Subsequently, the attacker created the *peacenotwar*[3] module, which was included in the affected *node-ipc* version and garnered over 1 million weekly downloads. Another example is the *chalk-next* package, which mimicked a well-known package called *chalk*[4] that modifies string styles in the console. The attacker used the same `README.md` to deceive developers and enable information theft. These and others involving packages like *eslint-scope*[5] and *left-pad*, highlight that poisoning attacks targeting npm are no longer isolated cases but are increasingly common occurrences [58].

In this paper, we propose DONAPI, which takes sensitive application programming interfaces (APIs) and behavior sequences as the primary research object and combines static and dynamic analysis techniques to detect malicious packages and classify their categories automatically. The primary objective of DONAPI is to assist developers in establishing a secure

† Corresponding Author

[1]https://www.npmjs.com/package/next
[2]https://www.npmjs.com/package/node-ipc
[3]https://www.npmjs.com/package/peacenotwar
[4]https://www.npmjs.com/package/chalk
[5]https://www.npmjs.com/package/eslint-scope

dependency base by facilitating fast package review, reducing the need for manual review, and proactively prevent of the usage of malicious packages. As such, the detector focuses on three key aspects: **speed**, **accuracy**, and **comprehensiveness**. To evaluate its effectiveness, we collected a dataset of over 4,000 publicly available malicious packages and performed several analyses, yielding the following findings:

- Batch poisoning: numerous malicious packages often possess similar names while containing identical payloads for executing malicious activities.
- Attack purposes: malicious packages have various objectives, such as importing malware, stealing information, creating reverse shells, and modifying files.
- Entry files: the attack payload of a malicious package is commonly embedded in entry files.
- Code transformation: malicious packages employ code transformation techniques such as obfuscation or compression to evade static detection.

Based on these findings, we primarily focus the analysis of our detector on the entry files. Research [52] has shown that malicious packages often involve extensive network and file operations. Therefore, we perform static analysis to extract API call sequences from the entry file and its dependent files. These extracted sequences serve as features for training classifiers to make an initial assessment of whether a package is malicious or not. Furthermore, the malicious packages that pass the static screening and obfuscated packages that cannot be handled undergo dynamic analysis. This step aims to extract additional API call sequences and convert them into sequences representing sensitive package behavior. Ultimately, our detector provide hierarchical classification results of behavior sequences, which is not available in the current work [16, 36, 59]. In summary, the main contributions of our work are as follows:

- *Behavior Knowledge Base*: a *hierarchical classification framework* [30] based on 806 sensitive API calls and 44 behavior sequences that can automatically map the five most common categories for malicious packages.
- *Detector* DONAPI: an automated malicious package detector, consisting of six primary modules, mixes code analysis, machine learning, and natural language processing techniques to directly map the final malicious category for each detected package.
- *Effective Results*: we build a local npm package cache of over 3.4 million packages, capable of synchronizing official replicate in near real-time and retaining deleted malicious packages. DONAPI found 325 new malicious packages with manual check, discovered 2 unusual API calls and 246 API call sequences that have not appeared in previous malicious samples.

2 Background

From an attacker's perspective, software supply chain attacks consists of three essential steps [59]: (1) publish a malicious package; (2) get users to install it; (3) run the malicious code. The first two steps are intricately connected, with the success of the second step greatly influenced by the approach employed in the first step. Moreover, these initial two steps form the foundation of a software supply chain attack. Consequently, we can further categorize these three steps into two distinct phases: preparation and execution.

2.1 Preparation

While the preparation phase is not the primary focus of our study, we will provide an overview here. One of the simplest methods in this phase is to release a new package. However, attackers often encounter obstacles at this step because of incomplete or suspicious information, which makes users cautious. In this context, attackers employ several methods.

Typosquatting [48, 67, 70]. Attackers employ names that closely resemble popular packages when registering names for malicious packages. This deceptive tactic leads some users to download these malicious packages when they make spelling mistakes unintentionally. One notable example is the malicious package *crossenv* [64] mentioned in a Synk blog, which imitates the popular package *cross-env*[6]. Despite having similar functionality, the malicious version also includes the unauthorized collection of user information.

Dependency Confusion [28]. This technique, known as dependency repository hijacking, often employs strategies similar to *Namespacing* [5]. It involves uploading a package with the same name as certain packages in a private registry but with a higher version number to the public npm registry. As a result, when users synchronize their packages, they unknowingly download the malicious package from a public source. For instance, during one test attack, Birsan [2] discovered the presence of a package called *auth-PayPal* being used for PayPal development, even though it does not exist in the official npm registry at that time.

On the other hand, a more challenging approach for an attacker is to contribute directly as a maintainer of a legitimate project. In this case, the attacker would then have to gain the privileges needed to modify the code base of the legitimate project by means that typically include using Social Engineering (SE) techniques on legitimate project maintainers [23], taking over legitimate accounts (e.g., reusing compromised credentials [14] and account privilege transfers [25]), by compromising the maintainer system (e.g., exploiting vulnerabilities [24]), and also by exploiting vulnerable points in code maintenance (e.g., outdated domain names of maintainers or lack of package maintenance [26, 79]). Finally, they distribute malicious packages that cause users to introduce

[6] https://www.npmjs.com/package/cross-env

malicious code (both explicit and implicit) when downloading or updating packages.

2.2 Execution

Before the user can execute it, a package must go through two processes, installation and import, which is the stage where the Open Source Security Foundation (OpenSSF)[7] has found frequent security problems [8]. A package in npm consists of the `package.json`[8] file and various code files that will play a significant role in the above process.

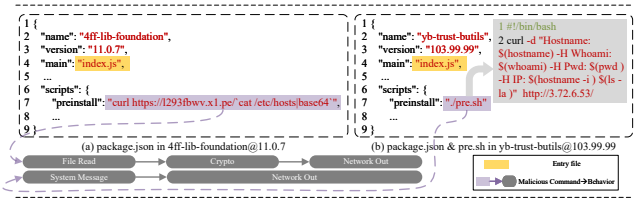


Figure 1: The malicious commands during installation

Installation. The `package.json` file involved in the installation process includes a field called `scripts`[9] that provides `pre` and `post` hooks for tasks such as preparation and cleanup. While most of these commands are under the user's control, specific fields, such as `preinstall`, pose a significant security risk [77]. As the name suggests, the commands defined under this field are executed automatically before the users install the package, allowing attackers to launch malicious attacks. As shown in Figure 1, the package *4ff-lib-foundation@11.0.7* uses a combination of commands in the `preinstall` field to encode the contents of the local file `/etc/host` and send it to the external address; the package *yb-trust-butils@103.99.99* calls the `pre.sh` file in the `preinstall` and sends the sensitive information to the external address with the `curl` command.

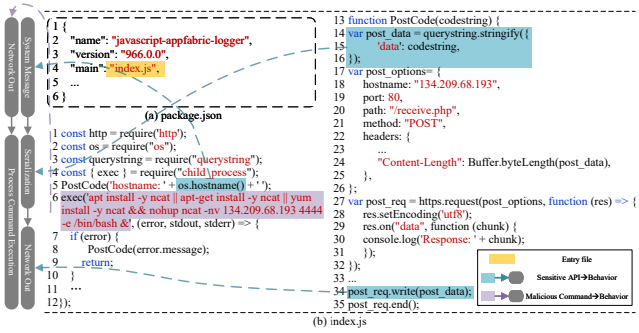


Figure 2: The malicious code in **javascript-appfabric-logger@966.0.0** detected by DONAPI on June 5, 2023.

Import. In Node.js, when an object in one code file needs to be accessed by another, it is often necessary to specify it as an exported object. However, some packages import without providing specific export content still have some side effects, such as setting global configurations or executing initialization code. In addition, when the user imports a new module, it automatically executes the file in the `main` (or `exports`) field. Therefore, these files is important to consider when detecting malicious packages. As depicted in Figure 2, if another package imports the package *javascript-appfabric-logger@966.0.0*, it will call multiple APIs to obtain and send sensitive information to an external address.

Code transformation techniques. Code transformation techniques, called code obfuscation in our research scenario, can convert a source program into a target program that is difficult to detect for static analysis without losing its behavior or functionality [16]. Thus, attackers often use these techniques, including renaming, dead code insertion, control flow flattening, and encoding obfuscation [57], to mask their malicious intent or increase the analysis complexity. However, it is essential to note that these techniques are not limited to attackers alone. Many legitimate developers also use code transformation techniques to reduce code size or protect their privacy and intellectual property [45]. Therefore, obfuscated code alone is not a definitive indicator of malicious intent.

3 Methods

In this section, we present the entire processing flow of the detector. The primary labeling technique of the proposed detector is hierarchical classification using behavior sequences derived from API call sequences of the packages. As shown in the Figure 3, diagrams 1-6 represent different modules, which we will describe in detail in the subsequent sections.

3.1 Code Dependencies Reconstructor

To better capture the API call sequences, we designed a code dependency reconstructor for npm packages that simulates the code execution during the processes of installation and import of the packages by utilizing the Abstract Syntax Trees (ASTs) technique, extracting and merging all the code involved into a single `.js` file, and rename its parameters, functions, and classes. The process is shown in the Algorithm 1, and the basic steps and key points are described in detail below.

Entry files extraction. As mentioned in section 2.2, the installation and import process is crucial for detecting malicious packages, so we focus on the entrances involved in the above process, including `scripts`, `main`, `exports`, `imports`, `bin`. Within the `scripts` field, apart from the previously mentioned hooks (i.e., `preinstall` and `postinstall`), developers can define custom fields executed via the `npm run <field>` command. We use regular expressions to extract filenames specified under the script field to capture these fields. The `main` (or `exports`) field, which serves as the default entry point, automatically executes the appropriate files during

[7] https://openssf.org/
[8] https://docs.npmjs.com/cli/v9/configuring-npm/package-json
[9] https://docs.npmjs.com/cli/v9/using-npm/scripts

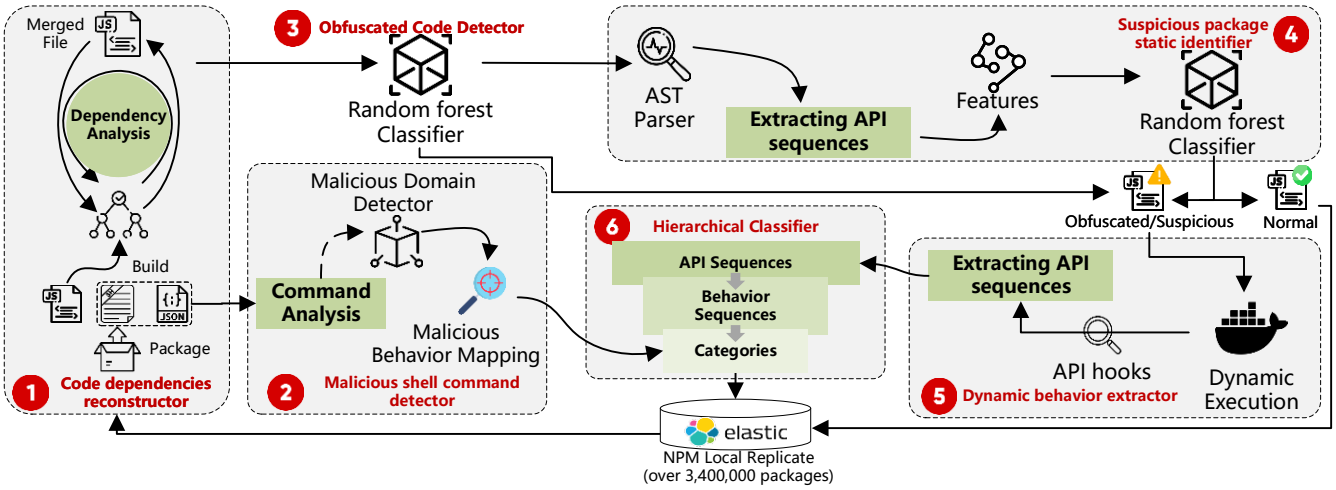


Figure 3: The overall framework of DONAPI

package import. The `imports` field defines the imported subpaths for the current package. `bin` field creates some CLIs that, in turn, execute specific files. We extract them from the configuration file whenever present, regardless of whether the package executes the specified file upon import.

Dependencies parsing. The purpose of dependencies parsing is to resolve imports of other code files declared dynamically in the code and retrieve the content of the imported target files. In Node.js, different module systems have different import methods, which are `require()` function for *CommonJS*[10] and `import` statement for *ECMAScript*[11] modules. Thus, we design two AST node parsing rule sets to adapt to these module specifications. Although a package may contain two module systems, typically, only one module system is used per file, so we generated ASTs in file units. In the AST, each code block (such as function declaration and class declaration) and code lines not contained in code blocks are defined as top-level nodes. The dependencies parsing process traverses the top-level nodes, matches all nodes, including import action, parses them recursively, and inserts the returned AST into the original positions.

Objects modifying. To solve the problem of variable ambiguity after merging different codes, we need to unify the identifiers of imported and exported objects based on dependency resolution. Since the dependency resolution process will merge code from different files, and the merged code does not need to define any exported objects, we first need to modify the exported objects into general object declarations and unify their names as `export_<file_name>_<object_name>`, where `file_name` does not contain extensions and special characters are replaced by "$" to avoid duplication of variable names. On the other hand, the import statement usually assigns the imported external object to a new identifier to call the external function in the subsequent code, so each identifier member corresponds to an external object during the actual code execution. Based on this principle, we rewrite the value assigned to the new identifier as a new object that includes all exported objects defined as its members in the target file.

Algorithm 1: Code Dependencies Reconstructor

Input: Package (package.json, code files)
Output: Merged codes
Hyperparameters : recursion = 2

```
// Parsing package.json to get 5 entry points
entries = ExtractEntries (package.json);
foreach entry E_i in entries do
    // Determine the type of file module specification
    moduleType = GetModuleType (E_i, package.json);
    // Generating abstract syntax trees using parsers
    AST_i = GenerateAST (E_i, moduleType);
    foreach top-level node N_j in AST_i do
        if N_j is require/import statement then
            // Get dependencies and download
            dependencies = GetDependency (N_j);
            // Recursive resolving of dependencies
            AST_sub=ResolveDep (dependencies, recursion);
            MergeAST (AST_i, AST_sub)
        end
        else if N_j is exports/export statement then
            // Modify export statement
            variableName = variable name of N_j;
            fileName = file name of E_i;
            remove export keyword in N_j;
            variableName → export_fileName_variableName;
        end
    end
    // Convert the modified AST back to JS code
    restoreE_i = RestoreCodeFromAST (AST_i);
    allEntry.append(restoreE_i);
end
return allEntry;
```

[10] https://wiki.commonjs.org/wiki/CommonJS
[11] https://www.ecma-international.org/publications-and-standards/standards/ecma-262/

3.2 Malicious Shell Command Detector

In supply chain poisoning attacks, in addition to directly implanting malicious code, executing the attack through malicious shell commands is a typical tactic hackers use. These commands may exist in various places such as `package.json` files, `.sh` files, or code. Therefore, we propose a malicious shell command detector to address this issue.

Basic detection process. For the `package.json` file, our detector parses and extracts all `scripts` field values as bash commands and parses them to build the corresponding ASTs using *bashlex*[12] library. For `.sh` files, the detector executes the file within a Docker sandbox in debug mode using the command */bin/sh -x*, capturing the executed command sequence. In the case of code, we focus on the parameters of standard API calls used for command execution (including *child_process.spawn, child_process.spawnSync, child_process.exec, and child_process.execSync*). In a similar process to `package.json` file, these parameters are treated as bash commands and processed using the same methodology mentioned above. Then, the YARA rules shown in Appendix A.2 are used to analyze the information extracted from these bash commands and judge whether they are malicious.

Table 1: Malicious URL features

URL Fea.	Description
UF1	Longest subdomain entropy
UF2	Length of the longest subdomain
UF3	% of vowel letters in the longest subdomain
UF4	% of consonant letters in the longest subdomain name
UF5	% of consecutive letters in the longest subdomain
UF6	% of repeated letters in the longest subdomain
UF7	% of numeric characters in the longest subdomain
UF8	Modify file permissions and create processes
UF9	gibberish test to determine the readability of the longest subdomain
UF10	Top-level domain types, including: xyz, br, us, etc.

Malicious URL detection process. In practical scenarios, specific commands rely solely on accessing URLs to fulfill their intended functions, which means that the degree of maliciousness of the URLs will determine whether the command is malicious. With the attacker's ability improvement [6], the similarity between malicious and benign URLs among text features has dramatically increased, challenging the static detection methods of malicious URLs based on text features. At present, a feasible way is to access the target URL in a sandbox environment, capture the traffic during the communication, and analyze the maliciousness of the URL on this basis. However, this method is time-consuming because the URL needs to be accessed, which makes it challenging to meet the detection efficiency requirement of DONAPI. Therefore, we consider a compromise: allowlist combined with machine learning. Specifically, first, we build a domain allowlist based on Alexa Top 1M and use it to filter the target URLs. Next, if the URL fails to hit any items in the allowlist, a machine learning model is introduced to detect it to balance accuracy and efficiency. Regarding machine learning models, referring to existing works [10, 38], we use features shown in Table 1.

After completing the above detection process and obtaining the results, we record all malicious commands and forward the packages without malicious commands to the downstream tasks for further analysis.

[12] https://github.com/idank/bashlex

3.3 Obfuscated Code Detector

As described in Section 2.2, code obfuscation affects the effectiveness of static analysis, so we designed a machine-learning-based model specifically to identify packages with obfuscated code during installation and import. Ultimately, we send obfuscated packages to the *dynamic behavior extractor* and unobfuscated to the *suspicious package static identifiers*.

Table 2: Obfuscation features. "*" in "Ref." indicates that the feature was partially modified in our research

Obfuscation Fea.	Description	Ref.
OF1	The ratio of compressed lines of code to original lines of code	new
OF2	The ratio of the number of spaces in the compressed code to the number of spaces in the original code	new
OF3	# of string function calls, e.g. "subString", "charAt"	[1]
OF4	# of encoding function calls, such as "escape", "String"	new
OF5	# of occurrences of special characters, such as "%", "$", "\"	[69]*
OF6	# of lines of code	[42]
OF7	# of white spaces	[42]
OF8	# of special numbers, such as hex and unicode encoding	[42]
OF9	Ave. length of identifiers	new
OF10	Shannon entropy of identifiers	[35]*
OF11	Max. string length	[29]
OF12	# of strings over a certain length	new
OF13	# of prototype method calls	new
OF14-OF25	Frequency of the keyword, such as 'if', 'else'	[69]*

Numerous recent advancements [20, 29] have been in obfuscated code detection, but we remain committed to building feature-based detectors due to the two distinct advantages. First, previous research work [1, 29, 35, 42, 69] has extensively used and effectively validated the effectiveness of features in detecting obfuscated code. Second, the feature extraction process can be executed quickly, thus improving overall efficiency. Therefore, based on the study of commonly used obfuscation tools and methods [33, 57], we have carefully distilled a set of 25 features based on keywords and code structure and evaluated their importance, as shown in Appendix B. Table 2 details this comprehensive set of features.

For the model, we chose the Random Forest (RF) model based on several primary reasons. First, RF excels at handling high-dimensional data with many features. Second, in real-world applications with often missing data, RF is just as good at managing missing data while maintaining model accuracy, which may be difficult for other algorithms to produce. Third, RF can help identify the most influential features in distinguishing obfuscated data from unobfuscated data, thereby revealing the underlying mechanisms of code obfuscation. Furthermore, researchers [31, 39] have generally favored RF

algorithms for detection tasks over other traditional machine learning algorithms, such as Support Vector Machines (SVM), Decision Trees (DT), and Naïve Bayes (NB).

3.4 Suspicious Package Static Identifier

The main objective of the static detection module is to perform a robust and efficient maliciousness analysis of packages detected by the previous module as unobfuscated files (involved in the process of installation and import). However, considering that static analysis has certain limitations (e.g., poor environment awareness and low precision [7]) compared to dynamic analysis and the combination of them can perform better [34, 61], another goal of this module is to perform preliminary maliciousness assessments on a large number of unobfuscated packages and screen out suspicious samples to be sent to the dynamic module for further examination, to reduce the burden of dynamic analysis and speed up the overall process. The module includes the following two main functional components:

API call sequence extraction. In addition to referring to existing works [16, 59], we manually analyze many actual samples and ultimately focus on four different behaviors during script code execution. 1) *Network requests* allow communication over various protocols, such as sockets, HTTP, FTP, etc. They are often used to leak sensitive information [4], obtain malicious payloads [12], etc. 2) *File system accesses* allow file operations such as read, modify, chmod, etc. They have been used to compromise SSH private keys [4], overwrite files [65], etc. 3) *Process operations* allow process creation, termination, and privilege changes. They have been used to spawn individual malicious processes [60]. 4) *Arbitrary code execution* allows code generation and execution. The infamous `eval` implements almost all possible functions.

To capture these behaviors as much as possible, we use Node.js native APIs that implement the four behaviors described above in the underlying layers of Node.js. Additionally, even if executed through dependencies or wrapper functions, the code reconstructor can merge code into a single file for static analysis, meaning that this approach does not miss sensitive behaviors. Drawing parallels to taint analysis [34, 73], the foundation of our static analysis relies on API call sequences, which aims to ascertain if there is a data flow from source to sink. However, due to efficiency constraints inherent in taint analysis, we opt to utilize solely the location information of API calls to establish the presence of data flow.

We adhere to the "**AST first, regex later**" methodology to extract sequences. AST is a potent tool for syntax analysis and identifying pivotal system calls. However, it is essential to acknowledge that the diverse nature of JavaScript syntax and the inherent limitations of parsing tools may impede successful parsing across the board. When conventional parsing encounters obstacles, we resort to regex matching as a viable alternative. While regex exhibits more constraints than AST, it proves satisfactory in this context, especially considering its role as a fallback option.

Feature engineering. Feature selection plays a key role, as we initially chose the RF model to identify malicious packages. Striking the right balance is essential: if the feature selection is overly strict, it may lead to a substantial number of missed detections, thereby impacting the effectiveness of subsequent analysis; conversely, if the feature selection is too lenient, it can result in a high number of false positives, significantly impeding the overall efficiency of the DONAPI.

Table 3: Behavior features

Behavior Fea.	Description
BF1	Send sensitive information to the outside
BF2	Query system environment variables
BF3	Download the content and execute it as a string
BF4	Write to file and execute
BF5	Read the contents of the file and execute
BF6	Read files and execute code dynamically
BF7	Download content and execute code dynamically
BF8	Modify file permissions and create processes
BF9	Identify operating system platforms
BF10	Modify the data flow of system command execution results
BF11	Execute system commands
BF12	Number of Performing sensitive file operations

Through exploration, we found that identifying malicious behaviors only by individual suspicious behaviors would generate high false positives and decrease the overall analysis efficiency. So, we chose to detect potentially malicious behaviors during the installation phase of software packages through a combination of behaviors as selected features. In addition, existing studies have found significant differences between benign and malicious packages in performing critical behaviors, mainly including a range of native APIs performing essential functions such as file system access, network requests, process manipulation, and arbitrary code execution. For example, when stealing sensitive information, the code execution behavior manifests as a network request and attempts to access sensitive information before that step. In addition, we are interested in modules such as the *fs* module, the *https* module, and the *child_process* module. Ultimately, we summarized 12 features in Table 3.

3.5 Dynamic Behavior Extractor

As described in Sections 3.3 and 3.4, dynamic analysis helps to overcome the challenges encountered in static analysis, especially in encryption, obfuscation and compression in JavaScript code. Therefore, this module aims to perform runtime API monitoring with API call sequence extraction for the obfuscated and suspicious packages in the previous sections.

Monitoring methods. Most of the existing research [15, 56] on capturing API call sequences uses system call tracing

tools (e.g., Strace[13], Sysdig[14]), which are simple and effective, but lack interpretability. Based on the open-source nature of Node.js, we use API instrumentation that avoids capturing non-package behaviors while having broader API coverage (especially APIs that do not result in system calls). Specifically, we insert additional functional JavaScript code into key API implementations to capture API calls and explicit parameters generated by packages during installation and import. Moreover, we can locate the APIs called by the package to specific code snippets based on call stack information, thus allowing for an intuitive understanding of the package logic at the code level and improving interpretability. For the deployment, we engineered a Docker image that recompiles and integrates modified Node.js source code atop the foundational Ubuntu image. Then, we generate separate Docker containers for each package and perform the processes of installation and import within them, closely mimicking the behavior of actual packages and capturing API call sequences precisely.

Monitoring scope. We monitored 132 APIs, including native implementations such as file manipulation, network connection, and process creation. Our analysis suggests these APIs cover all potential API calls that facilitate malicious operations. However, due to the low-level nature of these APIs, package installation will result in some non-user *side effects*. For example, the *https.request* is implicitly called to download package dependencies and file manipulation APIs are used to log the installation process. To address this issue, we have filtered out these extraneous behaviors by analyzing the function call stack of the API and determining the location of parent function calls and function parameters. Our strategy monitors malicious behaviors at the package installation and import stages. Our research indicates that over 90% of malicious packages activate during these stages (rather than during exported function invocation), so we only considered the execution of code involved in these phases. In our future work, we aim to continually iterate the functional code of API instrumentation and customize the API return values for different scenarios. This strategy will enable us to emulate a range of execution conditions and external dependencies, thereby improving code coverage.

3.6 Hierarchical Classifier

API calls are critical for packages to achieve specific behaviors (benign or malicious) and are the basis of many existing malware classification studies [3, 9, 37]. However, considering that the implementation of package behaviors is not only related to some specific APIs but also greatly depends on the call order of these APIs, e.g., the execution of a malicious software import, which first requires a connection to an external network to download the software, and subsequently to execute the corresponding commands, which can lead to a large number of false positives if the order is not taken into account as can be demonstrated in Appendix A.1. Therefore, to enhance the understanding of malicious package behaviors and complement the insufficiency of using API call sets alone, we chose to take the API calls order into account, and we propose a *hierarchical classification framework* centered on the API call sequences, as shown in Figure 4.

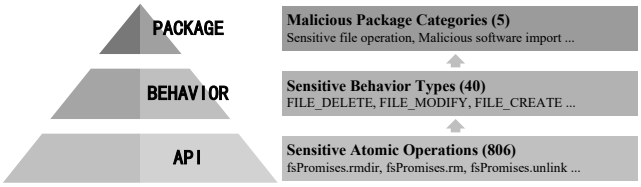


Figure 4: Hierarchical classification framework of malicious packages

First, we map dynamically extracted APIs to specific behaviors and subsequently classify packages based on the forward and backward order of these behaviors (API call sequence order), using a bottom-up abstraction approach for the whole process. In designing the framework, however, we used a top-down approach to avoid explicitly modeling specific packages based on their characteristics and to be more inclusive. Next, we introduce the three layers of the framework: malicious package categories, sensitive behaviors, and sensitive APIs. For more detailed information, please visit the page [30].

Malicious packages. After studying the existing reports of npm malicious packages [4, 53], we found that most malicious npm packages steal user information by various means, and a few other types exist [12]. Furthermore, we combed through some of the studies [44, 54] on malware classification, manually analyzed the collected malicious samples, and now classified the malicious packages into five categories. The categories are Sensitive information theft (**M1**), Sensitive file operation (**M2**), Malicious software import (**M3**), Reverse shell (**M4**), Suspicious command execution (**M5**). These categories represent different attack purposes despite variations in their specific implementations.

Sensitive behaviors. Malicious packages often use different behaviors and sequences to achieve different purposes. Therefore, we associate the category of malicious packages with the corresponding sequence of behaviors. For example, the sensitive information theft category follows the behavior sequence: "Access sensitive information first and then send it over the network." Therefore, based on the following criteria, we define 12 behavior types (40 behavior subtypes in total).

- **Mutual exclusion** (types between atomic behaviors should not overlap)
- **Completeness** (covering all possible sensitive behaviors)
- **Non-ambiguous** (type division is clear)

[13] https://github.com/strace/strace
[14] https://github.com/draios/sysdig

- **Repeatability** (multiple classification results for an atomic behaviors are consistent)
- **Acceptability** (logical and intuitive)
- **Practical** (can be used for in-depth research)

Sensitive APIs. APIs are fundamental to software functionality and are not inherently malicious. However, depending on the user's different behavioral intentions, certain APIs may contribute to malicious behavior to varying degrees. Therefore, we supplemented the APIs gathered from existing research [16] with the native API functionality descriptions from the official Node.js documentation[15]. Ultimately, we identified 226 APIs (combined with parameters to form 806 sensitive APIs) that could potentially be used for malicious purposes to represent the sensitive behaviors defined above at the code level.

The hierarchical classification framework centered on API call sequences is one of our work's highlights, notable for its generality and language independence, allowing for the development of guidelines for different languages. Of course, for consistency, we have also included malicious shell commands in the relevant malicious package category. This additional aspect, while necessary, is separate from the central focus of this section. Therefore, we will present the rules for categorizing malicious shell commands in the Appendix A.2.

4 Experiments and Results

4.1 Datasets

Our dataset comprises multiple data sources, including security vendor blogs and existing sets of known malicious samples. However, in many cases, these sources only provide information on malicious package names and version numbers without disclosing the specific malicious code itself. Therefore, it poses a challenge when attempting to download packages directly from the official npm registry[16], which may have been unpublished or deleted, resulting in the unavailability of valid information through the official npm registry.

To overcome this challenge, we implemented a solution by building a local package cache using npm replicate[17], which is a CouchDB instance that offers a *Change API*[18] to track database changes. When building the local cache, we initially copied all the metadata from npm to the local and downloaded all the corresponding `.tgz` files. Later, we only need to use change messages to perform local synchronization during the synchronization process. However, unlike npm replicate, we only mark packages as officially deleted and do not delete the associated `.tgz` files. Thus, our dataset includes the raw metadata of the npm registry as well as the `.tgz` files of the removed packages, some of which may contain malicious packages. We summarize the details of the collected dataset in Table 4 for reference.

Table 4: Statistics of the malicious package datasets

Dataset	Source	Num
Redlili	https://red-lili.info/	1,214
Backstabber	https://dasfreak.github.io/Backstabbers-Knife-Collection/	1,504
ReversingLabs	https://blog.reversinglabs.com/blog	39
Maloss	https://github.com/osssanitizer/maloss	332
Cuteboi	https://cuteboi.info/	500
Synk-blog	https://snyk.io/blog/	32
Lofygang	https://gist.github.com/jossef	10
Sonatype-blog	https://blog.sonatype.com/	315
Local cache	-	600+
Total	-	**4,546+**
Total (in used)	-	**1,159**

We found several problems when studying these datasets: 1) Intersection of data exists; 2) The malicious code is the same except for the external address; 3) The malicious behaviors cannot be triggered (not in installation and import). Therefore, in our analysis, we manually reviewed and de-duplicated these samples and constructed a malicious dataset (1,159 in total). Since some modules of Donapi play complementary roles in detecting malicious packages, this part of the data is highly varied, and we will present it in the corresponding evaluation.

4.2 Experiment Design

To evaluate whether our method achieves the three objectives mentioned in the previous sections, we have formulated the following research questions (RQs) to guide the experimental design:

RQ1 Accuracy. How does the proposed method perform on the dataset in terms of detection accuracy? (§4.3)

RQ2 Efficiency. Can the model handle a large number of packages within a limited time? (§4.4)

RQ3 Validity. Can the detector find malicious packages in the wild? How does it compare to other detectors? (§4.5)

4.3 Accuracy Evaluation (RQ1)

The DONAPI comprises multiple modules, each with specific roles, working collaboratively to achieve malicious package detection. To evaluate the detector, we conducted individual evaluations on each module and assessed the overall performance of the entire detector. The evaluation metrics include precision, recall, and F1 score, considering malicious package detection as a binary classification task. The following sections present each module's experimental setup and results. Furthermore, Section 4.3.2 on integrated evaluation comprehensively explains the causes of false alarms and omissions.

4.3.1 Partial Evaluation

Code dependencies reconstructor. We aim to merge the actual running code into a single file while ensuring that the

[15] https://nodejs.org/en/docs
[16] https://registry.npmjs.org/
[17] https://replicate.npmjs.com/
[18] https://replicate.npmjs.com/_changes?descending=true&include_docs=true

syntax is correct, and we are not concerned with whether or not the merged code will run successfully. In addition, we use the AST for reconstruction, and if there are no errors in the process, we consider the output correct. Since the code merging process is file-based, we use the success rate of the output file as an evaluation metric. We tested all newly released npm packages from May 30, 2023, to June 1, 2023, totaling 28,874 packages and 579,269 files. The overall results, as shown in Table 5, demonstrate that the code dependencies reconstructor exhibits excellent usability, achieving an overall error probability of less than 1%.

Table 5: Evaluation results for code dependencies reconstructor. "No." is the n^{th} day of the test period

No.	#Packages	#Files	#Reconstruct error files	#Total error files
#1	11,917	216,615	43	1,367 (0.63%)
#2	10,481	248,901	37	1,773 (0.71%)
#3	6,476	113,753	18	1,003 (0.88%)
Total	28,874	579,269	98	4,143 (0.72%)

[1] The total error consisting of Reconstruct errors and Parsing errors.

Table 6: Evaluation results for sub-detectors and DONAPI

Detector	#Malicious/Obfuscated	#Benign	Prec.	Recall	F1
MSCD	208	92	98.54%	97.12%	97.82%
OCD	88	337	94.25%	93.18%	93.71%
SPSI	147	567	99.32%	100.00%	99.66%
DONAPI (Integral detector)	**1,159**	**3,000**	**98.88%**	**91.63%**	**95.12%**

Sub-detectors. The main modules for this part of the verification are *Malicious shell command detector (MSCD)*, *Obfuscated code detector (OCD)*, and *Suspicious package static identifier (SPSI)*, which will be replaced with name abbreviations later in the paragraph. As these sub-detectors identify different aspects of a package, we prepared distinct datasets for each of these three components. For *MSCD*, we collected 208 malicious and 92 benign commands, covering the five malicious categories mentioned in Section 3.6, for validation purposes. For *OCD* and *SPSI*, we assembled corresponding datasets of obfuscated and malicious packages and split the training and validation set into a 4:1 ratio. Table 6 shows the specific number of packages and evaluation results.

Table 7: Comparative experiments on URL classifiers

Model	Acc.	Recall	F1	Speed
LSTM+CNN+Attention [47]	0.91	0.68	0.79	CPU: 1.465ms/it GPU: 0.041ms/it
Features+RF	0.82	0.58	0.62	CPU: 0.041ms/it
Features+RF+AllowList	0.82	0.58	0.62	CPU: 0.022ms/it

In addition, we evaluated the URL classifiers involved in *MSCD* to demonstrate the effectiveness of our design. As shown in Table 7, in a CPU-only environment, the machine-learning model, despite being slightly less effective, is much faster than the deep-learning model. In addition, allowlist can further improve the detection speed without affecting the model's effectiveness.

Dynamic behavior extractor. The Dynamic behavior extractor complements the static identifier and is critical to the overall detection process. We first tested three days of data with a total of 6,766 packages, and 376 were in error (about 5.6%), and we manually analyzed some error samples. We found that the main reasons for installation failures are code or command errors, such as code syntax errors, no dependency version exists, connection timeouts due to C2 address failures, no installation environment supported, etc. If the dynamic behavior extraction is incomplete for the above reasons, the package cannot successfully attack the victim host despite being malicious. It is worth mentioning that we can still capture network request behavior for packages that fail to install due to connection failures.

Table 8: Evaluation results for hierarchical classifier

Module	Category	Recall
Hierarchical classifier	Sensitive information theft (M1)	93.14%
	Sensitive file operation (M2)	100.00%
	Malicious software import (M3)	82.28%
	Reverse shell (M4)	97.22%
	Suspicious command execution (M5)	68.75%

Hierarchical classifier. To verify the validity of the criteria we defined for our hierarchical classification framework, we used the malware set mentioned in Section 4.1, which covers all five malicious categories we defined. However, it is worth noting that the actual number of categories for packages is slightly more than 1,159 due to the complex behavior of some packages, which results in the possibility of belonging to more than one malicious type, e.g., both sensitive information theft and sensitive file tampering. The specific recall rates for each category are shown in Table 8.

We initially designed Suspicious command execution (M5) to encompass more suspicious commands to avoid underreporting. However, after analyzing the missed samples, we found the recall rate unsatisfactory due to their obfuscated code and inability to obtain the complete behavior sequences through dynamic execution. In future work, we will refine the detector based on the characteristics of such samples.

4.3.2 Integrated Evaluation

In the previous sections, we conducted detailed performance evaluations for each module, but assessing the detector as a whole is also essential. The malicious samples used in the experiments come from the data mentioned in Section 4.1 (1,159 in Total). For the benign samples, we randomly selected and manually validated 3,000 packages.

In the accuracy evaluation of RQ1, we found that the detection could be better, so we analyzed the samples in detail. For underreporting, we discovered that some malicious samples

are time-sensitive, e.g., failed external links cause us to extract API call sequences incompletely. For false positives, we found that the rich behavior of large software packages may satisfy part of our defined API call sequences, which leads to false positives; moreover, the dependencies between packages and between packages and other software lead to the fact that these packages inevitably import external software, but we are unable to determine the degree of maliciousness of the imported software accurately.

4.4 Efficiency Evaluation (RQ2)

Before the efficiency evaluation, we analyzed the update frequency of the npm repository. Figure 5(a) shows the number of package updates in the local npm package cache every 14 days from *December 12, 2022* to *June 24, 2023*. The box plots and marked points represent the distribution of package updates over this period, and we observe that approximately 219,834 packages are released or updated every 14 days, an average of 16,102 package updates per day. Thus, with such many package updates, our detector must operate efficiently without compromising the quality of the analysis.

4.4.1 Timeout Analysis

The dynamic and static API call sequence extraction is a critical detector component and significantly impacts the overall processing time. However, complex package behaviors may lead to excessively long processing times, e.g., large packages may lead to longer static API call sequence extraction times, and network issues may lead to unstable dynamic installation times. Therefore, we employ a timeout mechanism to prevent the detector from taking too long to process specific packages.

To determine suitable timeout values, we conducted tests on a randomly selected subset of packages. For static API call sequence extraction, we tested 50,000 packages and calculated the empirical cumulative distribution function (eCDF) under different time durations, as shown in Figure 5(c). We found that when the detection time was less than 50s, 150s, and 300s, the coverage rate for packages reached 82%, 93%, and 97%, respectively. Based on these results, we set the static processing timeout to 300s to ensure sufficient coverage. However, it is essential to note that most packages require significantly less than 300s for processing.

Similarly, we randomly selected 15,000 packages for dynamic API extraction for testing and analyzed the eCDF under different time durations, as depicted in Figure 5(d). Due to the variations in package functions, there is variability in the duration distribution. Therefore, to achieve comprehensive coverage, a timeout of 600s was set for dynamic execution, as this duration covered nearly 89% of packages. It is worth mentioning that this does not imply that the processing time is inherently long, as not all packages require dynamic detection.

4.4.2 Processing Time

After determining the timeout, we randomly selected actual update packages totaling three days (September 1-3, 2023) for processing duration evaluation. It is worth noting that not all packages go through the dynamic analysis step. Therefore, the experiment tested a total of 15,479 samples, of which 4,571 were processed by the dynamic behavior extractor.

Table 9: Evaluation results for efficiency. Statistics are from September 1-3

Object	Result
Num of detected packages	15,479 (4,571 through dynamic)
Processing time	21 h 48 m 36s
Total lines of all codes	168,610,774 rows
Total lines of reconstruction codes	19,989,837 rows
Num of detected packages in 24 hours (estimated)	$\approx$ 17,033 (> 16,102)

Table 9 lists the test results, showing that the detector could process 15,479 packages in 22 hours. Meanwhile, to prevent the number of update packages from fluctuating over time, we estimated the number of packages (about 17,033) processed by the detector over 24 hours, and the results show that it is similar to the average number of updates (16,102) per day that we have counted before. In addition, given the variation in the number of lines per package, measuring the detection efficiency in terms of code lines is necessary. On this basis, the detection efficiency of the detector is about 1.29 million lines per 10 minutes for all codes and about 152,757 lines per 10 minutes for reconstruction code.

4.5 Validity Evaluation (RQ3)

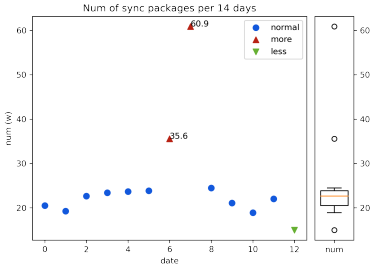
To validate the effectiveness of the detector, we conducted experiments with a larger number of packages and compared its performance with other tools on some packages. This extensive evaluation allows us to understand better the detector's effectiveness in detecting and mitigating malicious packages.

Table 10: Evaluation results for validity

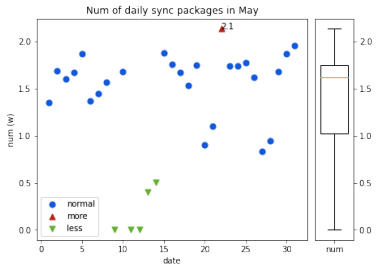
Detector	Term	Total	Det.	Pos. Det.
DONAPI	Jan-May	2,764,022	1,727	325 (+165)
DONAPI	May	420,395	792	148 (+83)
GUARDDOG [27]			49,070	$\approx$ 6 in 1,000
AMALFI [59]			2,678	$\approx$ 22 in 1,000
SAP [36]			50,043	$\approx$ 6 in 1,000

Note Numbers in parentheses are the number of malicious packets detected by the model but not visually analyzed manually due to code obfuscation.

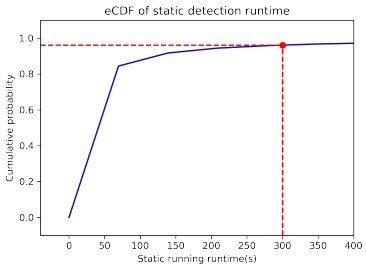
Long term. Starting in 2023, we deployed DONAPI to real-world environments to detect packages updated daily from our local caches. We identified and manually confirmed 325 malicious packages (and tagged with npm or Synk) from January through June. It is worth noting that the results of



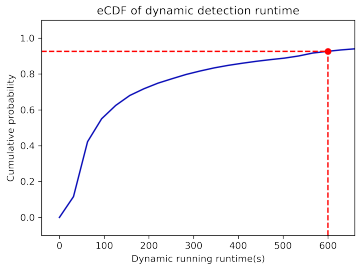
(a) Synchronization of local npm package cache per 14 days



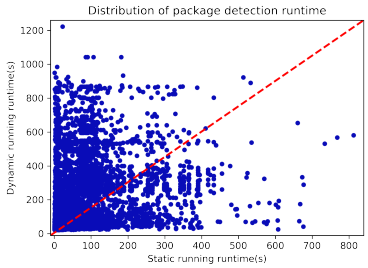
(b) Synchronization of local npm package cache in May



(c) eCDF of Static detection runtime



(d) eCDF of Dynamic detection runtime



(e) Distribution of detection runtime

Figure 5: Evaluation results for efficiency

our detector needed to be more satisfactory during the early stages of deployment, and the number of samples detected was relatively low. However, as we iterated and improved our detector, by May, our model stabilized, and the detections aligned with our experimental results and expectations. DONAPI detected 148 actual malicious packages out of 420,395 packages throughout May. The specific detection results are shown in Table 10.

Table 11: Evaluation results of different tools on datasets

Detector	TP	FP	Acc.	Prec.	Recall	F1
AMALFI [59]	1,031	27	0.97	0.97	0.89	0.97
SAP [36]	1,083	355	0.93	0.75	0.93	0.83
GUARDDOG [27]	1,052	512	0.90	0.67	0.91	0.77
DONAPI	1,062	116	0.97	0.90	0.92	0.93

Comparative study. In this study, it is necessary to demonstrate the effectiveness of our detector by comparing it with other npm package detection tools. *GUARDDOG*, a heuristic rule-based tool developed by Google, is fully open-source and thus easily used by others for comparison. *AMALFI* and *SAP*, leading approaches based mainly on machine learning, hide part of their code for security reasons, so we reconstructed the feature extraction function according to the paper description. In addition, since *AMALFI* does not provide a training set, we use our dataset for training here, and *SAP* uses the dataset supplied by their repository.

We first evaluated them using the malicious samples(1159) used in the Integrate evaluation and the npm Top 5000 packages as benign samples. As shown in Table 11, *GUARDDOG*

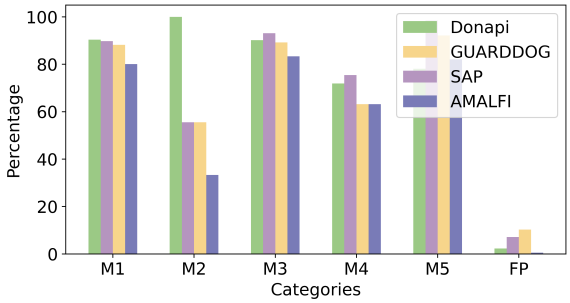


Figure 6: Percentage coverage of different package categories by different tools

and *SAP*'s performance in the false positives and evaluation metrics is significantly worse than that of AMALFI and our detector. Moreover, the approximate detection numbers do not imply similar detection capabilities. As shown in Figure 6, different detectors have different detection effects on different malicious categories, and our detector performs more balanced and significantly outperforms the other tools in the M2 category. In addition, we found that out of the 1052 instances detected by *GUARDDOG*, 972 were determined to be malicious based solely on the presence of the `scripts` field, and *SAP* and *AMALFI* rely heavily on `package.json`, which contrasts with our API call sequence-based approach and highlights a significant difference in detection methods.

In addition, for better comparison, we also analyzed all packages locally synchronized in May 2023 using these tools, and the results are shown in Table 10. The other three tools reported more malicious samples than `Donapi`, and *GUARD-*

DOG and *SAP* even reported nearly 50,000. Subsequently, we randomly sampled and manually reviewed the samples reported by the other three tools, as shown in Table 10. We found that only 6, 22, and 6 of them are truly malicious samples, respectively, which indicates that our detector has a lower false positive rate in real scenarios, thus reducing the workload of security researchers more effectively.

5 Discussion

During our experiments and daily detection, we have gained valuable insights into malicious packages and our detector. Here, we will provide a concise overview of these findings.

Table 12: Case study of unconventional behavior sequences

Cat.	Behav.	Conv.	A Case of Unconv.
Sensitive information theft (M1)	FILE_CREATE	○	◐
	SYSTEM_MESSAGE	●	●
	SERIALIZATION	◐	○
	PROCESS_COMMAND_EXECUTION	○	◐
	NETWORK_OUT	◐	◐
Reverse shell (M4)	NETWORK_IN	◐	○
	PROCESS_FILE_EXECUTION	○	◐
	PROCESS_COMMAND_EXECUTION	◐	◐
	NETWORK_OUT/IN	●	○

[1] ○miss; ◐hit once; ●Hit more than once;

Case study. We will illustrate the advantages of our detector using a few examples from our experiments. During our daily detection operations, we have observed that dynamic execution significantly complements static detection in our detector. This behavior was captured through dynamic execution, providing valuable insights that static analysis alone would have missed. For instance, the packages *businessemail-validator@99.10.9* and *azure-sdk-v3@99.10.11* employed obfuscation techniques to transform their code, rendering static parsing ineffective. However, our dynamic analysis was still able to capture the behavior of these packages.

Furthermore, our hierarchical classification framework can encompass many malicious samples. For instance, we can accurately classify *@m365-admin/customizations@999.9.15* to M1 and *tslib-tool@1.6.1* to M4, even though their behavior sequences significantly differ from the conventional behavior sequences of our known samples, as indicated in Table 12. As an example, the case of M1 sent host information to a remote server by decrypting the A file using AES in the code and executing the malicious code contained within it. Our analysis also discovered two previously unseen APIs: *fs.fchown* and *fs.writeSync*. Thus, using a comprehensive and well-developed list of APIs helps our framework effectively detect these new APIs used in malicious packages.

Findings. After analyzing collected malicious datasets and the malicious samples newly found by our detectors, we found the following phenomena. **1) Code reuse.** We observed instances where packages released by different authors simultaneously exhibited identical malicious behavior and code, with the only difference being the outgoing network addresses. For example, packages like *smart-jsonapi@1.1.1* and *uitk-build-tasks@10.0.0* showcased this phenomenon. While we cannot definitively conclude that they originated from distinct attackers, it is crucial to acknowledge the prevalence of such cases. Tracing these malicious packages back to the same attacker could imply an association with a specific attack template, enabling us to trace and address the attacking organization more effectively. To facilitate this effort, we are currently compiling a list of payloads from frequently encountered malicious packages, some of which are accessible on our web page. **2) Sophisticated attack.** As the technology for malicious package detection advances, many malicious packages attempt to bypass detection by collaborating with multiple packages, primarily through dependencies. For instance, in our investigation on the *@alfalab* series, 47 packages were released on a specific day. However, the malicious code was present only in one package named *@alfalab/core-components-spinner*, while the rest of the packages, such as *@alfalab/core-components-action-button* and *@alfalab/core-components-button*, leveraged dependencies to carry out malicious behavior. Similar to the above are packages such as *jpeg-metadata@1.5.1/ttf-metadata@1.5.2* [68], which in their first step obtain a token from one of several potential remote servers, and in their second step use this token to obtain another attack script from a remote server. **3) 0-days.** During the deployment of DONAPI, we found and manually confirmed 325 new malicious packages, all discovered on their release day. We first investigated the inclusion of two vulnerability databases (Synk[19] and Mend.io[20]). We found that nearly 20% of the packages were not indexed by either, while only 50% were indexed by both. Then, we investigated how long these malicious packages existed in the npm source and found that they lived for about one week on average, with some of them being able to exist for dozens or even hundreds of days. Therefore, our work can help npm officials find and take down these malicious packages faster.

Novelty. Table 13 summarizes some existing methods for checking the maliciousness of npm packages, compared to which our proposed DONAPI has the following advantages. First, DONAPI effectively combines both dynamic and static analysis techniques. Most existing works lack dynamic analysis, which prevents them from effectively analyzing obfuscated packages. Second, DONAPI has a more targeted and highly automated detection mechanism. It simulates package installation and import process, analyzes multiple aspects, including command scripts, code, and URLs, and builds a complete knowledge base. Third, DONAPI introduces the code dependencies reconstructor. The module simulates the code execution during installation and import, thus avoiding analyzing all files in the package [36, 59]. In addition, it preserves dependency information between files, which

[19] https://security.snyk.io/

[20] https://www.mend.io/vulnerability-database/

is missing when analyzing each file individually [16]. Finally, DONAPI presents some technical details that previous work overlooked. 1) Improve robustness through anti-evasion measures. (e.g., modifying container environment variables to prevent malicious packages from evading dynamic execution). 2) Using API instrumentation instead of Strace [16, 76] provides better interpretability by avoiding tracing non-package behaviors and enabling code stalking analysis. 3) Broader API coverage can help to reduce underreporting.

Table 13: Existing tools for analyzing npm packages

Package scanner	Detection Granularity	Technique used
npm-audit [49]	Metadata of dependencies	Static (Rules)
Ferreira et al. [21]	Package	Static (Rules)
Ohm et al. [51]	Artifact	Static (ML)
Zahan et al. [77]	Metadata	Static (Rules)
Liang et al. [40]	Package	Static (ML)
SAP [36]	Package	Static (ML)
GUARDDOG [27]	Package	Static (Rules)
AMALFI [59]	Package	Static (ML)
MALOSS [16]	Package	Static & Dynamic
DONAPI	Package	Static & Dynamic

Limitations. We know that no system is entirely foolproof, and neither are our detectors. As our strategy tries to use appropriate sub-detectors for possible attack surfaces during the package installation and import stages, this leads to some inevitable limitations in DONAPI. First, we currently only have parsers for the bash and sh commands, which are the main shell syntax used by attackers, so we need to continue to extend for other shell commands for specific scenarios (e.g., Windows shell). In addition, we set a timeout to prevent the detector from being stuck waiting for a long time, but this can lead to partial misses, such as attackers using code obfuscation and *sleep* methods to escape detection, so we also need to handle such behavior during dynamic detection. Further, we found that some packages need specific conditions (e.g., environment variables [13, 63] and external factors [65]) in dynamic execution, and their absence will lead to a decrease in code coverage, making it impossible to accomplish dynamic monitoring accurately and generating false negatives.

Finally, distinct from the testing scenario, too many false positives can lead to significant manual review costs due to the massive volume of software packages in real-world situations, which is reflected in RQ3. Therefore, we should adjust the hierarchical classification framework according to the specific usage scenarios, e.g., if we focus on ensuring the security of the environment, we should relax the policy, i.e., more sound; if we focus on the accuracy of the report, we should tighten the policy, i.e., more complete.

6 Related Work

Numerous studies [17, 18, 19] for detecting malicious JavaScript code exist. However, our research explicitly targets the detection of malicious packages within the npm ecosystem rather than solely focusing on JavaScript file detection or vulnerability detection. Therefore, we will emphasize relevant work that aligns with our research direction. We have categorized these research techniques into three distinct categories.

Machine learning. Machine learning has emerged as a well-established approach in various research domains, including malicious package detection. Garrett et al. [22] proposed an anomaly detection technique for identifying and flagging anomalous update behavior since attackers tend to deviate from the regular pattern in terms of timing and frequency when using developer credentials to update packages associated with them. Liang et al. [41] are similar, except they focus on differences in API call sequences and identify anomalous packages through cluster analysis. Wyss et al. [75] proposed a machine learning-based package difference metric in a related study designed to identify packages that share the same attack code or vulnerabilities, thus mitigating malicious code reuse. Ladisa et al. [36] proposed using language-independent features and attempted to train monolingual and cross-language models using algorithms such as XGBoost, discovering 58 previously unknown malicious packages.

Another noteworthy contribution in this field comes from Ohm et al. [51]. Their approach involves clustering malicious packages and utilizing code similarity to detect potentially malicious packages. However, the work closest to our goal is conducted by Sejfia et al. [59]. Their research uses metadata and static APIs as features, enabling classifiers for package detection. This approach offers a promising method for accurately identifying malicious packages.

Permission system. At the core of this approach lies program analysis and privilege control. These studies emphasize the vital role of program analysis and privilege control in identifying and mitigating potential threats associated with malicious packages. Ferreira et al. [21] introduced a permission system based on their investigation of package updates [22]. They achieved this by executing updated component packages within a sandbox environment to identify whether they contain sensitive permission calls.

Similarly, Vasilakis et al. [50] centered on evaluating third-party libraries. They utilized a combination of dynamic and static approaches to scrutinize whether the permissions associated with these packages went beyond pre-defined boundaries. Building upon this line of inquiry, they introduced a domain-specific language (DSL) specifically tailored to express Read-Write-eXecute (RWX) permissions [71]. By leveraging a detailed and granular model of RWX permissions, they effectively tackled the dynamic compromise problem, where the security of the system is compromised due to runtime modifications. In more recent work, Wyss et al. [76] introduce a novel permission system that evaluates malicious packages in the list and blocks any identified malicious behavior by generating a list of install-time behaviors combined with a lightweight policy language.

Dependency analysis. Dependency analysis has emerged as a prominent technique for examining code propagation paths and has found widespread application in various programming languages [66, 74, 78], and JavaScript also benefits from this approach. In a study conducted by Chinthanet et al. [11], they analyzed the entire process of code updates and vulnerability fixes. The researchers identified a significant propagation delay within the npm ecosystem, potentially resulting in delayed vulnerability fixes updates. This lag raises concerns for developers and researchers alike. Similarly, Liu et al. [43] delved into the specific parsing rules involved in the package installation process. They proposed a knowledge graph-based dependency parsing technique that enables a more precise analysis of the dependency tree for each package. This technique facilitates better identification and tracking of vulnerability propagation. While the primary focus of these studies may not be malicious package detection, the concepts and methodologies they embody offer new perspectives and ideas for effectively detecting the propagation of malicious packages within software ecosystems.

7 Conclusion

In this paper, we first build a local npm package cache containing more than 3.4 million packages for data support. Then, based on the analysis of many samples, we propose DONAPI, an malicious npm package detector that combines dynamic and static analysis to achieve a hierarchical classification of npm malicious packages. Our detector is experimentally verified to achieve good results and has advantages over *GUARDDOG*, *AMAFLI* and *SAP*. Ultimately, we also identified and confirmed 325 malicious packages as well as 2 API calls and 246 API call sequences never seen before, concrete proof of the value and effectiveness of our approach.

As we advance, we aim to further increase the granularity of our classification by analyzing more samples and improving the accuracy and efficiency of the classification through technological advances. Ultimately, we envision applying our techniques and methods more concretely and practically to diverse language ecosystems, not limited to the npm.

8 Ethics and Disclosure

This research uncovered real-world malicious packages. Despite the risks involved, the code is essential for a complete study presentation and has previously appeared in similar forms in the literature.

Regarding security disclosure protocol, we did not publicly disclose the malicious packages we found for security reasons. Although npm officials have unpublished most malicious packages, we hope to share them with professional security researchers via institutional email requests.

Finally, a large multinational enterprise has already deployed our detector internally for security interception, so we cannot release the code due to contract limitations and copyright issues. However, we are willing to share separately the malicious packages found in the official npm sources and their detection results for use by other researchers. We hope that our work will contribute to future research in these directions.

9 Acknowledgements

We would like to thank our shepherd and anonymous reviewers for their constructive comments and suggestions. This work was supported in part by National Key Research and Development Program of China (No.2021YFB3100500), Sichuan Science and Technology Program (No.2023YFG0162).

References

[1] Ammar Alazab, Ansam Khraisat, Moutaz Alazab, and Sarabjot Singh. Detection of obfuscated malicious javascript code. *Future Internet*, 2022.

[2] Birsan Alex. Dependency confusion: How i hacked into apple, microsoft and dozens of other companies. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610, 2021.

[3] Eslam Amer, Ivan Zelinka, and Shaker El-Sappagh. A multi-perspective malware detection approach through behavioral fusion of api call sequence. *Computers & Security*, 2021.

[4] Ionut Arghire. Dozens of malicious npm packages steal user, system data. https://www.securityweek.com/dozens-of-malicious-npm-packages-steal-user-system-data/, 2023.

[5] Gershon Aviad. Attacking the software supply chain with a simple rename. https://checkmarx.com/blog/attacking-the-software-supply-chain-with-a-simple-rename/, 2022.

[6] Balaji. New phishing attack hijacks email thread to inject malicious url. https://gbhackers.com/phishing-hijacks-email-thread/, 2023.

[7] Richard Bellairs. What is static analysis? static code analysis overview. https://www.perforce.com/blog/sca/what-static-analysis, 2023.

[8] Caleb Brown and David A. Wheeler. Introducing package analysis: Scanning open source packages for malicious behavior. https://openssf.org/blog/2022/04/28/introducing-package-analysis-scanning-open-source-packages-for-malicious-behavior/, 2022.

[9] Xiaohui Chen, Zhiyu Hao, Lun Li, Lei Cui, Yiran Zhu, Zhenquan Ding, and Yongji Liu. Cruparamer: Learning on parameter-augmented api sequences for malware detection. *IEEE Transactions on Information Forensics and Security*, 2022.

[10] Yu-Chen Chen, Yi-Wei Ma, and Jiann-Liang Chen. Intelligent malicious url detection with feature analysis. In *ISCC*, 2020.

[11] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering*, 2021.

[12] Catalin Cimpanu. Malicious npm packages caught installing remote access trojans. https://www.zdnet.com/article/malicious-npm-packages-caught-installing-remote-access-trojans/, 2020.

[13] Jimmy Cleveland. How to use environment variables in npm scripts safely across operating systems. https://blog.jimmydc.com/cross-env-for-environment-variables/, 2021.

[14] Tute Costa. strong_password v0.0.7 rubygem hijacked. https://withatwist.dev/strong-password-rubygem-hijacked.html, 2019.

[15] K Deepa, Radhamani G, Vinod P, MOHAMMAD SHOJAFAR, Neeraj Kumar, and M Conti. Identification of android malware using refined system calls. *Concurrency and computation*, 2019.

[16] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. In *NDSS*, 2021.

[17] Yong Fang, Chaoyi Huang, Minchuan Zeng, Zhiying Zhao, and Cheng Huang. Jstrong: Malicious javascript detection based on code semantic representation and graph neural network. *Computers & Security*, 2022.

[18] Yong Fang, Cheng Huang, Yu Su, and Yaoyao Qiu. Detecting malicious javascript code based on semantic analysis. *Computers & Security*, 2020.

[19] Aurore Fass, Michael Backes, and Ben Stock. Jstap: A static pre-filter for malicious javascript detection. In *ACSAC*, 2019.

[20] Aurore Fass, Robert P Krawczyk, Michael Backes, and Ben Stock. Jast: Fully syntactic detection of malicious (obfuscated) javascript. In *DIMVA*, 2018.

[21] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. Containing malicious package updates in npm with a lightweight permission system. In *ICSE*, 2021.

[22] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. Detecting suspicious package updates. In *ICSE-NIER*, 2019.

[23] Luiz Giovanini, Daniela Oliveira, Huascar Sanchez, and Deborah Shands. Leveraging team dynamics to predict open-source software projects' susceptibility to social engineering attacks. *arXiv*, 2021.

[24] Betul Gokkaya, Leonardo Aniello, and Basel Halak. Software supply chain: review of attacks, risk assessment strategies and security controls. *arXiv*, 2023.

[25] Danny Grander. Malicious code found in npm package event-stream downloaded 8 million times in the past 2.5 months. https://snyk.io/blog/malicious-code-found-in-npm-package-event-stream/, 2018.

[26] Yacong Gu, Lingyun Ying, Yingyuan Pu, Xiao Hu, Huajun Chai, Ruimin Wang, Xing Gao, and Haixin Duan. Investigating package related security threats in software registries. In *S&P 2023*, 2023.

[27] Guarddog. https://github.com/DataDog/guarddog, 2022.

[28] Hacktricks. Dependency confusion. https://book.hacktricks.xyz/pentesting-web/dependency-confusion, 2023.

[29] Xincheng He, Lei Xu, and Chunliu Cha. Malicious javascript code detection based on hybrid analysis. In *APSEC*, 2018.

[30] Donapi's hierarchical classification framework. https://das-lab.github.io/Donapi/, 2024.

[31] Soodeh Hosseini and Mehrdad Azizi. The hybrid technique for ddos detection with supervised learning algorithms. *Computer Networks*, 2019.

[32] Williams James and Dabirsiaghi Anand. The unfortunate reality of insecure libraries. Technical report, Aspect Security, 3 2012.

[33] Jscrambler. https://jscrambler.com/, 2023.

[34] Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, VN Venkatakrishnan, and Yinzhi Cao. Scaling javascript abstract interpretation to detect and exploit node. js taint-style vulnerability. In *S&P*, 2023.

[35] Byung-Ik Kim, Chae-Tae Im, and Hyun-Chul Jung. Suspicious malicious web site detection with strength analysis of a javascript obfuscation. *International Journal of Advanced Science and Technology*, 2011.

[36] Piergiorgio Ladisa, Serena Elisa Ponta, Nicola Ronzoni, Matias Martinez, and Olivier Barais. On the feasibility of cross-language detection of malicious packages in npm and pypi. In *ACSAC*, 2023.

[37] Ce Li, Qiujian Lv, Ning Li, Yan Wang, Degang Sun, and Yuanyuan Qiao. A novel deep framework for dynamic malware detection based on api sequence intrinsic features. *Computers & Security*, 2022.

[38] Tie Li, Gang Kou, and Yi Peng. Improving malicious urls detection via feature engineering: Linear and nonlinear space transformation methods. *Information Systems*, 2020.

[39] XuKui Li, Wei Chen, Qianru Zhang, and Lifa Wu. Building auto-encoder intrusion detection system based on random forest feature selection. *Computers & Security*, 2020.

[40] Genpei Liang, Xiangyu Zhou, Qingyu Wang, Yutong Du, and Cheng Huang. Malicious packages lurking in user-friendly python package index. In *TrustCom*, 2021.

[41] Wentao Liang, Xiang Ling, Jingzheng Wu, Tianyue Luo, and Yanjun Wu. A needle is an outlier in a haystack: Hunting malicious pypi packages with code clustering. In *ASE*, 2023.

[42] Peter Likarish, Eunjin Jung, and Insoon Jo. Obfuscated malicious javascript detection using classification techniques. In *International Conference on Malicious and Unwanted Software*, 2009.

[43] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *ICSE*, 2022.

[44] Pascal Maniriho, Abdun Naser Mahmood, and Mohammad Jabed Morshed Chowdhury. A study on malicious software behaviour analysis and detection techniques: Taxonomy, current trends and challenges. *Future Generation Computer Systems*, 2022.

[45] Marvin Moog, Markus Demmel, Michael Backes, and Aurore Fass. Statically detecting javascript obfuscation and minification techniques in the wild. In *DSN*, 2021.

[46] Dotan Nahum. Review of recent npm-based vulnerabilities. https://blog.checkpoint.com/securing-the-cloud/review-of-recent-npm-based-vulnerabilities/, 2023.

[47] Juhong Namgung, Siwoon Son, and Yang-Sae Moon. Efficient deep learning models for dga domain detection. *Security and Communication Networks*, 2021.

[48] Shradha Neupane, Grant Holmes, Elizabeth Wyss, Drew Davidson, and Lorenzo De Carli. Beyond typosquatting: An in-depth look at package confusion. In *USENIX Security*, 2023.

[49] Npm-audit. https://docs.npmjs.com/cli/v10/commands/npm-audit, 2023.

[50] Grigoris Ntousakis, Sotiris Ioannidis, and Nikos Vasilakis. Detecting third-party library problems with combined program analysis. In *CCS*, 2021.

[51] Marc Ohm, Lukas Kempf, Felix Boes, and Michael Meier. Supporting the detection of software supply chain attacks through unsupervised signature generation. *arXiv*, 2020.

[52] Marc Ohm, Arnold Sykosch, and Michael Meier. Towards detection of software supply chain attacks by forensic artifacts. In *ARES*, 2020.

[53] Lindsey O'Donnell-Welch. Dozens of malicious data-harvesting npm packages were found. https://duo.com/decipher/dozens-of-malicious-data-harvesting-npm-packages-found, 2022.

[54] Nagababu Pachhala, S Jothilakshmi, and Bhanu Prakash Battula. A comprehensive survey on identification of malware types and malware classification using machine learning techniques. In *ICOSEC*, 2021.

[55] QualityClouds. Open source libraries & security vulnerabilities. https://qualityclouds.com/open-source-libraries-and-security-vulnerabilities/, 2021.

[56] Asma Razgallah and Raphaël Khoury. Behavioral classification of android applications using system calls. In *APSEC*, 2021.

[57] Kunlun Ren, Weizhong Qiang, Yueming Wu, Yi Zhou, Deqing Zou, and Hai Jin. An empirical study on the effects of obfuscation on static machine learning-based malicious javascript detectors. In *ISSTA*, 2023.

[58] ReversingLabs. The state of software supply chain security (sscs) 2024. Technical report, ReversingLabs, 1 2024.

[59] Adriana Sejfia and Max Schäfer. Practical automated detection of malicious npm packages. In *ICSE*, 2022.

[60] Jonathan Sar Shalom. Common payloads attackers plant in malicious software packages. https://jfrog.com/blog/malware-has-a-way-

of-hiding-even-after-the-attack-is-over-get-to-know-these-common-payload-examples/, 2022.

[61] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. Silent spring: Prototype pollution leads to remote code execution in node.js. In *USENIX Security*, 2023.

[62] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. Anything to hide? studying minified and obfuscated code in the web. In *WWW*, 2019.

[63] Tyler Smith. Prevent npm from installing packages outside of a docker container. https://dev.to/tylerlwsmith/prevent-npm-from-installing-packages-outside-of-a-docker-container-akh, 2021.

[64] Liran Tal. What is typosquatting and how typosquatting attacks are responsible for malicious modules in npm. https://snyk.io/blog/typosquatting-attacks/, 2021.

[65] Liran Tal. Alert: peacenotwar module sabotages npm developers in the node-ipc package to protest the invasion of ukraine. https://snyk.io/blog/peacenotwar-malicious-npm-node-ipc-package-vulnerability/, 2022.

[66] Wei Tang, Zhengzi Xu, Chengwei Liu, Jiahui Wu, Shouguo Yang, Yi Li, Ping Luo, and Yang Liu. Towards understanding third-party library dependency in c/c++ ecosystem. In *ASE*, 2022.

[67] Matthew Taylor, Ruturaj K Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. Spellbound: Defending against package typosquatting. *arXiv*, 2020.

[68] Phylum Research Team. Phylum discovers sophisticated ongoing attack on npm. https://blog.phylum.io/sophisticated-ongoing-attack-discovered-on-npm/, 2023.

[69] Bernhard Tellenbach, Sergio Paganoni, and Marc Rennhard. Detecting obfuscated javascripts from known and unknown obfuscators using machine learning. *International Journal on Advances in Security*, 2016.

[70] Nikolai Philipp Tschacher. *Typosquatting in programming language package managers*. PhD thesis, Universität Hamburg, Fachbereich Informatik, 2016.

[71] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. Preventing dynamic library compromise on node. js via rwx-based privilege reduction. In *CCS*, 2021.

[72] Veracode. Esg survey report: Modern application development security. Technical report, Veracode, 8 2020.

[73] Chao Wang, Ronny Ko, Yue Zhang, Yuqing Yang, and Zhiqiang Lin. Taintmini: Detecting flow of sensitive data in mini-programs with static taint analysis. In *ICSE*, 2023.

[74] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of third-party libraries in java projects. In *ICSME*, 2020.

[75] Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. What the fork? finding hidden code clones in npm. In *ICSE*, 2022.

[76] Elizabeth Wyss, Alexander Wittman, Drew Davidson, and Lorenzo De Carli. Wolf at the door: Preventing install-time attacks in npm with latch. In *Asia-CCS*, 2022.

[77] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. What are weak links in the npm supply chain? In *ICSE-SEIP*, 2022.

[78] Xian Zhan, Lingling Fan, Tianming Liu, Sen Chen, Li Li, Haoyu Wang, Yifei Xu, Xiapu Luo, and Yang Liu. Automated third-party library detection for android applications: Are we there yet? In *ICSE*, 2020.

[79] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *USENIX Security*, 2019.

[80] Nikola Đuza. Javascript growing pains: From 0 to 13,000 dependencies. https://blog.appsignal.com/2020/05/14/javascript-growing-pains-from-0-to-13000-dependencies.html, 2020.

Appendix

A Hierarchical Classifier

A.1 The Importance of Sequences

Table 14: The performance on Malicious software import (M3) using sets and sequences

Category	APIs Representation	TP	FP	Instances
Malicious software import (M3)	Sets	56	44	FP: ping-me-maybe@0.0.0, create-sanity@3.11.5
	Sequences	54	26	FN: noblox.js-proxies@1.0.0, noblox.js-proxies@1.0.3

We obtained 100 packages from malicious datasets and real-world scenarios containing sets corresponding to our defined M3 sequences for testing. Table 14 shows that API call sequences have fewer false positives (18) than API sets, although there are a few misses (only 2).

A.2 Hierarchical Classification for Shell Command

Table 15: YARA rules for capturing sensitive behavior of bash commands and shell scripts

Rule	Description	Examples
R1	Capturing Executable Files	/bin/bash, /bin/sh
R2	Capturing file manipulation commands	scp, cat, binary, chmod
R3	Capturing Commands for Obtaining Sensitive Information	whoami, hostname, pwd
R4	Capturing Networking Commands	wget, curl, nc, nslookup
R5	Capturing sensitive commands	base64, b64
R6	Capturing Sensitive Files or Folders	etc/rc, etc/passwd, /.profile
R7	Capturing commands that execute .exe file	.*.exe
R8	Capturing Suspicious Files	.sh, .exec

Table 16: Mapping rules for malicious command behavior

Module Name	Category	Rule Sequences
Malicious shell command detector	Sensitive information theft (M1)	[R4 → R3, R4 → R6]
	Sensitive file operation (M2)	[R2 → R6, R5 → R1]
	Malicious software import (M3)	[R4 → R8, R7]
	Reverse shell (M4)	[R4 → R1]

As depicted in Table 15, we formulate a set of rules to capture suspicious behaviors within the commands. Leveraging these rules, we then devise a rule combination analogous to API call sequences, enabling precise categorization of both the associated malicious commands and their respective packages. This categorization is illustrated in Table 16.

B Obfuscation Feature Evaluation

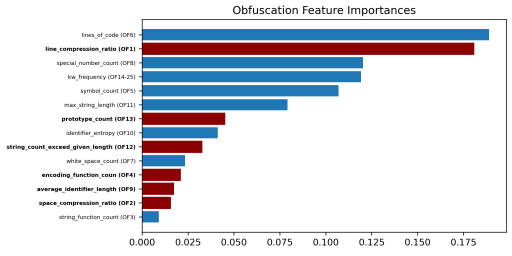


Figure 7: Obfuscation feature importance rankings. The bolded parts are new features that we propose.

C Additional experiments on our detectors

In addressing RQ1, we scrutinized our detectors' accuracy via a validation set, yielding a commendable performance. Nevertheless, considering that both training and validation sets originated from our accumulated malicious samples (though without overlap), these experiments inherently carry the potential for certain metric inflation. Consequently, to substantiate the stability of our detectors' accuracy in the face of sample variations, we deliberately curated distinct test sets for each subdetector and for DONAPI independently. Our focus here is solely on the capability to detect unknown malicious or obfuscated samples, i.e., the Recall.

C.1 Dataset

Local cache. Given the inherent variability and diversity of malicious samples and to verify the validity of the local cache, we collected meta-information on 151 malicious packages disclosed by the Synk database in May and June. Subsequently, we tried and obtained all the original `.tgz` files from the local cache (not available at the official NPM), effectively proving that we can retain the repository deletion copies promptly.
Obfuscated code detector. Considering that different obfuscation means may have different characteristics [62], we randomly selected 66 obfuscated packages from daily synchronization as obfuscated samples.
Static identifier & DONAPI. Due to the static recognizer's lack of ability to detect commands, we excluded a subset of malicious packages (8 in total) during this testing phase. In contrast, for the DONAPI test, we used the complete set of 151 malicious samples.

C.2 Result

Table 17: Additional evaluation results of recall for subdetectors and DONAPI

Idx	Module Name	Malicious/Obfuscated	FN	Recall
#1	Obfuscated code detector	66	2	96.97%
#2	Suspicious package static identifier	143	8	94.41%
#3	**DONAPI (Integral detector)**	**151**	**8**	**94.70%**

As shown in Table 17, the detector maintains a comparable recall rate for unknown samples.

D API & Behavior

We focus on 132 APIs (see the website for their combinations with parameters). At the same time, we outline abstractions for 12 different behavior types, with each behavior type precisely defined and described. For comprehensive insights, please refer to https://github.com/das-lab/Donapi.