Software Evolution Metrics for the Detection of Trojan Code in npm Packages

Giacomo Benedetti[1], Luca Caviglione[1,*], Giovanni Lagorio[2] and Marco Zoratti[1,2]

[1]Institute of Applied Mathematics and Information Technologies, Genova, Italy
[2]University of Genova, Genova, Italy

Abstract

The increasing sophistication of attacks against the software supply chain imposes the need for lightweight and effective countermeasures. The recent threat RATatouille targeting the `rand-user-agent` package of the `npm` ecosystem has demonstrated that attackers can stealthily embed trojans directly into source code by exploiting "visual deceiving" techniques. Therefore, this paper focuses on the detection of trojan injection mechanisms leveraging bursts of blank spaces and homoglyphs to evade manual inspection and traditional countermeasures. To perform tests, we compiled a dataset consisting of the $2,800$ most popular `npm` packages and we considered seven classic metrics used to describe the evolution of the code as well as three ad-hoc indicators. Our results showcase that generic metrics are inadequate for fast-evolving ecosystems, whereas ad-hoc indicators can help to spot the presence of trojans cloaked within the source code.

Keywords

software supply chain security, trojan source, code evolution

1. Introduction

Mitigating the impact of malicious software has been a prime security concern for more than two decades. Over the years, researchers and industrial experts proposed many countermeasures against malware, such as detectors based on signatures, static analysis tools, reverse engineering frameworks, and sandboxing environments [1, 2]. The recent surge of AI ignited the proliferation of advanced approaches, e.g., graph neural networks for classifying malicious software [3, 4], and LLMs for anticipating offensive trends [5]. However, better countermeasures triggered an "arms race" leading to a composite panorama of defensive tactics and practices [6, 7].

To evade detection, attackers regularly use advanced techniques ranging from information hiding to obfuscation [8]. Several elusive mechanisms leverage a multi-stage loading blueprint, where a monolithic malware is decoupled into different components, e.g., dropper and payload. Due to the growing adoption of third-party packages and the ubiquitous adoption of open-source software, a major offensive template aims at injecting malicious contents in popular repositories [9]. Many recent threat campaigns targeted software packages for which the attacker already obtained the access, e.g., via social engineering or fake identities [10]. The threat actor could also create a "rogue" dependency for importing the malicious payload when merged against the target project. Despite the used template, the aim of the attacker is to avoid the detection by leveraging the burden of source code, commits, and turnovers of maintainers, which are often difficult to assess and model even with the aid of AI [11].

The injection of trojan code in a package requires deceiving the developer to hinder the identification of the unwanted payload. In general, attackers take advantage of two major techniques. The first exploits *blank spaces* for shifting the malicious payload outside the visible area of the development environment, e.g., the editor or the diff console. The second exploits *homoglyphs* preventing the developer to recognize additional functions or variables due to their visual similarity with legitimate

Joint National Conference on Cybersecurity (ITASEC & SERICS 2026), February 09-13, 2026, Cagliari, IT
*Corresponding author.
giacomo.benedetti@ge.imati.cnr.it (G. Benedetti); luca.caviglione@ge.imati.cnr.it (L. Caviglione); giovanni.lagorio@unige.it (G. Lagorio); marco.zoratti@unige.it (M. Zoratti)
0000-0003-2609-6787 (G. Benedetti); 0000-0001-6466-3354 (L. Caviglione); 0000-0002-6632-1523 (G. Lagorio); 0009-0005-7438-1461 (M. Zoratti)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-04-23

CEUR-WS.org/Vol-4198/paper11.pdf

ones. A major example is the RATatouille malware that has been injected in the `rand-user-agent` package version 1.0.100 hosted by the npm ecosystem. Specifically, the attacker appended malicious instructions to a benign line of code interleaved with a trail of blank spaces [12]. Both blank spaces and homoglyphs are effective for evading manual code analysis practices. The "visual" ability of the human is highly impaired since many editors (e.g., the code renderer of the npm web interface) do not alert the developer nor automatically wrap lines. Mitigating this new-wave of *visual deceiving* attacks is then of prime importance, especially to prevent the need of deploying more complex approaches, such as dynamic analysis. Unfortunately, spotting subtle variations of the code could be a time- and resource-expensive process, especially for large open-source projects. A promising defensive approach entails *code evolution*, which aims to modeling the evolution of a package to identify alterations. For instance, [13] showcased a prime attempt of analyzing GitHub repositories to detect injections into the distributed software. However, the comparison is limited to files within the registry, thus may not be effective to spot additional statements added directly to the sources.

Therefore, this paper provides a first analysis to how metrics related to the software development life-cycle can enable the identification of trojan code injected via blank spaces and homoglyphs. To pursue this goal, we shortlisted seven metrics taken from the literature dealing with code analysis (e.g., number of lines of code) and designed three metrics for our threat model (e.g., blank space ratio). To assess the performance of our approach, we conducted tests on $2,800$ most popular npm packages. We selected npm since is one of the most used and composite ecosystem at the basis of modern software supply chains [14]. Our results support the intuition that ad-hoc metrics are required to detect the injection of trojans in source code, whereas more general indicators are impaired by the fast development pace characterizing modern ecosystems.

Summing up, the contributions of this work are: *i*) a dataset containing evolution metrics for npm packages, and *ii*) a quantitative evaluation on whether they can identify blank spaces and homoglyphs attacks.

The rest of the paper is structured as follows. Section 2 provides background information and the motivations, while Section 3 reviews the literature. Section 4 introduces the threat model, Section 5 outlines the investigation methodology, and Section 6 discusses a brief characterization of npm packages. Section 7 showcases numerical results and Section 8 concludes the paper.

2. Background and Motivation

This section introduces basic concepts and motivates the need of new detection approaches. Specifically, Section 2.1 provides an overview of how open-source packages can be analyzed to identify malicious activities. Section 2.2 briefly outlines main malware detection techniques and why new metrics are needed to face trojan injections.

2.1. The Open-Source Development Process

Modern software supply chains heavily rely on open-source ecosystems, where packages evolve through a continuous stream of releases, contributions, and automated updates [15]. In ecosystems such as npm, the development workflow is typically mediated by version control services (e.g., GitHub) and driven by a composite set of actors, including maintainers, external contributors, and bots handling dependency updates or maintenance tasks. The evolution of a package is then captured by the *commit history*, which could be composed of more than thousands of entries for the most popular projects [14]. Despite differences in contributor roles and commit styles, software projects tend to preserve structural regularities that emerge from coding conventions and languages. Examples include limiting the maximum line length for readability, consistent formatting practices, and the stable use of a specific text encoding across the codebase [16]. The major behaviors that emerge across the commit history form an *evolutionary signature* for the project.

The ability of computing a baseline behavior becomes particularly relevant in the context of threat detection since it allows to recognize subtle anomalies that may surface during the development

process. This is especially useful to identify attackers seeking to compromise repositories by "blending" their malicious modifications within legitimate software [10]. For instance, the introduction of a line hiding a payload after an excessive sequence of blank spaces, or the appearance of identifiers encoded with characters outside the usual encoding range, could be captured by analyzing deviations from the "historical" behavior of the project. Hence, software evolution metrics can serve as lightweight indicators to spot out-of-pattern modifications without requiring a deep code analysis. This is critical for ecosystems like `npm`, where the number and frequency of releases make manual inspection impractical and where the development pace favors attackers exploiting visual ambiguity or formatting-based evasion techniques [17].

2.2. Limitation in Malware Detection Strategies

Despite being usually very effective, classic detection methods tend to struggle when adversaries compromise the source code itself. This is why an increasing number of threat actors targets the software supply chain and injects trojans directly into upstream repositories [13]. When the malicious payload becomes part of the official release, traditional security mechanisms may overlook the hazard, as the artifact originates from a legitimate maintainer.

Software supply chain threats like trojan source attacks exacerbate this problem by exploiting discrepancies between what humans see and what compilers interpret [18, 19]. Techniques such as the insertion of long sequences of blank spaces or the use of homoglyphs allow the concealment of instructions/compounds or introduce deceptive function names that visually resemble legitimate ones. Static analysis tools often normalize blank spaces or character encodings, causing to discard the features that can reveal an additional payload [18]. Dynamic analysis could be insufficient since malicious payloads may remain dormant, require specific activation conditions, or serve as exfiltration helpers embedded in apparently harmless functions. Moreover, the volume and churn of modern ecosystems make dynamic inspection resource-intensive and impractical at scale [20]. The analysis of code evolution then offers a strategy that can complement classic static/dynamic malware detection techniques. Rather than understanding the semantics of the injected malicious logic, the idea is to focus on measurable deviations from the historical development patterns of the project.

3. Related Work

The security of package registries, such as `npm` and `PyPI`, has been extensively investigated in recent years due to the surge in supply chain attacks. The fragility of the `npm` ecosystem has been already highlighted in [21] by showing how the account of a maintainer can act as a single point of failure for thousands of linked packages. A curated dataset of malicious open-source `Node.js` packages, which have been used in real-world attacks, has been provided in [22]. More recently, this view has been expanded by proposing a general taxonomy independent of a specific programming language or ecosystem [10]. While these works focus on identifying malicious packages via metadata (e.g., author reputation and package naming patterns) or dependency graphs, our work focuses on detecting the malicious payload itself embedded within the source code during the maintenance life-cycle.

As regards "visual deceiving" vulnerabilities like *trojan source*, they have been formally introduced in [18]. In essence, the authors demonstrated how Unicode control characters can be exploited to reorder tokens in source code to deceive human reviewers, while remaining valid for compilers. This class of attacks exploits the "visual gap" in development tools: while IDEs and diff viewers render code for readability, compilers process the underlying raw byte stream. Although compilers can give warnings when they encounter these kind of patterns, attackers can circumvent these countermeasures by using homoglyphs or, as highlighted in the recent RATatouille campaign, by exploiting widely accepted formatting characters, such as blank spaces, to push malicious logic out of the visible viewport [12]. Unlike general obfuscation used to thwart reverse engineering [8], visual spoofing attacks are designed to survive human-driven code review processes.

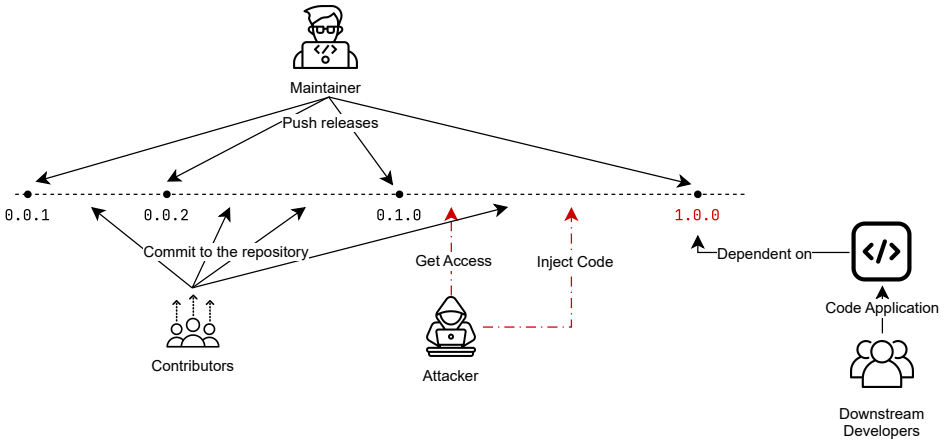


Figure 1: Threat model considered in this work.

Detecting malware through software metrics is a well-established practice. However, within the context of high-level interpreted languages like JavaScript, these metrics can be ineffective due to *minification* and legitimate code density. Previous work proposed the use of repository history to identify anomalous commits [13], a concept that aligns with our methodology. Nevertheless, the approach primarily targets file additions or metadata changes, while we address suspicious source code changes. However, a relevant corpus of research on security of open-source ecosystems focuses on package naming attacks, which are exploited via two main prevalent techniques. The first is *typosquatting* and relies on users making typographical errors when issuing installation commands (e.g., omitting a hyphen or swapping two characters). This causes to download malicious packages instead of the intended ones, and allows the attackers to execute code on the machine of the victim. Typosquatting may leverage a wide-range of cases, for instance the malicious code could be cloaked in test cases or in pre-/post-install scripts [22, 23]. The second technique is *combosquatting* and creates plausible-sounding package names by combining a familiar target name with legitimate keywords (e.g., `analytics`, `proxy`, or `helper`) to deceive developers into trusting a rogue dependency [24].

Attacks based on "squatting" target the distribution phase by mimicking package identifiers, while our work addresses threats injected directly into source code artifacts. Thus, our goal is to introduce lightweight evolution metrics, specifically focusing on blank space ratios and encoding variations.

4. Threat Model

The *trojan source* umbrella refers to a hiding method that injects characters into the source code to force text editors to display a content that differs from what compilers and interpreters process [18, 19]. Figure 1 showcases the threat model considered in this work. In more detail, the attacker hijacks a code repository and injects malicious code at a specific point in time, resulting in the malicious payload being included in the release made by the project maintainer. Since the attacker applies a minimal modification using a trojan source technique, the injected code is hardly detectable.

In this work, we consider an attacker injecting code via two major templates, i.e., blank spaces and homoglyphs. The first class consists of an attacker capable of appending a one-line malicious payload after a burst of blank space characters. The assumption here is that the developer uses an editor without any active line/text wrap functionalities, i.e., it does not automatically apply carriage returns to fit the text in the editor window. Figure 2 showcases how the online editor of GitHub can be exploited to mislead the user. As shown, the attack only alters the size of the horizontal bar for managing the area rendering the code. If this detail is missed, the developer will fail to spot that a payload has been cloaked outside the visible area. The second class allows the attacker to deceive a developer to use a malicious function via a homoglyph. Figure 3 showcases an example where a package has been hijacked. The code

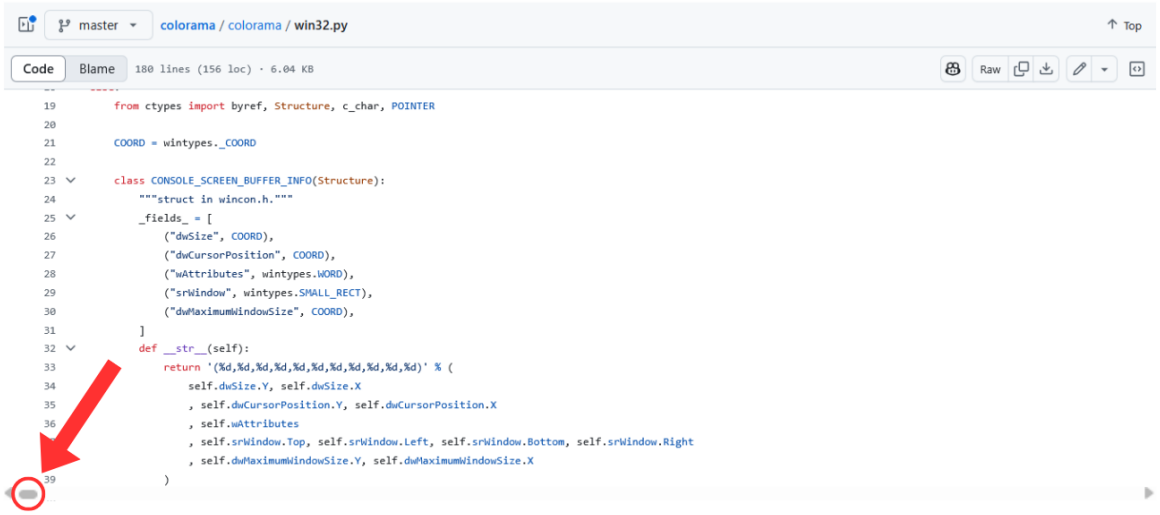


Figure 2: Example of blank spaces used to hide a malicious payload outside the editor frame.

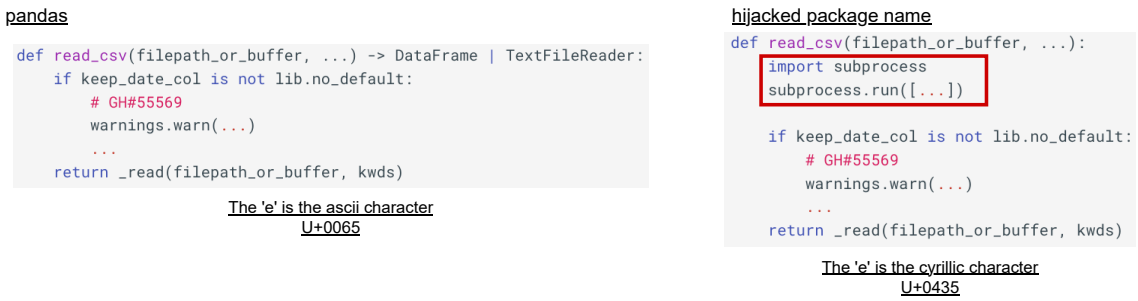


Figure 3: Example of a homoglyph used to invoke an alternative version of `read_csv()`.

has been altered to invoke a "bogus" version of the `read_csv` function instead of the one implemented in the `pandas` package. This has been made possible since the Unicode characters U+0065 and U+0435 are rendered in a similar manner, i.e., the ASCII and cyrillic 'e' are visually equivalent. To not disclose the attack, the injected `read_csv` mimics all the functionalities of the original code. Yet, it also contains some commands added by the attacker, which have been highlighted with a red box in the example of Figure 3. Downstream developers could then be misled into using the homoglyphed `read_csv` function via poisoned code-completion features or fake examples that can be copied/pasted into the editor.

We point out that, both the blank space and the homoglyphs techniques are effective only when a shallow analysis of the code is applied. In fact, deeper techniques (e.g., dynamic analysis) may easily reveal their usage. However, the complexity of the supply chain that makes packages available to developers hides the distribution of potentially infected artifacts.

5. Investigation Methodology

This section discusses our approach for evaluating the evolution of the npm ecosystem. In more detail, Section 5.1 introduces the framework used for collecting the needed information, while Section 5.2 describes the metrics for modeling the evolution of npm packages. Lastly, Section 5.3 describes how malicious samples have been prepared.

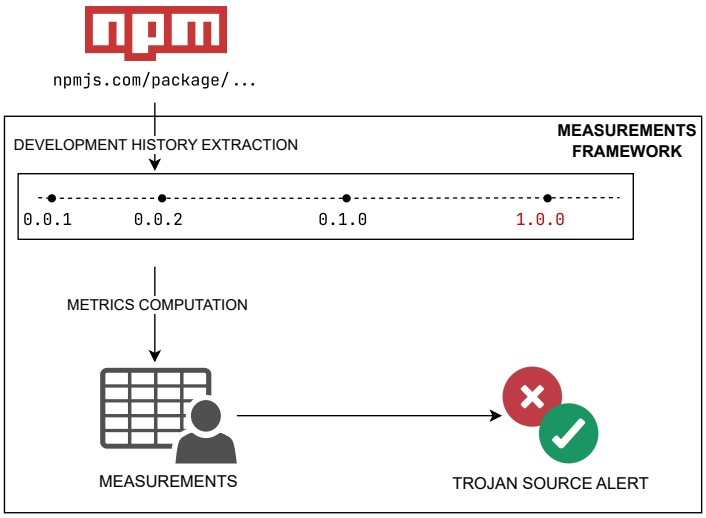


Figure 4: Methodology used for collecting and analyzing data from the npm registry.

5.1. Measurement Framework

To collect information from real npm packages and to evaluate the feasibility of detecting trojan code injections via metrics describing the code evolution, we implemented a set of command line tools written in Python[1]. Figure 4 showcases the overall framework, which relies on three stages: extraction of evolutionary data, computation of the selected metrics, and assessment of anomalies arising across the life-cycle of a package.

As a first step, our measurement framework interacts with the npm registry to reconstruct the development history for a given package. To this aim, a suitable function retrieves all tagged releases: by relying upon *tags* rather than individual commits, our approach reduces the burden caused by temporary or experimental changes. The collected information for a package contributes to a clearer view of the evolution patterns relevant to downstream users. As a second step, the source files composing a single release are "sanitized". Our tool removes minified, auto-generated, or test artifacts that would skew the measurements. The remaining source files are then parsed for computing the various evolution metrics. Obtained data captures how the package evolves over time and allows to establish a baseline describing its expected behavior. Our framework can iterate through a list of packages for computing a csv file with measurements and indications on potential trojanized code. However, performing automatic detection is outside the scope of this paper and is part of our ongoing research.

5.2. Selection and Design of Metrics

The selection and design of suitable indicators for assessing the evolution of npm packages are driven by two objectives. Metrics should be general enough to effectively capture the evolution of the code base but they should also guarantee a sufficient sensitivity against the (subtle) alterations characterizing the use of blank spaces or homoglyphs. To this aim, we shortlisted a set of ten indicators, summarized in Table 1.

Specifically, we considered the *commit frequency* to identify potential discrepancies in the habits of developers that may suggest a malicious behavior. The *commit time* may indicate one or more commits at an unusual time, e.g., an attacker residing in a different timezone. The *cyclomatic complexity* can capture code with an alternative form compared to regular contributions, such as a malicious payload implementing advanced offensive techniques. The *lines of code* added across the history of the package

[1]To reproduce results presented in the paper and to conduct further research, both code and data are publicly available online at: https://github.com/marchacio/ITASEC26-source-attack-metrics

Table 1
Metrics used to spot trojan code (**bold**: metrics effective for our threat model).

Metric name	Brief description
Maximum line length	Longest sequence of characters without a line break in a file.
Blank space ratio	Fraction of blank space characters over total characters.
Number of encodings	Count of characters with a different encoding observed across files.
Commit frequency	Intervals when commits happen from contributors in the projects.
Commit time	Date and time of the commit.
Cyclomatic complexity	Complexity of the code according to the cyclomatic algorithm.
Line of code added per release	Lines of code added in a release.
Code size in bytes	Size of the project in bytes.
Number of changed files	Difference between the total number of files in consecutive releases.
Shannon entropy	Amount of information contained in a file.

can be suitable to identify the presence of blank lines to push contents in a portion of the file "difficult" to inspect. The *code size* in bytes can determine whether a malicious payload is present, whereas the *number of changed files* may indicate the introduction of ad-hoc files containing a trojan, configuration data, or supplemental attack stages. Lastly, the *Shannon entropy* can provide useful information to reveal the presence of obfuscated code, e.g., trojanized statements or compounds.

To complete these general indicators, we also introduce three metrics tailored for our threat model. The first is the *maximum line length*, i.e., the longest sequence of characters without a newline within any file of a release. This metric is effective for detecting malicious payloads appended after long bursts of characters, as commonly used in visual-deceiving attacks. The second is the *blank space ratio*, defined as the fraction of blank space characters over the total number of non-blank characters. Large and unexpected increases suggest spacing-based hiding strategies. The third is the *number of encodings* used within a file, which counts the occurrences of "uncommon" characters that may include homoglyphs.

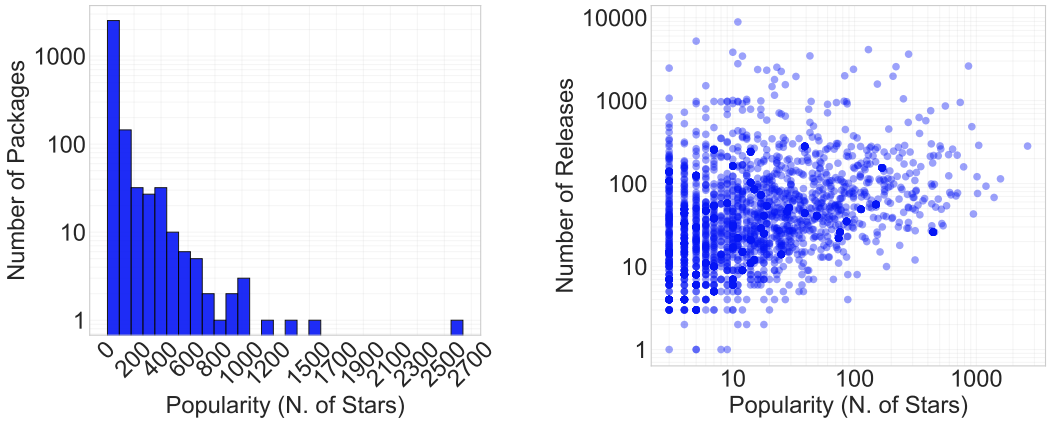
5.3. Payload Injection

To assess the metrics presented in Table 1 under fair conditions, we injected a real trojan in 100 randomly-chosen packages. In more detail, we modified the last tagged release of a package with the RATatouille malware, i.e., a one-line trojan designed to initiate a command-and-control channel on the host of the victim. We selected such a class of trojans since they are characterized by a minimal footprint and allow us to model the operational simplicity of real software supply chain attacks, where the typical goal is to inject only the code necessary for persistence or exfiltration [25].

To inject the payload, we implemented the two variants of the threat model discussed in Section 4. For the case of blank space, the remote trojan has been added by appending its code after a sequence of 800 blank spaces. For the case of homoglyphs, we injected the trojan by modifying four characters in the respective homoglyphs, i.e., four Cyrillic 'a' (U+0430) instead of the Latin version (U+0061). For both cases, the injection of the payload has been done by randomly selecting both the line at which "insert" the payload and the targeted `.js` source file. The repository for the paper contains both scripts and payloads to create `npm` packages "poisoned" by the malicious payloads.

6. Data Collection and Preliminary Analysis

To have a suitable dataset to evaluate both the feasibility and the performance of our approach, we collected the 2,800 top-ranked `npm` packages from the `npm-rank` list [26]. Apart being representative of realistic open-source software used in a wide variety or settings, focusing on popular packages prevents to avoid data poisoning. In fact, many packages available in the `npm` registry are test or unfinished projects that do not represent actual packages. Resorting to 2,800 packages also allows for a suitable trade-off between the relevance of the result and the execution time. Specifically, our tests



(a) Popularity distribution

(b) Number of releases vs. popularity

Figure 5: Popularity metrics of the considered 2,800-package dataset.

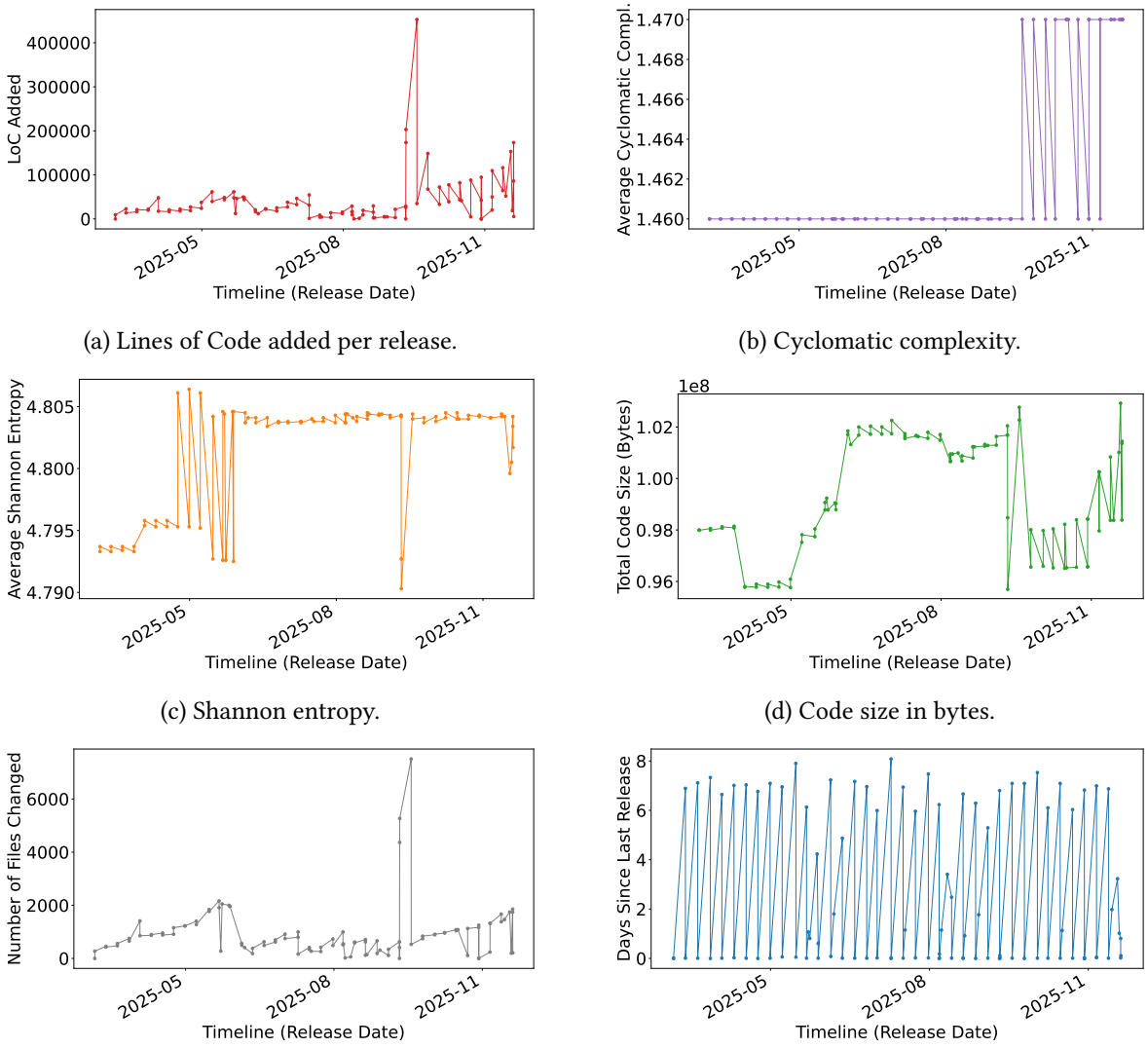
Table 2
High-level view of the dataset composed of 2,800 npm packages.

Versions per Package (Release)		Files per Release	
Pkg. with 1 version	20	Rel. with 1 file	44,276
Pkg. with < 10 versions	528	Rel. with < 20 files	275,727
Pkg. with > 50 versions	933	Rel. with > 20 files	89,801
Mean	145.45	Mean	171.44
Median	31.00	Median	4.00
Max	10,161	Max	44,709

have been conducted on an Ubuntu 20.04 machine equipped with an Intel i9-9900KF CPU @3.60 GHz and 32 GB of RAM. On average, the complete analysis of a package release required 18 seconds.

Table 2 provides a high-level view of the computed dataset. As shown, the number of versions per package is generally high. However, the median is 31 versions, which hints at a dataset heavily influenced by a few packages with many versions. There is also a significant number of packages with more than 50 versions. This supports our intuition of trying to spot the presence of attacks via "evolutionary" information, i.e., the larger amount of evolutionary information can be derived, the better. The collected 2,800 packages also vary in the number of files: many packages have a small number of items, with a mean of 171.44 and a median of 4. Many releases have fewer than 20 files, hence our approach is manageable in terms of resources and execution time. Since the popularity plays a major role in the attractiveness of a package from the perspective of the attacker, Figure 5 showcases core popularity metrics computed over our dataset. In more detail, Figure 5a depicts the distribution of popularity as computed by the official npm registry. As shown, the maximum number of stars is 2,638 with a mean of 15.76. According to [21], the popularity of packages in our sample follows the same trend as the overall npm ecosystem, making our subset representative of mainstream development practices. Besides, Figure 5b shows that the number of releases for highly-popular packages (i.e., more than 100 stars) is clustered largely above 10 releases, indicating that our dataset is particularly suitable for being investigated through evolution metrics.

Before conducting large-scale tests, we performed a preliminary analysis to identify the most discriminative indicators for revealing the presence of trojans. As an example, Figure 6 depicts the results for the seven general metrics reported in Table 1 when computed on the last 100 releases of the `Angular` package also by considering a version containing the trojan. As shown, indicators such as the commit frequency, the cyclomatic complexity, or the code size turned out to be too sensitive to benign factors such as refactoring, dependency changes, or verbosity differences. For instance, version 19.2.5 of



(a) Lines of Code added per release.

(b) Cyclomatic complexity.

(c) Shannon entropy.

(d) Code size in bytes.

(e) Number of changed files.

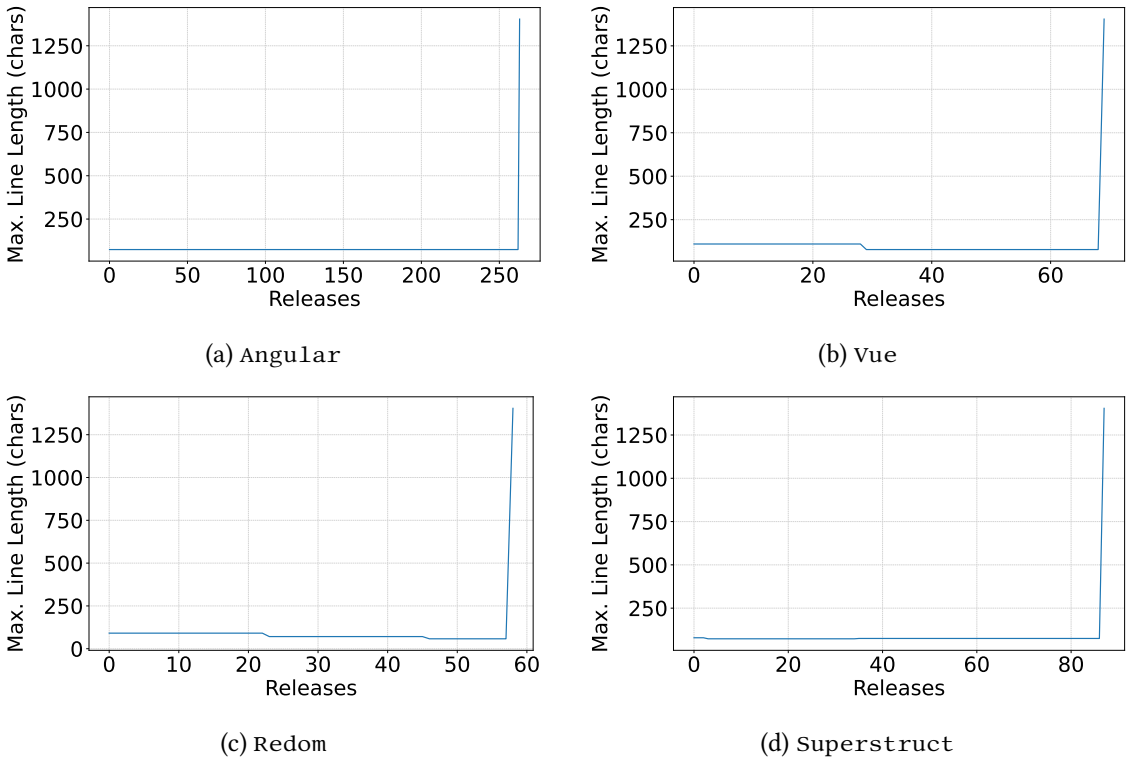
(f) Commit frequency and time.

Figure 6: Example of general metrics behavior on the last 100 versions of the `Angular` package.

`Angular` reduced the dependence against `NgModules` functionalities, thus leading to a considerable fluctuation of the lines of code indicator. Alterations in indicators were then properly due to the refactoring of many functionalities of the `Angular` framework. In other words, general metrics do not provide a solid baseline for the identification of attacks carried out through the injection of blank spaces and homoglyphs. Therefore, the metrics designed for our threat model and reported in bold in Table 1 are expected to exhibit a stronger stability across the "normal" development process.

7. Results

This section presents the results obtained by considering $2,900$ samples, i.e., the collected $2,800$ `npm` packages and 100 injected packages. For the sake of clarity, we only report results for the blank space ratio, maximum line length, and number of encodings. The online dataset contains `.csv` files for each package-metric pair having the same layout. Specifically, rows correspond to the content of a `npm` package and columns correspond to versions. When an asset is not present in a specific version of the package, the "cell" is empty. In the following, Section 7.1 reports the results for blank spaces, while Section 7.2 for homoglyphs.



(a) `Angular` (b) `Vue`

(c) `Redom` (d) `Superstruct`

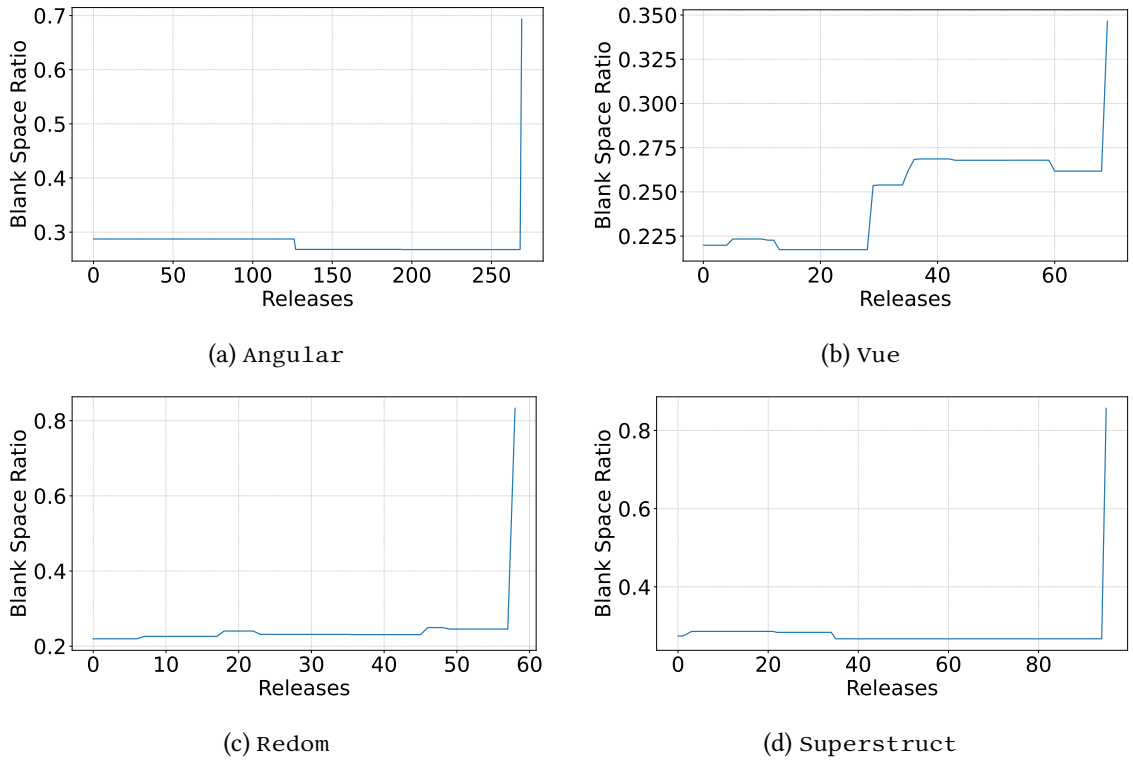
Figure 7: Maximum line length metric computed on four packages from our dataset.

7.1. Blank Spaces

When computed against our dataset, the blank space ratio and the maximum line length exhibit a stable behavior. This could be ascribed to the coding convention of not exceeding 80 characters per line when editing textual contents [16]. The metrics should be then considered particularly informative for capturing anomalies associated with the injection of blank spaces.

To make some examples, Figure 7 portrays the behavior of the maximum line length for four representative packages in our dataset that have been targeted by the trojan. As shown, the metric exhibits similar trends regardless of the development process or the package type. For instance, `Angular` with more than 250 releases and `Redom` with less than 60 releases are both consistently under 100 characters per line, as reported in Figure 7a and Figure 7c, respectively. As a result, the presence of a release poisoned with a trojan will cause the metric to diverge sharply. Conversely, Figure 8 shows that the blank space ratio exhibits a greater variability and it is less susceptible to the insertion of very long lines of legitimate code. For instance, Figure 8b reports the behavior for the package `Vue`, which is characterized by an "irregular" development process. As shown, the blank space ratio increases by roughly 21% between releases `2.6.14` and `2.7.0-alpha.1` due to legitimate stylistic and structural modifications, i.e., the introduction of templates for graphical rendering of web pages containing many blank spaces. However, the injected payload causes a pronounced rise (i.e., approximately 35%), which stands clearly apart from typical evolution-induced fluctuations. The metric is even more expressive for projects with dense release cycles, such as `Angular`, where the large number of versions (> 300) amplifies deviation patterns and makes the anomaly evident, as depicted in Figure 8a. The blank space ratio remains also effective for slow-paced projects, such as `Superstruct` and `Redom`. These repositories show small and incremental modifications over time, making code changes appear more visible in the metric trend.

Overall, our approach was able to correctly identify the injected payload in 70% of the 100 injected `npm` packages and it only triggered false positives for 73 out of the 37,462 analyzed files. The main cause of false positives is the use of formatting patterns, which can be ascribed to well-known templates.



(a) `Angular` (b) `Vue`

(c) `Redom` (d) `Superstruct`

Figure 8: Blank space ratio metric computed on four packages from our dataset.

Table 3
Impact of the trojan on indicators based on encodings and characters for three versions of four representative packages.

`Angular`				`Vue`			
Version	**Homoglyphs**	**Chars**	**Encodings**	**Version**	**Homoglyphs**	**Chars**	**Encodings**
v14.2.9	0	3,589	0	v3.2.43	0	8,732	0
v14.2.10	0	3,589	0	v3.2.44	0	8,321	0
v14.2.11	0	4,341	0	v3.2.45	0	9,822	0
v14.2.11*	4	4,683	4	**v3.2.45***	4	9,916	4

`Redom`				`Superstruct`			
Version	**Homoglyphs**	**Chars**	**Encodings**	**Version**	**Homoglyphs**	**Chars**	**Encodings**
v3.25.0	0	3,281	0	v0.6.0	0	696	0
v3.26.0	0	3,281	0	v0.6.0	0	541	0
v3.27.0	0	3,310	0	v0.7.0	0	1,458	0
v3.27.0*	4	4,404	4	**v0.7.0***	4	1,790	4

For instance, some packages (e.g., `akkeris` and `d4`) include the description of functionalities through textual drawings and diagrams that embed a large number of blank spaces, causing a legitimate increase in the blank space ratio. Besides `figlet` exploits ASCII art to present its functionalities to implement a "banner" that heavily influences the count of characters and spaces. To partially mitigate this issue, the blank space ratio should be computed by ignoring comments in the code or by identifying drawings containing characters. Despite an overhead for "sanitizing" the `.js` files, this could open up to threats cloaking payload in comments or ASCII-intensive assets, e.g., `Base64`-encoded attack routines [27].

7.2. Homoglyphs

As expected, the number of encodings confirmed as the most sensitive indicator for spotting the presence of trojan source leveraging homoglyphs. In fact, we observed a generalized absence of "alien" characters, i.e., we found that samples using alternative encodings are close to zero throughout the development history of the npm packages in our dataset.

For the case of the 100 packages trojanized via homoglyphs, such characters were extremely rare: out of 41,903 tags, only 156 versions from 10 projects (i.e., 0.3% of tags) contained characters requiring additional or uncommon encodings. Table 3 shows a compact snapshot on the behavior of encodings for four representative npm packages in our dataset. Despite differences in the development process, rules, and habits, each package consistently exhibits a lack of homoglyphs, i.e., 0. The only exception is the last version when plagued with the hidden trojan payload, which has been denoted in bold and with an asterisk. Here, the value of the metric increases to 4, providing a clear indication of manipulation. The table also reports the number of characters for the various source files composing a release. As shown, the metric is too general to offer a reliable mechanism to discriminate whether an increasing amount of entries should be ascribed to additional functionalities or malicious data. Our framework also generated false positives for 5 packages in our dataset. By manually inspecting them, we found three main causes: mathematical symbols, localization files, and potential (benign) copy-pastes. Specifically, the first affects `Fraction.js` and `num2fraction` that contain algorithms or numerical/mathematical operations, e.g., the developer inserted comments describing the implemented functionalities. The second concerns `Mastodon` and `Rocket.Chat` where text localization (i.e., translation) is offered for some components, implying the presence of characters with different encodings. The third has been observed in `Bootstrap` and it is probably due to copy-pasting from files in other formats, such as PDFs, which introduce non-visible characters.

Therefore, our preliminary analysis showcased that the evolution of encodings can be considered a valid metric for detecting threats leveraging homoglyphs. Yet, the number of false positives could be non-negligible for packages requiring mathematical symbols or gathering developers from different countries (e.g., those using cyrillic characters). A possible improvement may analyze the abstract syntax tree to discriminate whether homoglyphs have been used in functions or variable names.

8. Conclusions

In this paper, we investigated a prime mechanism for the identification of trojans directly injected in source code via blank spaces and homoglyphs to impair manual software review and inspection practices. To this aim, we investigated seven metrics commonly deployed for code analysis and we introduced three ad-hoc indicators. Our analysis targeted 2,800 npm packages and revealed that metrics capturing standard code evolution behaviors (e.g., lines of code) are not effective to spot malicious payloads. Modern repositories characterized by a fast-paced development process require more precise indicators, such as the presence of an abnormal burst of blank spaces. Results highlight the need for lightweight and scalable tools for processing large code bases, which characterize modern software supply chains.

However, this work is still preliminary and has some limitations. First, even if we used 100 realistic npm packages, we evaluated our approach on synthetic malicious releases. Since the attack template is known *a priori*, the design of the metrics could lead to an array of highly-specialized indicators. Moreover, an attacker could employ more sophisticated offensive techniques, hindering the identification process offered by an approach based on evolutionary metrics. Additionally, we considered only JavaScript, thus whether our idea can be generalized to different languages requires further investigation.

Future research aims at refining our approach, especially to go beyond the aforementioned limitations and to validate the metrics in more realistic scenarios. Moreover, part of our ongoing research is devoted to exploring automated detection strategies, especially to deploy repair mechanisms for sanitizing code without requiring human intervention. A relevant part of our efforts aims to integrate our approach into a minimal detection pipeline.

Acknowledgments

This work was partially supported by Project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] Ö. A. Aslan, R. Samet, A comprehensive review on malware detection approaches, IEEE Access 8 (2020) 6249–6271.
[2] E. Debas, N. Alhumam, K. Riad, Unveiling the Dynamic Landscape of Malware Sandboxing: A Comprehensive Review, International Journal of Advanced Computer Science and Applications 15 (2024).
[3] J. Busch, A. Kocheturov, V. Tresp, T. Seidl, NF-GNN: Network Flow Graph Neural Networks for Malware Detection and Classification, in: Proceedings of the 33rd International Conference on Scientific and Statistical Database Management, 2021, p. 121–132.
[4] J. Gu, H. Zhu, Z. Han, X. Li, J. Zhao, GSEDroid: GNN-based Android Malware Detection Framework Using Lightweight Semantic Embedding, Computers & Security 140 (2024) 103807.
[5] D. Devadiga, G. Jin, B. Potdar, H. Koo, A. Han, A. Shringi, A. Singh, K. Chaudhari, S. Kumar, Gleam: GAN and LLM for Evasive Adversarial Malware, in: 2023 14th International Conference on Information and Communication Technology Convergence, 2023, pp. 53–58.
[6] L. Caviglione, M. Choraś, I. Corona, A. Janicki, W. Mazurczyk, M. Pawlicki, K. Wasielewska, Tight Arms Race: Overview of Current Malware Threats and Trends in Their Detection, IEEE Access 9 (2020) 5371–5396.
[7] MITRE, MITRE ATT&CK, https://attack.mitre.org/, 2024.
[8] J. A. Marpaung, M. Sain, H.-J. Lee, Survey on Malware Evasion Rechniques: State of the Art and Challenges, in: 2012 14th International Conference on Advanced Communication Technology, 2012, pp. 744–749.
[9] R. Dyer, H. A. Nguyen, H. Rajan, T. N. Nguyen, Boa: Ultra-Large-Scale Software Repository and Source-Code Mining, ACM Trans. Softw. Eng. Methodol. 25 (2015).
[10] P. Ladisa, H. Plate, M. Martinez, O. Barais, SoK: Taxonomy of Attacks on Open-Source Software Supply Chains, in: 2023 IEEE Symposium on Security and Privacy, 2023, pp. 1509–1526.
[11] Y. Chen, J. Wu, X. Ling, C. Li, Z. Rui, T. Luo, Y. Wu, When Large Language Models Confront Repository-level Automatic Program Repair: How Well They Done?, in: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, 2024, pp. 459–471.
[12] Aikido Security, Ratatouille: A malicious recipe hidden in rand-user-agent, https://www.aikido.dev/blog/catching-a-rat-remote-access-trojian-rand-user-agent-supply-chain-compromise, 2024.
[13] D. L. Vu, I. Pashchenko, F. Massacci, H. Plate, A. Sabetta, Towards Using Source Code Repositories to Identify Software Supply Chain Attacks, in: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 2093–2095.
[14] A. Zerouali, T. Mens, G. Robles, J. M. Gonzalez-Barahona, On the Diversity of Software Package Popularity Metrics: An Empirical Study of npm, in: 2019 IEEE 26th international conference on software analysis, Evolution and Reengineering, 2019, pp. 589–593.
[15] Sonatype, Inc., State of the Software Supply Chain, https://www.sonatype.com/state-of-the-software-supply-chain/introduction, 2024.
[16] D. Oliveira, R. Santos, F. Madeiral, H. Masuhara, F. Castor, A Systematic Literature Review on the

Impact of Formatting Elements on Code Legibility, Journal of Systems and Software 203 (2023) 111728.
[17] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, 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, 2022, pp. 331–340.
[18] N. Boucher, R. Anderson, Trojan Source: Invisible Vulnerabilities, in: Proceedings of the 32nd USENIX Security Symposium, 2023.
[19] E. Buchicchio, L. Grilli, E. Capobianco, S. Cipriano, D. Antonini, Invisible Supply Chain Attacks Based on Trojan Source, Computer 55 (2022) 18–25.
[20] B. Hammi, S. Zeadally, Software Supply-Chain Security: Issues and Countermeasures, Computer 56 (2023) 54–66.
[21] M. Zimmermann, C.-A. Staicu, C. Tenny, M. Pradel, Small World with High Risks: A Study of Security Threats in the npm Ecosystem, in: Proceedings of the 28th USENIX Security Symposium, 2019, pp. 995–1010.
[22] M. Ohm, H. Plate, A. Sykosch, M. Meier, Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, 2020, pp. 23–43.
[23] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, W. Lee, Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages, in: 28th Annual Network and Distributed System Security Symposium, 2021.
[24] P. Kintis, N. Miramirkhani, C. Lever, Y. Chen, R. Romero-Gómez, N. Pitropakis, N. Nikiforakis, M. Antonakakis, Hiding in Plain Sight: A Longitudinal Study of Combosquatting Abuse, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 569–586.
[25] Z. Tan, S. P. Parambath, C. Anagnostopoulos, J. Singer, A. K. Marnerides, Advanced Persistent Threats Based on Supply Chain Vulnerabilities: Challenges, Solutions, and Future Directions, IEEE Internet of Things Journal 12 (2025) 6371–6395.
[26] Tristan F.-R., npm-rank, https://tristan-f-r.github.io/npm-rank/PACKAGES.html, 2024.
[27] L. Caviglione, W. Mazurczyk, Never Mind the Malware, Here's The Stegomalware, IEEE Security & Privacy 20 (2022) 101–106.