Doctoral Dissertation

Characterizing the Quality of Third-Party Libraries through Runnability and Risk Assessment in the Open Source Ecosystem

Bodin Chinthanet
Program of Information Science and Engineering
Graduate School of Science and Technology
Nara Institute of Science and Technology

Supervisor: Kenichi Matsumoto
Software Engineering Lab. (Division of Information Science)

Submitted on September 21, 2021

A Doctoral Dissertation
submitted to Graduate School of Science and Technology,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of Engineering

Bodin Chinthanet

Characterizing the Quality of Third-Party Libraries through Runnability and Risk Assessment in the Open Source Ecosystem[*]

Bodin Chinthanet

Abstract

Third-party library usage becomes the important part of software development within the open source community. These library packages provide software developers with useful features without the need to "reinvent the wheel", with each package often depending on several others. Since there are millions of packages available online, the understanding of the package quality is needed for choosing the suitable package in the software development project.

This thesis characterizes the package quality through (1) the package selection and (2) the package security risk. The first part of this thesis finds how to choose the good package from user and contributor perspectives through the developer survey and the analysis of package runnability. The results show that both users and contributors share similar views on how to assess the package quality. Runnability of the package could be used for choosing the good package. The second part of this thesis investigates the risk of vulnerability in the package through the vulnerability fix adoption analysis and the code-centric vulnerability detection. The results show that lags of the fix adoption are affected by factors (i.e., severity and freshness). Additionally, most of the vulnerable codes are not reachable in the application.

Keywords:

Open Source Software, Software Libraries, Software Ecosystem, Mining Software Repository, Security Vulnerabilities

[*]Doctoral Dissertation, Graduate School of Science and Technology, Nara Institute of Science and Technology, September 21, 2021.

Acknowledgements

First, I would like to express my deep gratitude to Prof. Kenichi Matsumoto for giving me opportunities to study in his laboratory during the past five years as an intern, master's student, and Ph.D. student. He also provides me guidance and encouragement during my Ph.D. journey. This thesis would never have been accomplished without his generous support.

I would like to express my gratitude to Assist. Prof. Raula Gaikovina Kula for teaching, suggesting, and sharing many things from the beginning of my study in Japan. He also helps me relieve the stress and solve the problems while struggling with research and daily life.

I would also like to express my gratitude to the rest of my thesis committee, including Prof. Hajimu Iida, Assoc. Prof. Takashi Ishio, Assoc. Prof. Shane McIntosh, and Dr. Akinori Ihara. They give me invaluable comments and suggestions to improve the quality of my research.

I would like to extend my gratitude and appreciation to Assoc. Prof. Hideaki Hata, Dr. Christoph Treude, and Dr. Markus Wagner for their insightful discussions to improve my research skills.

During my internship at SAP Labs France, Dr. Serena Elisa Ponta, Dr. Henrik Plate, and Dr. Antonino Sabetta were great mentors and showed how researchers could work in a big company. I am very grateful for the opportunity to work with them. Special thanks to my colleagues at SAP, who assisted me with many things during my stay in France.

I sincerely thank Shade Ruangwan and Rungroj Maipradit, who have been like brothers to me, for sharing their experiences and advice. I also thank my colleagues and staff in NAIST for the delightful time during my study.

Special thanks to Pattaraporn Tulathum for always being by my side. We went through many obstacles during this long journey, and finally, I am able to see the end of the tunnel. I am sure that she will see the light at the end of her Ph.D. journey soon.

Last but not least, I would like to thank my family for their love, support, and encouragement to study until now. Without them, I would not have had a chance to pursue my dream in Japan.

List of Publications

- **Lags in the Release, Adoption, and Propagation of npm Vulnerability Fixes**
  Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. Empirical Software Engineering (EMSE), 26(3): 1-28, 2021. (Accepted as a journal paper)
  - Received SIGSE Distinguished Paper Award 2021.
- **Code-based Vulnerability Detection in Node.js Applications: How far are we?**
  Bodin Chinthanet, Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Raula Gaikovina Kula, Takashi Ishio, Kenichi Matsumoto. In Proceeding of the 35th International Conference on Automated Software Engineering (ASE): Industry Showcase, 2020. (Accepted as a conference paper)
- **SōjiTantei: Function-Call Reachability Detection of Vulnerable Code for `npm` Packages**
  Bodin Chinthanet, Raula Gaikovina Kula, Rodrigo Eliza Zapata, Takashi Ishio, Kenichi Matsumoto, Akinori Ihara. IEICE Transactions on Information and Systems, Vol.E105-D, No.1, pp.-, Jan. 2022. (Accepted as a letter)

Contents

List of Figures

List of Tables

1 Introduction

Software ecosystem is a collection of software packages, which are developed, shared, and co-evolve in the same environment [25, 75, 78]. Packages in the ecosystem are reusable, allowing the new software applications to use them as third-party libraries. The usage of libraries in the ecosystem forms a network of packages. The examples of existing software ecosystems are npm (JavaScript ecosystem), Maven (Java ecosystem), and RubyGems (Ruby ecosystem).

Third-party library usage becomes an important part in the software ecosystem as developers could reuse functionalities from existing packages in their own applications [144]. The popularity of library usage is confirmed by GitHub, as around 90% of JavaScript packages depend on at least one open source component [46]. Using those packages not only prevents reinventing the wheel, but also saves the cost of implementing the new features [144]. By using the available package manager for each ecosystem, developers can easily access those millions of packages at their fingertip. For example, npm (i.e., Node.js package manager) provides the access of more than a million packages on the registry for Node.js developers to find and install packages in their applications [90].

Despite the benefits of third-party library usages, those packages have a potential to introduce maintenance issues in the application [8]. For example, updating the new release of packages with backward incompatible changes is likely to break the applications that rely on those changed functions [14]. On the other hand, keeping the outdated packages can potentially increase the risk from bugs and security vulnerabilities [51, 64, 73].

In order to ensure the value of the package and avoid potential negative consequences, developers have to concern about the quality of the package [40, 69, 144].

From the international organization for standardization (ISO), ISO25010 was published to define the quality aspects for developing a software-based on static properties of software and dynamic properties of the computer system [40]. The defined aspects are (1) functional suitability, (2) performance efficiency, (3) compatibility, (4) usability, (5) reliability, (6) security, (7) maintainability, and (8) portability. This standard helps developers in identifying and validating quality requirements during the software development process.

Security vulnerability, one of quality aspects, is a rising concern in the software development life cycle from academia, government, and industry [42, 94, 97]. As shown by the recent studies, the risks from vulnerabilities are not only posed to the direct user of libraries, but also the transitively affected through the dependency network [32, 51, 64, 70, 73]. In order to eliminate the vulnerability in the application, the US government introduces secure software development framework (SSDF) which is a set of practices for creating secure software [97]. One of the key practices in the framework is that third-party libraries should comply with security requirements. This practice could be realized by using industrial grade tools available for detecting vulnerabilities and mitigating the risks in their applications [47, 94, 108, 109, 122].

As there are millions of packages available in the ecosystem, it is challenging for developers to choose a good packages that have high-quality for their application. Thus, the theme of this thesis is to find the characterization of package quality in addition to ISO25010.

1 Problem Statement

Choosing a package is important for application development as the quality of the application depends on the quality of the chosen packages [40, 69, 144]. However, recent studies show that developers choose packages arbitrarily for their applications [69]. Moreover, other studies show that a small set of security vulnerabilities can cause a cascade effect to the entire ecosystem [51, 64, 73]. These increase the risks of encountering security issues while carelessly choose packages. Developers have to choose a high-quality package to avoid risks; however, they comment that there are too many packages that have the same functionality and hard to measure package quality [143]. In order to find a high-quality package systematically, the characterization of package quality is needed. Therefore, I state this thesis as follows:

Thesis Statement: Choosing a package is important for developers to avoid risks posed by vulnerabilities. However, it is hard for them to measure the quality of those packages and adopt high-quality ones. Hence, the characterization of package quality is needed for finding a suitable package in the software development project.

To validate this thesis, I focus on the package quality characterization through the analysis of packages in the software ecosystem from two different dimensions:

Package Selection. In the software ecosystem, developers now have an easy access to millions of packages at their fingertips. Developers could use the package manager to automatically find and install desired packages. The interested audience of the package is not limited to developers who would like to adopt the package into their application, but also to developers who are interested in contributing to the package. Therefore, investigating of features to quantify the a package quality by analysing the interested audience could contribute the understanding of a good package.

Vulnerabilities and Fixes Assessment. The potential risk of vulnerability is not restricted to the direct users of packages, but it also extends to the broader software ecosystem through the dependency network. To mitigate the risk, developers require to adopt the vulnerability fixing release from the package as soon as it is available. However, developers who use vulnerable packages transitively cannot adopt this fix right away as they have to wait the upstream packages to adopt the fix and propagate to them first. Therefore, it is a challenging task to reduce the lags in adoption and propagation of the fix.

2 Contributions

The main contributions of this thesis can be classified into three categories: empirical observations, survey insights, and tools.

Empirical Observations

1. Package features from the same type have strong positive correlations. Other feature combinations present trade-offs - in particular runnability related features tend to be negatively correlated with other features (Chapter 4).

2. Runnability of the package is predictable with high F1 score. Repository features are particularly important for prediction the runnability (Chapter 4).

3. The vulnerability fix is not always released as its own patch update as it released with unrelated commits (Chapter 5).

4. While the package release the fix as a patch, the client tend to slowly adopt the fix and cause lags (Chapter 6).

5. Fixing releases that occur on the latest lineage and medium severity suffer the most lags (Chapter 6).

6. Creating the bill of materials of Node.js project is not trivial (Chapter 7).

7. Majority of vulnerable functions in dependencies are not reachable from the application (Chapter 7).

Survey Insights

1. User and contributor of packages share similar view on which features they use to assess package quality (Chapter 3).

2. Runnability of packages is one of the most important features for the high-quality package apart from the documentation (Chapter 3).

Tools

1. Implement of a viable code-based vulnerability detection tool for Node.js applications (Chapter 7).

2. Implement of a prototype vulnerable function reachability detection tool for Node.js applications (Chapter 7).

3 Outline

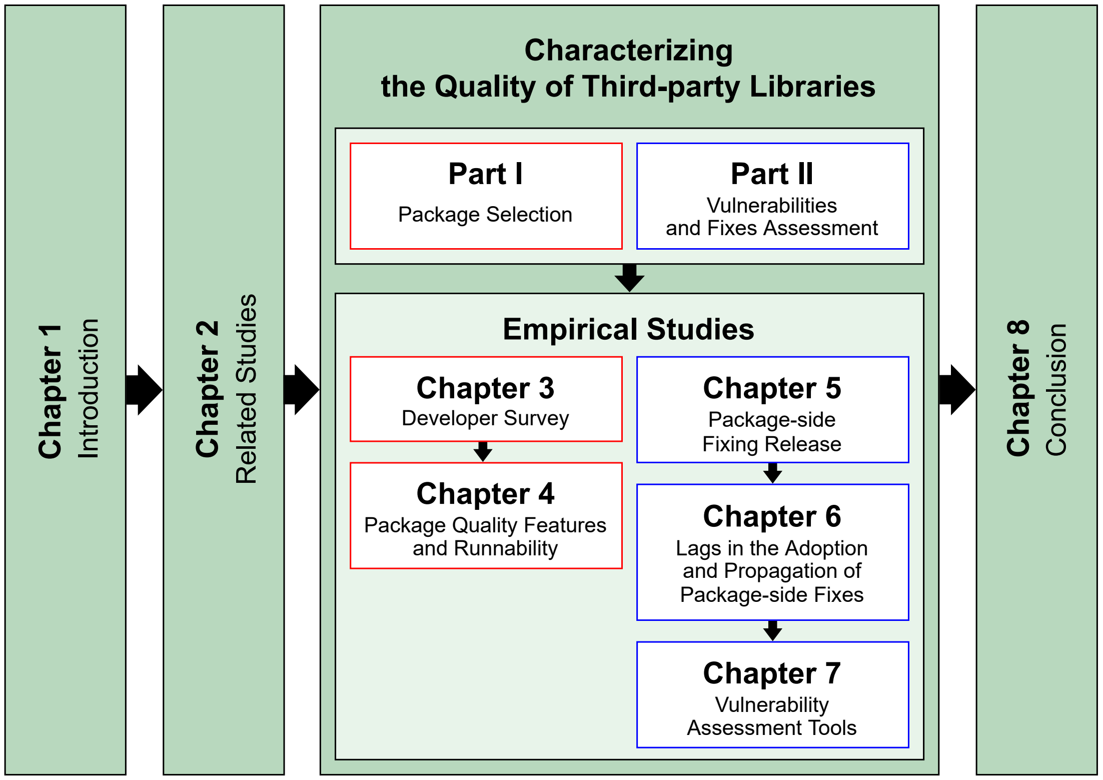

Chapter 1
Introduction
Chapter 2
Related Studies
Characterizing the Quality of Third-party Libraries
Part I
Package Selection
Part II
Vulnerabilities and Fixes Assessment
Empirical Studies
Chapter 3
Developer Survey
Chapter 4
Package Quality Features and Runnability
Chapter 5
Package-side Fixing Release
Chapter 6
Lags in the Adoption and Propagation of Package-side Fixes
Chapter 7
Vulnerability Assessment Tools
Chapter 8
Conclusion

Figure 1.1: An overview of the scope of the thesis.

In this section, I provide an outline of this thesis. Figure 1.1 illustrates the structure of the thesis and the outcomes of each chapter. The details of the rest of this thesis is structured as follows:

Part I: Package Selection. In the second part of this thesis, I present a developer survey and an empirical study to understand what features of packages that users and contributors consider important when searching for a good npm package. I also present the possibility to predict the runnability of the package which considered as one of the important package for assessing the package quality.

Chapter 3 presents a developer survey to understand what features users and contributors consider important when searching for a good npm package including GitHub activity, software usability, and properties of the package repository and documentation. The results show that both users and contributors share similar views on which features they use to assess package quality, especially by using the runnability of packages.

Chapter 4 presents an empirical study to investigate the correlation between package features and to find possibility to predicting the runnability of npm package. The results show that (1) several features negatively correlated with runnability-related features and (2) predicting the runnability of packages is viable.

Part II: Vulnerabilities and Fixes Assessment.

Chapter 5 presents an empirical study to investigate the vulnerability fixing release in packages. The results show that (1) most of the commits are not related to the fix and (2) the fix itself tends to have only a few lines of code.

Chapter 6 presents an empirical study of the adoption and propagation tendencies of fixing releases that impact throughout a network of dependencies. The results show that (1) outdated clients require additional migration effort, even if the fixing release was quick and (2) factors such as the severity of vulnerability and the branch that the fixing release lands on influence the propagation of fixes.

Chapter 7 presents two supporting tools to help developers (1) detecting the vulnerable code and (2) detecting the reachbility of vulnerable function inside their Node.js applications.

Part III: Conclusion. In this last part of the thesis, I focus on the big picture of the studies done in this thesis and discuss about the findings, contributions, and opportunities for future research.

> **Chapter 8** concludes this thesis by discussing the findings from each studies, listing contributions of this thesis, and suggesting the future research opportunities.

2 Related Studies

Complementary related works are introduced throughout the paper, in this section, I discuss some key related works.

Analysis of Repositories - Recent studies have explored dependency networks in different aspects. Some studies investigated the structure and evolution of the dependency networks and revealed their issues such as dependency hell and technical lag [31, 64, 147]. Several studies focused on how to detect known security vulnerabilities from third-party libraries in the applications [17, 146]. Cogo et al. [19] performed an empirical study of dependency downgrades and found that downgrades occur because developers want to avoid some defects from a specific version and some incompatibility issues. Hejderup et al. [52] extracted the call graph for software to build a fine-grained representation of the dependency network. Social coding in the ecosystem is also an emerging research area. Some studies focused on how developers have social interactions with each other [21, 22, 30, 99]. The relationship between different groups of developers in coding collaboration is explored in various ecosystems [38, 49, 99]. Qiu et al. [112] focused on how social coding impacts the chances of long-term engagement. Work such as Blincoe et al. [13] looked at the reference coupling between projects in the ecosystem. Other studies have shown that the maintenance of these dependencies is critical to the ecosystem [32, 68, 83, 146]. ***This thesis*** have analysis in both history and GitHub interaction of repositories. The aim of this thesis is to characterize package quality in terms of package features from different perspectives and package vulnerability fix responses.

Code Snippet Executability - Documentation is an important part of choosing a library [69] and the popularity of GitHub repositories [3], and example code is in itself an important aspect of good documentation. Code snippets are primarily used online in documentation, tutorials, and collaborative sites like Stack Overflow to demonstrate how software and APIs should be used; however, not all code snippets are usable as-is [54, 80, 114, 145]. For developers learning APIs, insufficient or inadequate examples can be a major obstacle [116], and many code snippets online are incomplete, contain errors, or simply do not work. In many cases, code snippets become outdated as software evolves, and despite being the first resource many developers will see, official software documentation is frequently out-of-date, and changes to the software are not immediately reflected in documentation [71]. There is also little support for developers writing documentation, which makes maintaining up-to-date, executable code snippets in GitHub repositories a non-trivial task [26]. Studies of the executability of online code snippets show that most are not executable; a study of online coding tutorials found that only 26% of code snippets could be executed successfully, and no tutorial could be executed to completion [80]. Gistable [54], a framework for running Python code snippets found on GitHub using the gist system, found that only 25% of code snippets were executable by default. These numbers appear to be consistent with other research into code snippet executability [55, 107, 145], with some variance depending on the language. ***This thesis*** expands the knowledge from existing studies by considering the ability to execute code snippets and test cases for assessing the package quality.

Mining-related Studies - These studies relate to the mining techniques in the software repository and software ecosystem with the focus of security vulnerability. The first step of software repository mining is data collection and extraction. Researchers need to have data sources and know about which part of the data can use in their work. In the case of the npm package repository, I can extract the information of packages from `package.json` meta-file [79, 142]. In the case of the security vulnerability, I can collect data from Common Weakness Enumeration (CWE) [82] and Common Vulnerabilities and Exposures (CVE) [81] database [5, 16, 18, 70, 73, 84]. To study the issues within the software ecosystem, I also define the traversal of the downstream clients by using the dependency list of clients. These studies introduce some techniques to model the dependency graph [9, 51, 64]. ***This thesis*** uses similar mining techniques to extract the dependencies as well as construct the ecosystem. Specifically, I manually extract and investigate the commits to understand the contents of the fix.

On Updating Dependencies - These studies relate to the migration of libraries to the latest versions of libraries. With new libraries and newer versions of existing libraries continuously being released, managing a system library dependencies is a concern on its own. As outlined in Bogart et al. [14], Raemaekers et al. [113], Teyton et al. [131], dependency management includes making cost-benefit decisions related to keeping or updating dependencies on outdated libraries. Additionally, Bavota et al. [9], Hora et al. [53], Ihara et al. [59], Robbes et al. [115], Sawant et al. [119] showed that updating libraries and their APIs are slow and lagging. Decan et al. [31] showed the comparison of dependency evolution and issue from three different ecosystems. Their results showed that these ecosystems faced the dependency update issue which causes a problem to downstream dependencies, however, there is no perfect solution for this issue. Kula et al. [68] found that such update decisions are not only influenced by whether or not security vulnerabilities have been fixed and important features have been improved, but also by the amount of work required to accommodate changes in the API of a newer library version. Decan et al. [33] performed an empirical study of technical lag in the npm dependency network and found that packages are suffered from the lags if the latest release of dependencies are not covered by their version ranges in *package.json* file. They also found that semantic versioning could be used in order to reduce the technical lag. Mirhosseini and Parnin [79] studied about the pull request notification to update the dependencies. Their results showed that pull request and badge notification can reduce lags, however, developers are often overwhelmed by lots of notifications. Another study by Abdalkareem et al. [2] focused on the trivial packages and found that they are common and popular in the npm ecosystem. They also suggested that developers should be careful about the selection of packages and how to keep them updated. ***This thesis*** focuses on the lag of the fixing release adoption and propagation with the influencing factors (i.e., lineage freshness and severity). I also expand my study from prior works by taking transitive clients (i.e., downstream propagation) into consideration. My work complements the findings of prior work, with the similar goal of encouraging developers to update.

Malware and Vulnerabilities - These studies relate to the security vulnerability within a software ecosystem from various aspects. Decan et al. [32] explored the impact of vulnerability within the npm ecosystem by analyzing the reaction time of developers from both vulnerable packages and their direct dependent packages to fix the vulnerabilities. They also considered the reactions of developers from different levels of vulnerability severity. They found that the vulnerabilities were prevalent and took several months to be fixed. Several studies also explored the impact of vulnerability within various ecosystems by analyzing the dependency usage [51, 64, 70, 73]. Some studies tried to characterize the vulnerability and its fix in various ecosystems other than the npm ecosystem [72, 106]. There is a study about the relationship between bugs and vulnerabilities, to conclude that the relationship is weak [84]. In order to increase the developers' awareness of the security vulnerability, some studies tried to create a tool to detect and alert vulnerability when it disclosed [16, 44]. There is a study about addressing the over-estimation problem for reporting the vulnerable dependencies in open source communities and the impact of vulnerable libraries usage in the industry [101]. Additionally, some studies tried to predict the vulnerability of software systems by analyzing the source code [5, 18, 120]. ***This thesis*** takes a look at the package-side fixing release and its fix at the commit-level of npm package vulnerabilities, instead of the release-level in prior studies. I also propose a set of definitions to characterize the package-side fixing release and client-side lags, which covers both direct and transitive clients. Prior works, instead, only analyze the direct clients.

Part I

Package Selection

3 Developer Survey on Goodness of Package

1 Introduction

The Node.js Package Manager (i.e., npm) serves as a critical part of the JavaScript community and provides support to one of the largest developer ecosystems in the world, with over 1.7 million packages [90]. These packages provide developers with useful features and libraries without the need to “reinvent the wheel", with each package often depending on several others. Searching for a suitable third-party library is a well-known problem for software developers [143], especially in a massive network like the npm ecosystem.

Most research revolves around selection criteria based on quantifying a library goodness of fit for a certain scenario. For example, works such as LibRec [133] and other proposed library recommendation techniques present a scenario where a developer is interested in using a library that has also been used in other similar projects [29, 57, 87, 98, 100, 118]. Other related work focuses on the motivations behind why a software developer selects a package. For example, Larios Vargas et al. [69] argue that software developers often choose libraries arbitrarily, without considering the consequences of their decisions. These studies confirm that developers do struggle with library selection and updates [34, 57, 102, 139, 149]. In all of these works, the common assumption is that the *interested audience* of a library is a typical software developer who would like to adopt the package into their application.

Unlike related work, the novelty of this work is to provide a comprehensive investigation of features from previous work to quantify the goodness of a package by analysing the interested audience perspective. Based on the literature, I identified the audience from two overlapping perspectives:

1. *Users* - who are looking to adopt a package into their applications. They are typically software developers [11, 69], who interested in how well documented a package is and how it can be integrated into their existing project.

2. *Contributors* - who are interested in contributing to a package. They are likely to be newcomer developers who view the package as a software project they would like to onboard [126, 130].

To characterise Node.js packages published on npm registry (i.e., npm packages) from these two perspectives, I conducted an online survey to ask Node.js developers what features they consider before adopting or contributing to assessing the quality of npm packages. I separate features of npm packages into four types: (1) *Documentation*: the properties of the package documentation, (2) *Repository*: the properties of the git repository, (3) *Software*: the ability to install, build, and execute the test of the package, and (4) *GitHub activity*: the interaction between developers and the repository on GitHub. I formulate the first research question as follows:

(RQ1) *Which features of an npm package do practitioners find relevant for assessing package quality?*

In summary, this chapter presents the following contributions:

- A definition of the 'goodness' of an npm package from two perspectives, user and contributor.

- A survey of Node.js developers to investigate how different perspectives select npm packages.

2 Audience Perspectives of an npm package

In this chapter, I characterise the audience of an npm package into two different perspectives. These perspectives are based on the various ways in which npm packages or software projects, in general, have been discussed in related work:

Table 3.1: List of 30 features for package quality assessment presented in the survey. The features are grouped by their characteristic (feature type).

Feature type	Feature	Description
Documentation	[D1] hasAReadmemd	the existence of the README.md file in the root directory of the package [60, 110]
	[D2] linesInReadmemd	the number of lines in the README.md file [60, 110]
	[D3] numberOfCodeBlocks	the number of code snippets extracted from the README.md file [60, 110]
	[D4] readmemdCodeSnippetsNumber	the number of JavaScript snippets in the README.md file [60]
	[D5] hasAnInstallExample	the existence of an install example in the README.md file [50, 60, 110]
	[D6] hasARunExample	the existence of a run example in the README.md file [60, 110]
	[D7] hasAContributionGuideline	the existence of a CONTRIBUTING.md file in the root directory of the package [37, 65, 110, 121]
	[D8] hasACodeOfConduct	the existence of a CODE_OF_CONDUCT.md file in the root directory of the package [110, 135]
	[D9] hasALicense	the package has either license file or license description in REAMDE.md file [6, 60, 85, 110]
	[D10] githubLink	the GitHub repository link of the package [60]
Repository	[R1] hasASourceDirectory	the existence of a source directory (i.e., /src) within the root directory of the package [62]
	[R2] numberOfFiles	the number of files in the package [10]
	[R3] numberOfJsFiles	the number of JavaScript files (.js) in the package [10]
	[R4] numberOfHtmlFiles	the number of HTML files (.html) in the package [10]
	[R5] sizeOfRepository	the size of the package repository in bytes [89]
	[R6] mostPopularFileExtension	the most popular file extension appearing in the repository [10]
	[R7] hasATestDirectory	the existence of a test directory (i.e., /test, /tests) in the root directory of the package [85]
	[R8] numberOfRepositoryTags	the number of git tags in the repository [61]
	[R9] numberOfCommits	the number of commits in the main branch of the package [10]
Software	[S1] ableToInstall	whether the package is able to install the project [76]
	[S2] ableToBuild	whether the package is able to build (i.e., install dependencies, prepare working environment) [10, 41]
	[S3] ableToExecuteATest	whether the package is able to execute all test cases [10]
GitHub activity	[G1] numberOfNewcomerLabels	the number of newcomer labels in the GitHub issues [130]
	[G2] numberOfContributors	the number of GitHub contributors in the package [48, 85]
	[G3] numberOfIssues	the number of GitHub issues [48, 148]
	[G4] numberOfPullRequests	the number of GitHub pull requests [48]
	[G5] mostRecentIssue	the most recent GitHub issue date in an epoch format [48]
	[G6] mostRecentPullRequest	the most recent pull request date in an epoch format [48]
	[G7] latestRepositoryUpdate	the most recent committed date in an epoch format [48]
	[G8] latestReadmemdUpdate	the most recent README.md update date in an epoch format [48]

1. *User perspective* - according to Lethbridge et al. [71], apart from very good code design and reusability aspects, high quality documentation is the key to learning a software system. Bennett [11] explained that "The greatest library in the world would fail if the only way to learn it was reading the code (and, in fact, it already has to a large extent). Some packages have managed to overcome this by way of lots of unofficial documentation – blog entries and the like – but there is absolutely no substitute for full, well-written documentation." Therefore, I consider the user perspective essential to my investigation of what makes a good Node.js package.

2. *Contributor perspective* - according to Steinmacher et al. [126], newcomers can be attracted to a project with characteristics such as license type,

project visibility, or a number of existing contributors. Moreover, providing good support for the onboarding of newcomers to contribute to a project is important. It can be done by identifying good first issues and creating informative descriptions for them [130]. A package that is good for users (e.g., easy to install) might not necessarily be good for contributors (e.g., presence of a CONTRIBUTING.md) and vice versa. Investigating such different perspectives is one of the goals of this work.

2.1 Developer Survey on Perspective

To answer (RQ1) *Which features of an npm package do practitioners find relevant for assessing package quality?* I start my investigation with a list of 30 package features derived from related work. I extracted these features from prior works on npm in particular or software reuse in general. Table 3.1 shows the 30 features I identified along with the reference(s) for each. I group the features into four types based on their characteristics: (1) *Documentation*, (2) *Repository*, (3) *Software*, and (4) *GitHub activity*. However, related work has not validated the usefulness of these features to contributors and users. To address this, I conducted an online survey to identify developer opinions on what perspectives npm features belong to and what features are relevant for assessing npm package quality.

In the survey, I first asked the developers about their demographics, i.e., their experience with Node.js and npm packages, including their contributions on GitHub. After that, I asked the developers "which perspective do these features belong to?" for each of the 30 features, to which they can answer either (1) user perspective, (2) contributor perspective, (3) both of them, or (4) none of them. To summarise the result of this question, I assign the top voted choice as the perspective of each feature. In the case of the top voted choices are "user" and "contributor" (i.e., scores are tied), I assign "both of them" to the feature. I also confirmed that there were no cases where the "none of them" choice tied with others. Next, I asked developers to evaluate "how relevant are these features for assessing package quality?" for each of the 30 features using a five-point Likert scale, i.e., ranging from strongly agree to strongly disagree with a neutral option the middle strongly. To find participants for the survey, I contacted 2,150 npm developers and received 33 responses.

Table 3.2: Demographic of the respondents (33 Node.js developers).

How many years of experience do you have with Node.js?	
2 to 3 years	3
4 to 5 years	11
6 to 7 years	12
8 to 9 years	3
10 years or more	4
What do you identify yourself as?	
User	19
Contributor	2
Both	12
How often do you use npm packages in your projects?	
Once a month or more often	30
Less than once a month but more than once per year	2
Less than once per year	1
How often do you contribute to npm packages on GitHub?	
Once a month or more often	8
Less than once a month but more than once per year	18
Less than once per year	6
Never	1

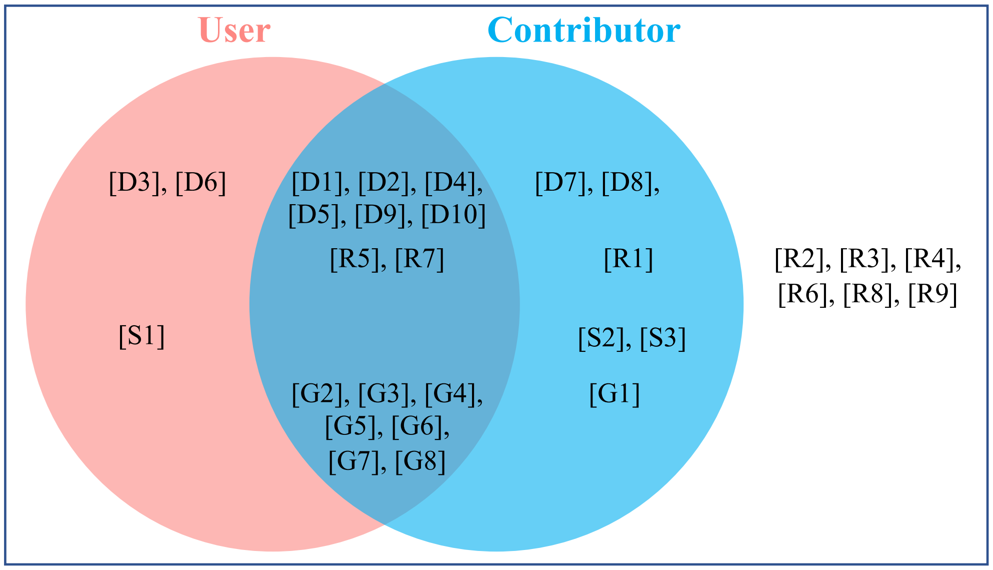
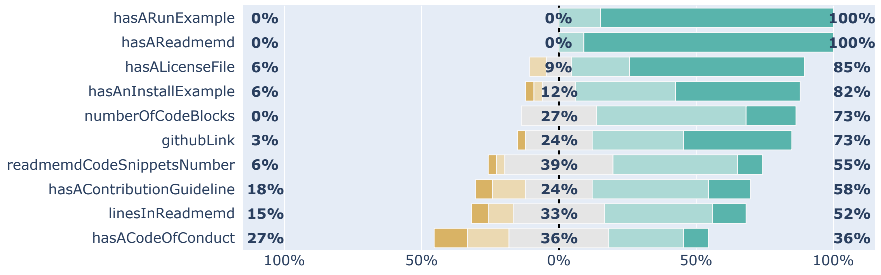

User
Contributor
[D3], [D6]
[S1]
[D1], [D2], [D4], [D5], [D9], [D10]
[R5], [R7]
[G2], [G3], [G4], [G5], [G6], [G7], [G8]
[D7], [D8],
[R1]
[S2], [S3]
[G1]
[R2], [R3], [R4], [R6], [R8], [R9]

Figure 3.1: Respondents mapping of features to the perspectives.

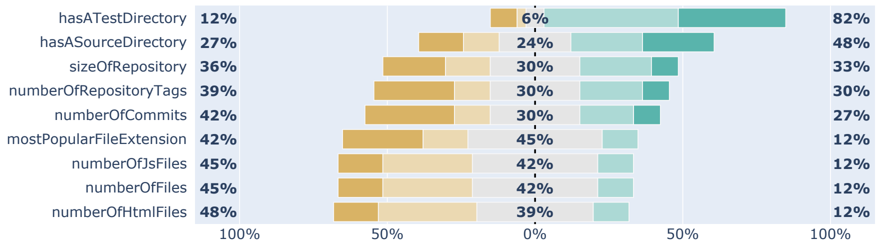
Table 3.2 shows the demographics of the participants. All have at least two years of experience with Node.js and npm. 19 developers described themselves as only a user of packages, two as a contributor to packages and 12 as both a user and contributor. Regarding the usage of npm packages, 30 developers responded that they used packages once a month or more often, two used them less than once a month but more than once per year, and the only one used npm packages less than once per year. Eight developers contribute to packages on GitHub once a month or more often, 18 less than once a month but more than once per year, six contribute less than once per year, and only one never makes any contributions.

Figure 3.1 shows which perspective package features belong to according to the opinion of developers. I find that half of the features are shared among both user and contributor perspectives. If features are specific to either perspective, users focus on the example of code or snippet (D3, D6) and ability to install the package (S1). On the other hand, contributors focus on guideline and code of conduct (D7, D8), source code directory (R1), ability to build and test (S2, S3), and newcomer labels in GitHub issues (G1). Interestingly, most repository features (R2, R3, R4, R6, R8, R9) do not belong to any perspectives since their usefulness does not convince participants to assess the package quality.

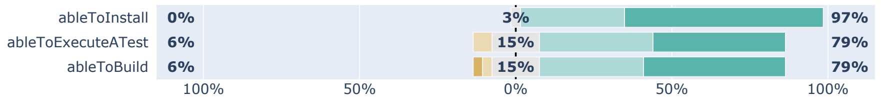
(a) Documentation features (Median agreement: 73%, neutral: 24%, disagreement: 6%)



(b) Repository features (Median agreement: 27%, neutral: 34.5%, disagreement: 42%)



(c) Software features (Median agreement: 79%, neutral: 15%, disagreement: 6%)



(d) GitHub activity features (Median agreement: 43.5%, neutral: 37.5%, disagreement: 19.5%)

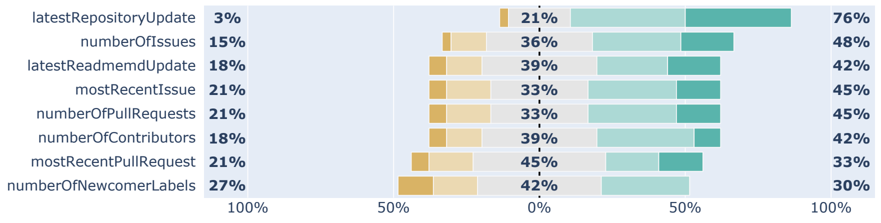


Figure 3.2: Survey results on the relevance of features for package quality assessment. Left hand (yellow) shows levels of disagreement, middle (grey) shows neutral, and right (green) shows levels of agreement.

Figure 3.2 shows the survey results on how relevant the features are for assessing package quality. The green, grey, and yellow bars represent the level of agreement, neutrality, and disagreement from developer votes. Overall, I find that developers agree that documentation, GitHub activity, and software features could be used to measure the quality of npm packages – on average, 73%, 43.5%, and 79% respectively agree (i.e., agree and strongly agree). There are three features that get more than 90% of agreement includes (1) hasARunExample (100%), (2) hasAReadmemd (100%), and (3) ableToInstall (97%). On the other hand, developers disagree that repository features can reflect the quality of packages – on average, 42% of the respondents disagree (i.e., disagree and strongly disagree).

From the additional comments from developers about unlisted features for package quality assessment, users have a concern in security features when they packages. For example, developers commented that they concerned about "*Perceived project activity, support, and security*" when adopting packages. The other comment showed that developers choose packages that have a "*use of a fail-proof security features (like Passport for authentication) to avoid related risk*". Features such as "*code coverage*" and "*security scans*" of packages are also mentioned by developers.

Summary for RQ1: Choosing a good package from both perspectives is slightly different based on their needs. Developers agree that software and documentation features are highly relevant for package quality assessment. Repository features such as the number of commits and git tags in the repository are not useful in both perspectives. Additionally, security features are also in the consideration of developers.

3 Summary

In this chapter, I conducted an online survey to ask Node.js developers what features they consider before adopting or contributing to assessing the quality of npm packages. The key results of this chapter are (1) users and contributors of npm packages share similar views on which features are important for selecting a good package; users focus more on how to use the package, while contributors focus on the contribution guideline and how to build and test the package, (2) from lots of features, developers still agree that software and documentation features are highly relevant to assess package quality, (3) developers believe that repository features do not belong to any perspectives and do not reflect the package quality. (4) developers comment that security is one of the important feature for assessing the package quality, but was not mentioned in the survey. Hence, Chapter 4 explores the characteristic of these listed features in packages for a better understanding how to choose a good package.

4 | Package Quality Features and Runnability

1 Introduction

To investigate what trade-offs exist between features, I collected diverse data related to Node.js packages to create GH-Node.js, a dataset of 723,218 Node.js package repositories. First, I extracted and analysed the npm package features from 103,364 npm packages to understand the correlation among the features. I then explored the possibility of predicting package runnability since software features are highly relevant for assessing package quality, as shown in my survey. Through the two lenses of perspectives and extracted features, I formulate the two following research questions to guide my study:

> (RQ2) *What features of an npm package correlate from different perspectives?*

The key results of RQ2 are that (1) features from the same type usually have strong correlations. (2) Software features are negatively correlated with other features. (3) Features that are less likely to be considered by any perspective, such as a number of files, are correlated with the ability to build a package.

> (RQ3) *What features of an npm package predict whether it is runnable or not?*

The key results of RQ3 are that (1) predicting runnability of the package is viable. (2) Based on the permutation feature importance, I find that repository features are important for predicting the runnability.

In summary, this chapter presents the following contributions:

- Three measures of the runnability of an npm package.
- GH-Node.js, a dataset that contains a curated set of 723,218 Node.js packages containing repository and social interaction information.
- A large scale analysis of 104,364 npm packages for correlations and predictions of runnability.

2 GH-Node.js: A Node.js Repository and Interaction Dataset

To extract the required features needed by the two perspectives, I compose my own dataset of npm repositories named GH-Node.js. GH-Node.js is an open dataset contains a curated snapshot of Node.js and npm packages that focuses on the social, technical, and documentation aspects. GH-Node.js is based on two existing datasets (GHTorrent and Libraries.io). GHTorrent provides a mirror of git repositories and developer interactions gathered from GitHub [48], while Libraries.io provides meta-data and relationships among packages hosted on popular software ecosystems, e.g., npm, Maven, and PyPI [134].

Table 4.1 shows the summary statistics of GH-Node.js from November 30, 2020, with 723,218 Node.js repositories, which only 104,364 repositories are identified as npm packages. I also collected 1,960,345 issues, 1,083,828 pull requests, and 283,360 contributors associated with npm packages by using the GitHub API. In total, I took four months, from August to December 2020, to acquire and process all information for GH-Node.js.

3 Experiment Setup

To answer RQ2 and RQ3, I extracted a subset of GH-Node.js as shown in Table 4.2. For my data preparation, I first extract metrics that relate to each perspective. Note that to answer RQ2, I focus on all 11,127 packages that were

Table 4.1: Dataset Snapshot Statistics. The full dataset estimations are approximate values.

Node.js and npm Package Repository Information	
Repository Snapshot	Nov 2, 2020
# Node.js packages	723,218
# total npm packages	104,364
- # last update 2013	7
- # last update 2014	9,867
- # last update 2015	25,680
- # last update 2016	26,115
- # last update 2017	15,057
- # last update 2018	8,869
- # last update 2019	7,642
- # last update 2020	11,127
- # commits	6,402,982
- # repository tags	674,258

last updated in 2020 from the total of 104,364 packages. My motivation was to retrieve projects that were likely to be active.

3.1 Runnable Code: Build and Run Tests

Building a package (79%) and running its unit tests (also 79%) are two features that received positive feedback in RQ1 survey. They are also among the first tasks that contributors do to ensure that the selected package is ready to run and test new features or fixes in the local environment. To confirm that an npm package is runnable from its source code, I automatically create a virtual environment, build, and test the package from scratch.

My approach to building and testing each package from its repository consists of four steps:

1. Create a docker container to construct an isolated environment.

2. Clone the repository of the npm package to the docker container.

Table 4.2: Dataset for my experiment.

Runnable Code	
# repositories (last updated in 2020)	11,127
# successfully built	8,199
- # passed tests	607
- # failed tests	4,262
- # no test	3,330
# unsuccessfully built	2,928
Executable Documentation	
# repositories	104,364
# collected code snippets	233,826
- Max	302
- Min	0
- Mean	2.2405
- Median	1.0000
- SD	4.8990
# successfully installed	97,006
# executed code snippets	220,324
- # successfully executed	33,484
- # unsuccessfully executed	186,840

3. Build the repository by using `npm install -no-audit`.

4. If the package is built successfully, then execute `npm test` to run a unit test script as detailed in the package.json file.

Note that I set a timeout of five minutes for processing to detect frozen processes.

As shown in Table 4.2, I selected 11,127 repositories with at least one commit in 2020 as the input of the runnable code experiment. I find that 8,199 packages (73.68%) are successfully built. From these built packages, only 607 packages (7.40%) have at least one test case and are able to pass all their test cases. Note that the building and the testing process took around 21 days of execution.

3.2 Runnable Package: Install and Execute Code Snippets

Installing a package (97%) and trying the run example (100%) are two features from the user perspective, which also received a positive result from RQ1. Both features are among the first tasks that package users do to ensure that the selected package is executable in their applications. To confirm that an npm package is runnable after installed in any application, I automatically create a virtual environment, install, and execute the extracted code snippets.

I extracted 233,826 Node.js code snippets from the README files of all 104,364 repositories (last updated from 2013 up to 2020). Example code in markdown files can be identified by the surrounding special characters (‘‘‘) that allow for the rendering of code blocks, and additional language information may be provided to allow for syntax highlighting; for example, adding the tag `js` to the same line (‘‘‘`js`) will enable JavaScript syntax highlighting on GitHub. I extracted these code blocks, along with any language data, and then filtered my dataset to contain only Node.js code. To do this, I discarded any code blocks that were not in the JavaScript language, such as bash commands used to demonstrate package installation, but kept those without a specified language, as not all READMEs utilise syntax highlighting. Then, to discard more unrelated code snippets, I filtered out common install commands such as `npm install`. Within Node.js repositories, it is also common to see code blocks containing the results of the previous snippet, often in the form of a JSON data object or array; as such, I filtered out any singular objects or arrays from my set of snippets.

To simulate how developers include the package in their projects and test its usage, I use a similar approach to Section 3.1. Instead of building the package itself, I built the empty Node.js package, which depends on the selected npm package. After that, I executed the code snippets inside my empty package.

As shown in Table 4.2, I selected all 104,364 npm package repositories as the input of the executable documentation experiment. I found that 97,006 packages (92.95%) are successfully installed. From these installed packages, I found that only 64,280 packages (61.59%) have at least one code snippet in their README. In the end, I executed 220,324 code snippets and found that only 33,484 (15.20%) of them were successfully executed without any error. Note that the code snippet executing process took around 14 days of execution.

4 Correlating features by Perspective

To answer (RQ2) *What features of an npm package correlate from different perspectives?* I initially perform a descriptive analysis of all features. From this analysis, I first describe the most positive and most negative correlations in the dataset to understand the trade-off between features and perspectives. I then manually investigate correlations between different perspectives. The following analysis characterises the 11,127 packages for which all 30 feature values are available; I ignore githubLink and mostPopularFileExtension. This was necessary to achieve a consistent picture, as not all used techniques allow for missing values.

Figure 4.1 shows the Spearman rank-order correlation coefficients between the 28 features and clustered with Wards hierarchical clustering approach. I find that features from the same type are more likely to have strong positive correlations among them. The GitHub activity features like numberOfIssues and numberOfPullRequests have the strongest positive correlation among any feature pairs. For documentation features, numberOfCodeBlocks and linesInReadmemd have a strong positive correlation, but less than the GitHub activity features. For repository features, numberOfCommits and numberOfRepositoryTags also have a strong positive correlation and belong to the same cluster. Software features (i.e., ableToExecuteATest, ableToInstall, ableToBuild) have positive correlations among each other, but the correlations are not strong. On the other hand, the features from different types tend to have either small positive correlations or negative correlations.

Table 4.3 shows the top negative correlations between software features and other features, which are top negative correlations among any feature pairs shown in Figure 4.1. I find that various features that measure the size of a repository (e.g., number of files, commits, issues, pull requests, contributors) tend to be negatively correlated with the ability to install, build, and execute tests on a package. This makes intuitive sense as larger repositories are more complex and more likely to encounter issues regarding runnability. In addition, I note that correlations between several of the documentation features and runnability also tend to be slightly negative. Curiously, even the correlation between hasAnInstallExample and the ability to install a package is not positive.

Table 4.3: Top 5 negative correlations of features grouped by their types (U: user, C: contributor, B: both of them, N: none of them).

Feature type	Software feature	Correlation	Corresponding feature
Documentation	ableToBuild	-0.0962	hasAContributionGuideline (C)
		-0.0536	hasACodeOfConduct (C)
		-0.0448	linesInReadmemd (B)
		-0.0426	numberOfCodeBlocks (U)
		-0.0393	hasARunExample (U)
	ableToExecuteATest	-0.0518	hasAContributionGuideline (B)
		-0.0281	hasACodeOfConduct (C)
		-0.0199	hasARunExample (U)
		-0.0074	hasAReadmemd (B)
		-0.0065	numberOfCodeBlocks (U)
	ableToInstall	-0.0255	hasACodeOfConduct (C)
		-0.0216	hasAContributionGuideline (C)
		-0.0096	hasAnInstallExample (B)
		-0.0092	linesInReadmemd (B)
Repository	ableToBuild	-0.2485	numberOfFiles (N)
		-0.2291	numberOfCommits (N)
		-0.2224	sizeOfRepository (B)
		-0.2126	hasASourceDirectory (C)
		-0.1847	numberOfRepositoryTags (N)
	ableToExecuteATest	-0.1141	sizeOfRepository (B)
		-0.1114	numberOfCommits (N)
		-0.0997	numberOfFiles (N)
		-0.0897	hasASourceDirectory (C)
		-0.0818	numberOfRepositoryTags (N)
	ableToInstall	-0.0947	sizeOfRepository (B)
		-0.0808	numberOfFiles (N)

Table 4.3 Continued table (U: user, C: contributor, B: both of them, N: none of them).

Feature type	Software feature	Correlation	Corresponding feature
		-0.0683	numberOfCommits (N)
		-0.0648	numberOfJsFiles (N)
		-0.0270	numberOfRepositoryTags (N)
GitHub activity	ableToBuild	-0.2249	numberOfIssues (B)
		-0.2139	numberOfPullRequests (B)
		-0.1893	numberOfContributors (B)
		-0.0662	numberOfNewcomerLabels (C)
	ableToExecuteATest	-0.1099	numberOfIssues (B)
		-0.1064	numberOfContributors (B)
		-0.1029	numberOfPullRequests (B)
		-0.0235	numberOfNewcomerLabels (C)
	ableToInstall	-0.0484	numberOfIssues (B)
		-0.0466	numberOfContributors (B)
		-0.0379	numberOfPullRequests (B)
		-0.0274	numberOfNewcomerLabels (C)

Figure 4.2 shows the distributions of the feature values. Note that the strip plots cannot show all 11,127 data points; however, they provide a subjectively better qualitative and quantitative characterisation of the distribution than violin plots.

Figure 4.3 shows the clusters of npm packages using the t-distributed Stochastic Neighbour Embedding (t-SNE) [137] to project the 28D data points into 2D. t-SNE's reduction process attempts to preserve the distances in the high-dimensional space as much as possible. Interestingly, a large number of clusters has formed. I have manually inspected the clusters and added labels for cases that I have found interesting. While I have to be careful not to over-interpret this reduced representation, I can observe co-located areas where the neighbourhood seems intuitively reasonable. For example, at the right, repositories are listed with contribution guidelines and codes of conduct, both of which can be qualities of mature repositories – and I can indeed find these repositories in the vicinity of those with large numbers of issues; further to the left, I can find repositories for which it is possible to execute tests. Similarly, I can find repositories that cannot be built or do not have a license in the lower part of the plot. Lastly, and a little bit to the left of the centre of the figure, I have repositories where the README.md has a small number of code snippets: this seems to place them at the (fuzzy) boundary between possible immature projects and mature ones.

Summary for RQ2: There are strong positive correlations among features from the same type. Other feature combinations present trade-offs – in particular software features tend to be negatively correlated with other features.

5 Predicting whether or not an npm package is runnable

To answer (RQ3) *What features of an npm package predict whether it is runnable or not?* I first need to clarify what I mean by runnable that can be expressed by my features defined in Table 3.1 in my experiments. To do so, I apply the DUO principle (Data Mining Algorithms Using/Used-by Optimizers [4]) to mine insights by using optimisers: I employ auto-sklearn [39] to automatically search over the space of machine learning models to achieve an optimised insight, i.e., to achieve a good approximation of the truth while reducing potential limitations of modelling technology and its default or manual configuration. auto-sklearn is a combination of the machine learning library sklearn [105], the algorithm configurator smac [58], and a set of preprocessing strategies for data.

I investigate three different binary target features as my interpretation of code being runnable, focusing on essential aspects of software engineering:

1. ableToInstall: while this appears to be a weak condition, it can be a necessary condition for subsequent code development;

2. ableToBuild: stronger than the previous one, and can indicate higher usefulness;

3. ableToExecuteATest: if true, then this can indicate correctness with respect to the specification;

I consider all 104,364 repositories, and I consider all features as inputs except for the three ableTo* (to avoid the accidental explanation through strong correlations), as well as the two string-based features; this leaves us with 25 input features. For the model search, I perform 5-fold cross-validation. For each fold, auto-sklearn has 10 minutes allocated (single CPU core) to find a well-performing model for a target feature.[1] The average F1-scores and accuracy scores over the five folds (per target) are as follows:

1. ableToInstall: F1=0.96, accuracy=0.92. This is not too surprising, as 92% of the data set is installable.

2. ableToBuild: F1=0.83, accuracy=0.73. 72% can be built.

3. ableToExecuteATest: F1=0.09, accuracy=0.95. The F1 score is very low, although this is partly due to the imbalanced dataset: it is possible to execute at least one of the tests only for 5% of the repositories.

To dig deeper into the results, I take (for each target) one of the generated models and investigate the permutation importance of the features [15]. In this post-hoc approach, a single column of the validation data is randomly shuffled, leaving the target and all other columns in place. This metric measures the effect on the accuracy of predictions in that shuffled data. Due to randomised effects, this process is repeated five times for each feature (sklearn default).

Figure 4.4, 4.5, and 4.6 show the results, where larger values indicate higher importance in these tuned models. As I can observe, to predict ableToInstall (Figure 4.4), a large number of features is necessary, with number-based and size-based features being the four most important ones. To predict ableToBuild (Figure 4.5), these features remain important, although the now most important

[1]As a performance reference: a random forest from sklearn is typically trained in about one second on my standard laptop.

one is hasASourceDirectory. To predict ableToExecuteATest (Figure 4.6), the previously mentioned features remain important, but they are now joined by linesInReadmemd.

Figure 4.4, 4.5, and 4.6 also indicate positive and negative correlations with the respective target features. In general, all correlations with the targets are very weak, i.e., almost all correlations with the targets are within $[-0.2, 0.2]$ and are neither marked with a + or a −. ableToBuild stands out a bit as a number of features are weakly negatively correlated with it.

Summary for RQ3: Predicting the runnability of a package is viable (i.e., high F1 score). Repository features are particularly important for predicting the runnability.

6 Discussion

I now discuss my results and provide suggestions for both researchers and practitioners.

1. *Audience perspective matters.* As shown by my analysis of package features, not all features are relevant from all perspectives, and some are even negatively correlated. For example, the user perspective is more interested in documentation than the contributor perspective, which may be interested in workflow characteristics such as the pull-request and issue management systems. My survey results also confirmed that package users are more focused on documentation features. For example, one survey respondent mentioned that they adopt a package due to "*Usefulness, good documentation, recent releases*". On the other hand, contributors are more focused on the contribution guidelines as mentioned by another respondent: "*simple rules for issues and pull requests*".

   For practitioners, I suggest that taking into account both the user and contributor may help the overall attractiveness of their Open Source projects. Furthermore, opportunities for future work may be tool support for recommending projects based on the identified features and the different scenarios of these perspectives. My results confirm that automatically predicting the runnability of a package may be viable. For researchers, I suggest that my perspectives open up different scenarios for the different practitioners and a re-evaluation of existing metrics as well as investigating new metrics that can capture these two perspectives. The simple heuristics of using the popularity, dependency usage, stars and downloads would need to be re-evaluated, especially for selecting representative samples for empirical studies.

2. *Maintaining the runnability of npm packages is non-trivial.* According to my results, the definition of successfully running a project is not trivial. Although some work has looked at a single definition, no work looks at multiple different types of runnability. For practitioners, I suggest considering these different types when developing projects. This could be an issue if the project is constantly evolving, causing especially code snippets in the documentation (e.g., README) to be outdated. Furthermore, these snippets may be the usage examples or installation instructions, and therefore important for users and contributors. Another answer from a survey respondent supports this: "*One thing that indicates a good package is an example of someone using it to solve an existing problem. For npm packages, those are usually not found in the package's documentation, but on someone's blog*".

7 Threats to Validity

Threats to *construct validity* exist in the appropriateness of my feature list and perspectives. I mitigated these threats by conducting the survey to verify the relevance of features and perspectives for assessing the quality of packages. My feature list is taken from related works related to npm in particular or software reuse in general; however, only a few features have been used for assessing the package quality. In this case, I received 33 responses to confirm the relevant level of features (i.e., 23 out of 30 features vote agree more than disagree).

Threats to *internal validity* involve the correctness of tools and techniques used in this study. I use multiple features extracted from GH-Node.js. The threat is that sometimes some packages do not have some features or are unavailable to extract, so I applied a filter to remove packages that have at least one missing feature for RQ2, thus making this threat minimal.

Threats to *external validity* correspond to my ability to generalise results. Because my data and conclusions are based on a large number of npm packages hosted on GitHub, I cannot generalise my results to all projects in general. Not all npm packages are hosted on GitHub, and some Node.js libraries use alternative package managers like Yarn. My research is also focused on Node.js and npm only; there are similar package management systems for other languages, such as PyPI for Python and Maven for Java.

8 Summary

With more than 1.7 million packages to choose from, selecting a suitable npm package is a difficult task – and the factors that influence this decision vary based on the developer intentions. Most existing research into library selection has focused on the perspective of a user of a package; in this chapter, I have investigated the perspective of user and contributor.

In this part, I first conducted a survey with 33 respondents to understand which npm package features do practitioners find relevant for assessing package quality (Chapter 3). I found that users and contributors of npm packages share similar views when choosing a package, as half of the considered features belong to both perspectives. However, the user perspective is more focused on the documentation and the usage of the package. The contributor perspective, on the other hand, is more focused on contribution guidelines. Interestingly, developers believe that most repository features do not belong to any perspectives and do not reflect the quality of packages.

I then created the GH-Node.js dataset, a dataset containing a curated snapshot of npm packages that the research community can utilise (Chapter 4). I identified a set of 30 features important to one or both perspectives and identified correlations between different perspectives. I found that features from different perspectives are not necessarily correlated; in fact, in some cases, negatively correlated. This suggests that my different perspectives are important and that trade-offs exist; users and contributors have different priorities. This also suggests a need for new metrics to capture these perspectives, which complement the existing features such as popularity, dependency usage, stars, and download count. I also evaluated the runnability of packages in terms of the runnability of test cases and example code snippets (Chapter 4). Out of 11,127 npm packages that were updated in 2020, I find that 8,199 (73.68%) are able to be built, and only 607 (7.40%) are able to pass all test cases. Out of 104,364 npm packages from my dataset, I found that 97,006 (92.95%) could be successfully installed, 64,280 (61.59%) have at least one code snippet in their README, and out of 220,324 code snippets, 33,484 (15.20%) were able to execute successfully. Using this data, I investigated if I could predict the runnability of a package. I find that predicting a package runnability depended heavily on what metric of runnability I use; for example, I find that there is a strong correlation between the ability to install a package and its number of files and commits; however, these features are less important in predicting the ability to build a package.

My work lays the groundwork for future work on understanding how users and contributors select appropriate npm packages. I suggest that practitioners should take package audiences into account to help the attractiveness of their packages. Package owners and contributors should maintain the runnability of their package for attracting and helping new users and newcomer contributors since this is not trivial. Potential future avenues for researchers include (1) a package recommendation system based on the runnability of packages and (2) an exploration of new metrics that can measure the package quality with the consideration of both user and contributor perspectives.

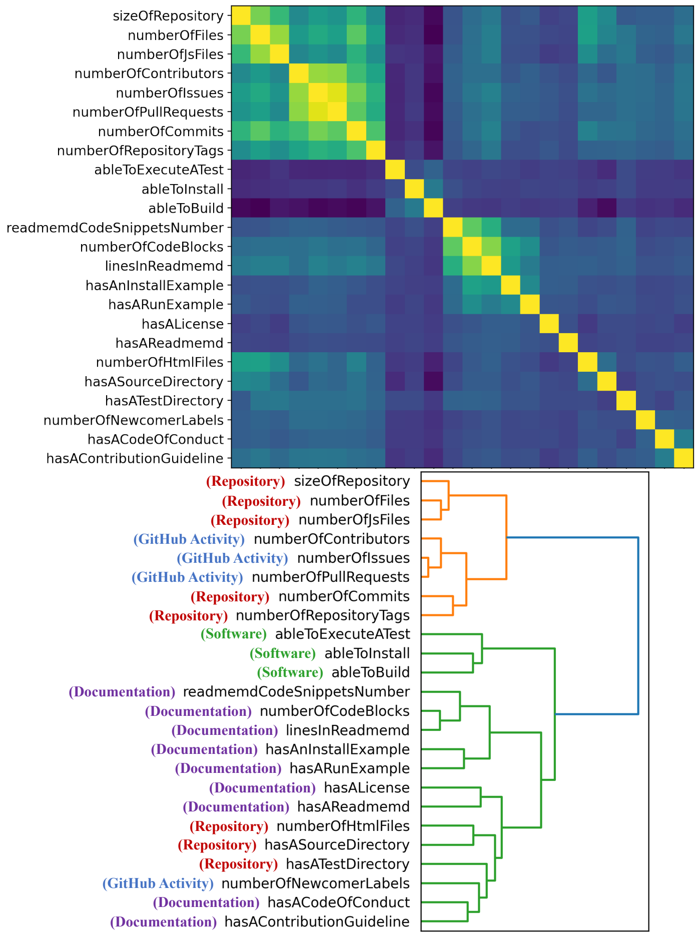


Figure 4.1: Features: correlations (top) and clustering (bottom). Lighter fields correspond to a strong positive correlation between the features, and darker fields to a strong negative correlation. X-labels are omitted as they follow the order (optimised to co-locate correlated features) of the y-labels. The dendrogram groups correlated features closely together. Shown are the correlations based on the 11,127 data points for which all feature values are available; not shown are the four timestamp-related features.

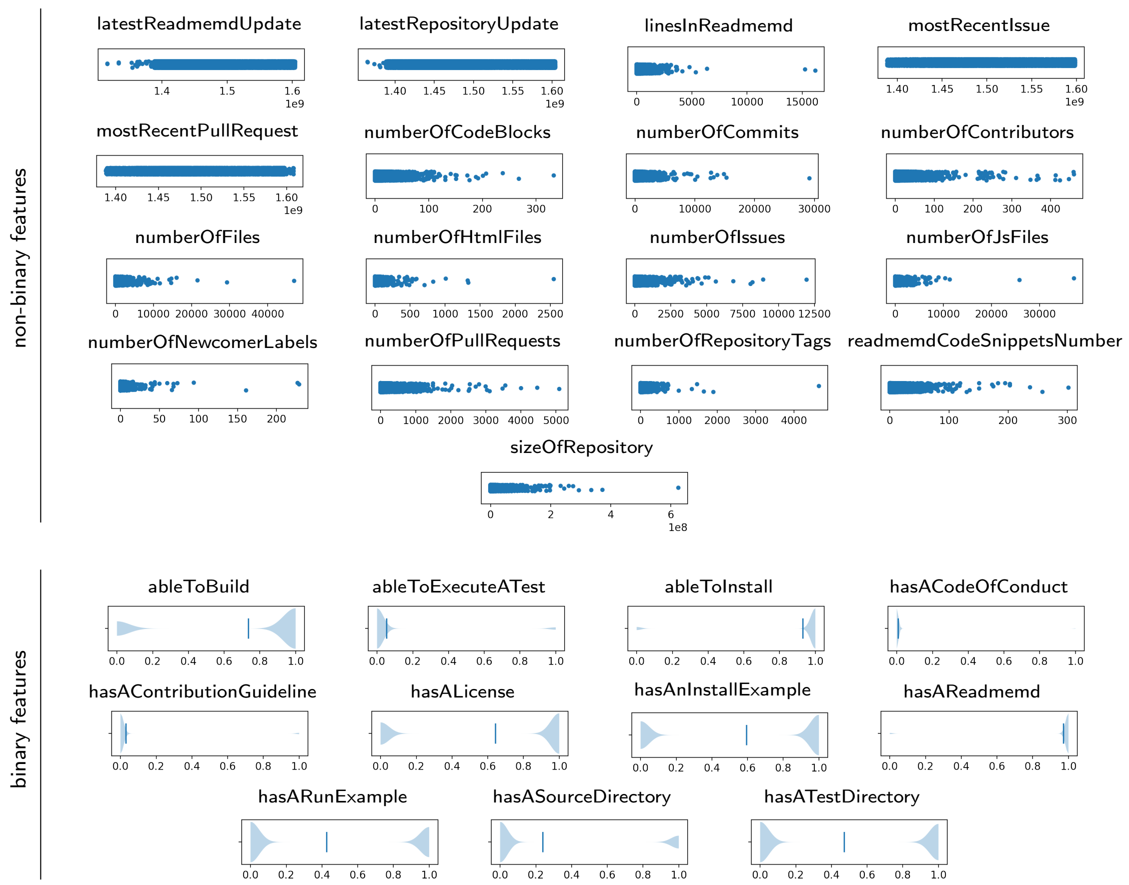


Figure 4.2: Value distribution of 28 features in alphabetical order. For non-binary features (top five rows), I use strip plots to show the distributions qualitatively by showing single data points; for binary features (bottom three rows), I use violin plots with shown means to illustrate the amount of data at either end of the distribution. Shown are all data points (11,127 to 104,364).

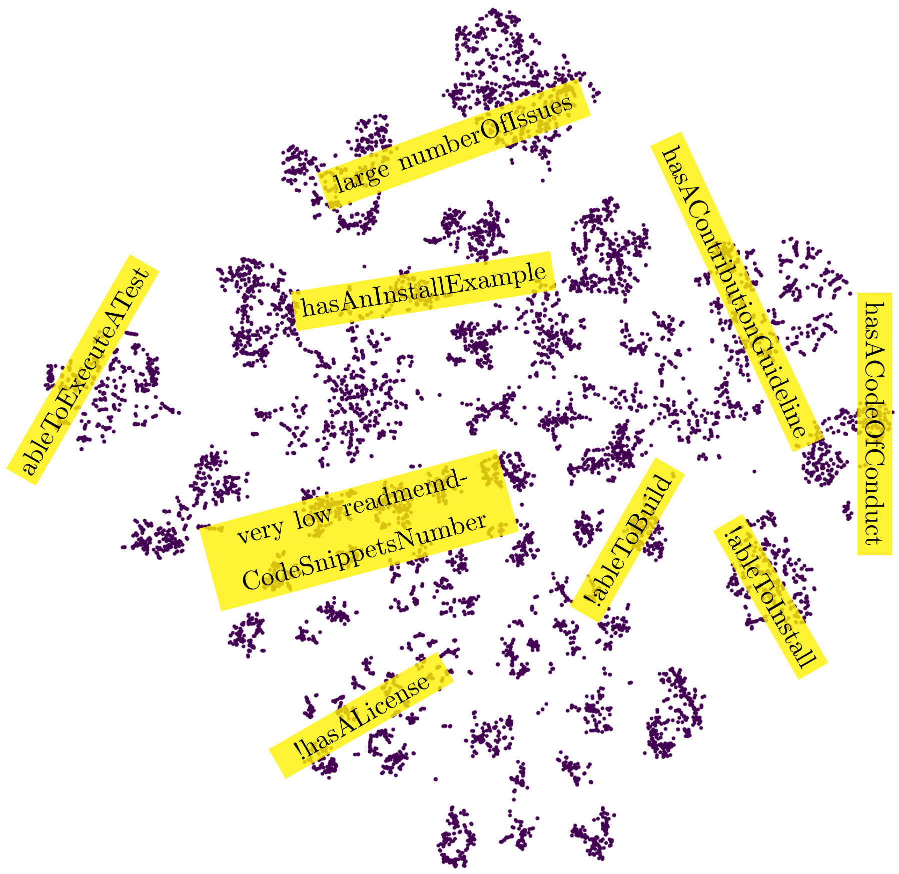


Figure 4.3: Repositories in 2D. The axes do not have any particular meaning in projections like these, which is why I removed them. Note that the clusters at the very bottom end of this figure are not easily characterised by a single feature but by combinations of features.

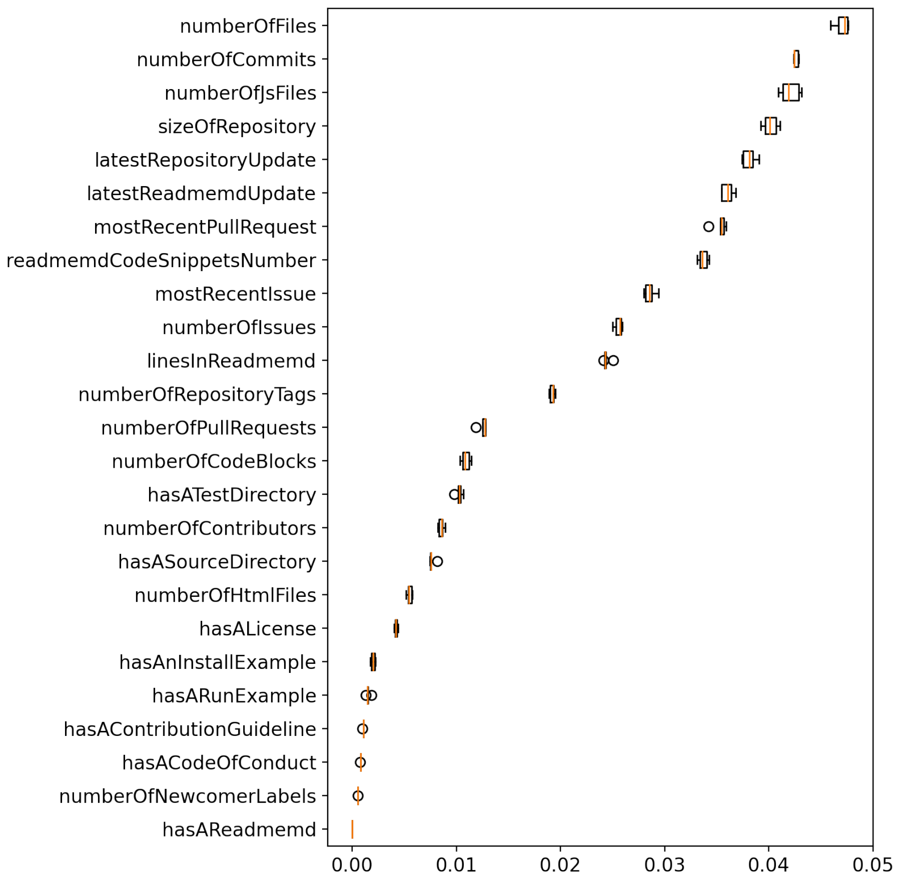


Figure 4.4: Permutation importance of a good predictive model for ableToInstall. Larger values mean that the prediction is more sensitive with respect to that parameter; hence it can be seen as more important. The + and − mark features with weak correlations with the respective target > 0.2 and < -0.2; almost all correlations are indeed in $[-0.2, 0.2]$, and no correlation was outside of $[-0.4, 0.4]$.

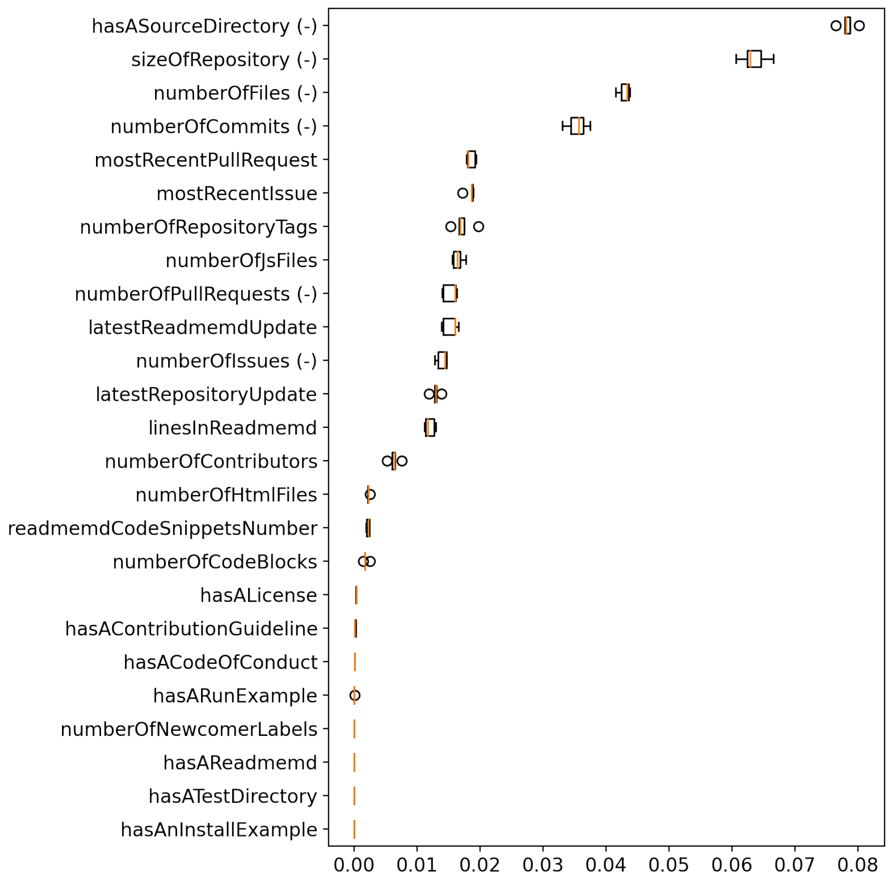


Figure 4.5: Permutation importance of a good predictive model for ableToBuild. Larger values mean that the prediction is more sensitive with respect to that parameter; hence it can be seen as more important. The + and − mark features with weak correlations with the respective target > 0.2 and < -0.2; almost all correlations are indeed in $[-0.2, 0.2]$, and no correlation was outside of $[-0.4, 0.4]$.

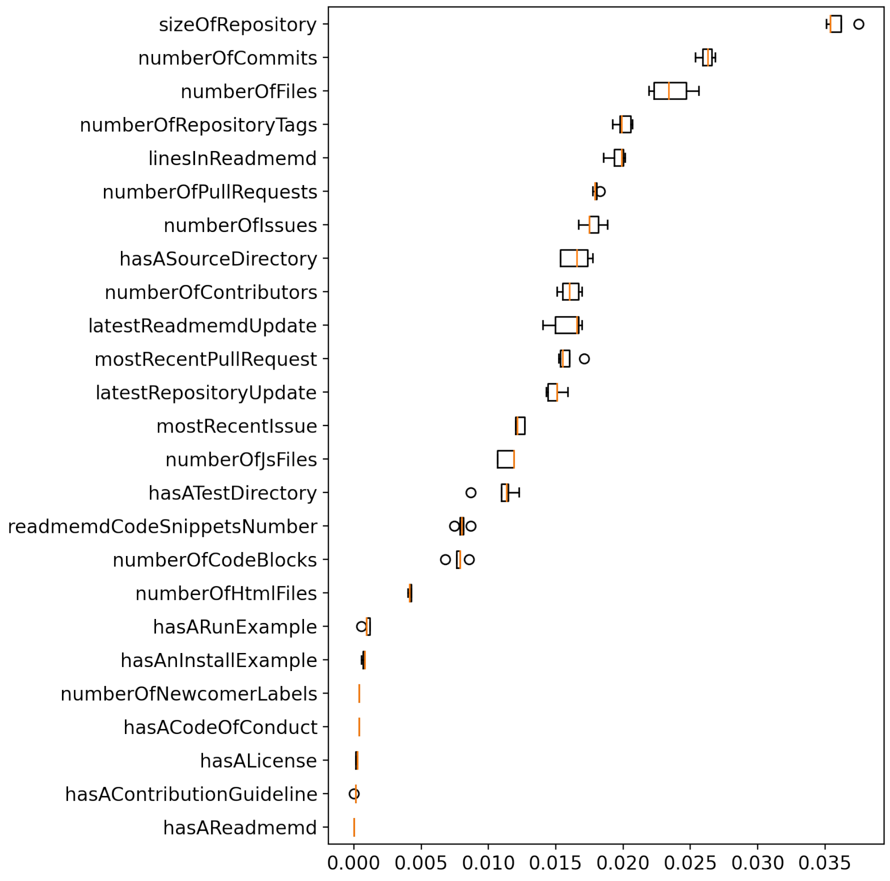


Figure 4.6: Permutation importance of a good predictive model for ableToExecuteATest. Larger values mean that the prediction is more sensitive with respect to that parameter; hence it can be seen as more important. The + and − mark features with weak correlations with the respective target > 0.2 and < -0.2; almost all correlations are indeed in $[-0.2, 0.2]$, and no correlation was outside of $[-0.4, 0.4]$.

Part II

Vulnerabilities and Fixes Assessment

5 | Package-side Fixing Release

1 Introduction

Vulnerability in third-party dependencies is a growing concern for the software developer. In a 2018 report, over *four million vulnerabilities* were raised to the attention of developers of over 500 thousand GitHub repositories [43]. The risk of vulnerabilities is not restricted to the direct users of these software artifacts, but it also extends to the broader software ecosystems to which they belong. Examples include the ShellShock [12] and Heartbleed [128] vulnerabilities, which caused widespread damages to broad and diverse software ecosystems made up of direct and indirect adopters. Indeed, the case of Heartbleed emphasized its critical role in the modern web. Durumeric et al. [35] shows that OpenSSL, i.e., the project where the Heartbleed vulnerability originated, is presented on web servers that host (at least) 66% of sites and (at least) 24% of the secure sites on the internet were affected by Heartbleed.

The speed at which ecosystems react to vulnerabilities and the availability of fixes to vulnerabilities is of paramount importance. Three lines of prior works support this intuition:

1. Studies by Hejderup [51], Howard and Leblanc [56], Munaiah et al. [84], Nguyen et al. [88], Pashchenko et al. [101], Ponta et al. [108], Williams et al. [141] encourage developers to use security best practices, e.g., project validation, security monitoring, to prevent and detect vulnerabilities in deployed projects.

2. Studies by Cox et al. [23], Decan et al. [31], Kikas et al. [64] show that

vulnerabilities can cascade transitively through the package dependency network. Moreover, they observe that security issues are more likely to occur in the field due to stale (outdated) dependencies than directly within product codes.

3. Studies by Bavota et al. [9], Bogart et al. [14], Ihara et al. [59], Kula et al. [68] show that developers are slow to update their vulnerable packages, which is occasionally due to management and process factors.

While these prior studies have made important advances, they have tended to focus on (i) a coarse granularity, i.e., releases, that focused only on the vulnerable dependency and only the direct client. For example, Decan et al. [32] analyzed how and when package releases with vulnerabilities are discovered and fixed with a single direct client, while Decan et al. [33] analyzed releases to explore the evolution of technical lag and its impact. Commit-level analysis similar to Li and Paxson [72], Piantadosi et al. [106] is important because it reveals how much development activity (i.e., migration effort) is directed towards fixing vulnerabilities compare to the other tasks. Furthermore, there is also a research gap that relates to (ii) the analysis of the package vulnerability fixes with respect to the downstream clients, and not a single direct client.

To bridge these two research gaps, I set out to identify and characterize the release, adoption, and propagation tendencies of vulnerability fixes. I identify and track a release that contains the fix, which is defined as a *fixing release*. I then characterize the fixing release for each npm JavaScript package in terms of commits for fixing the vulnerability, which is defined as the *package-side fixing release*. Based on semantic versioning [111], a package-side fixing release is a `package major landing`, `package minor landing`, or `package patch landing`. From a client perspective, I identify *client-side fixing release* lags to classify how a client migrates from a vulnerable version to the fixing version as a `client major landing`, `client minor landing`, `client patch landing`, or `dependency removal`. By comparing the package-side fixing release with the client-side fixing release, I identify lags in the adoption process as clients keep stale dependencies.

My empirical study is comprised of two parts. First, I perform a preliminary study of 231 package-side fixing releases of npm packages on GitHub (Chapter 5).

I find that the package-side fixing release is rarely released on its own, with up to 85.72% of the bundled commits being unrelated to the fix. Second, I conduct an empirical study of 1,290 package-side fixing releases to analyze their adoption and propagation tendencies throughout a network of 1,553,325 releases of npm packages (Chapter 6). I find that quickly releasing fixes does not ensure that clients will adopt them quickly. Indeed, I find that only 21.28% of clients reacted to this by performing a client patch landing of their own. Furthermore, I find that factors such as the branch upon which a fix lands and the severity of the vulnerability have a small effect on its propagation trajectory throughout the ecosystem, i.e., the latest lineage and medium severity suffer the most lags. To mitigate propagation lags in an ecosystem, I recommend developers and researchers to (i) develop strategies for making the most efficient update via the release cycle, (ii) develop better awareness mechanisms for quicker planning of an update, and (iii) allocate additional time before updating dependencies.

The contributions of Chapter 5 and 6 are three-fold. The first contribution is a set of definitions and measures to characterize the vulnerability discovery and fixing process from both the vulnerable package, i.e., package-side fixing release, and its client, i.e., client-side fixing release. The second contribution is an empirical study that identified potential lags in the release, adoption, and propagation of a package-side fixing release. The third contribution is a detailed replication package, which is available at https://github.com/NAIST-SE/Vulnerability-Fix-Lags-Release-Adoption-Propagation.

2 Concepts and Definitions

2.1 Package-side Vulnerability Fixing Process

Figure 5.1 illustrates the timeline of the package-side vulnerability fixing process of package $\mathbb{P}$ in the lower part of the figure. I break down this process into two steps:

Step one: Vulnerability Discovery. Figure 5.1 shows the vulnerability of package $\mathbb{P}$ being detected after the release of $\mathbb{P}_{V1.1.0}$. As reported by Kula et al. [68], CVE defines four phases of a vulnerability: (i) threat detection, (ii) CVE assessment,

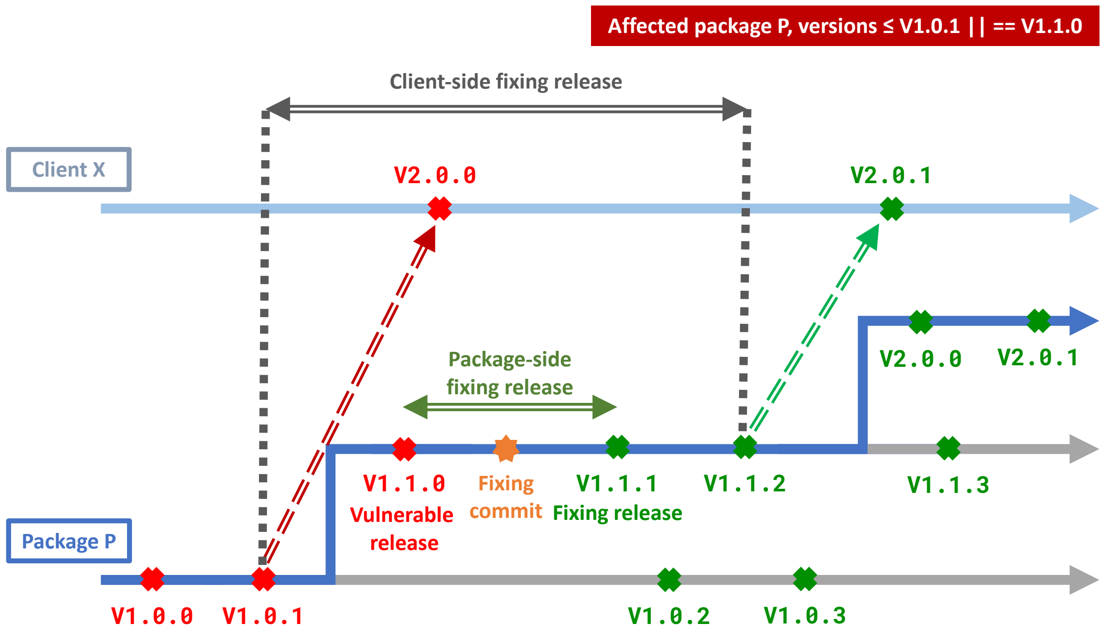


Figure 5.1: The relationship between package-side and client-side regarding vulnerability discovery, fixing, and release process of package $\mathbb{P}$ and client $\mathbb{X}$ over time. Red and green releases indicate whether releases are vulnerable or not.

(iii) security advisory, and (iv) patch release. I define the vulnerability discovery as the period between the threat detection and before the patch release. It is most likely that the fixing process starts after the CVE assessment, i.e., a vulnerability has been assigned a CVE number. In addition to a CVE number, a vulnerability report details the affected packages, which can include releases up to an upper-bound of reported versions. In this step, the developers of an affected package are notified via communication channels such as a GitHub issue for GitHub projects. Step two: Vulnerability Fix and Release. Figure 5.1 shows the vulnerability of package $\mathbb{P}$ being fixed and released as $\mathbb{P}_{V1.1.1}$. I define the vulnerability fix and release as the period where developers spend their efforts to identify and mitigate the vulnerable code. Once the fix is ready, developers merge that fix to a package repository. In most GitHub projects, developers will review changes via a GitHub pull request. Semantic versioning convention is also used to manage the release version number of a package [111]. I define the *fixing release* as the first release that contains a vulnerability fix ($\mathbb{P}_{V1.1.1}$). I also define *package-side* *fixing release* to describe the fixing release of the vulnerable package and to show how a package bumped the release version number from a vulnerable release to a fixing release. There are three kinds of package-side fixing release based on the semantic versioning: (i) `package major landing` (major number is bumped up), (ii) `package minor landing` (minor number is bumped up), and (iii) `package patch landing` (patch number is bumped up). As shown in Figure 5.1, package-side fixing release of package $\mathbb{P}$ is package patch landing, i.e., $\mathbb{P}_{V1.1.0}$ to $\mathbb{P}_{V1.1.1}$.

2.2 Client-side Fixing Release

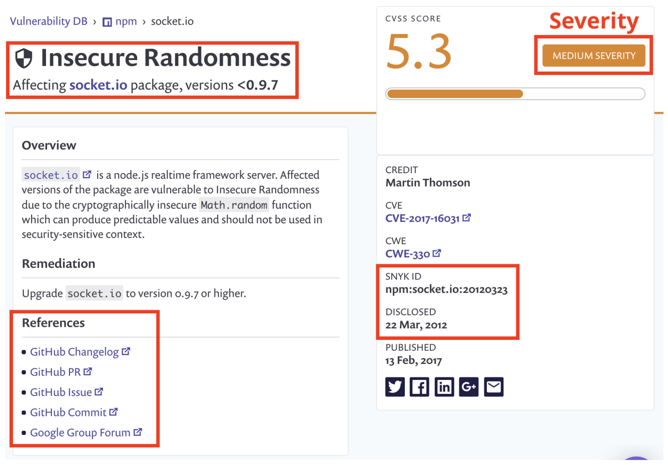
Prior work suggests that lags in adoption could be the result of migration effort [68]. Thus to quantify this effort, I compare a vulnerable release that a client is using, i.e., listed in the report, against the package-side fixing release to categorize as a *client-side fixing release*. Developers of the vulnerable package fulfilled their responsibility, thus the adoption responsibility is left to the client.

Figure 5.1 shows the timelines of a fixing release and its clients. As illustrated in the figure, client $\mathbb{X}$ suffers a lag in the adoption of a package-side fixing release by switching dependency branches, i.e., client minor landing. Also, client $\mathbb{X}$ directly depends on package $\mathbb{P}$. Client $\mathbb{X}$ is vulnerable (V2.0.0) due to its dependency ($\mathbb{P}_{V1.0.1}$). To mitigate the vulnerability, package $\mathbb{P}$ creates a new branch, i.e., minor branch, which includes the package-side fixing release ($\mathbb{P}_{V1.1.1}$). Client $\mathbb{X}$ finally adopts the new release of package $\mathbb{P}$ which is not vulnerable (V1.1.2). I consider that client $\mathbb{X}$ has a lag, which was not efficient because it actually skipped the package-side fixing release ($\mathbb{P}_{V1.1.1}$). Instead, client $\mathbb{X}$ adopted the next release ($\mathbb{P}_{V1.1.2}$). A possible cause of lags is the potential migration effort needed to switch branches, i.e., from $\mathbb{P}_{V1.0.1}$ to $\mathbb{P}_{V1.1.2}$. The migration effort for major or minor changes may include breaking changes or issues in the release cycle.

2.3 Motivating Example

Figure 5.2 shows a practical case of the vulnerability fixing process which affects a library for network communication, i.e., `socket.io`. Figure 5.2a and Figure

(a) `socket.io` vulnerability report with medium severity.



Vulnerability DB › npm › socket.io

Insecure Randomness

Affecting **socket.io** package, versions **<0.9.7**

CVSS SCORE
5.3

Severity
MEDIUM SEVERITY

Overview

`socket.io` is a node.js realtime framework server. Affected versions of the package are vulnerable to Insecure Randomness due to the cryptographically insecure `Math.random` function which can produce predictable values and should not be used in security-sensitive context.

Remediation

Upgrade `socket.io` to version 0.9.7 or higher.

References

- GitHub Changelog
- GitHub PR
- GitHub Issue
- GitHub Commit
- Google Group Forum

CREDIT
Martin Thomson

CVE
CVE-2017-16031

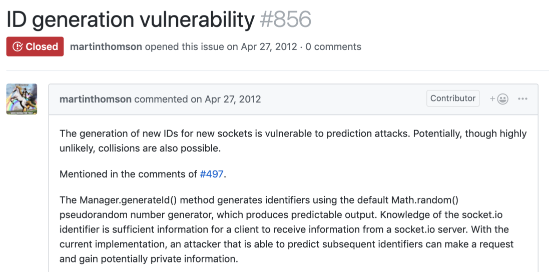
CWE
CWE-330

SNYK ID
npm:socket.io:20120323

DISCLOSED
22 Mar, 2012

PUBLISHED
13 Feb, 2017

(b) GitHub Issue reporting the vulnerability.



ID generation vulnerability #856

Closed **martinthomson** opened this issue on Apr 27, 2012 · 0 comments

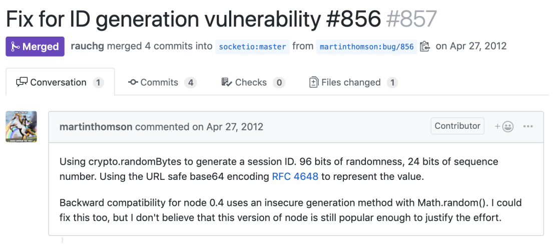
martinthomson commented on Apr 27, 2012 — Contributor

The generation of new IDs for new sockets is vulnerable to prediction attacks. Potentially, though highly unlikely, collisions are also possible.

Mentioned in the comments of #497.

The Manager.generateId() method generates identifiers using the default Math.random() pseudorandom number generator, which produces predictable output. Knowledge of the socket.io identifier is sufficient information for a client to receive information from a socket.io server. With the current implementation, an attacker that is able to predict subsequent identifiers can make a request and gain potentially private information.

(c) Pull request containing the fixing commit.



Fix for ID generation vulnerability #856 #857

Merged **rauchg** merged 4 commits into `socketio:master` from `martinthomson:bug/856` on Apr 27, 2012

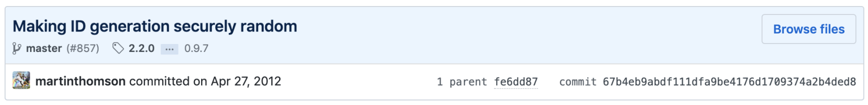
Conversation 1 · Commits 4 · Checks 0 · Files changed 1

martinthomson commented on Apr 27, 2012 — Contributor

Using crypto.randomBytes to generate a session ID. 96 bits of randomness, 24 bits of sequence number. Using the URL safe base64 encoding RFC 4648 to represent the value.

Backward compatibility for node 0.4 uses an insecure generation method with Math.random(). I could fix this too, but I don't believe that this version of node is still popular enough to justify the effort.

(d) Fixing commit to mitigate the vulnerability.



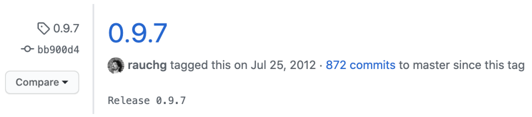
Making ID generation securely random

master (#857) 2.2.0 … 0.9.7

Browse files

martinthomson committed on Apr 27, 2012 — 1 parent `fe6dd87` commit `67b4eb9abdf111dfa9be4176d1709374a2b4ded8`

(e) New package-side fixing release has been release.



0.9.7
bb900d4
Compare

0.9.7

rauchg tagged this on Jul 25, 2012 · 872 commits to master since this tag

```
Release 0.9.7
```

Figure 5.2: Developer artifacts that mitigate a vulnerability (`socket.io`) on GitHub.

5.2b show step one, where the vulnerability is reported as a GitHub issue,[1] and summarized in snyk.io.[2] The vulnerability report contains detailed information regarding the identified problem, its severity, and a proof-of-concept to confirm the threat. In this example, `socket.io` was vulnerable to a medium severity vulnerability. I also found that the reporter is the same person who also created the fix. Figure 5.2c and 5.2d show step two. Figure 5.2c shows a fix that will be merged into the code base. The fix is submitted in the form of a pull request.[3] Figure 5.2d shows that there are four commits in a pull request with one commit actually fixes the vulnerability.[4] The other three commits were found to be unrelated, e.g., `"removing fixes for other bug"`. Figure 5.2e shows that the package-side fixing release was available on July 25, 2012.[5]

Interestingly, there is a lag in the vulnerability fix and release step. This example shows that there is an 89 days period between when the fix was created and released for any client to use. I found that `socket.io` merged its fixes on April 27, 2012, however, it was actually released on July 25, 2012 and classified as a package patch landing, i.e., V0.9.7.

3 Package-side Fix Commits and Landing: Preliminaries

From the motivating example in Section 2.3, I reveal how much development activity is directed towards fixing vulnerabilities compare to the other tasks. Thus, I conduct a preliminary study to characterize package-side fixing release at the commit-level which including (i) changes of package-side fixing release, (ii) contents of package-side fixing release. I first highlight the motivation, approach, and analysis to answer my preliminary questions. I then show my data collection and finally provide the results. The following two preliminary questions guide the study:

[1]https://github.com/socketio/socket.io/issues/856

[2]https://snyk.io/vuln/npm:socket.io:20120323

[3]https://github.com/socketio/socket.io/pull/857

[4]https://github.com/socketio/socket.io/commit/67b4eb9abdf111dfa9be4176d1709374a2b4ded8

[5]https://github.com/socketio/socket.io/releases/tag/0.9.7

(PQ_1) What is the prevalence of package patch landing?

- *Motivation* My motivation for PQ_1 is to analyze the package-side fixing release. Different from Decan et al. [32], I manually investigate the version changes in the package-side fixing release itself. My assumption is that *every fix is applied as a package patch landing.*

- *Approach* The approach to answer PQ_1 involves a manual investigation to identify the package-side fixing release, i.e., package major landing, package minor landing, package patch landing. This is done in three steps.

  1. The first step is to extract the fix-related information on GitHub. The extracted information is captured into three types as (i) an issue, (ii) a commit, and (iii) a pull request.
  2. The second step is to identify the release that contains a fix. This step involves an investigation of the package history. From the link in the first step, the first author manually tracked the git commit history to identify when the fix was applied.
  3. The final step is to identify a difference between a vulnerable release and a package-side fixing release. I compare a vulnerable release, i.e., listed in the report, against a package-side fixing release to categorize the changes: (i) package major landing, (ii) package minor landing, and (iii) package patch landing.

- *Analysis* The analysis to answer PQ_1 is the investigation of package-side fixing releases. I use a summary statistic to show the package-side fixing release distribution. Furthermore, an interesting example case from the result is used to confirm and explain my findings.

(PQ_2) What portion of the release content is a vulnerability fix?

- *Motivation* Extending PQ_1, my motivation for PQ_2 is to reveal what kinds of contents are bundled within the package-side fixing release. I complement recent studies, but at the commit-level [32, 33, 51]. I would like to evaluate the assumption that *commits bundled in a package-side fixing release are mostly related to the fixing commits.*

- *Approach* The approach to answer PQ_2 involves a manual investigation of contents inside the package-side fixing release. This is done in three steps.

  1. The first step is to gather information of a fixing commit. In this case, I tracked the git commit history similar to the second step of PQ_1.
  2. The second step is to list commits in a package-side fixing release. I use GitHub comparing changes tool to perform this task.[6]
  3. The final step is to identify the type of commits bundled in a package-side fixing release. Similar to PQ_1, the first author manually tracked and labeled the commit as either (i) fixing commit or (ii) other commit. To label commits, the first author uses source codes, commit messages and GitHub pull request information. For validation, other co-authors confirmed the results, i.e., one author found the evidence and the other validated.

- *Analysis* The analysis to answer PQ_2 is to examine the portion of fixing commits in a package-side fixing release. I show the cumulative frequency distribution to describe the distribution of fixing commits for 231 package-side fixing releases. I use a box plot to show fixing commit size in terms of lines of code (LoC). Similar to PQ_1, I show interesting example cases from the result.

3.1 Data Collection

My dataset contains the vulnerability reports with fix-related information on GitHub. For the vulnerability reports, I crawled the data from snyk.io [122] that were originally disclosed in CVE and CWE database. For the fix-related information, I focus on packages from the npm JavaScript ecosystem that is one of the largest package collections on GitHub [90] and also has been the focus of recent studies [2, 31, 32, 33, 51, 64]. At the time of this study, I extracted fix-related information links directly from snyk.io (e.g., GitHub issue, commit, pull request).

[6] https://help.github.com/en/github/committing-changes-to-your-project/comparing-commits

Table 5.1: A summary of package-side dataset information for preliminary study.

npm Vulnerability Report Information	
Disclosures period	Apr 9, 2009 – Aug 7, 2020
# vulnerability reports	2,373
# samples of vulnerability reports (with fix references)	231
# vulnerable packages	172

Table 5.2: A summary statistic of package-side fixing release distribution in PQ_1.

Package-side fixing release	**# of fixing releases**
Major package landing	17 (7.36%)
Minor package landing	47 (28.14%)
Patch package landing	149 (64.50%)
	231

As shown in Table 5.1, I crawled and collected all reports from April 9, 2009 to August 7, 2020, i.e., in total 2,373 reports. To identify the reports with fix-related information, I removed reports that (i) do not have the fixing release or (ii) do not provide any fix-related information. After that, I randomly select around 237 reports (10% of 2,373) and manually filter reports that the vulnerable package does not follow semantic versioning. In the end, the dataset for PQ_1 and PQ_2 analysis consists of 231 reports that affect 172 packages.

3.2 Results to the Preliminary Study

(PQ_1) What is the prevalence of package patch landing?

Table 5.2 shows the evidence that not every fix is applied as a patch. This evidence contradicts my assumption. I find that 64.50% of fixes are a package patch landing. On the other hand, I find that 7.36% and 28.14% of fixes are package major landing and package minor landing respectively. From my result, I suspect that some releases, especially package major landing and package minor landing might contain unrelated contents to the fixing commits.

The example case is a package major landing of an HTTP server framework,

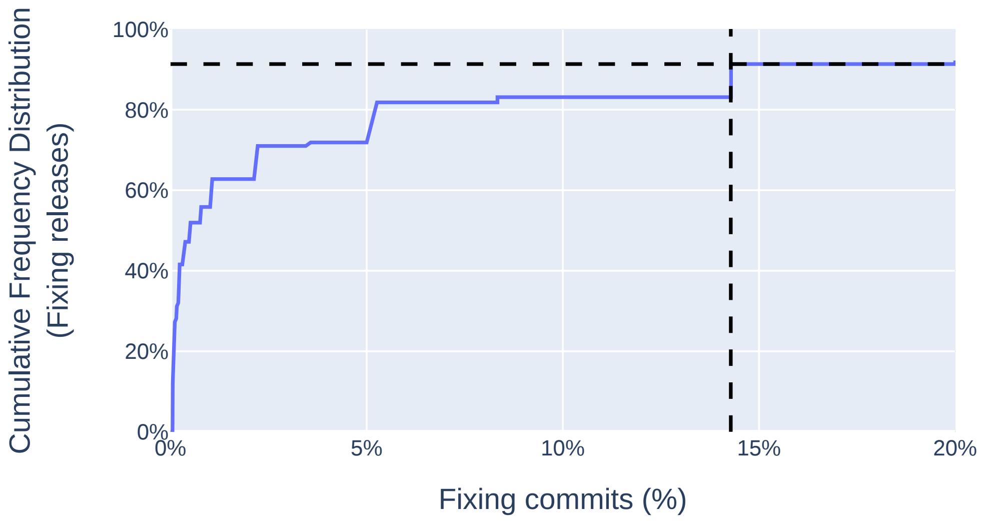


Figure 5.3: I find that 91.77% out of 231 fixing releases have fixing commits up to 14.28% of commits in a package-side fixing release.

i.e., `connect` (V2.0.0). This package was vulnerable to Denial of Service (DoS) attack [123]. Under closer investigation, I manually validated that the other fixes were bundled in a package-side fixing release, including API breaking changes (i.e., removed function).[7] This fix also takes 53 days before it gets released. I suspect that this may cause a lag in the package-side fixing release, especially if the project has a release cycle.

(PQ_2) What portion of the release content is a vulnerability fix?

Figure 5.3 is evidence that fixes are usually bundled with other kinds of changes. I find that 91.77% out of 231 fixing releases have up to 14.28% commits that related to the fix, which means that 85.72% of commits were unrelated. Figure 5.4 shows that the fix itself tends to contain only a few lines of code, i.e., median of 10 LoC. Similar to the commit-level analysis, the package-side fixing release tends to contain a lot of changes, i.e., median of 219 LoC. These results complement the finding of PQ_1 about package-side fixing release might contain unrelated changes to the fixing commit.

[7]https://github.com/senchalabs/connect/blob/fe28026347c2653a9602240236fc43a8f0ff8e87/History.md#200--2011-10-05

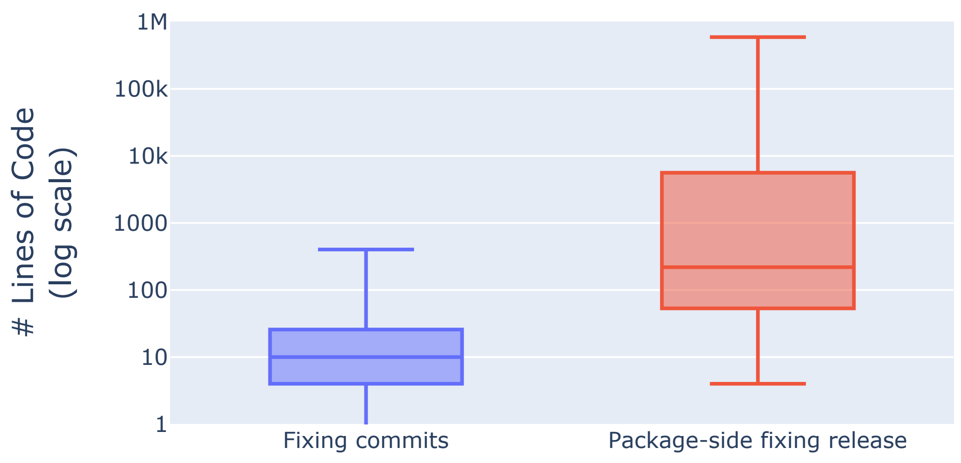


Figure 5.4: LoC of the fixing commits for 231 vulnerabilities. I find that there are only few fixing fix commits in the package-side fixing release, i.e., median of 10.

I show two examples to investigate the content of a fix and its size. The first example is a package patch landing of a simple publish-subscribe messing for a web, i.e., `faye` [124]. Under closer manual inspection, I find that there is one commit that updates the default value of variables.[8] However, a package patch landing includes a total of 45 commits that is not related to the fix.[9] In the second example, I show that the actual fix is only a few lines of code. The `npm` package, which is the command line interface of a JavaScript package manager [125], took seven lines of code to fix the vulnerability.[10]

[8] https://github.com/faye/faye/commit/e407e08c68dd885896552b59ce65503be85030ad

[9] https://github.com/faye/faye/compare/0.8.8...0.8.9

[10] https://github.com/npm/npm/commit/f4d31693

4 Summary

In this chapter, I first find the prevalence of package patch landing. The results shows that the vulnerability fix is not always released as its own patch update. Also, only 64.50% of fixing releases are a package patch landing. The rest of fixing releases are either package major landing, i.e., 7.36%, or package minor landing, i.e., 28.14%. After that, I show the portion of the release content that is a vulnerability fix. I find that a small portion of the release contents related to the vulnerability fix, with 91.77% of 231 fixing releases have up to 85.72% unrelated commits. Furthermore, the fix itself tends to have only a few lines of code (i.e., median of 10 lines of code). The results shows the characterization of vulnerability fixing process of vulnerable packages. Chapter 6 looks further than packages by characterizing the fix adoption of their direct clients and the fix propagation to transitive clients.

6 | Lags in the Adoption and Propagation of Package-side Fixes

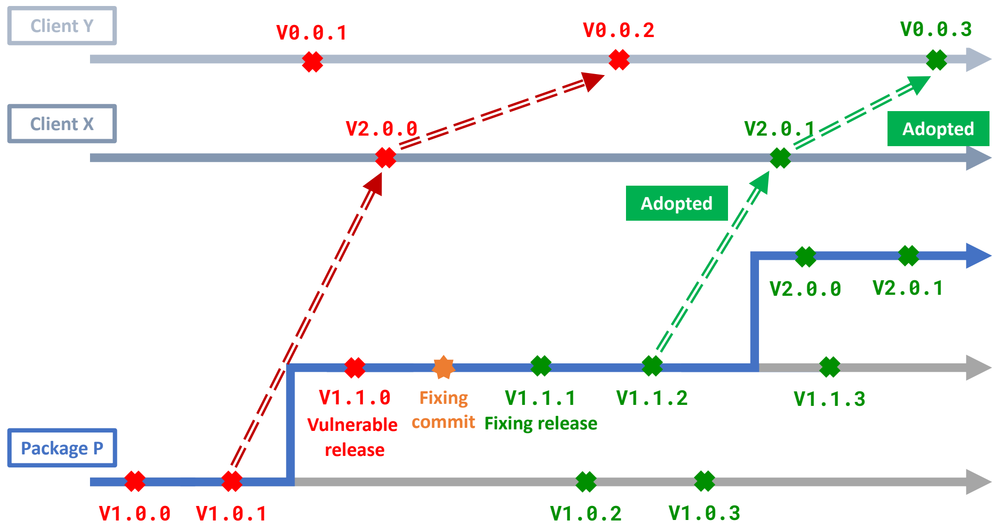
1 Introduction

The results of my preliminary study in Chapter 5 characterize the package-side fixing release, where I find that (i) up to 64.50% of vulnerability fixes are classified as a package patch landing and (ii) up to 85.72% of commits in a release are unrelated to the actual fix. Based on these results, I suspect that potential lags might occur while the package-side fixing release get adopted by the clients and transitively propagate throughout the dependency network. Hence, I perform an empirical evaluation to explore potential lags in the adoption and propagation of the fix.

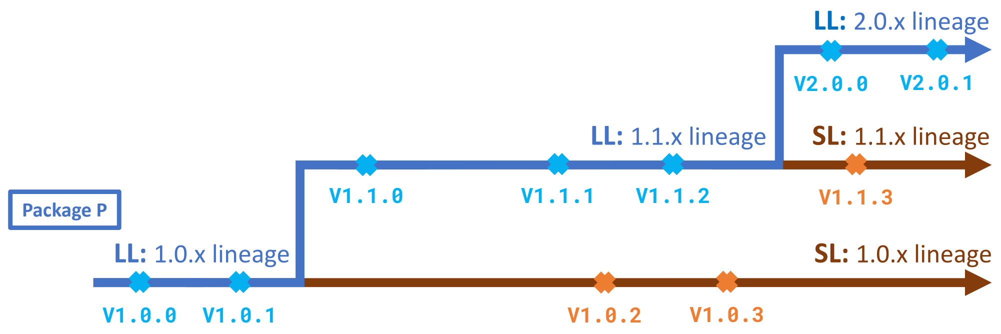
2 Model and Track Lags

To explore potential lags in both adoption and propagation, I model and track the package-side fixing release and client-side fixing release as illustrated in Figure 6.1.

Released and Adopted by Version - I identify lags in the adoption by analyzing the prevalence of patterns between a package-side fixing release and



(a) An example of a lag in the propagation, caused by cascade delays from upstream clients.



(b) An example of a Latest Lineage (LL: a package-side fixing release in the latest branch at any given point of time) and a Supported Lineage (SL: a package-side fixing release in the outdated branch at any given point of time) classify to track the freshness of a package-side fixing release.

Figure 6.1: These figures show the terms that are used to model and track the lags.

client-side fixing release, which is similar to technical lag Zerouali et al. [147] and based on semantic versioning. The definition of *package-side fixing release* was explained in Section 2.1 which describes how the package bumped the release version number. Note that pre-releases or special releases are not considered in this study. I then define a new term called a *client-side fixing release*. Client-side fixing release describes how clients bumped the version of an adopted package up from vulnerable version to fixing release. There are four kinds of client-side fixing release: (i) `client major landing` (major number of an adopted package is bumped up), (ii) `client minor landing` (minor number of an adopted package is bumped up), (iii) `client patch landing` (patch number of an adopted package is bumped up), and (iv) `dependency removal` (adopted package is removed from a client dependency list).

Figure 6.1a shows an example of the two terms defined above. First, I find that the package-side fixing release for package $\mathbb{P}$ is classified as a package patch landing. This is because of the difference between a fixing release ($\mathbb{P}_{V1.1.1}$) and its vulnerable release ($\mathbb{P}_{V1.1.0}$). Furthermore, I find that the client-side fixing release for client $\mathbb{X}$ is a client minor landing. This is because of the difference between the adopted fixing release ($\mathbb{P}_{V1.1.2}$) and its previous vulnerable release ($\mathbb{P}_{V1.0.1}$). In the reality, client $\mathbb{X}$ could decide to adopt $\mathbb{P}_{V1.0.3}$ instead of $\mathbb{P}_{V1.1.2}$. In this case, the client-side fixing release for client $\mathbb{X}$ is a client patch landing.

Propagation Influencing Factors - I define *Hop* as the transitive dependency distance between a package-side fixing release and any downstream clients that have adopted this fix, i.e., one, two, three, and more than or equal to four hops. As shown in Figure 6.1a, client $\mathbb{X}$ is one hop away from package $\mathbb{P}$. I consider two different factors to model and track lags in the propagation:

1. *Lineage Freshness:* refers to the freshness of the package-side fixing release as inspired by Cox et al. [23] and Kula et al. [67]. Figure 6.1b shows two types of lineage freshness based on the release branches including: *Latest Lineage (LL)*: the client has adopted any package-side fixing release on the latest branch, and *Supported Lineage (SL)*: the client has adopted any package-side fixing release not on the latest branch. My assumption is that a package-side fixing release in the latest lineage is adopted faster than a

package-side fixing release in a supported lineage, i.e., suffer less lags. Figure 6.1b shows that three versions of package $\mathbb{P}$ (V1.0.2, V1.0.3, V1.1.3) are classified as SL.

2. *Vulnerability Severity:* refers to the severity of vulnerability, i.e., H = high, M = medium, L = low, as indicated in the vulnerability report (as shown in Figure 5.2a from Section 2). My assumption is that a package-side fixing release with higher severity is adopted quicker, i.e., less lags.

3 Empirical Evaluation

The goal of my empirical study is to investigate lags in the adoption and propagation. I use these two research questions to guide my study:

(RQ_1) Is the package-side fixing release consistent with the client-side fixing release? My motivation for RQ_1 is to understand whether developers are keeping up to date with the package-side fixing releases. I define that package-side and client-side fixing releases are consistent if client-side fixing release follow package-side fixing release. For example, client minor landing and package minor landing combination is consistent, but client major landing and package patch landing combination is not consistent. My key assumption is that the inconsistent combination requires more migration effort than the consistent one, which in turn is likely to create lags.

(RQ_2) Do lineage freshness and severity influence lags in the fix propagation? My motivation for RQ_2 is to identify the existence of lags during a propagation. Concretely, I use my defined measures, i.e., propagation influencing factors, to characterize a propagation lag. My assumption is that a package-side fixing release on the latest lineage with high severity should propagate quickly.

Data Collection - My data collection consists of (i) vulnerability reports and (ii) the set of cloned npm package and client git repositories. I use the same 2,373 vulnerability reports as shown in my preliminary study which crawled from snyk.io [122]. As inspired by Wittern et al. [142], I cloned and extracted information of npm package and client from public GitHub repositories. In this study, I consider only normal dependencies listed in the `package.json` file to make sure that the packages are used in the production environment. Hence, other types of dependencies including: (1) devDependencies, (2) peerDependencies, (3) bundledDependencies, and (4) optionalDependencies are ignored in this study since they will not be installed in the downstream clients in the production or cannot be retrieved directly from the npm registry. To perform the lags analysis, I first filter reports that do not have the fixing release. I then used the package name and its GitHub link from the reports to automatically match cloned repositories.

As shown in Table 6.1, my data collection included 2,373 vulnerability reports that disclosed from April 9, 2009 to August 7, 2020. There are 1,290 reports that already published the fixing releases which affect 786 different packages. The statistics of vulnerable packages and reports are presented in the table. For package and client repositories, I collected a repository snapshot from GitHub on August 9, 2020 with 152,074 repositories, 611,468 dependencies, and 1,553,325 releases.

Approach to Answer RQ1 - The data processing to answer RQ_1 involves the package-side fixing release and client-side fixing release extraction. Similar to PQ_1, I first identify the package-side fixing release by comparing a vulnerable release and a fixing release. To track the client-side fixing release, I then extract the direct clients' version history of the vulnerable packages. A client is deemed vulnerable if its lower-bound dependency falls within the reported upper-bound as listed in a vulnerability report.

To ensure quality, I additionally filter out packages and clients that did not follow semantic versioning as shown Table 6.3. My key assumption is to keep packages and clients that follow a semantic version release cycle, i.e., packages and clients should have all the update patterns of major landing, minor landing, and patch landing. As a result, 4,000 packages and clients were filtered out from the dataset. As shown in Table 6.2, my final dataset for RQ_1 consists of 410 vulnerability reports that affect 230 vulnerable packages and 5,417 direct clients.

The analysis to answer RQ_1 is the identification of lags in the adoption. I show

Table 6.1: A summary of the data collection which used to populate the dataset to answer RQ_1 and RQ_2.

npm JavaScript Ecosystem Information	
Repository snapshot	Aug 9, 2020
# package & client GitHub repositories	152,074
# total dependencies (with downstream)	611,468
# total packages releases	1,553,325
npm Vulnerability Report Information	
Disclosures period	Apr 9, 2009 – Aug 7, 2020
# reports	2,373
# reports (with fixing release)	1,290
# vulnerable packages	786
# vulnerabilities per package	
- mean	1.64
- median	1.00
- SD	2.58
# high severity vulnerabilities	566
# medium severity vulnerabilities	647
# low severity vulnerabilities	77

the frequency distribution of client-side fixing release in each package-side fixing release. In order to statistically validate my results, I apply Pearson's chi-squared test (χ^2) [104] with the null hypothesis *'the package-side fixing release and the client-side fixing release are independent'*. To show the power of differences between each package-side fixing release and client-side fixing release combination, I investigate the effect size using Cramér's V (ϕ'), which is a measure of association between two nominal categories [24]. According to Cohen [20], since the contingency Table 6.4 has 2 degrees of freedom (df*), effect size is analyzed as follows: (1) $\phi' < 0.07$ as Negligible, (2) $0.07 \leq \phi' < 0.20$ as Small, (3) $0.20 \leq \phi' < 0.35$ as Medium, or (4) $0.35 \geq \phi'$ as Large. To analyze Cramér's V, I use the `researchpy` package.[1]

[1] https://pypi.org/project/researchpy/

Table 6.2: A summary of dataset information for the empirical study to answer RQ_1 and RQ_2.

RQ_1 **Dataset**	
# vulnerability reports (follow semver)	410
# vulnerable packages (follow semver)	230
# direct clients (follow semver)	5,417
# filtered clients (break semver)	4,000
RQ_2 **Dataset**	
# vulnerability reports (with fix released)	617
# vulnerable packages (with fix released)	344
# downstream clients	416,582

Table 6.3: A summary number of filtered clients grouped by their update pattern in RQ_1. There are 4,000 packages and clients that excluded in the RQ_1.

semver Update Patterns	**# clients**
Major only	94
Minor only	293
Patch only	2,812
No change	801
Filtered packages & clients	4,000

Approach to Answer RQ2 - The data processing to answer RQ_2 involves propagation influencing factors extraction. There are three steps to track downstream clients and classify lineage freshness and severity. First, I build and traverse in a dependency tree for each package-side fixing release using a breadth-first search (BFS) approach. The meta-data is collected from each downstream client which includes: (i) version, (ii) release date, and (iii) dependency list, i.e., exact version and ranged version. I then classify whether or not a client is vulnerable using an approach similar to RQ_1. My method involves removing duplicated clients in the dependency tree, which is caused by the npm tree structure. Second, I classify the lineage freshness of a fixing release by confirming that it is on the latest branch. Finally, I extract the vulnerability severity from the report. As shown in Table 6.2, my final dataset for RQ_2 consists of 617 vulnerability reports, 344 vulnerable packages with fixing releases, and 416,582 downstream clients.

The analysis to answer RQ_2 is the identification of lags in the propagation. I show a summary statistic of lags in terms of days, i.e., the mean, the median, the standard deviation, and the frequency distribution, with two influencing factors. In order to statistically validate the differences in the results, I apply Kruskal-Wallis non-parametric statistical test [66]. This is a one-tailed test.[2] I test the null hypothesis that *'lags in the latest and supported lineages are the same'*. I investigate the effect size using Cliff's δ, which is a non-parametric effect size measure [117]. Effect size are analyzed as follows: (1) $|\delta| < 0.147$ as Negligible, (2) $0.147 \leq |\delta| < 0.33$ as Small, (3) $0.33 \leq |\delta| < 0.474$ as Medium, or (4) $0.474 \leq |\delta|$ as Large. To analyze Cliff's δ, I use the `cliffsDelta` package.[3]

4 Results to the Empirical Study

(RQ_1) Is the package-side fixing release consistent with the client-side fixing release? My results are summarized into two findings. First, Table 6.4 shows the evidence that most of package-side fixing releases are package patch landings. As shown in the first row of a table, I find that 245 out of 410 fixing releases have package patch landings (highlighted in red). I also find that

[2] https://github.com/scipy/scipy/issues/12231#issuecomment-660404413
[3] https://github.com/neilernst/cliffsDelta

Table 6.4: A contingency table shows the frequency distribution of client-side fixing release for each package-side fixing release. I find that (i) there is a dependency between package-side fixing release and client-side fixing release and (ii) only the case of package minor landing is consistent.

	Package major landing (66)	Package minor landing (99)	Package patch landing (245)	**All**
Client major landing	448 (46.18%)	453 (30.00%)	657 (22.37%)	1,558
Client minor landing	2 (0.21%)	761 (50.40%)	1,082 (36.84%)	1,845
Client patch landing	0 (0.00%)	0 (0.00%)	625 (21.28%)	625
Dependency removal	520 (53.61%)	296 (19.60%)	573 (19.51%)	1,389
All	970	1,510	2,937	5,417

there are 66 package major landings and 99 package minor landings. This finding complements the result of PQ_1.

Second, Table 6.4 shows the evidence that there is a dependency between package-side fixing release and client-side fixing release variables. However, there is no consistency across package-side fixing release and client-side fixing release. As highlighted in *Client patch landing* row of Table 6.4, I find that there are only 21.28% of clients adopt a package patch landing as client patch landings. Instead, clients are more likely have client minor landings, i.e., 36.84% of clients (highlighted in red). For the case of package major landing, there are 53.61% of clients remove their dependencies to avoid vulnerability (highlighted in yellow). The majority of clients that still adopt the package major landing are around 43.18% as client major landing. The only case that I find consistent is package minor landing which 50.40% of clients adopt the fix as client minor landing (highlighted in green).

For the statistical evaluation, I find that there is an association between the package-side fixing release and the client-side fixing release. Table 6.5 shows that my null hypothesis on *'the package-side fixing release and the client-side fixing release are independent'* is rejected (i.e., $\chi^2 = 1,484.48$, *p-value* < 0.001). From the Cramér's V effect size (ϕ'), I got a value of 0.37 which shows the large level of association.

Table 6.5: A result of statistical test for RQ_1. I find that differences between each distribution are significant and have a large level of effect size.

Statistic	**Value**
Pearson's chi-squared test (χ^2)	$1,484.48$
p-value	< 0.001
Cramér's V (ϕ')	0.37

Answer to RQ_1: No for the case of package patch landing, but yes for the others. I find that only 21.28% of package patch landing are adopted as a client patch landing in clients. Instead, 36.84% of package patch landing are adopted as a client minor landing. The evidence suggests that since clients keep stale dependencies, more migration effort is required to fix that client due to the potential risk from backward incompatible changes (i.e., client major landing or client minor landing).

Table 6.6: A summary statistic of lags in the propagation (# days) categorized by lineage freshness to show the difference between lags in LL and SL. Lags in the table is not accumulative.

	# hop	**# clients**	**Mean**	**Median**	**SD**
LL	1	18,444	311.69	164.00	370.92
	2	55,044	299.70	157.00	359.80
	3	74,257	255.21	130.00	313.86
	≥ 4	239,128	212.11	112.00	270.62
		386,873			
SL	1	2,880	217.90	89.00	298.30
	2	5,675	269.55	139.00	318.41
	3	5,948	216.43	101.00	276.27
	≥ 4	15,206	181.55	96.00	240.11
		29,709			

Table 6.7: A summary statistic of lags in the propagation (# days) categorized by vulnerability severity to show the difference of lags between high, medium, and low severity vulnerability fixes. Lags in the table is not accumulative.

	# hop	**# clients**	**Mean**	**Median**	**SD**
H	1	5,569	187.09	91.00	239.72
	2	16,160	212.04	103.00	277.36
	3	18,444	190.99	88.00	255.29
	≥ 4	34,189	184.91	94.00	242.13
		74,362			
M	1	14,320	350.54	194.00	399.91
	2	39,758	341.29	193.00	386.66
	3	55,625	280.18	151.00	331.87
	≥ 4	210,571	215.75	116.00	274.23
		320,274			
L	1	1,435	219.39	123.00	248.54
	2	4,801	214.78	127.00	246.28
	3	6,136	184.29	98.00	223.92
	≥ 4	9,574	180.46	94.00	234.38
		21,946			

(RQ_2) Do lineage freshness and severity influence lags in the fix propagation? My results are summarized into two findings. First, Table 6.6 shows the evidence that the lineage freshness influences lags in a propagation. As highlighted in red, I find that LL has more lags than SL in terms of days for every hops, e.g., median of lags for the first hop: 164 days > 89 days.

Second, Table 6.7 shows the evidence that the vulnerability severity influences lags in a propagation. As highlighted in green, I find that the high severity fixing release has the least lags than others in every hop, e.g., the first hop: 91 days. I also find that the medium severity fix has the most lags than others as highlighted in red, e.g., the first hop: 194 days.

For the statistical evaluation, I find that lags in the latest and supported lineage showed to have a significant (p-value < 0.001), but negligible to small

Table 6.8: A comparison of lags in the propagation between clients that adopt the latest lineage and supported lineage fixing release, i.e., by the median. I find that difference between each distribution is significant mostly in the case of medium severity. The effect sizes of those differences are negligible and small level. (LL: median of the latest lineage, SL: median of the supported lineage, *: p-value < 0.001).

# hop	**H**	**M**	**L**
1	LL > SL negligible	LL > SL * small	LL > SL negligible
2	LL > SL * negligible	LL > SL * negligible	LL > SL * small
3	LL > SL negligible	LL > SL * negligible	LL > SL negligible
≥ 4	LL > SL negligible	LL > SL * negligible	LL < SL * negligible

association. Table 6.8 shows that my null hypothesis on whether *'lags in the latest and supported lineages are the same'* is rejected, i.e., the first hop to the more than the fourth hop for medium severity, the second hop and more than the fourth hop for low severity; and the second hop for high severity.

Answer to RQ_2: Yes, lineage freshness and severity influence lags in the propagation. I find that fixing releases that occur on the latest lineage and medium severity suffer the most lags.

5 Discussion

5.1 Lessons Learned

This section discusses three main implications based on my results in PQ_1, PQ_2, RQ_1, and RQ_2. These are presented as lessons learned and could have implications for both practitioner and researcher.

1. Release cycle matters. According to the results of PQ_2, fixing commits are small with less than 14.28% of fixing commits in the package-side fixing release. I suspect that vulnerability fix repackage is a cause for lags. Hence, developers of the vulnerable packages are recommended to release fixes as soon as they have applied the fix, if not, they should highlight these fixes when bundling the fix. Additionally, from 10 randomly selected vulnerabilities, I found that discussions between developers did not include an explicit mention of the vulnerability, i.e., GitHub issue, commit, and pull request. Since developers bundled the fix with other updates, developers may have been unaware. *In summary, researchers should provide strategies for making the most efficient update via the release cycle. For example, (i) releasing an emergency patch that does not introduce any new features for security fixes to maintain backward compatibility and (ii) providing a security support for an active version, i.e., long-term-release version. Furthermore, practitioners can upgrade security fixes as first class citizens, so that the vulnerability fix can travel quicker throughout the ecosystem.*

2. Awareness is important. According to PQ_1 and RQ_1, 64.50% of fixes are a package patch landing. However, clients are more likely to have client major landing and client minor landing, i.e., 22.37% and 36.84%, than the patch client landing, i.e., 21.28%. Security fixes need to be highlighted in the update note, as a possible reason is for failure to update because client developers are more interested in major features that are highlighted in an update. Recently, some open source communities start to make tools to highlight the vulnerability problems in a software ecosystem. GitHub [42, 47] made a new function for notifying a new vulnerability from the dependency list of clients by using a bot. However, GitHub stated that the tool will not be able to catch everything and send the alert notification within a guaranteed time frame. Also, npm [94] made a new command for listing the vulnerability information in downloaded dependencies of clients and try to automatically fix them called `npm audit`. However, there are some cases that `npm audit` does not work. For example, the immediate dependency does not adopt the package-side fixing release from the vulnerable package [95]. In these cases, a manual review is required. Thus, client

developers have to wait for the propagation of the fixing release, i.e., a lag exists. According to RQ_1, 1,389 of 5,417 clients, i.e., 25.64%, decided to remove vulnerable dependencies to mitigate the risk of vulnerabilities. Instead of waiting for the fix, clients might remove the vulnerable dependency if they are able to find a similar package as a replacement or do not want to use the vulnerable dependency anymore.

From a result of PQ_2, explicit package-side fixing release with a highlight of the vulnerability is needed to speed up the adoption. *In summary, researchers and practitioners need to provide developers more awareness mechanisms to allow quicker planning of the update. The good news is current initiatives like GitHub security are trending towards this.*

3. Migration cost effort. From my first finding of RQ_2, the package-side fixing release in the latest lineage suffers more lags than the supported lineage in terms of days. A possible reason is that migrating to the latest version of packages required a new testing process that incurs some cost for the downstream clients. This also might be due to the risk from using the latest version as sometimes they are unstable, introducing new bugs, or breaking changes.

   In terms of security, according to the official documentation of npm [93], when a security threat is identified, the following severity policy is put into action: (a) P0: Drop everything and fix!, (b) P1: High severity, schedule work within 7 days, (c) P2: Medium severity, schedule work within 30 days, (d) P3: Low severity, fix within 180 days. Surprisingly, the second finding of RQ_2 shows evidence that low severity fixes are adopted quicker than medium severity fixes. A possible reason for the quicker adopting of low severity could be because the fix is easier to integrate into the application. *In summary, researchers and practitioners that are package developers in npm seem to require additional time before updating their dependencies. Although migrating to the latest version of packages might require extra effort, they still have to prioritize their works and consider adopting the vulnerability fixes if possible.*

5.2 Threats to Validity

Internal Validity - I discuss three internal threats. The first threat is the correctness of the tools and techniques used in this study. I use the listed dependencies and version number as defined in the `package.json` meta-file. The threat is that sometimes some dependencies are not listed or the semantic version is invalid and vice-versa, so I applied a filter to remove clients that do not follow the semantic versioning, thus making this threat minimal. The second threat is the tools used to implement my defined terminology (i.e., `numpy`, `scipy`, `gitpython`, and `semantic-version`). To mitigate this, I carefully confirmed my results by manually validating the results for RQ_1 and RQ_2, then also manually validating results with statistics on the npm website. For the existing tool for suggesting the package-side fixing release like `npm audit`, I found that this tool was inappropriate to use in this work due to its limitation of the `package-lock.json` file is required for analyzing repositories, which only 2.27% of repositories of the dataset and the tool assumes the latest information. Unlike my work, I analyzed the data available in the historical snapshot. As the correctness of dependency relations depends on getting all dependencies, the final internal threat is the validity of my collected data. In this study, my ecosystem is made up of packages and clients that were either affected directly or transitively by at least a single vulnerability. I also make sure that the packages and their repositories are actually listed on a npm registry. I are confident that the results of RQ_2 are not affected by invalid data.

External Validity - The main external threat is the generality of other results to other ecosystems. In this study, I focused solely on the npm JavaScript ecosystem. However, my analysis is applicable to other ecosystems that have similar package management systems, e.g., PyPI for Python, Maven for Java. Immediate future plans include studying the lags in other ecosystems. Another threat is the sample size of the analyzed data. In this study, I analyzed only 1,290 vulnerability reports with package-side fixing releases from 2,373 extracted reports from snyk.io. This small size of sample data might not be able to represent the population. However, I are confident of the data quality and reduced bias as I followed strict methods to validate by two authors for PQ_1, PQ_2, and RQ_1 data.

Construct Validity - The key threat is that there may be other factors apart from the two factors, i.e., lineage freshness and vulnerability severity. These factors are based on prior studies, i.e., measuring of dependency freshness from Cox et al. [23], exploring the impact of vulnerability to transitive dependencies from Kikas et al. [64], responding to a vulnerability NPM [93]. For future work, I would like to investigate other factors.

6 Summary

Security vulnerability in third-party dependencies is a growing concern for software developers as the risk of it could be extended to the entire software ecosystem. To ensure quick adoption and propagation of a fixing release, I conduct an empirical investigation to identify lags that may occur between the vulnerable release and its fixing release from a case study of npm JavaScript ecosystem. I found that the package-side fixing release is rarely released on its own, with up to 85.72% of the bundled commits in a package-side fixing release being unrelated to the fix (Chapter 5). I then found that a quick package-side fixing release (i.e., package patch landing) does not always ensure that a client will adopt it quicker, with only 17.69% of clients matching a package patch landing to a client patch landing (Chapter 6). Furthermore, factors such as the lineage freshness and the vulnerability severity have a small effect on its propagation (Chapter 6).

In addition to theses lags that I identified and characterized, this chapter lays the groundwork for future research on how to mitigate these propagation lags in an ecosystem. I suggest that researchers should provide strategies for making the most efficient update via the release cycle. Practitioners also need more awareness to allow quicker planning of the update. Potential future avenues for researchers include (i) a developer survey to a better understanding of the reason for releasing and adopting fixes, (ii) a performance improvement plan for highlighting the fixing release tool, (iii) a tool for managing and prioritizing vulnerability fixing process. As a result, Chapter 7 introduces the vulnerability assessment tools for prioritizing vulnerability fixing process by detecting vulnerable codes and their execution traces inside vulnerable packages.

7 | Vulnerability Assessment Tools

1 Introduction

As of 2020, the Node.js package manager (i.e., npm) is reported to serve over 1.3 million packages to roughly 12 million developers, who download such packages 75 billion times a month, and all at a growing rate [132]. Furthermore, as evidence of its influence, the industry giant Microsoft's GitHub had completed its acquisition of npm earlier in April, 2020.

Raising the awareness for developers to quickly update their third-party dependencies is now regarded seen as the priority [64, 68], especially if the threat includes malice intent. As well as fixing bugs and adding new features, migration to a new version (i.e., update) sometimes includes fixes to prevent these threats. Such threats on dependency are regarded as *vulnerable dependency*. Recent epidemic vulnerabilities such as `heartbleed` [1] are examples of how vulnerable dependencies can affect all users in an ecosystem of users such as the `npm` ecosystem. Recent studies have shown evidence that known vulnerabilities can affect both open source and industrial applications alike [129].

Most detection methods for vulnerabilities has been at meta-detection [32, 33, 42, 64, 94, 147]. Meta-detection capabilities rely on the assumption that the metadata associated to Open Source Software (OSS) libraries (e.g., name, version), and to vulnerability descriptions (e.g., technical details, list of affected components) are always available and accurate. The metadata, which are used to map each library onto a list of known vulnerabilities that affect it, are often incomplete, inconsistent, or missing altogether. Recent studies [52, 109, 146] have supported the claim of overestimation of vulnerability alerts, where the client does not actually call the vulnerable code. Ponta et al. [109] showed that, for `Java` projects, many clients do not actually call the affected function.

Ponta et al. [108, 109] proposed *Eclipse-Steady*, a code-centric and usage-based approach to detect open source vulnerabilities. The *Eclipse-Steady* project [36] is able to identify, assess and mitigate open source dependencies with known vulnerabilities for Java and Python industry grade applications. It supports software development organizations in regard to the secure use of open source components during application development. A code-centric approach reduces the number of false positives and false negatives as it accounts for the actual presence of vulnerable constructs (i.e., constructs that are modified by the patch), no matter where they occur [108, 109]. Having identified the vulnerable constructs, it is then possible to establish whether they are reachable in the context of an application thereby assessing the potential impact of the vulnerability.

Lauinger et al. [70] provides evidence that JavaScript issues are prevalent in most web applications, strengthening the argument for a Node.js code-centric solution. Due to the dynamic event-based nature of JavaScript code, the performance of a code-centric approach is unknown.

To address this gap, in this chapter, I present two supporting tools to detect open source vulnerabilities and their reachability in the Node.js application. Section 2 present a code-based vulnerability detection in Node.js applications by extending the industry-grade tool (i.e., *Eclipse-Steady*). Section 3 present a prototype of a vulnerability function reachability detection tool (i.e., SōjiTantei).

2 *Eclipse-Steady*: Node.js Vulnerable Code Detection Extension

2.1 Overview

In this section, I present an experience report on a code-centric approach to detect open source vulnerabilities using bill of materials to determine whether vulnerable code is repackaged within Node.js applications. First, I discuss the challenges associated with the construction of bill of materials of Node.js applications. I then propose my solution to counter these challenges. To evaluate my approach, I perform a case study on 65 Node.js applications under development at SAP. Preliminary results show that my method is viable, with vulnerable code from five vulnerabilities being detected in 18 applications under development. The study highlights three lessons learned and the challenges that require attention by both researchers and practitioners dealing with Node.js applications and JavaScript in general.

Table 7.1: Defined List of Constructs in hierarchical chain for Node.js Applications. This is based on Figure 7.1 and Listing 7.1

Construct Type	Description	Fully Qualified Name
Package (PACK)	Package and Directory name	ProjectA
Module (MODU)	File name	ProjectA.utils.util_b
Function (FUNC)	Function name with arguments	ProjectA.utils.util_b.buy(item)
Class (CLAS)	Class name with extended class	ProjectA.utils.util_b.Car()
Method (METH)	Method name with arguments	ProjectA.utils.util_b.Car().drive(distance,direction)
Constructor (CONS)	Constructor name with arguments	ProjectA.utils.util_b.Car().constructor(name,age)
Object (OBJT)	Object name	ProjectA.utils.util_b.item_list

2.2 Perils of JavaScript Node.js Analysis

Analysis of JavaScript code is not trivial, as server-side Node.js applications (including npm packages) involve sockets, streams, and files performed in an asynchronous manner, where the execution of listeners is triggered by events [77]. The challenge for such dynamic code is the proper identification of a function call, which has been the issue for static analysis tools [28, 127]. Moreover, JavaScript allows anonymous functions, i.e., functions without a name [136]. To avoid this complexity of the reachability analysis, my approach is based on the detection of vulnerable code.

I reuse the approach proposed by Ponta et al. [108, 109], where a vulnerability is detected whenever an application dependency contains *program constructs* (such as methods) that were modified, added, or deleted to fix that vulnerability. I extend *Eclipse-Steady* to support the analysis of JavaScript code [86]. In particular, I add the ability to construct the list of program constructs modified to fix JavaScript vulnerabilities, as well as the list of program constructs which are part of a JavaScript application and dependencies (its bill of materials).

2.3 Bill of Materials for Node.js

When compared to the classical model (i.e., Java or C++), JavaScript does not provide a true class implementation. Instead, it has only the object construct with its private property (i.e., prototype) to imitate the constructs from the classical model [140]. A `program construct` is defined as a set of structural elements with a language, a type, and a unique fully-qualified name identifier as defined in Ponta et al. [108, 109].

Constructs for a Node.js application

```
class Car {
    constructor(name, age) { ...
    }
    drive(distance, direction) { ...
    }
}
var item_list = { ...
}
function buy(item) { ...
}
```

Listing 7.1: Example of a class of util_b.js

Figure 7.1 illustrates a hierarchical structure of a Node.js project, which will be used as my running example. Complementary, Listing 7.1 shows a code snippet from the JavaScript file `util_b.js` of Figure 7.1. I use these running examples to explain my proposed constructs.

Table 7.1 shows a summary of the seven construct types I use for Node.js applications. I now explain each construct in detail. The `PACK` construct represents an application scope and its internal directories e.g., `ProjectA`, `/utils`. The `MODU` construct represents a JavaScript file in an application e.g., `util_a.js`. The `FUNC` construct represents a function declaration in a `MODU` e.g., `buy(item)`.

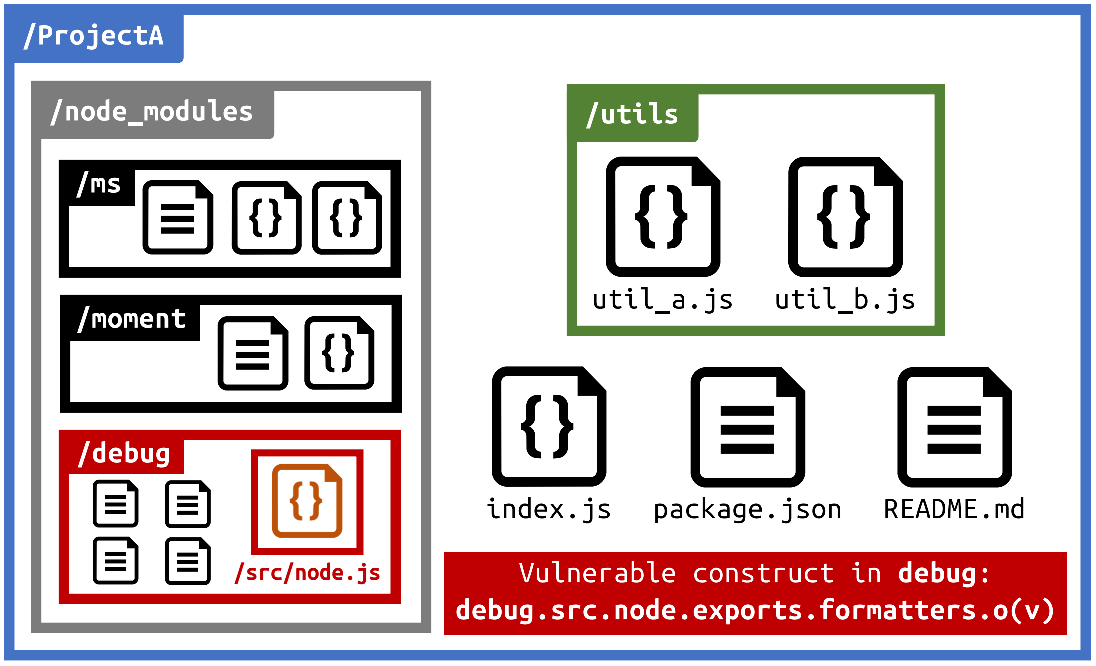


Figure 7.1: Running example of the Node.js project with its hierarchical structure.

The `CLAS` construct represents a class declaration in a `MODU` e.g., `Car()`. The `METH` construct represents a method declaration in a `CLAS` e.g., `drive(distance, direction)`. The `CONS` construct represents a constructor declaration in a `CLAS` e.g., `constructor(name, age)`. The `OBJT` construct represents an object in a `MODU` e.g., `item_list`.

As shown in Table 7.1, I use the PACK construct (i.e., ProjectA) hierarchy to form my fully qualified name. Since JavaScript does allow for anonymous functions, classes, or objects, I use the (LoC) position for the fully-qualified name of anonymous constructs.

Dependency Constructs and their Features

The program constructs defined in Section 2.3 are also used to obtain the bill of materials of the third-party dependencies that are contained within the application (i.e., npm package). Following my running example in Figure 7.1, I use the `package.json` configuration file and the `nodes_modules` directory. In my example, the vulnerable construct is in the debug package at the `OBJT` level (i.e., debug.src.node.exports.formatters.o(v))

```
...
 },
 "dependencies": {
   "moment": "2.25.3"
 },
 "devDependencies": {
   "debug": "3.0.0"
 },
 ...
```

Listing 7.2: Dependency snippet from package.json

Listing 7.2 shows that my example `ProjectA` depends on the packages `debug` and `moment`. On top of collecting the bill of materials of each dependency, I analyze them according to two features. The first is related to whether or not the dependency will be used in production. There are two types of dependencies. Runtime dependencies (i.e., `moment` package) are those intended to be used in production. Test dependencies (i.e., `debug` package) are intended as development-only packages, unneeded in production.

```
ProjectA@1.0.0 /ProjectA
|--- debug@4.1.1
|  |--- ms@2.1.2
|--- moment@2.25.3
```

Listing 7.3: Dependency tree of ProjectA

The second feature is the dependency tree depth. Listing 7.3 shows the dependency tree that depicts the relationships among the packages `debug`, `moments` and `ms`. There are two types of dependencies: direct and transitive. *Direct* dependencies are directly required by the application. As shown in the example, the packages `debug` and `moment` are direct dependencies and are listed in the `package.json` file. *Transitive* dependencies are not directly required by the application but are required by its dependencies. As shown in Listing 7.3, the `ms` package is a transitive package required by `debug`.

Table 7.2: Experimental Dataset

OSS npm package vulnerabilities	
number of vulnerabilities	60
number of vulnerable package	24
number of valid vulnerabilities	32
number of valid vulnerable package	15
Industrial Node.js applications	
number of applications	65
number of valid application	42

2.4 Case Study of Node.js Applications

To evaluate my proposed constructs, I conducted an assessment of vulnerable code from under-development projects at SAP.

Experiment Design

The experiment consisted of the detection of a set of known open source vulnerabilities against a set of industrial applications. Note that my experiment was conducted in September, 2019.

Bill of materials extraction I create a bill of materials (BOM) which consists of construct lists of the application and its dependencies (i.e., both direct and transitive). To do this, I use ANTLR-v4 with a JavaScript grammar [7] to model and extract the Node.js application source code. This grammar is able to partially extract JavaScript with ES6 features at the beginning of my development (July 24, 2019).

To obtain the BOM from an application, I first download the application dependencies by using `npm install`. I then explore and build the dependency tree by looking at the `package-lock.json` file. After that, I use ANTLR-v4 to extract the list of constructs from the JavaScript files of the application. Next, I traverse the dependency tree depth-first to extract the list of constructs from each dependency.

Vulnerability knowledge base I build my own Node.js vulnerability dataset which includes the vulnerability information and its fix for *Eclipse-Steady*. I first retrieve the list of Node.js vulnerabilities and their information from the National Vulnerability Database (NVD) [27]. I selected only vulnerabilities that have fixes and affect the top-100 most depended npm packages [96]. I then manually annotate the set of commits that correspond to the vulnerability fix. The set of commits has to be confirmed as it appeared on the master branch of the library git repository. Given the fix commit(s) for a vulnerability, I use my extension of *Eclipse-Steady* to determine the changes that were applied to the code by the fix commit. As shown in Table 7.2, I end up with 60 vulnerabilities in my study.

SAP Node.js Applications I used SAP GitHub enterprise to identify Node.js applications suitable for my case study. I considered only applications under development having the `package.json` file in their root directory. I selected a sample of 65 applications, as shown in Table 7.2.

Table 7.3: Dependency Type information.

# Dependencies	Median	Min	Max	Q1	Q3	SD
# All Dep.	464.5	3	1,226	229.75	748.5	339.55
# Runtime Dep.	108.5	0	586	40.75	193	146.18
# Test Dep.	257	0	1,067	117.25	561.5	335.31

Table 7.3 shows the distribution of dependencies, showing more than a hundred dependencies in each application by median (i.e. 464.5 dependencies) with some applications having up to a thousand dependencies (i.e., 1,226 dependencies). I observe that the number of test dependencies is bigger than the one of runtime dependencies by two times (i.e., 257 $>$ 108.5).

Results

I present my results in terms of: (i) detected vulnerabilities, and (ii) dependency constructs.

Detected Vulnerabilities My prototype was able to detect five vulnerabilities that affected the `lodash` and `debug` npm packages. Lodash [74] is "A modern JavaScript utility library delivering modularity, performance, and extras". According to the npmjs website [92], lodash is a very popular package, with over 27,500,000 weekly downloads and 114,917 other packages that are dependent on this package. Debug [138] is "A tiny JavaScript debugging utility modelled after Node.js core's debugging technique". According to the npmjs website [91], debug is also considered a popular package, with over 66,800,000 weekly downloads and 34,494 dependents.

Dependency Constructs and features In my case study, my extension to *Eclipse-Steady* could analyze 42 out of 65 applications.

Table 7.4 shows the distribution of the BOM extracted from the applications. My prototype was able to extract more than a hundred constructs from an application and its dependencies (i.e., 164.5 constructs). In more detail, the number of application constructs is bigger than the one of dependency constructs by three times (i.e., 75 > 26).

Table 7.4: Summary of Construct Information from the experiment.

# Constructs	Median	Min	Max	Q1	Q3	SD
# App Consts.	75	0	3,083	28.25	167.5	573.99
# Dep Consts.	26	0	9,549	1.25	114.25	2,144.69
# App + Dep Consts.	164.5	1	9,671	83.25	609.75	2,224.14

Table 7.5: Frequency count of Dependent Construct Changes per vulnerability

CVE	Construct Change Type		
	Added	Modified	Removed
CVE-2017-16137	FUNC:1	MODU:1, FUNC:1	
CVE-2018-3721	FUNC:2, OBJT:1	MODU:2, FUNC:7	
CVE-2018-16487	FUNC:2, OBJT:4	MODU:2, FUNC:4	FUNC:1
CVE-2019-10744	FUNC:1, OBJT:3	MODU:2, FUNC:5	
CVE-2019-1010266	FUNC:1	MODU:2, FUNC:3	

Table 7.6: Frequency distribution of Dependency Constructs based on the dependency features.

Vulnerability	**Runtime (26)**		**Test (31)**	
	Direct	**Trans.**	**Direct**	**Trans.**
CVE-2017-16137	0	1	0	11
CVE-2018-3721	0	6	1	2
CVE-2018-16487	0	6	1	3
CVE-2019-10744	0	7	1	8
CVE-2019-1010266	0	6	1	3
	0	26	4	27

Table 7.5 and Table 7.6 show the affected dependency constructs and their construct type. Table 7.5 shows that the construct changes were detected at the `OBJT`, `MODU` and `FUNC` level. I observe that the majority of the vulnerable dependencies are transitive (28 runtime dependencies and 27 test dependencies), i.e., usually out of the control of the application developer. I also observe that most of the vulnerable constructs are detected in test dependencies.

2.5 Experience Report

My results indicate that a Node.js vulnerable code detector is viable. I now report three lessons learned and their potential future roadmap.

Mapping JavaScript Object to Constructs

In my approach, I defined a more classical inheritance of constructs (like Java and C++) on top of the JavaScript prototypal inheritance model. With this choice, one of the main issues is to ensure that I capture all the different ways to create objects and their constructs. For instance, there are at least six way to declare a function in JavaScript [103]. Furthermore, it is still an open question whether the implementation efforts required to extract the finer-level constructs (e.g., OBJT) are worth. As shown in Table 7.5, in most of the cases the `MODU` constructs were sufficient for the detection of the vulnerabilities.

Potential future avenues are two-fold. First, I would like to consider all the ways in which objects can be created in JavaScript. Second, I intend to evaluate the detection capabilities of my approach at different levels of the construct hierarchy (i.e., MODU vs. FUNC vs. OBJT).

Node.js application reliance on the npm ecosystem

The applications in my case study rely on npm packages, and as such, are potentially prone to attacks targeting popular packages, like the `lodash` and `debug` packages. Since the npm ecosystem is considered one of the biggest and most popular, it does also suffer the most in terms of known vulnerabilities, with the GitHub Advisory Database reporting npm as having the highest number of vulnerabilities (i.e., 681) when compared to six other ecosystems [45].

With the GitHub acquisition of npm, I envisage that Node.js applications will need to be aware of changes within the npm ecosystem. The creation and evaluation of such reporting mechanisms are seen a future work.

Faster Technology Adoption

Officially known as ECMAScript, the JavaScript language has been in constant evolution with its technology, with new specifications released every year. In response, Node.js keeps up to date [63]. Since industrial projects struggle with migration due to various migration or compatibility issue, it is a struggle for applications to keep up with the Node.js technology. For example, practitioners would like to control or specify the supported platform of the language. As mentioned in Section 5.2, the usage of npm packages requires industrial applications to keep up with the npm ecosystem evolution. Like most tools, I find that JavaScript static tools (such as ANTLRv4) struggle to keep up to date.

Potential future avenues for both researchers and practitioners should include strategies that help application developers to properly manage backward compatibility or guidelines to keep up with the ever-evolving technology.

2.6 Summary

In this section, I present an experience report on the implementation of a code-centric vulnerability detection tool for open source dependencies of Node.js applications. Using extracted constructs, I show that a code-centric detection tool is viable, although there are challenges related to the JavaScript language and the complexity of the application dependencies.

Future work would be to tackle the challenges of JavaScript analysis, or extending the tool to analyze the reachability of vulnerable constructs using static and dynamic analysis techniques. I believe that my results and experience is not only useful for the *Eclipse-Steady* project, but also in regards to the overall analysis of Node.js applications and their npm packages.

3 SōjiTantei: Node.js Vulnerable Function Reachability Detection

3.1 Overview

The prior work [146] showed that there is indeed an overestimation, with 73.3% of outdated clients were actually not in direct danger from the threat. In this section, I analyze the effect of vulnerabilities within the npm ecosystem on a larger scale than the manual study of 60 clients from prior work. I develop SōjiTantei that automatically detect the reachability of the vulnerable code in the application. This tool is evaluated in two experiments: (i) a replication study for accuracy and (ii) a larger scale analysis of vulnerabilities.

I refer to these projects as being *clean* (i.e., they do not execute the affecting code in their client applications). Conversely, I refer to *reached* clients as projects that adopt and execute the vulnerability code.

3.2 Experiments

I carried out two experiments. The first experiment was a replication study of the prior work. For the first experiment, I evaluate the performance by using the same data collected in Zapata et al. [146]. I plan to present a comparison of my results against this manual work.

For the second experiment, my aim is to analyze a larger statistical sample set of projects. I draw from the Decan et al. [32] study, using a stratified sample

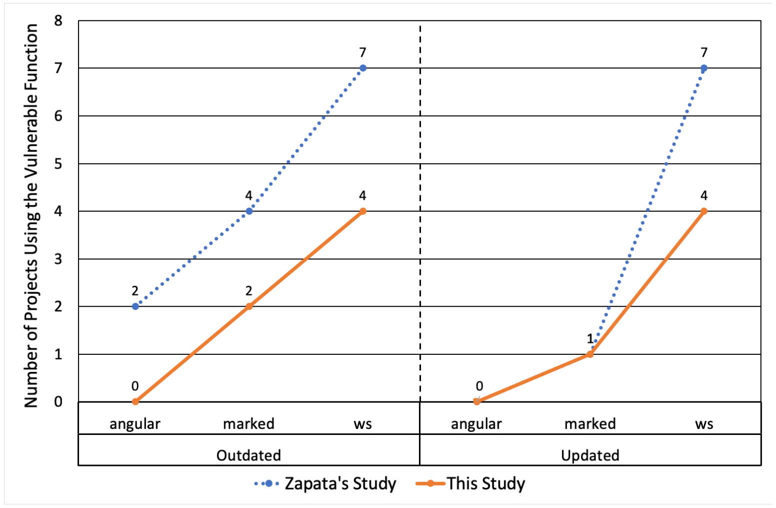


Figure 7.2: Comparison of results between the study of Zapata et al. and this study. Results show that my method does not capture all vulnerable projects.

from the 400 vulnerabilities. (with a confidence level of 95% and a confidence interval of 5.[1]) The final dataset that matched my criteria included 780 clients that were affected by 78 vulnerabilities, ending with 196 vulnerabilities. To ensure a quality dataset, I then selected vulnerabilities that met the following criteria: (i) were accessible to be downloaded (ii) had at least 10 clients (i.e., similar to the prior study) that I could test, and (iii) the vulnerability had to have a fix in which, I could identify the vulnerable function. It is important to note that the detection of the vulnerable function was still performed manually. For my results, I will report the proportion of projects using the vulnerable function. Note that listed dependencies refer to clients that list the vulnerable dependency, but the client code does not call any functions of the dependency. Furthermore, I also investigate whether the vulnerable dependency was updated or not (i.e., for the SōjiTantei performance).

Results for SōjiTantei performance. Table 7.7 presents results for the first experiment. Importantly, I find that SōjiTantei has an accuracy of 83.3% out of the 60 projects that were studied. This indicates that my method is reliable for the sample projects. As shown in Figure 7.2, SōjiTantei is not perfect, as not report all detected functions (i.e., these are reported as 10 false negatives).

[1]https://www.surveysystem.com/sscalc.htm

Table 7.7: Results of Performance Metrics

Metric	**Value**	**Description**
Statistical Measures		
n	60	# projects
TN	39	True Negatives.
FN	10	False Negatives.
FP	0	False Positives.
TP	11	True Positives.
Performance Measures		
Accuracy	0.833	(TN + TP)/N
Miss-classification rate	0.167	(FP + FN)/N
TP rate	0.524	TP/(FN + TP)
FP rate	0	FP/(TN + FP)
TN rate	1	TN/(TN + FP)
FN rate	0.476	FN/(FN + TP)

Additionally, it is important to mention that the average execution time for the function-call extraction was 0.73 seconds per project, making this approach significantly faster than the manual execution. This is especially significantly faster when compared to a manual analysis task.

Results for the Case Study Table 7.8 shows the results of the analysis of the output generated by SōjiTantei, finding that 249 of the clients were using the vulnerable package, but not actually have a direct function-call to the vulnerable code. Overall I found that a majority (61 vulnerabilities) of the clients did not reach the vulnerable code. I found that 61 of the vulnerabilities had no clients that were directly using the vulnerable code (0% reached). There were vulnerabilities that reached at least one client (i.e., 1%∼99%). In this case, around a median of 42.86% of the total clients for each vulnerability had clean clients.

3.3 Summary

I summarize the implications of the study and its contributions as follows: (1) It is likely for the client not to reach the dependency vulnerable code. (2) Automa-

Table 7.8: Summary of client classifications by SōjiTantei

Client Classification	**# Clients**
Clean	249
Reached	33
Listed Only	445
No Data Available	53
Total	780

tion is promising with the potential for improvement. (3) I believe that knowing that a client is clean can motivate client developers to update dependencies and that the community can contribute to finding accurate methods to report vulnerabilities threats. The immediate future is integrating SōjiTantei with the existing automated pull request bot such as Dependabot.[2]

[2] `https://dependabot.com/`

Part III

Conclusion

8 | Conclusion

Third-party library usages have become common among software developers in the open source software ecosystem. The npm ecosystem is one of the examples of the ecosystem that provides over a million packages to more than 11 million developers worldwide. These packages help developers to save their costs and efforts to implement their desired features. Choosing high-quality packages is essential for developers to avoid risks posed by vulnerability. However, it is hard for developers to measure the quality of those packages and adopt high-quality ones.

In this thesis, I focus on package quality characterization through the analysis of packages in the npm ecosystem. To validate this thesis, I explore two dimensions in the ecosystem: (1) the package selection and (2) the vulnerabilities and fixes assessment.

For the first dimension, I identified the list of features that developers use for assessing the package quality through the survey. As a result, developers could use this feature list to help them while searching for new packages. I then empirically identified a correlation between features and use them to predict the runnability of packages. The results show that those features can predict the runnability, which can be used in the package recommendation system. For the second dimension, I identified a small portion of the release contents related to the vulnerability fix. By analyzing the lags of adoption and propagation of fixes in the dependency networks, I found that developers do not keep adopting the new fixing release potentially due to the backward incompatibility risk. I also found that the lineage freshness of the fixing release and the severity of the vulnerability influence lags in propagation. These results suggest that developers need more awareness not only to the vulnerability but also to the fix. To help developers to assess their vulnerability in their application, I created the tools to identify vulnerable constructs in the source code and trace the vulnerable code execution. The outcomes of this thesis contribute (1) understanding of different perspectives on assessing the package quality, (2) understanding of how vulnerability fixes are propagated within the ecosystem, and (3) vulnerability assessment improvement by using tools. Based on the results of this thesis, I conclude that a good package should be able to run in a developer environment and quickly react to the security vulnerability threat. These two conclusions supply the knowledge to maintainability and security aspects of ISO25010. Specifically, the first part of this thesis contributes to the maintainability aspect as developers highlight runnability for assessing the package quality. The second part of this thesis contributes to security aspects as the proposed vulnerability assessment tools help developers to detect vulnerable codes and their execution traces. This thesis also adds knowledge about the lags of vulnerability fix adoption, which could affect the package quality.

1 Implications and Suggestions

The main goal of this thesis is to help developers (1) choosing high-quality packages and (2) understanding lags from the adoption and propagation of vulnerability fixes towards creating the high-quality application from their library usages. To achieve these goals, I perform a developer survey and three empirical studies with two supporting tools. Therefore, the empirical findings from this thesis could be valuable to both practitioners and researchers. I summarize the findings and suggestion for each part as follows:

Part I Package Selection.

1. User and contributor of packages share similar view on which features they use to assess package quality (Chapter 3).
   Suggestion. Practitioners should take both user and contributor perspectives into account to help the overall attractiveness of their packages.

2. Runnability of packages is one of the most important features for the high-quality package apart from the documentation (Chapter 3).
   Suggestion. Practitioners could consider runnability and other important features for selecting good packages.

3. Package features from the same type have strong positive correlations. Other feature combinations present trade-offs - in particular runnability related features tend to be negatively correlated with other features (Chapter 4).
   Suggestion. Developers should aware of features that negatively correlated with runnability, as it might potentially affect the package goodness.

4. Runnability of the package is predictable with high F1 score. Repository features are particularly important for prediction the runnability (Chapter 4).
   Suggestion. It is possible to use the predicted runnability feature to improve the package recommendation system.

Part II Vulnerabilities and Fixes Assessment.

1. The vulnerability fix is not always released as its own patch update as it released with unrelated commits (Chapter 5).
   Suggestion. Researchers should provide strategies for making the most efficient update via the release cycle. Furthermore, practitioners can upgrade security fixes as first class citizens, so that the vulnerability fix can travel quicker throughout the ecosystem.

2. While the package release the fix as a patch, the client tend to slowly adopt the fix and cause lags (Chapter 6).
   Suggestion. Researchers and practitioners need to provide developers more awareness mechanisms to allow quicker planning of the update. Since clients keep stale dependencies, more migration effort is required to fix that client due to the potential risk from backward incompatible changes.

3. Fixing releases that occur on the latest lineage and medium severity suffer the most lags (Chapter 6).

Suggestion. Even though the dependencies are from the latest lineage, developers still need to keep eyes on the vulnerability fix. While waiting for the propagation of the fix, developers should assess the possibility of executing the vulnerable code in their applications.

4. Creating the bill of materials of Node.js project is not trivial (Chapter 7).
   Suggestion. As the standard of JavaScript is evolving every year, the JavaScript static analysis tool has to keep up to date with the new specification.

5. Majority of vulnerable functions in dependencies are not reachable from the application (Chapter 7)
   Suggestion. Developers could prioritize their task to adopt the fixing release by using the reachability of the vulnerable code.

2 Opportunities for Future Research

In this thesis, I investigate (1) package selection and (2) vulnerabilities and fixes assessment. However, there are still a lot of research aspect that can be done in order to help developers towards creating the high-quality application. In the following, I outline the research opportunities for the immediate future.

An exploration of new metrics that can measure the package quality. In Chapter 3, I identified 30 features that important to one or both perspectives and identified correlations between different perspectives. This list of features were obtained from the prior studies, which might not be able to capture the all characteristics of packages. I suggest that there is a need for new features to capture both user and contributor perspectives, which complement the existing features such as package popularity, dependency usage, GitHub star, and download count.

A package recommendation system based on the runnability of packages. In Chapter 3 and 4, I found that runnability of packages is one of the most important feature from both user and contributor perspective for assessing the package quality. I also found that there is a possibility to use the presented features to predict the runnability of packages instead of actually install and run the package. I envision that the package recommendation system could benefit from the runnability features to improve the quality of the ranking result.

A developer survey to a better understanding of the reason for releasing and adopting vulnerability fixes. In Chapter 6, the evidences show that developer tend to keep their outdated dependencies even though their package client landing. I suspect that this phenomenon occur due to the potential risk of incompatibility issues. Hence, a developer survey could help the research community to understand the reason of the fix adoption.

A tool for managing and prioritizing the vulnerability fixing process. In Chapter 6, I found that the severity of vulnerability do influence the lags of the fix propagation. However, the fixing releases of medium severity vulnerabilities are taking longer time than to low severity to propagate to through out the dependency network. This suggest that there is the need for tool for managing and prioritizing the vulnerability fixing process.

Extend the vulnerability detection tool to analyze the reachability of vulnerable constructs by using both static and dynamic analysis. In Chapter 7, I created two supporting tools to (i) detect the vulnerable construct in the source code and (ii) trace the reachability of vulnerable function. In the current state, the vulnerable construct detection tool is derived from Eclipse Steady, the industry-grade tool which come with the web service for assessing the risk of vulnerability online. Due to the limitation of the implementation, I cannot include the vulnerable code execution tracer to this tool, but instead implement the prototype tool for the empirical study by using the static analysis. In the new future, implementing the vulnerable code execution tracer by using both static and dynamic analysis is needed which could help developers to secure their application dependencies.

References

[1] Heartbleed bug. http://heartbleed.com/, 2017. (Accessed on 06/16/2018).

[2] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why Do Developers Use Trivial Packages? An Empirical Case Study on npm. In Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pages 385–395, 2017.

[3] Karan Aggarwal, Abram Hindle, and Eleni Stroulia. Co-evolution of project documentation and popularity within github. In Proceedings of IEEE/ACM Mining Software Repositories Conference (MSR), pages 360–363, 2014.

[4] Amritanshu Agrawal, Tim Menzies, Leandro L. Minku, Markus Wagner, and Zhe Yu. Better software analytics via "duo": Data mining algorithms using/used-by optimizers. Empirical Software Engineering (EMSE), 25(3):2099–2136, 2020.

[5] O. H. Alhazmi, Y. K. Malaiya, and I. Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. Computers and Security, 26(3):219–228, 2007.

[6] Daniel A. Almeida, Gail C. Murphy, Greg Wilson, and Mike Hoye. Do software developers understand open source licenses? In Proceedings of International Conference on Program Comprehension (ICPC), May 2017.

[7] ANTLR. grammars-v4/javascript at master · antlr/grammars-v4. https://github.com/antlr/grammars-v4/tree/master/javascript, 2017. (Accessed on 08/11/2020).

[8] Veronika Bauer, Lars Heinemann, and Florian Deissenboeck. A structured approach to assess third-party library usage. In Proceedings of International Conference on Software Maintenance (ICSM). IEEE, September 2012.

[9] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. How the Apache Community Upgrades Dependencies: An Evolutionary Study. Empirical Software Engineering (EMSE), 20(5):1275–1317, October 2015.

[10] Moritz Beller, Georgios Gousios, and Andy Zaidman. TravisTorrent: Synthesizing travis CI and GitHub for full-stack research on continuous integration. In Proceedings of IEEE/ACM Mining Software Repositories Conference (MSR), May 2017.

[11] James Bennett. Choosing a javascript library. https://www.b-list.org/weblog/2007/jan/22/choosing-javascript-library/. (Accessed on 01/13/2021).

[12] James T Bennett. Shellshock in the Wild - Shellshock in the Wild. https://www.fireeye.com/blog/threat-research/2014/09/shellshock-in-the-wild.html, 2014. (Accessed on 08/11/2020).

[13] Kelly Blincoe, Francis Harrison, Navpreet Kaur, and Daniela Damian. Reference coupling: An exploration of inter-project technical dependencies and their characteristics within large software ecosystems. Information and Software Technology (IST), 110:174–189, 2019.

[14] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems. In 24th International Symposium on the Foundations of Software Engineering (FSE), pages 109–120, 2016.

[15] Leo Breiman. Random forests. Machine Learning, 45(1):5, 2001.

[16] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. Tracking known security vulnerabilities in proprietary software systems. In Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pages 516–519, mar 2015.

[17] Bodin Chinthanet, Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. Code-based vulnerability detection in node.js applications: How far are we? In Proceedings of

IEEE/ACM International Conference on Automated Software Engineering (ASE), Sept 2018.

[18] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. Journal of Systems Architecture (JSA), 57(3):294–313, mar 2011.

[19] F. R. Cogo, G. A. Oliva, and A. E. Hassan. An empirical study of dependency downgrades in the npm ecosystem. IEEE Transactions on Software Engineering (TSE), pages 1–1, 2019.

[20] Jacob Cohen. Statistical Power Analysis for the Behavioral Sciences. Routledge, 1988.

[21] E. Constantinou and T. Mens. Socio-technical evolution of the ruby ecosystem in github. In Proceedings of International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 34–44, 2017.

[22] Eleni Constantinou and Tom Mens. An empirical comparison of developer retention in the RubyGems and npm software ecosystems. Innovations in Systems and Software Engineering (ISSE), 13(2-3):101–115, 2017.

[23] J. Cox, E. Bouwers, M. Eekelen, and J. Visser. Measuring dependency freshness in software systems. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), pages 109–118, 2015.

[24] Harald Cramér. Mathematical Methods of Statistics. Princeton University Press, 1946.

[25] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in GitHub. In Proceedings of ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW), 2012.

[26] Barthélémy Dagenais and Martin P Robillard. Creating and evolving developer documentation: understanding the decisions of open source contributors. In Proceedings of International Symposium on the Foundations of Software Engineering (FSE), pages 127–136, 2010.

[27] National Vulnerability Database. Nvd - home. https://nvd.nist.gov/, 2007. (Accessed on 08/11/2020).

[28] James Davis, Arun Thekumparampil, and Dongyoon Lee. Node.fz: Fuzzing the server-side event-driven architecture. In Proceedings of the 12th European Conference on Computer Systems (EuroSys), page 145–160, 2017.

[29] Fernando López de la Mora and Sarah Nadi. Which library should I use? A metric-based comparison of software libraries . In Proceedings of International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), pages 37–40, may 2018.

[30] Cleidson R.B. de Souza, Fernando Figueira Filho, Müller Miranda, Renato Pina Ferreira, Christoph Treude, and Leif Singer. The social side of software platform ecosystems. In Proceedings of Conference on Human Factors in Computing Systems (CHI), page 3204–3214, 2016.

[31] Alexandre Decan, Tom Mens, and Maelick Claes. An empirical comparison of dependency issues in OSS packaging ecosystems. In Proceedings of the 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER), pages 2–12, feb 2017.

[32] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In Proceedings of the 15th International Conference on Mining Software Repositories (MSR), pages 181–191, 2018.

[33] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the evolution of technical lag in the npm package dependency network. In Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME), pages 404–414, 2018.

[34] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep Me Updated: An Empirical Study of Third-Party Library Updatability on Android. In Proceedings of ACM SIGSAC Conference on Computer and Communications Security, pages 2187–2200, oct 2017.

[35] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In Proceedings of the 2014 Conference on Internet Measurement Conference (IMC). ACM, November 2014.

[36] Eclipse. Eclipse steady 3.1.11 (incubator project). https://eclipse.github.io/steady/, 2018. (Accessed on 08/11/2020).

[37] Omar Elazhary, Margaret-Anne Storey, Neil Ernst, and Andy Zaidman. Do as i do, not as i say: Do contribution guidelines match the GitHub contribution process? In Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), September 2019.

[38] Pablo Estefo, Jocelyn Simmonds, Romain Robbes, and Johan Fabry. The Robot Operating System: Package reuse and community dynamics. Journal of Systems and Software (JSS), pages 226–242, 2019. ISSN 01641212.

[39] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. Auto-sklearn 2.0: The next generation. Computing Research Repository (CoRR), abs/2007.04074, 2020.

[40] International Organization for Standardization. Iso - iso/iec 25010:2011 - systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models. https://www.iso.org/standard/35733.html. (Accessed on 08/30/2021).

[41] Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. Noise and heterogeneity in historical build data: an empirical study of travis CI. In Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE), September 2018.

[42] GitHub. About security alerts for vulnerable dependencies. https://help.github.com/articles/about-security-alerts-for-vulnerable-dependencies/, 2017. (Accessed on 08/11/2020).

[43] GitHub. How security alerts are keeping your code safer. https://github.blog/2018-03-21-security-alerting-a-first-look-at-community-responses/, 2018. (Accessed on 08/11/2020).

[44] GitHub. Viewing and updating vulnerable dependencies in your repository. https://help.github.com/articles/viewing-and-updating-vulnerable-dependencies-in-your-repository/, 2018. (Accessed on 08/11/2020).

[45] GitHub. Github advisory database. https://github.com/advisories, 2019. (Accessed on 08/11/2020).

[46] GitHub. The state of the octoverse | the state of the octoverse explores a year of change with new deep dives into developer productivity, security, and how we build communities on github. https://octoverse.github.com/, 2020. (Accessed on 05/31/2021).

[47] GitHub. Keep all your packages up to date with dependabot - the github blog. https://github.blog/2020-06-01-keep-all-your-packages-up-to-date-with-dependabot/, 2020. (Accessed on 10/09/2020).

[48] Georgios Gousios. The ghtorrent dataset and tool suite. In Proceedings of IEEE/ACM Mining Software Repositories Conference (MSR), pages 233–236, 2013.

[49] H. Guercio, V. Stroele, J. M. N. David, R. Braga, and F. Campos. Complex network analysis in a software ecosystem: Studying the eclipse community. In Proceedings of International Conference on Computer Supported Cooperative Work in Design (CSCWD), pages 618–623, 2018.

[50] Foyzul Hassan and Xiaoyin Wang. Mining readme files to support automatic building of java projects in software repositories. In Proceedings of International Conference on Software Engineering Companion (ICSE-C), May 2017.

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

[52] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. Software ecosystem call graph for dependency management. In Proceedings of International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), page 101–104, 2018.

[53] Andre Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stephane Ducasse, and Marco Tulio Valente. How Do Developers React to API Evolution? The Pharo Ecosystem Case. In Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME), pages 251–260, 2015.

[54] Eric Horton and Chris Parnin. Gistable: Evaluating the executability of python code snippets on github. In Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 217–227, 2018.

[55] Md Monir Hossain, Nima Mahmoudi, Changyuan Lin, Hamzeh Khazaei, and Abram Hindle. Executability of python snippets in stack overflow. arXiv preprint arXiv:1907.04908, 2019.

[56] Michael Howard and David Leblanc. Writing Secure Code. Microsoft Press, Redmond, Wash, 2003. ISBN 978-0735617223.

[57] Kaifeng Huang, Bihuan Chen, Bowen Shi, Ying Wang, Congying Xu, and Xin Peng. Interactive, effort-aware library version harmonization. In Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 518–529, nov 2020.

[58] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In Learning and Intelligent Optimization (LION), pages 507–523, Berlin, Heidelberg, 2011. ISBN 978-3-642-25566-3.

[59] Akinori Ihara, Daiki Fujibayashi, Hirohiko Suwa, Raula Gaikovina Kula, and Kenichi Matsumoto. Understanding When to Adopt a Library: A Case Study on ASF Projects. In International Conference on Open Source Systems (OSS), pages 128–138, 2017.

[60] Shohei Ikeda, Akinori Ihara, Raula Gaikovina Kula, and Kenichi Matsumoto. An empirical study of README contents for JavaScript packages. IEICE Transactions on Information and Systems, E102.D(2):280–288, February 2019.

[61] Saket Dattatray Joshi and Sridhar Chimalakonda. RapidRelease - a dataset of projects and issues on github with rapid releases. In Proceedings of IEEE/ACM Mining Software Repositories Conference (MSR), May 2019.

[62] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining GitHub. In Proceedings of IEEE/ACM Mining Software Repositories Conference (MSR), 2014.

[63] William Kapke. Node.js es2015/es6, es2016 and es2017 support. https://node.green/, 2016. (Accessed on 08/11/2020).

[64] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and Evolution of Package Dependency Networks. In Proceedings of the 14th International Conference on Mining Software Repositories (MSR), pages 102–112, 2017.

[65] Naoki Kobayakawa and Kenichi Yoshida. How GitHub contributing.md contributes to contributors. In Proceedings of Computer Software and Applications Conference (COMPSAC), July 2017.

[66] William H Kruskal and W Allen Wallis. Use of Ranks in One-Criterion Variance Analysis. Journal of the American Statistical Association, 47(260):583–621, dec 1952. ISSN 0162-1459.

[67] R. G. Kula, C. De Roover, D. M. German, T. Ishio, and K. Inoue. A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem. In Proceedings of International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 288–299, 2018.

[68] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? Empirical Software Engineering (EMSE), 23(1):384–417, 2018.

[69] Enrique Larios Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. Selecting third-party libraries: the practitioners' perspective. In Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 245–256, nov 2020. doi: 10.1145/3368089.3409711.

[70] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In Proceedings of the 24th Network and Distributed System Security Symposium (NDSS), 2017.

[71] Timothy C Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practice. IEEE Software, 20(6):35–39, 2003.

[72] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In 24th ACM SIGSAC Conference on Computer and Communications Security (CCS), page 2201–2215, 2017.

[73] Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velásquez. An Empirical Study on Android-related Vulnerabilities. In Proceedings of the 14th International Conference on Mining Software Repositories (MSR), pages 2–13, 2017.

[74] Lodash. lodash/lodash: A modern javascript utility library delivering modularity, performance, & extras. https://github.com/lodash/lodash, 2012. (Accessed on 08/11/2020).

[75] Mircea Lungu. Towards reverse engineering software ecosystems. In Proceedings of International Conference on Software Maintenance (ICSM), September 2008.

[76] Christian Macho, Shane McIntosh, and Martin Pinzger. Automatically repairing dependency-related build breakage. In Proceedings of International Conference on Software Analysis, Evolution and Reengineering (SANER), March 2018.

[77] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven node.js javascript applications. In Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), page 505–519, 2015.

[78] Konstantinos Manikas and Klaus Marius Hansen. Software ecosystems – a systematic literature review. Journal of Systems and Software (JSS), 86(5):1294–1306, May 2013.

[79] Samim Mirhosseini and Chris Parnin. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 84–94, oct 2017.

[80] Samim Mirhosseini and Chris Parnin. Docable: Evaluating the executability of software tutorials. In Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), page 375–385, 2020.

[81] Mitre Corporation. CVE - Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/, 2018. (Accessed on 08/11/2020).

[82] Mitre Corporation. CWE - Common Weakness Enumeration. https://cwe.mitre.org/, 2018. (Accessed on 08/11/2020).

[83] Suhaib Mujahid, Rabe Abdalkareem, Emad Shihab, and Shane McIntosh. Using others' tests to avoid breaking updates. In Proceedings of IEEE/ACM Mining Software Repositories Conference (MSR), 2020.

[84] Nuthan Munaiah, Felivel Camilo, Wesley Wigham, Andrew Meneely, and Meiyappan Nagappan. Do bugs foreshadow vulnerabilities? An in-depth study of the chromium project. Empirical Software Engineering (EMSE), 22(3):1305–1347, 2017.

[85] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating GitHub for engineered software projects. Empirical Software Engineering (EMSE), 22(6):3219–3253, 2017.

[86] NAIST-SE. Naist-se/steady: Analyses your java and python applications for open-source dependencies with known vulnerabilities, using both static analysis and testing to determine code context and usage for greater accuracy. https://github.com/NAIST-SE/steady, 2020. (Accessed on 08/11/2020).

[87] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, and Massimiliano Di Penta. CrossRec: Supporting software developers by recommending third-party libraries. Journal of Systems and Software (JSS), 161:110460, mar 2020.

[88] Viet Hung Nguyen, Stanislav Dashevskyi, and Fabio Massacci. An automatic method for assessing the versions affected by a vulnerability. Empirical Software Engineering (EMSE), 21(6):2268–2297, Dec 2016.

[89] Ehsan Noei, Mark D. Syer, Ying Zou, Ahmed E. Hassan, and Iman Keivanloo. A study of the relation of mobile device attributes with the user-perceived quality of android apps. Empirical Software Engineering (EMSE), 22(6):3088–3116, March 2017.

[90] NPM. npm. https://www.npmjs.com/, 2010. (Accessed on 05/13/2021).

[91] npm. debug - npm. https://www.npmjs.com/package/debug, 2011. (Accessed on 08/11/2020).

[92] npm. lodash - npm. https://www.npmjs.com/package/lodash, 2012. (Accessed on 08/11/2020).

[93] NPM. Responding to Security Threats and Critical Updates . https://www.npmjs.com/policies/security#responding-to-security-threats-and-critical-updates, 2016. (Accessed on 08/11/2020).

[94] NPM. Auditing package dependencies for security vulnerabilities. https://docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities, 2018. (Accessed on 08/11/2020).

[95] NPM. Security vulnerabilities found requiring manual review. https://docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities#security-vulnerabilities-found-requiring-manual-review, 2018. (Accessed on 10/12/2020).

[96] npm. npm - most dependend upon. https://www.npmjs.com/browse/depended, 2020. (Accessed on 08/11/2020).

[97] National Institute of Standards and Technology. Secure software development framework | csrc. https://csrc.nist.gov/Projects/ssdf. (Accessed on 06/08/2021).

[98] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, Takashi Ishio, Daniel M. German, and Katsuro Inoue. Search-based software library recommendation using multi-objective optimization. Information and Software Technology (IST), 83:55–75, mar 2017.

[99] M. Palyart, G. C. Murphy, and V. Masrani. A study of social interactions in open source component use. IEEE Transactions on Software Engineering (TSE), 44(12):1132–1145, Dec 2018.

[100] Amantia Pano, Daniel Graziotin, and Pekka Abrahamsson. Factors and actors leading to the adoption of a JavaScript framework. Empirical Software Engineering (EMSE), 23(6):3503–3534, dec 2018.

[101] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vulnerable Open Source Dependencies: Counting Those That Matter. In Proceedings of the 12th International Symposium on Empirical Software Engineering and Measurement (ESEM), pages 42:1–42:10, 2018.

[102] Jibesh Patra, Pooja N. Dixit, and Michael Pradel. ConflictJS: finding and understanding conflicts between JavaScript libraries. In Proceedings of International Conference on Software Engineering (ICSE), pages 741–751, may 2018.

[103] Dmitri Pavlutin. 6 ways to declare javascript functions. https://dmitripavlutin.com/6-ways-to-declare-javascript-functions/, 2016. (Accessed on 08/11/2020).

[104] Karl Pearson. X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 50(302):157–175, 1900.

[105] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12:2825–2830, 2011.

[106] V. Piantadosi, S. Scalabrino, and R. Oliveto. Fixing of security vulnerabilities in open source projects: A case study of apache http server and apache tomcat. In 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pages 68–78, apr 2019.

[107] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. A large-scale study about quality and reproducibility of jupyter notebooks. In Proceedings of IEEE/ACM Mining Software Repositories Conference (MSR), pages 507–517, 2019.

[108] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Beyond Metadata: Code-centric and Usage-based Analysis of Known Vulnerabilities in Open-source

Software. In Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME), pages 58–68, 2018.

[109] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Detection, assessment and mitigation of vulnerabilities in open source dependencies. Empirical Software Engineering (EMSE), 2020. doi: 10.1007/s10664-020-09830-x.

[110] Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. Categorizing the content of GitHub README files. Empirical Software Engineering (EMSE), 24(3):1296–1327, October 2018.

[111] Tom Preston-Werner. Semantic Versioning 2.0.0. https://semver.org/, 2009. (Accessed on 08/11/2020).

[112] H. S. Qiu, A. Nolte, A. Brown, A. Serebrenik, and B. Vasilescu. Going farther together: The impact of social capital on sustained participation in open source. In Proceedings of International Conference on Software Engineering (ICSE), pages 688–699, 2019.

[113] S. Raemaekers, A. van Deursen, and J. Visser. Measuring software library stability through historical version analysis. In Proceedings of the 28th International Conference on Software Maintenance (ICSM), pages 378–387, Sept 2012.

[114] Brittany Reid, Christoph Treude, and Markus Wagner. Optimising the fit of stack overflow code snippets into existing code. In Proceedings of Genetic and Evolutionary Computation Conference (GECCO), page 1945–1953, 2020.

[115] Romain Robbes, Mircea Lungu, and David Röthlisberger. How Do Developers React to API Deprecation?: The Case of a Smalltalk Ecosystem. In Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE), pages 56:1–56:11, 2012.

[116] Martin P Robillard. What makes apis hard to learn? answers from developers. IEEE Software, 26(6):27–34, 2009.

[117] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, and Linda Devine. Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen'sd indices the most appropriate choices. In Proceedings of Annual meeting of the Southern Association for Institutional Research. Citeseer, 2006.

[118] Mohamed Aymen Saied, Ali Ouni, Houari Sahraoui, Raula Gaikovina Kula, Katsuro Inoue, and David Lo. Improving reusability of software libraries through usage pattern mining. Journal of Systems and Software (JSS), 145(October 2017): 164–179, 2018. ISSN 01641212.

[119] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of 25,357 clients of 4+1 popular Java APIs. In Proceedings of the 32th International Conference on Software Maintenance and Evolution (ICSME), pages 400–410, 2016.

[120] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement (ESEM), pages 315–317, 2008.

[121] Dan Sholler, Igor Steinmacher, Denae Ford, Mara Averick, Mike Hoye, and Greg Wilson. Ten simple rules for helping newcomers become contributors to open projects. PLOS Computational Biology, 15(9):e1007296, September 2019.

[122] Snyk. Vulnerability DB. https://snyk.io/vuln, 2015. (Accessed on 04/20/2020).

[123] Snyk. Denial of Service (DoS) - npm:connect. https://snyk.io/vuln/npm:connect:20120107, 2017. (Accessed on 04/20/2020).

[124] Snyk. Insecure defaults in faye | snyk. https://snyk.io/vuln/npm:faye:20121107, 2017. (Accessed on 04/20/2020).

[125] Snyk. Symlink attack due to predictable tmp folder names in npm | snyk. https://snyk.io/vuln/npm:npm:20130708, 2017. (Accessed on 04/20/2020).

[126] Igor Steinmacher, Marco Aurélio Gerosa, and David F. Redmiles. Attracting, onboarding, and retaining newcomer developers in open source software projects. In Proceedings of ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW), number February, pages 1–4, 2014.

[127] Chungha Sung, Markus Kusano, Nishant Sinha, and Chao Wang. Static dom event dependency analysis for testing web applications. In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), page 447–459, 2016.

[128] Synopsys. Heartbleed Bug. http://heartbleed.com/, 2014. (Accessed on 08/11/2020).

[129] Synopsys. 2020 open source security and risk analysis (ossra) report | synopsys. https://www.synopsys.com/software-integrity/resources/analyst-reports/2020-open-source-security-risk-analysis.html, 2020. (Accessed on 05/27/2020).

[130] Xin Tan, Minghui Zhou, and Zeyu Sun. A first look at good first issues on GitHub. In Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 398–409, nov 2020.

[131] C. Teyton, J.-R. Falleri, and X. Blanc. Mining Library Migration Graphs. In Proceedings of the 19th Working Conference on Reverse Engineering (WCRE), pages 289–298, Oct 2012.

[132] the npm blog. npm blog: Next Phase Montage. https://blog.npmjs.org/post/612764866888007680/next-phase-montage, 2020. (Accessed on 05/20/2020).

[133] Ferdian Thung, David Lo, and Julia Lawall. Automated library recommendation. In Proceedings of Working Conference on Reverse Engineering (WCRE), number October, pages 182–191, oct 2013.

[134] Tidelift. Libraries.io - The Open Source Discovery Service. https://libraries.io/, 2017. (Accessed on 01/13/2021).

[135] Parastou Tourani, Bram Adams, and Alexander Serebrenik. Code of conduct in open source projects. In Proceedings of International Conference on Software Analysis, Evolution and Reengineering (SANER), February 2017.

[136] JavaScript Tutorial. Javascript anonymous functions. https://www.javascripttutorial.net/javascript-anonymous-functions/, 2020. (Accessed on 08/11/2020).

[137] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. Journal of Machine Learning Research, 9:2579–2605, November 2008. ISSN 1533-7928 (electronic); 1532-4435 (paper).

[138] Visionmedia. visionmedia/debug: A tiny javascript debugging utility modelled after node.js core's debugging technique. works in node.js and web browsers. https://github.com/visionmedia/debug, 2011. (Accessed on 08/11/2020).

[139] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. Do the dependency conflicts in my project matter? In Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 319–330, oct 2018.

[140] MDN web docs. Inheritance and the prototype chain - javascript | mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain, 2020. (Accessed on 08/11/2020).

[141] L. Williams, G. McGraw, and S. Migues. Engineering Security Vulnerability Prevention, Detection, and Response. IEEE Software, 35(5):76–80, Sep. 2018.

[142] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A Look at the Dynamics of the JavaScript Package Ecosystem. In Proceedings of the 13th International Conference on Mining Software Repositories (MSR), pages 351–361, 2016.

[143] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E. Hassan, and Zhenchang Xing. What do developers search for on the web? Empirical Software Engineering (EMSE), 22(6):3149–3185, December 2017.

[144] Bowen Xu, Le An, Ferdian Thung, Foutse Khomh, and David Lo. Why reinventing the wheels? an empirical study on library reuse and re-implementation. Empirical Software Engineering (EMSE), 25(1):755–789, September 2019.

[145] Di Yang, Aftab Hussain, and Cristina Videira Lopes. From query to usable code: An analysis of stack overflow code snippets. In Proceedings of IEEE/ACM Mining Software Repositories Conference (MSR), page 391–402, 2016. ISBN 9781450341868.

[146] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages. In Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 559–563, 2018.

[147] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesus Gonzalez-Barahona. An empirical analysis of technical lag in npm package dependencies. In Proceedings of the 17th International Conference on Software Reuse (ICSR), pages 95–110, 2018.

[148] Jiayuan Zhou, Shaowei Wang, Cor-Paul Bezemer, Ying Zou, and Ahmed E. Hassan. Studying the association between bountysource bounties and the issue-addressing likelihood of GitHub issue reports. IEEE Transactions on Software Engineering (TSE), pages 1–1, 2020.

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