# (Nothing But) Many Eyes Make All Bugs Shallow

Elizabeth Wyss
University of Kansas
Lawrence, KS, USA
ElizabethWyss@ku.edu

Lorenzo De Carli
University of Calgary
Calgary, CA
Lorenzo.DeCarli@ucalgary.ca

Drew Davidson
University of Kansas
Lawrence, KS, USA
DrewDavidson@ku.edu

## ABSTRACT

Open source package repositories have become a crucial component of the modern software supply chain since they enable developers to easily and rapidly import code written by others. However, low quality, poorly vetted code residing in such repositories exposes developers and end-users to dangerous bugs and vulnerabilities at a large scale.

Such issues have recently led to the creation of government-backed verification standards pertaining to packages, as well as a significant body of developer folklore regarding what constitutes a reliable package. However, there exists little academic research assessing the relationships between recommended development practices and known package issues in this domain.

Motivated by this gap in understanding, we conduct a large-scale study that formally evaluates whether adherence to these guidelines meaningfully impacts reported issues and bug maintenance activity across the most widely utilized npm packages (encompassing **7,162** packages with over **100K** weekly downloads each), which unveiled wide disparities across package-level metrics.

We find that it is only recommendations pertaining to a broad notion of *scrutiny* that provide strong and reliable insights into the reporting and resolving of package issues. These findings pose significant implications for developers, who seek to identify well-maintained packages for use, as well as security researchers, who seek to identify suspicious packages for critical observation.

## CCS CONCEPTS

• **Software and its engineering** → **Software libraries and repositories**; **Software verification and validation**.

## KEYWORDS

software supply chain, open-source, package repositories

**ACM Reference Format:**
Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. 2023. (Nothing But) Many Eyes Make All Bugs Shallow. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '23), November 30, 2023, Copenhagen, Denmark*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3605770.3625216



*SCORED '23, November 30, 2023, Copenhagen, Denmark*

ACM ISBN 979-8-4007-0263-1/23/11.
https://doi.org/10.1145/3605770.3625216

## 1 INTRODUCTION

Modern software infrastructure often relies on third-party code dependencies, known as *packages*, residing in open source repositories. The rapid rise of these language-based package ecosystems highlights their widespread use and importance to the software development community. The largest of such language-based repositories is npm [70] for Node.js, which serves billions of weekly downloads of more than two million unique packages, all the while continually growing at a rate of nearly one thousand new packages per day [20].

npm's frontend package manager allows developers to easily import packages into their own codebases via a simple command line interface, which in turn has enabled wide-scale code sharing, extensive code reuse, and rapid software development cycles. Despite these advantages, this very process has also enabled the propagation of dangerous bugs and vulnerabilities, as illustrated by the almost three thousand common vulnerabilities and exposures (CVEs) which have been derived from npm packages alone [6].

Attempting to avoid these problems, the software development community surrounding npm asserts considerable claims regarding how to select a reliable package [9, 12, 54, 61, 74], but these claims are largely untested in empirical settings. In a similar vein, recent national cybersecurity initiatives [1] have led the U.S. government to issue official development standards [13], aimed at catching vulnerabilities and other software flaws.

This paper seeks to formally evaluate whether adherence to these guidelines meaningfully impacts the quality-driven outcomes of packages–including reported issues and bug maintenance activity. This particular notion of quality serves to aid users in avoiding software defects, which may expose attack surfaces to adversaries along the software supply chain.

We encounter several challenges in our work. A first challenge exists in defining a representative and effective dataset for our analysis. npm is a treasure trove of package data, but also an extremely noisy one, containing large amounts of empty and unused packages [68]. We opt to focus on the most frequently downloaded npm packages because they have the greatest overall impacts and represent the most typical use cases. Packages that garner more than 100K weekly downloads account for the majority of all package downloads in npm [68], and they are thus representative of general repository usage. Applying this insight, we extract a wide range of package metrics for 7,162 of the most widely-utilized npm packages, which each boast weekly download counts ranging from 100K to 191M. Finally, we utilize the bug maintenance activity and the issues reported against these packages to comparatively assess proposed advice and standards.

A second challenge of this study is that different software metrics may measure related, overlapping aspects, and thus may not be independent. It is important to distinguish metrics that truly

correlate with the reporting and resolving of package issues from metrics that only appear to do so due to the effects of confounding factors. We perform careful statistical analysis, leading to the identification of a small set of metrics which robustly and independently correlate with issue reports and bug maintenance activity.

Our study offers several benefits to the software development community. First, it critically assesses conventional guidance so as to provide developers with empirically-backed recommendations for identifying highly maintained and reliable packages to use in their projects. Moreover, it aids both package maintainers and security researchers alike in identifying packages which are under-scrutinized and thus more attractive for critical observation.

From this study, we uncover wide disparities across package-level metrics, issue reports, and bug maintenance amongst the most widely utilized npm packages. Our findings identify specific metrics, which we relate to a broad notion of *scrutiny*, that is strongly and independently associated with the reporting and resolving of package issues. Importantly, we also find that, when controlling for the observed effects of our notion of scrutiny, other package metrics, recommendations from conventional wisdom, and official standards offer very few insights into reported issues and bug maintenance.

Not only do our findings support Linus' Law [60], a common software development adage which posits that many eyes make all bugs shallow–they ultimately suggest that a better alternative has yet to be found. Motivated by these results, we propose that developers and security researchers alike would benefit from employing scrutiny-based metrics to select the packages that they use and critically analyze.

Overall, the contributions of this work are as follows:

- We construct a large-scale dataset of metrics pertaining to the most widely-utilized npm packages, which we publicly release via the Open Science Framework [3] for further analysis, study, and replicability[1]
- We evaluate the effectiveness of presently suggested package metrics and standards with respect to their impacts on the reporting and resolving of issues across npm packages.
- We propose recommendations and metrics for identifying npm packages that are highly scrutinized, and conversely, packages which might benefit from additional critical observation. Measured correlation coefficients demonstrate that packages with more contributors, forkers, and total commits are strongly associated (**+0.703** to **+0.732**) with increased issue reports relative to use, as well as moderately associated (**+0.475** to **+0.529**) with a proportional increase in commit activity dedicated to resolving issues.

# 2 BACKGROUND AND RELATED WORK

## 2.1 Software Defect Detection

Related to this work is the field of software defect detection, which seeks to characterize [24, 31, 36, 37, 40, 49, 50, 83] and discover [44, 73, 76] flaws in software, i.e. bugs. Many works [33, 51, 62, 64, 85] utilize software metrics to predict when and where defects occur across the lifecycle of software development, and more modern prediction approaches often deploy machine learning [10, 56].

[1]https://osf.io/mtrsj/?view_only=0bef73c2fb6d4363b72aa3c54fbefd22

Some works [24, 32, 66] have employed known software defects as a means of assessing software quality metrics. Spinellis et al. [66] propose a general system for calculating, storing, and assessing software quality via modular metrics, which they employ to identify key quality metrics pertaining to code complexity and software development communities. Greiler et al. [32] investigate software quality and identify significant relationships between known defects and project ownership structures.

We differ from these works both in our choice of domain and our overarching goal. Rather than seeking to predict where bugs occur or assess our own software quality metrics, our focus is on evaluating existing advice and standards which purport to impact defects encountered by npm users.

## 2.2 Characterization of Package Repositories

Also similar to our work is the body of research analyzing the properties and evolution of various package repositories [18, 29, 59, 75]. Researchers have discovered significant interdependence between packages [22, 39, 75], particularly in npm, where 93% of package code resides in third-party dependencies [41]. The highly-interdependent nature of these repositories has been shown to further propagate bugs and vulnerabilities across packages through their direct and indirect dependencies [25, 47, 78].

Studies into the nature of package popularity [23, 24, 48, 82] find wide download and usage disparities across packages. Within npm, most packages are practically never downloaded by real users (i.e. they are almost exclusively downloaded by bots and crawlers which routinely download the entire package registry), and the top less-than-1% of packages garner the majority of all package downloads [68].

Other works investigate and characterize specific phenomena discovered in package registries. Two works by Abdalkareem et al. [7, 8] analyze the use and impacts of *trivial* packages, which contain few lines of code and exhibit only very simple functionality. Wyss et al. [77] investigate the phenomenon of package cloning within npm, and Zerouali et al. [81] explore how npm packages delay updating their dependencies for significant periods of time.

Expanding upon these past works, we conduct a study which employs a broad range of key package metrics to formally assess conventional developer wisdom and government-backed development standards within a large sample of the most widely-utilized npm packages.

## 2.3 Software Supply Chain Security

There exist numerous security challenges intrinsic to open source package repositories and the software supply chains they enable. Many works demonstrate how flaws in code dependencies are leveraged to compromise the security and stability of both package repositories and the overall process of software development [11, 16, 17, 43, 55, 57, 70, 84]. The security impacts of malicious, compromised, or otherwise poorly vetted dependencies encompasses a significant body of work [15, 21, 34, 42, 58, 69, 79], and there further exists extensive and ongoing efforts to detect malicious and vulnerable packages through metadata and code-based signals [2, 25, 27, 28, 46, 53, 67, 68, 80].

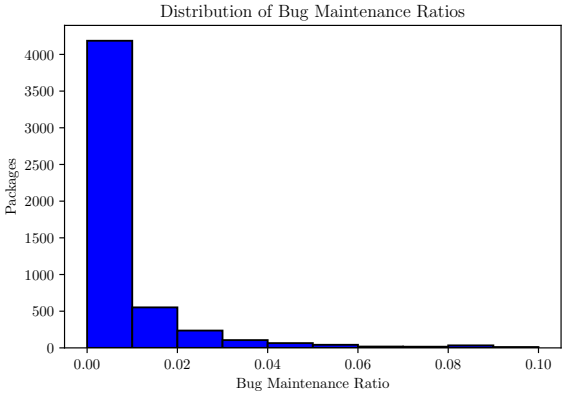


**Figure 1: Distribution of bug maintenance ratios across our package dataset.**

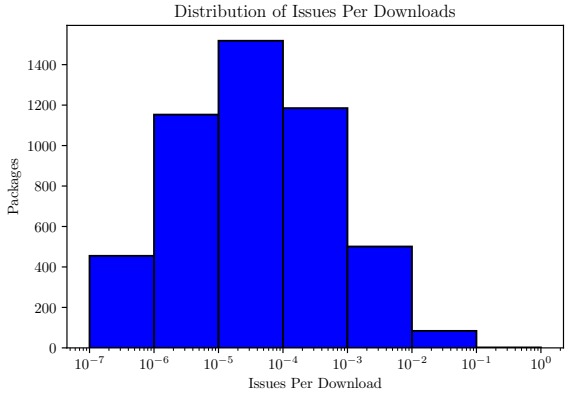


**Figure 2: Distribution of issues per downloads across our package dataset. Note the logarithmic scale on the x-axis.**

Our work has important security implications; identifying packages, metrics, and advice which are more effectively scrutinized helps to strengthen the integrity of the supply chain, and identifying those which are less effectively scrutinized helps to prioritize targets for critical observation. Moreover, our work benefits user experience, as well-scrutinized advice and standards aids developers in selecting high-quality, reliable packages for their own use.

## 3 OVERVIEW

Our study seeks to assess whether adherence to presently recommended advice and standards impacts the quality-driven outcomes of npm packages. We first describe the methodology of our study, including the construction of our dataset and the selection of our target metrics for correlative analysis. Then, we describe the processes we use to identify existing recommendations and the package metrics we extract from such recommendations.

### 3.1 Methodology

For the purposes of our study, we seek to build and analyze a large-scale dataset encompassing a wide range of recommended package metrics for the most popular npm packages. We choose to focus our study on the most popular npm packages–those which garner more than 100,000 weekly downloads–for two key reasons: (i) packages above this popularity threshold account for the majority of all npm package downloads [68] and thus are representative of npm's most typical use cases; and (ii) the most highly utilized packages have the greatest impacts on the software development community.

**Dataset Construction**:

At the initial time of our analysis (May 31st, 2022), we identified a total of 9,341 unique npm packages that met our popularity threshold and thus formed our preliminary package dataset.

The npm package repository provides rich metadata for each package in the form of a standardized *package.json* file, which we employ as a source of package metrics relating to usage, dependencies, versioning, size, and licensing. The overwhelming majority (93%) of these packages declare an associated GitHub repository in their metadata, which we extract and further employ as a source of package metrics relating to repository usage, project workflows, and development history.

We collect additional repository-level metrics from the Open Source Security Foundation (OSSF) Scorecards project [4], which analyzes GitHub workflows to determine whether repositories engage in development practices such as having security and code-review policies, inventorying their code dependencies, and using known static analysis and fuzzing tools. We successfully extracted associated GitHub repositories and collected such metrics for 8,644 of the packages in our dataset.

In some cases, multiple packages declare the same associated GitHub repository (for example, the `lodash` package, and its submodule, `lodash.isequal`, both declare *github.com/lodash/lodash* as their associated GitHub repository). To avoid duplicating or incorrectly separating out repository-level metrics across multiple packages, we merge packages which declare the same associated GitHub repository into a single, project-level data point for the purposes of conducting statistical analyses. Across our dataset, we identify a total of 1,863 packages which declare the same GitHub repository as another package in our dataset.

We perform additional analysis on the codebases[2] of packages to identify automated testing scripts, linting scripts, and use of built-in protections. Code-level metrics were successfully extracted for 83% of analyzed packages, with failures resulting from codebases that failed to generate valid intermediate representations needed for analysis, which typically occured due to nonstandard environmental dependencies that could not be automatically resolved. A cursory analysis of failed packages revealed that they spanned a representative range of package sizes, download counts, development histories, and other package metrics. Thus, we believe discarding those packages which could not be analyzed retains a representative sample of the most popular npm packages, which is sufficient for the aims of our study.

[2]Past work [71, 72] has identified that package code residing in GitHub is not always a perfect match of package code residing in the package manager. We account for this potential concern by obtaining all package code directly from the package manager. We only utilize GitHub to collect metadata regarding packages and their development histories.

| Label | Target Metrics | Description |
|---|---|---|
| *BMR.* | Bug Maintenance Ratio | Total bug references across all commits normalized per unit commit |
| *IPD.* | Issues Per Download | Total GitHub issues normalized per weekly download count |
| | **Developer-Proposed Metrics** | |
| *D.1.* | Weekly Downloads | Total package downloads from npm measured per week |
| *D.2.* | Dependencies | Total package dependencies |
| *D.3.* | Reverse Dependencies | Total packages that depend on the given package |
| *D.4.* | Size | Physical package size, measured in bytes |
| *D.5.* | Versions* | Total available package versions |
| *D.6.* | Vulnerabilities | Total disclosed vulnerabilities across all package versions |
| *D.7.* | License | Whether package declares a license |
| *D.8.* | Commits* | Total GitHub commits |
| *D.9.* | Contributors* | Total GitHub contributors |
| *D.10.* | Stars* | Total GitHub stars |
| *D.11.* | Forks* | Total GitHub forks |
| *D.12.* | Open Issues* | Total open GitHub issues |
| *D.13.* | Open Pull Requests* | Total open GitHub pull requests |
| *D.14.* | Security Policy* | Whether repository declares a security policy |
| *D.15.* | Code-Review Policy* | Whether repository declares a code-review policy |
| *D.16.* | Code Coverage Integration* | Whether repository is integrated with popular code coverage reporting framework |
| | **NIST Guidelines Associated Metrics** | |
| *N.1.* | Testing Framework* | Whether package declares automated testing scripts |
| *N.2.* | Static Analysis Tooling* | Whether repository is integrated with known static analysis frameworks |
| *N.3.* | Linting Framework | Whether package declares automated linting scripts |
| *N.4.* | Fuzzing Tooling* | Whether repository is integrated with known fuzzing frameworks |
| *N.5.* | Code Coverage Integration* | Whether repository is integrated with popular code coverage reporting framework |
| *N.6.* | Uses Strict Mode | Whether package utilizes Javascript's built-in strict mode checks |
| *N.7.* | Dependency-Update Tooling* | Whether repository is integrated with known dependency updating frameworks |
| *N.8.* | Dependency Pinning* | Whether repository inventories dependency versions at release |
| *N.9.* | Code-Review Policy* | Whether repository declares a code-review policy |

*Indicates Scrutiny-Related Metric

**Table 1: Index of our dataset's package metrics and their descriptions, separated by category.**

This process resulted in a final dataset containing complete package metrics for 7,162 npm packages spanning 5,299 unique GitHub repositories. A total index of analyzed package metrics by category, including labels and descriptions, is presented in Table 1.

We find that many of the metrics in our dataset pertain to scrutiny–a notion we employ to broadly refer to critical observation and engagement with a codebase–which we investigate further in Section 4. We note that this notion of scrutiny is subject to interpretation, which we discuss in Section 5.2 For clarity of presentation, we denote metrics we interpret as scrutiny-related in Table 1.

**Target Metrics**:

To characterize the effectiveness of existing recommendations, we first identify target metrics by which we will assess the quality of presently recommended advice and standards. To select our target metrics, we draw upon the body of research literature on software quality metrics.

Such works often evaluate a software's quality based on the discovery of defects across its development lifecycle, which are typically extracted from official bug tracking databases, e.g. Bugzilla, to supply ground truth for known software defects [33, 66, 83, 85]. One particular challenge inherent to our domain of npm is that there exists no official database for tracking defects discovered within npm packages (save for the small fraction of package vulnerabilities which have associated CVE numbers [6]). As such, we employ heuristics to broadly infer package defects from the GitHub repositories associated with those packages.

One potential source for identifying package defects is their GitHub commit history, which we utilize to infer defects from commit messages that contain textual references to discovered bugs. We note numerous past works that employed similar techniques to extract software defects from version control system logs [19, 26, 32, 65, 86]. Another potential source of package defects is GitHub issues, which past work has also employed [24].

An additional challenge intrinsic to our domain of npm is that fairly comparing discovered defects across packages is made difficult by vastly different levels of usage and development histories which span asynchronously across the period of more than thirteen years since the inception of npm. We overcome this challenge by

| WDs | Bug Maintenance Ratio | | | Issues Per Download | | |
|---|---|---|---|---|---|---|
| Part. | Avg. | Med. | SD | Avg. | Med. | SD |
| 1 | 0.0083 | 0.0 | 0.022 | 0.0018 | 1.6e−4 | 0.0087 |
| 2 | 0.0088 | 0.0 | 0.021 | 0.0012 | 9.3e−5 | 0.0048 |
| 3 | 0.0070 | 0.0 | 0.015 | 7.0e−4 | 4.7e−5 | 0.0031 |
| 4 | 0.0074 | 0.0 | 0.020 | 2.3e−4 | 1.2e−5 | 0.0011 |
| 5 | 0.0059 | 0.0 | 0.013 | 2.8e−5 | 1.5e−6 | 1.3e−4 |

| CMTs | Bug Maintenance Ratio | | | Issues Per Download | | |
|---|---|---|---|---|---|---|
| Part. | Avg. | Med. | SD | Avg. | Med. | SD |
| 1 | 0.0045 | 0.0 | 0.023 | 2.9e−5 | 1.7e−6 | 5.4e−4 |
| 2 | 0.0064 | 0.0 | 0.018 | 4.0e−5 | 7.1e−6 | 3.4e−4 |
| 3 | 0.0075 | 0.0 | 0.016 | 9.8e−5 | 3.0e−5 | 1.7e−4 |
| 4 | 0.0076 | 0.0035 | 0.016 | 3.4e−4 | 9.3e−5 | 6.7e−4 |
| 5 | 0.011 | 0.0065 | 0.019 | 0.0034 | 8.1e−4 | 0.010 |

**Table 2: Descriptive statistics of our target metrics across partitions of our dataset. Partitions along weekly downloads are depicted in the left table, and partitions along commits are depicted in the right table.**

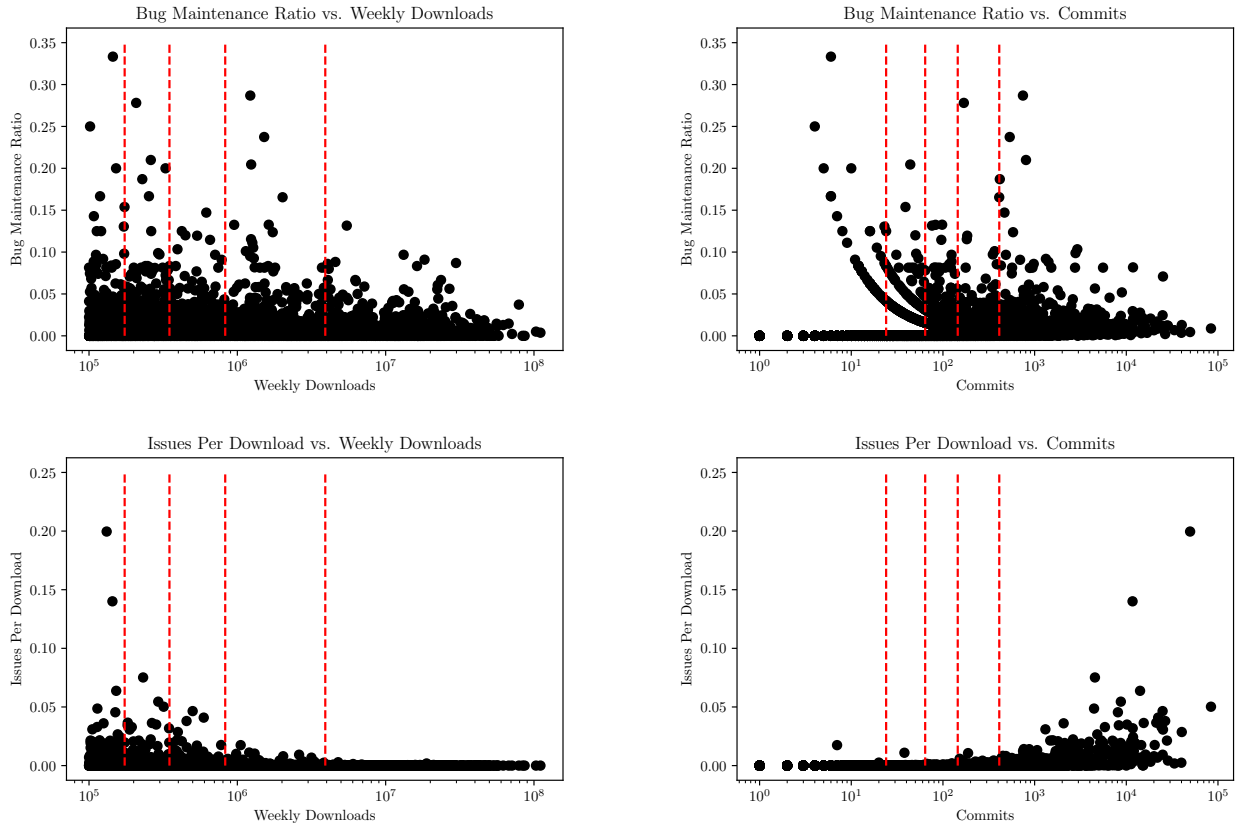


**Figure 3: Scatter plots of our target metric distributions across partitions of our dataset, with partition boundaries denoted by the dashed red lines. Note the logarithmic scale on the x-axes.**

| WDs | Bug Maintenance Ratio | | Issues Per Download | |
|---|---|---|---|---|
| Parts. | U-result | Significant? | U-result | Significant? |
| 1-2 | 578,293.5 | no | 508,398 | yes |
| 2-3 | 510,415 | no | 451,681 | yes |
| 3-4 | 529,858 | yes | 378,592 | yes |
| 4-5 | 596,726.5 | no | 327,061.5 | yes |

| CMTs | Bug Maintenance Ratio | | Issues Per Download | |
|---|---|---|---|---|
| Parts. | U-result | Significant? | U-result | Significant? |
| 1-2 | 619,260 | yes | 751,107.5 | yes |
| 2-3 | 591,939 | yes | 699,025.5 | yes |
| 3-4 | 641,633.5 | yes | 736,057 | yes |
| 4-5 | 777,720.5 | yes | 921,044 | yes |

**Table 3: Mann-Whitney U tests assessing differences in the distributions of our target metrics across partitions of our dataset. Partitions along weekly downloads are depicted in the left table, and partitions along commits are depicted in the right table.**

applying normalizations that account for such disparities in the construction of our target metrics.

To construct a software quality metric that is adjusted for total development history, we measure the total quantity of GitHub commits whose messages contain the word "bug" or "bugs", divided by the total quantity of GitHub commits, which we term the bug maintenance ratio of a package. This metrics is representative of the overall proportion of commits dedicated to addressing bugs, with larger values indicating a greater fraction of commits which handle bugs. Figure 1 presents the distribution of bug maintenance ratios across our package dataset. Most packages exhibit a bug maintenance ratio between 0.0 and 0.01, and only a few hundred packages garner bug maintenance ratios greater than 0.06, indicating that packages generally dedicate only a small proportion of their total commit activity to bug maintenance.

Additionally, Dey et al. [24] propose a usage-independent software quality metric, defined as the total quantity of GitHub issues per package download, which we utilize as an additional target metric. This metric, which we term as issues per download, is representative of user-provided defect reports relative to package use, with larger values indicating higher proportions of users reporting defects. We present the distribution of issues per downloads across our package dataset in Figure 2. Examined packages appear to exhibit a roughly normal distribution of issues per downloads centered around 0.00005, indicating that generally, proportionally few package users report encountering issues to package maintainers via GitHub.

To investigate whether our target metrics reveal interesting and distinguishing properties of packages, we examine their distributions across partitions of our dataset by usage and development history. We adopt a methodology similar to past work [36] that has analyzed the distributions of software issues relative to usage: first, we divide our dataset into a set of five equal-sized partitions, ranging from the lowest fifth of weekly download counts to the highest fifth of weekly download counts. We additionally create a second set of partitions along total package commits, following the same process. We present descriptive statistics of our target metrics across these partitions in Table 2, and we plot the distributions of our target metrics across these partitions–alongside the partition boundaries–in Figure 3.

Plotted distributions reveal that both of our target metrics generally increase with total commits and decrease with weekly downloads. We observe greater dispersion in the values of both of our target metrics within the lowest partition of both commits and weekly downloads, as well as much greater dispersion in the values of issues per downloads within the highest partition of commits. To determine whether the distributions of our target metrics across partitions demonstrate statistically significant differences, we conduct Mann-Whitney U tests, of which we present the results in Table 3. The results of our Mann-Whitney U tests demonstrate significant differences in the distributions of each target metric across adjacent partitions, except for the bug maintenance ratios across the first through third, and the fourth through fifth, weekly downloads partitions. The statistical significance of these distributions indicates that our target metrics do meaningfully distinguish packages across our dataset.

In tandem, our target metrics are representative of user experience and package maintenance. Higher values of issues per download indicate that proportionally more package users are reporting (and thus encountering) issues in their package usage experience, and higher values of bug maintenance ratio indicate that a package is dedicating proportionally more of its development history to the maintenance of defects. As such, these target metrics enable us to empirically assess whether existing advice and standards for packages provide meaningful insights into user experience and package maintenance, which we investigate via correlative analysis in Section 4.

There is no uniform standard in qualitatively describing the strength of correlation coefficients [35]. For our study, we refer to the strength of measured correlations as very weak when their magnitude is less than 0.2, weak when at least 0.2 but less than 0.4, moderate when at least 0.4 but less than 0.6, strong when at least 0.6 but less than 0.8, and very strong at or above 0.8. Throughout our study, we use Spearman correlations since the distribution of many metrics is decidedly non-normal.

## 3.2 Existing Recommendations

Our study seeks to analyze the existing body of recommendations regarding best practices in the use and development of npm packages, which includes both conventional developer wisdom and official software development standards. Providing an empirical backing from measurable impacts on our target metrics would help to validate this existing body of folklore and standards, thus building well-evidenced, actionable advice.

**Developer-Proposed Metrics**:

There exists a substantial body of developer folklore concerning what constitutes a 'good' npm package. Some past research has also interviewed developers about their processes for selecting third-party libraries [14, 38, 45], or proposed systems for recommending third-party libraries [52]. However, the effectiveness of developer-proposed recommendations remains largely untested within the academic literature. We seek to bridge this gap by investigating whether package metrics derived from conventional developer wisdom impact our target metrics.

Because npm is a rapidly evolving ecosystem, we opt to conduct a fresh analysis of developer-proposed metrics, which we compare against past research for completeness. To identify the software development community's proposed metrics for evaluating packages, we query the Google search engine for articles written by software developers detailing processes and metrics they utilize to select npm packages. We use the query terms "how to choose the right npm package", "choosing high quality npm packages", and "selecting the best npm package" to gather articles that meet the criteria of (i) being written by software developers and (ii) describing programatically measurable, actionable metrics. We employ this selection criteria so that the metrics we obtain are actively being recommended within the software development community and are feasible to calculate for our analysis. For each article, we extract and measure all suggested metrics, continuing this process until a steady state of extracted metrics is reached and remains unchanged for three consecutive articles. This process converged after just five articles [9, 12, 54, 61, 74]. Below, we describe the extracted metrics.

We find that developers value packages that are highly downloaded and depended on, have more version releases, are smaller in size, and declare fewer dependencies [9, 54, 61, 74]. We encapsulate these recommendations as metrics in weekly download counts, reverse dependencies, version releases, package size, and dependencies reported by npm metadata. We also find that developers value larger and more active GitHub communities, particularly those which harbor greater commits, contributors, stars, and forks, as well as fewer unresolved (open) issues and pull requests [9, 12, 54, 61, 74]. We obtain such metrics from packages' associated GitHub repositories. Further, we find that developers are concerned about known vulnerabilities in packages [9, 12, 54, 61, 74], which we collect from npm's advisory database [6]. Lastly, we find that developers value procedural metrics, including whether a package declares a license, a security policy, a code-review policy, and is integrated with a code coverage reporting framework [9, 54, 61, 74]. We obtain licensure information from npm metadata. The presence of a security policy is determined as whether a package's repository declares a standardized *SECURITY.md* file. We detect the presence of a code-review policy by leveraging analysis performed by the OSSF Scorecards project [4] which determines whether a package's GitHub repository is configured to require code-review prior to merging a pull request. Finally, integration with a code coverage reporting framework is encapsulated as whether a package's GitHub repository employs CodeCov [63], the most popular code coverage reporting application on the GitHub Marketplace [30].

Many of the developer-proposed metrics we identify are also noted by past research on third-party library selection, including GitHub stars, forks, contributors, commits, issues, pull requests, releases, and licensure [14, 45], as well as download counts [38, 45] and dependency relationships [45, 52].

We further analyze these developer-proposed metrics and assess how they relate to our target metrics in Section 4.1.

**Government-Backed Software Standards**:

As part of the May 2021 U.S. Executive Order on Improving the Nation's Cybersecurity, The National Institute of Standards and Technology (NIST) published official guidelines and standards for enhancing software supply chains and their security [5], which includes recommended minimal standards for developer verification of code [13]. NIST is an institution mandated by the U.S. government to set effective standards, making these guidelines the most prominent and current government-backed software standards that directly apply to supply chain components such as npm packages.

Since these standards designate minimally recommended development practices, it is applicable to investigate how they relate to our target metrics. Hence, we utilize NIST's standards to derive high-level, associated metrics, which we employ as an additional source of currently recommended package metrics. Below, we describe each of these standards and our associated package metrics.

NIST's automated testing standard involves the programmatic execution of unit tests, which we detect by identifying the presence of unit-testing scripts declared within npm package metadata.

NIST's standard for code-based static analysis encompasses the usage of static analysis tools to uncover bugs, hard-coded secrets, and other design flaws. Our associated metric utilizes analysis performed by OSSF Scorecards [4] to detect the presence of known static analysis applications in the GitHub workflows of packages. Additionally, NIST's standard includes linting tools, which we infer by identifying the presence of linting scripts declared by packages.

NIST's dynamic analysis standards involve the utilization of fuzzing tools, code coverage reporters, and built-in programming language protections provided to validate the execution of code. To detect fuzzing tools, we utilize OSSF Scorecards' [4] analysis which identifies known fuzzing applications in the GitHub workflows of packages. We identify code coverage reporting from GitHub workflow integrations with CodeCov [63]. Usage of built-in protections is identified by scanning package code for JavaScript's strict mode, which provides greater dynamic checks and error reporting.

NIST's standards for checking included software and dependencies involves inventorying and monitoring included software components, which we detect by identifying the presence of pinned-dependencies and dependency update tools. Pinned-dependencies are an up-to-date list of exact dependency versions published at package release, and dependency update tools raise pull requests for critical dependency updates, both of which we extract from analysis conducted by the OSSF Scorecards project [4].

Lastly, NIST's standard for code review and bug fixing encompasses development processes specifically designed to catch and prevent bugs. Our associated metric identifies whether packages are configured to require code review prior to merging pull requests, as determined by OSSF Scorecards [4].

Section 4.2 assesses how our operationalizations of these NIST standards correlate with our target metrics, in addition to how often they are followed by the most popular npm packages.

# 4 RESULTS

This section presents the findings of our study on package recommendations. Subsection 4.1 evaluates developer-proposed metrics, and Subsection 4.2 examines NIST-recommended standards.

## 4.1 Developer-Proposed Metrics

Table 4 presents identified developer-proposed metrics and their Spearman correlations with our target metrics.

We find the strongest and most reliable correlations between our target metrics and the metrics which we relate to scrutiny. This notion of scrutiny includes both project insiders, such as GitHub contributors–which have a +0.520 correlation with bug maintenance ratio and a +0.711 correlation with issues per download–as well as project outsiders, such as forkers–which have a +0.475 correlation with bug maintenance ratio and a +0.732 correlation with issues per download. Additional metrics relating to this notion of scrutiny include commits, open issues, version releases, stars, open pull requests, and the presence of security and code-review policies, which tend to demonstrate moderate to very strong correlations with our target metrics.

Outside of scrutiny-related metrics, other developer-proposed metrics have only very weak to moderate correlations with our target metrics. This finding highlights an important distinction between popularity and our notion of scrutiny. Packages with greater download counts tend to experience a use-relative decrease in defect reports. This is evidenced by the moderately negative (-0.489) correlation between weekly downloads and issues per download.

| | | | | Bug Maintenance Ratio | | Issues Per Download | |
|---|---|---|---|---|---|---|---|
| **Label** | **Developer-Proposed Metric** | **Avg.** | **SD** | **Corr** | **Adj. Corr** | **Corr** | **Adj. Corr** |
| *D.1.* | Weekly Downloads | 3 480 151.01 | 7 597 647.21 | -0.044 | -0.060 | -0.489 | -0.773 |
| *D.2.* | Dependencies | 2.54 | 5.36 | +0.189 | +0.008[x] | +0.324 | +0.139 |
| *D.3.* | Reverse Dependencies | 556.65 | 2770.16 | +0.229 | -0.074 | +0.251 | -0.395 |
| *D.4.* | Size | 658 096.85 | 4 538 186.03 | +0.458 | +0.170 | +0.637 | +0.267 |
| *D.5.* | Versions | 42.55 | 135.56 | +0.466 | +0.100 | +0.650 | +0.151 |
| *D.6.* | Vulnerabilities | 0.15 | 1.07 | +0.130 | +0.024[x] | +0.101 | -0.100 |
| *D.7.* | License | 0.97 | 0.18 | +0.022[x] | -0.019[x] | +0.016[x] | -0.046 |
| *D.8.* | Commits | 574.84 | 2375.21 | +0.529 | +0.170 | +0.703 | +0.197 |
| *D.9.* | Contributors | 44.42 | 168.17 | +0.520 | N/A | +0.711 | N/A |
| *D.10.* | Stars | 2329.04 | 8805.06 | +0.436 | -0.007[x] | +0.705 | +0.100 |
| *D.11.* | Forks | 354.69 | 2036.78 | +0.475 | N/A | +0.732 | N/A |
| *D.12.* | Open Issues | 46.67 | 188.76 | +0.422 | +0.065 | +0.717 | +0.329 |
| *D.13.* | Open Pull Requests | 7.6 | 20.22 | +0.330 | +0.027 | +0.571 | +0.203 |
| *D.14.* | Security Policy | 0.14 | 0.35 | +0.096 | +0.001[x] | +0.058 | -0.108 |
| *D.15.* | Code-Review Policy | 0.29 | 0.45 | +0.235 | -0.020[x] | +0.356 | +0.047 |
| *D.16.* | Code Coverage Integration | 8.93 | 29.03 | +0.088 | +0.006[x] | +0.110 | +0.008[x] |

[x]Note: Statistically Insignificant Correlation ($p > 0.05$)

**Table 4: Developer-proposed package metrics alongside their means, standard deviations, and Spearman correlations with our target metrics, both raw and adjusted for confounds of scrutiny.**

**Adjusting for Scrutiny**:

Motivated by the overwhelming impacts of metrics we relate to scrutiny and the underwhelming impacts of other metrics, we examine whether measured correlations meaningfully change and reveal any additional insights if we adjust for the effects of our notion of scrutiny by treating it as a confounding factor.

First, we identify a total set of potential scrutiny confounds, which includes versions, contributors, stars, forks, pull requests, security policy, code-review policy, and code coverage integration[3].

We then construct Random Forest regression models trained to predict our target metrics based on the values of our potential scrutiny confounds, and we extract average feature importance measures from the trained models.

This analysis revealed that contributors and forks represent the most distinct and impactful forms of our notion of scrutiny, accounting for the largest effects. Intuitively, this finding makes sense given that contributors are representative of critical observation of codebases by project insiders, and forks are representative of critical observation of codebases by project outsiders. Applying this finding, we define scrutiny-adjusted correlations as Spearman partial correlations which control for contributors and forks as confounding factors. Table 4 presents the scrutiny-adjusted correlations between developer-proposed metrics and our target metrics.

After controlling for the effects of our notion of scrutiny, developer-proposed metrics correlate only very weakly with bug maintenance ratio. As for issues per downloads, this scrutiny-adjustment causes the magnitude of correlations with popularity metrics to increase significantly. We observe a -0.773 scrutiny-adjusted correlation between issues per download and weekly downloads, as well as a -0.395 scrutiny-adjusted correlation between issues per download and reverse dependencies, which is a measure of how many packages depend on the given package–another form of package popularity. This finding demonstrates that defect reports relative to use is impacted heavily by both popularity and our notion of scrutiny, while the proportion of commit activity dedicated to bug fixing is most impacted by our notion of scrutiny alone. Ultimately, we find that developer-proposed metrics–outside of popularity and our notion of scrutiny–provide very little insight into package quality as guided by our target metrics.

[3]Although commits and issues also relate to our notion of scrutiny, we exclude them from the potential set of scrutiny confounds since they are factored into the calculations of our target metrics, which could bias the identification of confounding factors.

## 4.2 NIST Guidelines

We further analyze presently suggested package metrics by assessing our operationalizations of NIST's recently proposed guidelines concerning minimal standards for developer verification of software [13]. Table 5 summarizes NIST's standards and our associated package metrics, including their prevalence across examined packages and their Spearman correlations with our target metrics.

Surprisingly, many of the most widely utilized npm packages do not follow these instantiations of the minimal verification standards recommended by NIST. This result reveals large gaps between the current state of open source software development and official minimal standards for software verification.

With respect to their effects on bug maintenance ratios and issues per downloads, metrics associated with NIST's standards have very weak to moderate correlations with our target metrics. After adjusting for the effects of our notion of scrutiny, all of these correlations level out to very weak at best. This finding highlights that standards such as these can serve as further representations of of our notion of scrutiny, but they provide few additional insights into package quality as guided by our target metrics.

| | | | | | | Bug Maintenance Ratio | | Issues Per Download | |
|---|---|---|---|---|---|---|---|---|---|
| **NIST Guideline** | **Label** | **Associated Metric(s)** | **Avg.** | **SD** | **Prevalence** | **Corr** | **Adj. Corr** | **Corr** | **Adj. Corr** |
| Automated Testing | *N.1.* | Testing Framework | 0.75 | 0.43 | 74.86% | -0.071 | -0.029 | -0.151 | -0.113 |
| Static Analysis | *N.2.* | Static Analysis Tooling | 0.040 | 0.19 | 3.74% | +0.105 | +0.022$^{x}$ | +0.132 | +0.021$^{x}$ |
| | *N.3.* | Linting Framework | 0.34 | 0.48 | 34.44% | +0.112 | -0.025$^{x}$ | +0.191 | +0.031 |
| Dynamic Analysis | *N.4.* | Fuzzing Tooling | 0.0 | 0.02 | 0.06% | +0.034 | +0.016$^{x}$ | +0.037 | +0.012$^{x}$ |
| | *N.5.* | Code Coverage Integration | 0.12 | 0.32 | 11.64% | +0.095 | +0.010$^{x}$ | +0.114 | +0.007$^{x}$ |
| | *N.6.* | Uses Strict Mode | 0.71 | 0.45 | 71.13% | +0.091 | +0.028 | +0.124 | +0.047 |
| Check Dependencies | *N.7.* | Dep. Update Tooling | 0.30 | 0.46 | 29.55% | +0.215 | -0.007$^{x}$ | +0.373 | +0.151 |
| | *N.8.* | Dep. Pinning | 1.0 | 0.06 | 99.66% | -0.050 | -0.016$^{x}$ | -0.060 | -0.020$^{x}$ |
| Code Review | *N.9.* | Code-Review Policy | 0.29 | 0.45 | 28.65% | +0.235 | -0.020$^{x}$ | +0.356 | +0.047 |

$^{x}$Note: Statistically Insignificant Correlation ($p > 0.05$)

**Table 5: NIST's minimal standards for developer verification of software and associated metrics, alongside their means, standard deviations, prevalence, and Spearman correlations with our target metrics, both raw and adjusted for confounds of scrutiny.**

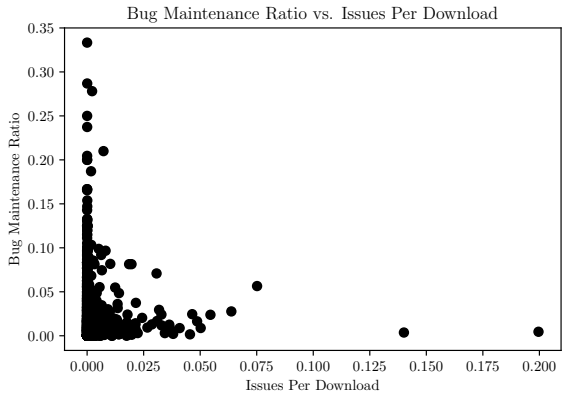


**Figure 4: Scatter plot of bug maintenance ratios versus issues per downloads across our package dataset.**

## 5 DISCUSSION

In this section, we discuss key insights, implications, and potential threats to validity pertaining to our findings.

### 5.1 Implications of Results

One key implication of our findings is that our target metrics can aid developers and researchers alike in identifying packages that are more attractive for use, as well as packages that potentially benefit from additional critical observation.

Packages with larger bug maintenance ratios and issues per downloads likely indicate highly active development communities which are dedicated to finding and fixing bugs, while packages with lower bug maintenance ratios and issues per downloads may indicate a state of relative stability, or perhaps the presence of latent bugs which are yet to be discovered. Meanwhile, packages with higher bug maintenance ratios yet lower issues per downloads likely indicate active development which is effective in catching and resolving defects before they reach end-users, while packages with lower bug maintenance ratios yet higher issues per downloads may indicate the presence of known, but unpatched defects. Although the correlation between bug maintenance ratios and issues per downloads is moderate (+0.433), we observe a sizable spread of packages with bug maintenance ratios that differ greatly from their issues per downloads. The distribution of bug maintenance ratios versus issues per downloads is presented in Figure 4

Packages that are highly active in uncovering and fixing bugs are more desirable for use, especially since we find that they tend to be backed by larger and more active development communities that have more resources for package maintenance, which in turn helps to build trust between package developers and package users. Conversely, packages with suspiciously low bug maintenance ratios or issues per downloads could warrant deeper investigation. Our findings on existing recommendations and our proposed notion of scrutiny can serve to inform and prioritize which packages the npm development community and the software supply chain community utilize and apply additional critical observation to.

### 5.2 Threats to Validity

Below, we discuss potential threats to the validity of our study.

**Construct Validity**:

Specific details in the constructions of our metrics and operationalizations may serve as a source of noise in our results.

Our calculation of bug maintenance ratio has the potential to miss bug fixing commits if they do not specifically mention bugs in their commit message. Additionally, the total number of bugs fixed in a single commit, as well as the amount of development effort encompassed by a single commit, are both obscured from our calculation of bug maintenance ratio, which may be relevant to a more complete understanding of bug fixing activity.

Our calculation of issues per download likely serves as an over-representation of issue reports, as GitHub issues not only encompass true software defects, but also feature requests and sometimes even clarifying questions posed by users.

Metrics which detect the presence of tooling may be incomplete, such as our determination of static analysis, fuzzing, and code

coverage tooling, which only identify specific known frameworks, thus likely undercounting the prevalence of such tooling.

Moreover, identifying which metrics relate to scrutiny, as well as the notion of scrutiny itself, are subject to interpretation. For example, we denote GitHub stars as pertaining to scrutiny since we assume that developers generally employ some critical process for starring repositories (which is supported by existing research [14]). However, we do not denote weekly downloads or reverse dependencies as pertaining to scrutiny because we view them as mere measurements of popularity. Under a different interpretation, it could be reasonably argued that GitHub stars are merely a measurement of popularity, and thus are not scrutiny-related. Rather yet, it could also be reasonably argued that all forms of popularity, including weekly downloads and reverse dependencies, indicate greater engagement and thus scrutiny. We propose one possible notion of scrutiny and an interpretation of how the identified metrics relate to it, but we acknowledge that other reasonable interpretations exist. Different interpretations may impact the definition of scrutiny-adjusted correlations and thus the results of our work.

Although such sources of noise may introduce bias in the results of our study, we believe that our metrics and operationalizations are still representative of the abstract concepts the seek to measure, and thus are useful objects of study.

**Internal Validity**:

Our study identifies significant correlations between package scrutiny, issue reports, and bug maintenance. It is possible that the relationships between these factors are not causative, or could be impacted by factors outside of our study, such as code complexity or project ownership structures, which have been noted by past research on software quality [32, 66]. Further research is needed to investigate causative links and control for such factors.

The developer-supplied link between npm packages and their associated GitHub repositories introduces the potential for an incorrect package-to-repository mapping, either due to developer error or malicious obfuscation, which could bias our results. However, given that our dataset consists only of highly popular npm packages, we believe the likelihood of such incorrect mappings to be low. Also related to package-to-repository mappings is the potential bias introduced by our decision to merge together data points corresponding to multiple packages that declare the same repository, thus changing the fundamental unit of our statistical analyses to be GitHub projects rather than packages, which will likely underreppresent package submodules across our analyses.

Additionally, some packages were excluded from our study because we could not generate complete metrics for them. Although our package dropout rate was only 23.3%, and a cursory manual analysis of dropout packages did not uncover significant distinctions, the exclusion of those packages may bias our results.

**External Validity**:

Our study analyzes highly popular packages within npm, which may impact the generalizability of our findings. Less popular packages, and other open source package repositories could differ in terms of scrutiny, issue reports, and bug maintenance. Further research is needed to investigate whether our results are generalizable to other packages and repositories.

# REFERENCES


[1] 2021. Executive Order on Improving the Nation's Cybersecurity. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/.

[2] 2022. Mining Node.js Vulnerabilities via Object Dependence Graph and Query. In *USENIX Security 22*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity22/presentation/li-song

[3] 2022. Open Science Framework. https://osf.io

[4] 2022. OpenSSF Scorecard. https://github.com/ossf/scorecard

[5] 2022. Software Security in Supply Chains. https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity/software-security-supply-chains

[6] 2023. GitHub Advisory Database. https://github.com/advisories?query=type%3Areviewed+ecosystem%3Anpm

[7] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why Do Developers Use Trivial Packages? An Empirical Case Study on Npm. In *ESEC/FSE 2017* (Paderborn, Germany). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3106237.3106267

[8] Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. 2020. On the impact of using trivial packages: an empirical case study on npm and PyPI. *Empirical Software Engineering* 25 (03 2020). https://doi.org/10.1007/s10664-019-09792-9

[9] Nitai Aharoni. 2020. How to Choose the Right NPM Package for Your Project. https://betterprogramming.pub/how-to-choose-the-right-npm-package-for-your-project-c3d1cc25285e

[10] Saiqa Aleem, Luiz Fernando Capretz, and Faheem Ahmed. 2015. Benchmarking machine learning technologies for software defect detection. *arXiv preprint arXiv:1506.07563* (2015).

[11] Anish Athalye, Rumen Hristov, Tran Nguyen, and Qui Nguyen. 2014. *Package Manager Security*. Technical Report. https://pdfs.semanticscholar.org/d398/d240e916079e418b77ebb4b3730d7e959b15.pdf

[12] Adrian Bece. 2019. Checklist for choosing an optimal npm package. https://dev.to/adrianbdesigns/checklist-for-choosing-an-optimal-npm-package-4dpm

[13] Paul E. Black, Vadim Okun, and Barbara Guttman. 2021. Guidelines on Minimum Standards for Developer Verification of Software. https://doi.org/10.6028/NIST.IR.8397

[14] Hudson Borges and Marco Túlio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *CoRR* abs/1811.07643 (2018). arXiv:1811.07643 http://arxiv.org/abs/1811.07643

[15] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. 2015. Tracking known security vulnerabilities in proprietary software systems. In *SANER*.

[16] Justin Cappos, Justin Samuel, Scott Baker, and John H Hartman. 2008. A look in the mirror: Attacks on package managers. In *CCS '08*. 565–574.

[17] Kyriakos Chatzidimitriou, Michail Papamichail, Themistoklis Diamantopoulos, Michail Tsapanos, and Andreas Symeonidis. 2018. Npm-miner: An infrastructure for measuring the quality of the npm registry. In *MSR 2018*. IEEE, 42–45.

[18] Filipe Roseiro Cogo, Gustavo A. Oliva, and Ahmed E. Hassan. 2021. An Empirical Study of Dependency Downgrades in the npm Ecosystem. *IEEE Transactions on Software Engineering* 47, 11 (2021). https://doi.org/10.1109/TSE.2019.2952130

[19] D. Cubranic and G.C. Murphy. 2003. Hipikat: recommending pertinent software development artifacts. In *ICSE 2003*. https://doi.org/10.1109/ICSE.2003.1201219

[20] Erik DeBill. 2021. Modulecounts. http://www.modulecounts.com/

[21] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *MSR 2018*.

[22] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems. *ESE* 24 (2019). https://doi.org/10.1007/s10664-017-9589-y

[23] Tapajit Dey and Audris Mockus. 2018. Are Software Dependency Supply Chain Metrics Useful in Predicting Change of Popularity of NPM Packages?. In *PROMISE'18*. 66–69. https://doi.org/10.1145/3273934.3273942

[24] Tapajit Dey and Audris Mockus. 2020. Deriving a usage-independent software quality metric. *ESE* 25 (2020). https://doi.org/10.1007/s10664-019-09791-w

[25] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *NDSS 2021*. Internet Society.

[26] M. Fischer, M. Pinzger, and H. Gall. 2003. Populating a Release History Database from version control and bug tracking systems. In *ICSM 2003*. 23–32. https://doi.org/10.1109/ICSM.2003.1235403

[27] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2019. Detecting suspicious package updates. In *ICSE-NIER 2019*. IEEE, 13–16.

[28] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. 2018. AFFOGATO: Runtime Detection of Injection Attacks for Node.Js. In *ISSTA/ECOOP 2018*. https://doi.org/10.1145/3236454.3236502

[29] Daniel M German, Bram Adams, and Ahmed E Hassan. 2013. The evolution of the R software ecosystem. In *CSMR*.

[30] GitHub. 2023. GitHub Application Marketplace; Code Coverage. https://github.com/marketplace?type=apps&query=code+coverage+sort%3Apopularity-desc+

[31] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. 2000. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering* 26, 7 (2000). https://doi.org/10.1109/32.859533

[32] Michaela Greiler, Kim Herzig, and Jacek Czerwonka. 2015. Code Ownership and Software Quality. In *MSR 2015*. https://doi.org/10.1109/MSR.2015.8

[33] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. 2010. Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. In *ICSE 2010*. https://doi.org/10.1145/1806799.1806871

[34] Joseph Hejderup. 2015. *In Dependencies We Trust: How vulnerable are dependencies in software modules?* Master's thesis. Delft University of Technology.

[35] James Hemphill. 2003. Interpreting the Magnitude of Correlation Coefficients. *The American psychologist* 58 (02 2003), 78–9.

[36] Israel Herraiz, Emad Shihab, Thanh Nguyen, and Ahmed E. Hassan. 2011. Impact of Installation Counts on Perceived Quality: A Case Study on Debian. 219–228. https://doi.org/10.1109/WCRE.2011.34

[37] J.P. Hudepohl, S.J. Aud, T.M. Khoshgoftaar, E.B. Allen, and J. Mayrand. 1996. Emerald: software metrics and models on the desktop. *IEEE Software* 13, 5 (1996), 56–60. https://doi.org/10.1109/52.536459

[38] Wenxin Jiang, Nicholas Synovic, Matt Hyatt, Taylor R. Schorlemmer, Rohan Sethi, Yung-Hsiang Lu, George K. Thiruvathukal, and James C. Davis. 2023. An Empirical Study of Pre-Trained Model Reuse in the Hugging Face Deep Learning Model Registry. arXiv:2303.02552 [cs.SE]

[39] Jaap Kabbedijk and Slinger Jansen. 2011. Steering Insight: An Exploration of the Ruby Software Ecosystem. In *Software Business*. Springer Berlin Heidelberg.

[40] T.M. Khoshgoftaar, E.B. Allen, N. Goel, A. Nandi, and J. McMullan. 1996. Detection of software modules with high debug code churn in a very large legacy system. In *ISSRE '96*. https://doi.org/10.1109/ISSRE.1996.558896

[41] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node.js Applications. In *RAID 2020*. USENIX Association. https://www.usenix.org/conference/raid2020/presentation/koishybayev

[42] R. G. Kula, C. D. Roover, D. German, T. Ishio, and K. Inoue. 2014. Visualizing the Evolution of Systems and Their Library Dependencies. In *IEEE VISSOFT*.

[43] Trishank Karthik Kuppusamy, Santiago Torres-Arias, Vladimir Diaz, and Justin Cappos. 2016. Diplomat: Using delegations to protect community repositories. In *NSDI 16*.

[44] Filippo Lanubile and Giuseppe Visaggio. 2000. Evaluating defect detection techniques for software requirements inspections. *ISERN* (2000).

[45] Enrique Larios Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. 2020. Selecting Third-Party Libraries: The Practitioners' Perspective. In *ESEC/FSE 2020*. https://doi.org/10.1145/3368089.3409711

[46] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2021. *Detecting Node.Js Prototype Pollution Vulnerabilities via Object Lookup Analysis*. 268–279. https://doi.org/10.1145/3468264.3468542

[47] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. *arXiv preprint arXiv:2201.03981* (2022).

[48] Suhaib Mujahid, Rabe Abdalkareem, and Emad Shihab. 2022. What are the characteristics of highly-selected packages? A case study on the npm ecosystem. https://doi.org/10.48550/ARXIV.2204.04562

[49] N. Nagappan and T. Ball. 2005. Use of relative code churn measures to predict system defect density. In *ICSE 2005*. https://doi.org/10.1109/ICSE.2005.1553571

[50] Nachiappan Nagappan and Thomas Ball. 2007. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In *ESEM 2007*. https://doi.org/10.1109/ESEM.2007.13

[51] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining Metrics to Predict Component Failures. In *ICSE '06*. https://doi.org/10.1145/1134285.1134349

[52] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, and Massimiliano Di Penta. 2020. CrossRec: Supporting software developers by recommending third-party libraries. *JSS* 161 (2020). https://doi.org/10.1016/j.jss.2019.110460

[53] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. 2019. Nodest: Feedback-Driven Static Analysis of Node.Js Applications. In *ESEC/FSE 2019*. https://doi.org/10.1145/3338906.3338933

[54] Kranti Nikam. 2022. Screening NPM Packages: Best Practices. https://medium.com/globant/screening-npm-packages-best-practices-a24930b2624e

[55] Chinenye Okafor, Taylor R. Schorlemmer, Santiago Torres-Arias, and James C. Davis. 2022. SoK: Analysis of Software Supply Chain Security by Establishing Secure Design Properties. In *SCORED*. https://doi.org/10.1145/3560835.3564556

[56] Logan Perreault, Seth Berardinelli, Clemente Izurieta, and John Sheppard. 2017. Using classifiers for software defect detection. In *SEDE 2017*.

[57] Brian Pfretzschner and Lotfi ben Othmane. 2017. Identification of Dependency-based Attacks on Node.Js. In *ARES*.

[58] H. Plate, S. E. Ponta, and A. Sabetta. 2015. Impact assessment for vulnerabilities in open-source software libraries. In *ICSME*.

[59] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2013. The maven repository dataset of metrics, changes, and dependencies. In *MSR*.

[60] Eric S Raymond. 1999. The Cathedral and the Bazaar. http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/ar01s04.html

[61] Alexis Regnaud. 2021. 7 Tools to Choose the Right NPM Package. https://javascript.plainenglish.io/7-tools-to-choose-the-right-npm-package-7baf47259ae0

[62] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. 2006. Predicting Component Failures at Design Time. In *ISESE '06*. https://doi.org/10.1145/1159733.1159739

[63] Sentry. 2023. Codecov.io. https://about.codecov.io/

[64] Yonghee Shin, Robert Bell, Thomas Ostrand, and Elaine Weyuker. 2009. Does calling structure information improve the accuracy of fault prediction?. In *MSR 2009*. https://doi.org/10.1109/MSR.2009.5069481

[65] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes?. In *MSR '05*. https://doi.org/10.1145/1083142.1083147

[66] Diomidis Spinellis, Georgios Gousios, Vassilios Karakoidas, Panagiotis Louridas, Paul J. Adams, Ioannis Samoladas, and Ioannis Stamelos. 2009. Evaluating the Quality of Open Source Software. *Electronic Notes in Theoretical Computer Science* 233 (2009). https://doi.org/10.1016/j.entcs.2009.02.058

[67] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2018. SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS. In *NDSS*.

[68] Matthew Taylor, Ruturaj Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. 2020. Defending Against Package Typosquatting. In *NSS 2020*.

[69] Jørgen Tellnes. 2013. *Dependencies: No Software is an Island*. Master's thesis. The University of Bergen.

[70] Ruturaj K. Vaidya, Lorenzo De Carli, Drew Davidson, and Vaibhav Rastogi. 2019. Security Issues in Language-based Sofware Ecosystems. *CoRR* abs/1903.02613 (2019). arXiv:1903.02613 http://arxiv.org/abs/1903.02613

[71] D. Vu. 2021. py2src: Towards the Automatic (and Reliable) Identification of Sources for PyPI Package. In *ASE*. https://doi.org/10.1109/ASE51524.2021.9678526

[72] Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. 2021. LastPyMile: Identifying the Discrepancy between Sources and Packages. In *ESEC/FSE 2021*. https://doi.org/10.1145/3468264.3468592

[73] Stefan Wagner. 2006. A literature survey of the quality economics of defect-detection techniques. In *ISESE '06*.

[74] Yasas Sri Wickramasinghe. 2021. 5 Best Practices to Choosing Third-Party NPM Packages. https://blog.bitsrc.io/5-best-practices-when-choosing-third-party-npm-packages-2198994357f9

[75] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A look at the dynamics of the JavaScript package ecosystem. In *MSR*.

[76] Murray Wood, Marc Roper, Andrew Brooks, and James Miller. 1997. Comparing and combining software defect detection techniques: a replicated empirical study. In *ESEC/FSE'97*.

[77] Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. 2022. What the Fork? Finding Hidden Code Clones in npm. In *ICSE 2022*. https://doi.org/10.1145/3510003.3510168

[78] Elizabeth Wyss, Alexander Wittman, Drew Davidson, and Lorenzo De Carli. 2022. Wolf at the Door: Preventing Install-Time Attacks in Npm with Latch. In *ASIA CCS '22*. https://doi.org/10.1145/3488932.3523262

[79] A. A. Younis, Y. K. Malaiya, and I. Ray. 2014. Using Attack Surface Entry Points and Reachability Analysis to Assess the Risk of Software Vulnerability Exploitability. In *HASE*.

[80] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. 2022. What are Weak Links in the npm Supply Chain?. In *ICSE-SEIP 2022*. https://doi.org/10.1145/3510457.3513044

[81] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesus Gonzalez-Barahona. 2018. An Empirical Analysis of Technical Lag in npm Package Dependencies. https://doi.org/10.1007/978-3-319-90421-4_6

[82] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus M. Gonzalez-Barahona. 2019. On the Diversity of Software Package Popularity Metrics: An Empirical Study of npm. In *SANER 2019*. https://doi.org/10.1109/SANER.2019.8667997

[83] Thomas Zimmerman, Nachiappan Nagappan, Kim Herzig, Rahul Premraj, and Laurie Williams. 2011. An Empirical Study on the Relation between Dependency Neighborhoods and Failures. In *ICST '11*. https://doi.org/10.1109/ICST.2011.39

[84] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *USENIX Security 19*.

[85] Thomas Zimmermann and Nachiappan Nagappan. 2008. Predicting defects using network analysis on dependency graphs. In *ICSE 2008*. https://doi.org/10.1145/1368088.1368161

[86] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting Defects for Eclipse. In *PROMISE '07*. https://doi.org/10.1109/PROMISE.2007.10