Untriviality of Trivial Packages

Md Atique Reza Chowdhury

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

December 2019

© Md Atique Reza Chowdhury, 2020

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Md Atique Reza Chowdhury**

Entitled: **Untriviality of Trivial Packages**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

______________________________ Chair
Dr. Weiyi Shang

______________________________ Examiner
Dr. Yann-Gaël Guéhéneuc

______________________________ Examiner
Dr. Tse-Hsun Chen

______________________________ Supervisor
Dr. Emad Shihab

Approved by ______________________________
Dr. Lata Narayanan, Chair
Department of Computer Science and Software Engineering

__________ 2019 ______________________________
Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

Untriviality of Trivial Packages

Md Atique Reza Chowdhury

Nowadays, developing software would be unthinkable without the use of third-party packages. Although such code reuse helps to achieve rapid continuous delivery of software to end-users, blindly reusing code has its pitfalls. Prior work investigated the rationale for using packages of small size, known as trivial packages, that implement simple functionality. This prior work showed that, although these trivial packages are simple, they are popular and prevalent in the *npm* ecosystem. This popularity and prevalence of trivial packages peaked our interest in questioning; first, the 'triviality' of these packages and, second, the impact of using these packages on the quality of the client software applications.

To better understand the 'triviality' of trivial packages and their impact, in this thesis we report on two large scale empirical studies. In both studies, we mine a large set of JavaScript applications that use trivial *npm* packages. In the first study, we evaluate the triviality of these packages from two complementary points of view: based on application usage and ecosystem usage. Our result shows that trivial packages are being used in important JavaScript files, by the means of their 'centrality', in software applications. Additionally, by analyzing all external package API calls in these JavaScript files, we find that a high percentage of these API calls are attributed to trivial packages. Therefore, these packages play a significant role in JavaScript files. Furthermore, in the package dependency network, we observe that 16.8% packages are trivial and in some cases removing a trivial package can break approximately 30% of the packages in ecosystem. In the second study, we started by understanding the circumstances which incorporate trivial packages in software applications. We analyze and classify commits that introduce trivial packages into software applications. We notice that developers resort to trivial packages while performing a wild range of development tasks that

are mostly related to ‘Building’ and ‘Refactoring’. We empirically evaluate bugginess of the files and applications that use trivial packages. Our result shows that JavaScript files and applications that use trivial packages tend to have a higher percentage of bug-fixing commits than files and applications that do not depend on trivial packages. Overall, the findings of our thesis indicate that although smaller in size and complexity, trivial packages are highly depended on packages. These packages may be trivial by the means of size, their usage in software applications suggests that their role is not so trivial.

Statement of Originality

I, Md Atique Reza Chowdhury, hereby declare that I am the sole author of this thesis. All ideas and inventions attributed to others have been properly referenced. This is a true copy of the thesis.

Dedication

To my parents.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to Almighty Allah for his abundance blessings and help to accomplish this work.

I would like to thank my supervisor Dr. Emad Shihab for his support and guidance in each step of this journey. He motivated me when I needed it most. I consider myself very lucky to work under his supervision.

I am grateful to Dr. Rabe Abdalkareem for guiding me in each step of this endeavor. Your advice, critical comments and feedback helped me immensely.

I am greatly indebted to my fellow colleagues Ahmad Abdellatif, Suhaib Mujahid, Mahmood AL Fadel, Diego Elias,Giancarlo Sierra and everyone else in the Data-driven Analysis of Software (DAS) Lab.

I owe my deepest gratitude to my parents for their fervent prayers, support, motivation and never loosing faith in me. Thanks to my son for inspiring me dream big and to my wife for helping me chase that dream.

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**

1.1 Motivating Example 2

1.2 Thesis Statement 3

1.3 Thesis Overview 3

1.3.1 Chapter 3: Untriviality of Trivial Packages: An Empirical Study of the npm JavaScript Packages 3

1.3.2 Chapter 4: An Empirical Study on the Impact of Using Trivial Packages on Software Quality 4

1.4 Thesis Contributions 5

1.5 Related Publications 5

2 Related Work **6**

2.1 Third-party Package Usage 6

2.2 Software Ecosystems 7

2.3 Impact of Reusing source code 9

2.4 Summary 10

3 Untriviality of Trivial Packages: An Empirical Study of the npm JavaScript Packages **11**

3.1 Introduction 11

3.2 Case Study Design 14
3.2.1 Dataset of Candidate Applications 14
3.2.2 Pruning List of Applications 14
3.2.3 Identifying JavaScript Applications that Use Trivial Packages 15
3.3 Case Study Result 17
3.3.1 RQ1: Are trivial packages used in important parts of JavaScript applications? 17
3.3.2 RQ2: How widely used are trivial packages in JavaScript Applications? . . 21
3.3.3 RQ3: Do trivial packages play an important role at the ecosystem level? . . 23
3.4 Discussion 28
3.5 Threats to validity 31
3.5.1 Construct validity 31
3.5.2 External validity 32
3.6 Chapter Summary 33

4 An Empirical Study on the Impact of Using Trivial Packages on Software Quality 34
4.1 Introduction 34
4.2 Case Study Design 37
4.2.1 Dataset of candidate applications 37
4.2.2 Select active and large JavaScript applications 38
4.2.3 Select applications with rich development history 38
4.2.4 Identify applications that use trivial packages 38
4.3 Case Study Result 40
4.3.1 RQ0: In which context trivial packages are introduced into a software application and what types of functionalities trivial packages provide? 40
4.3.2 RQ1: Does using trivial JavaScript packages impact the overall quality of applications? 46
4.3.3 RQ2: What is the impact of trivial packages on the quality of the files? . . 49
4.3.4 RQ3: Are commits that introduce trivial packages in JavaScript files risky? 53
4.4 Threats to validity 55

4.4.1 Internal validity 56
4.4.2 External validity 57
4.5 Chapter Summary 57

5 Summary, Contributions and Future Work **58**
5.1 Summary of findings 58
5.2 Contribution 59
5.3 Future Work 60
5.3.1 Detecting Trivial Packages That Provide Similar Functionalities: 60
5.3.2 Generate Automated Test Cases for the Packages: 60
5.3.3 Automate the Evaluation of Ecosystem Health: 60
5.3.4 Automatically Generate Smaller Packages: 61

Bibliography **62**

List of Figures

Figure 1.1 The source code of trivial package `isarray`. 2

Figure 3.1 An overview of our data collection approach. 14

Figure 3.2 Distribution of degree centrality rank of trivial dependent and non-trivial dependent files in different project groups based on number of files. 19

Figure 3.3 Distribution of percentage of API calls for trivial and non-trivial packages in JS files. 21

Figure 3.4 Distribution of trivial and non-trivial package API entropy. 22

Figure 3.5 Composite Dependency Network 24

Figure 3.6 Distribution of PageRank values for trivial and non-trivial packages. 25

Figure 3.7 The distribution of the percentage of trivial dependent files in all the studied applications based on TDDT segmentations. Dotted horizontal line present overall median. 30

Figure 4.1 Overview of the dataset selection process. 37

Figure 4.2 Distribution of percentage of bug-fixing commits in *trivial dependent* and *non-trivial dependent* applications. The solid horizontal lines represent the medians of the distribution. The dotted horizontal line is the overall median. 46

Figure 4.3 Distribution of percentage of bug-fixing commits before and after trivial package introduction in applications. The solid horizontal lines represent the medians of the distribution. The dotted horizontal line is the overall median. 48

Figure 4.4 The distribution of percentage of bug-fixing commits in *trivial dependent* and *non-Trivial dependent* files. The solid horizontal lines represent the medians of the distribution. The dotted horizontal line is the overall median. 49
Figure 4.5 The correlation between the percentage of number of trivial package used and percentage of number of the bug-fixing commits in file. 50
Figure 4.6 The distribution of bug-fixing commits before and after trivial package introduction in files that are converted from *non-trivial dependent* to *trivial dependent* in that commit. The solid horizontal lines represent the medians of the distribution. The dotted horizontal line is the overall median. 51
Figure 4.7 Distribution of percentage of fix-inducing commits in trivial introduction commits in files and other commits. The solid horizontal lines represent the medians of the distribution. The dotted horizontal line is the overall median. 53

List of Tables

Table 3.1 Filtering steps of the studied JavaScript applications. 15

Table 3.2 Summary of the number of developers, commits, watchers, and stars for 15,254 JavaScript projects. 16

Table 3.3 Distribution of number of *npm* packages in all the JavaScript applications in our dataset. 16

Table 3.4 Distribution of number of files in projects in our dataset. 17

Table 3.5 The distribution of the degree centrality of *trivial* and *non-trivial dependent* JavaScript files. 18

Table 3.6 The statistical summary of the distribution of technical bus factor (TBF) for the trivial and non-trivial packages in our composite dependency network. 26

Table 3.7 The top-20 most impactful trivial packages measured by *Technical Bus Factor (TBF)*. 27

Table 3.8 The statistical summary of the distribution of external package API call percentage in JavaScript files throughout application's development lifespan. The table shows the distribution for trivial packages (TP) and non-trivial packages (NPT). . . 30

Table 4.1 Summary statistic of the studied dataset. 40

Table 4.2 Type of development activities associated with introducing trivial packages into JavaScript applications. 41

Table 4.3 Categories of trivial packages based on functionality. 44

Table 4.4 The percentage of fix-inducing commits for each of the different type of functionalist that trivial packages provide. *The percentage of fix-inducing commits all commits for each of the different type of functionalities. **The percentage of fix-inducing commits for each type to all fix-inducing commits related to trivial packages. 54

Chapter 1

Introduction

Using third-party packages is becoming an integral part of today's software development practice. Developers share their code in the form of packages to different software package managers (e.g. Node Package Manager (*npm*) and Python Package Index (PyPI)). Developers are using these packages in their software applications to avoid reinventing the wheel (Abdalkareem, Nourry, Wehaibi, Mujahid, and Shihab (2017); Murphy-Hill et al. (2019); Wagner and Murphy-Hill (2019); Xu, An, Thung, Khomh, and Lo (2019)). Because of this demand and supply, entire ecosystems have been created around these package managers, e.g., the Node.js ecosystems are largely supported by *npm* (Cox (2019)).

Previous studies showed that blindly using external-packages has pitfalls like an increase of unforeseen maintenance cost, exposer to vulnerabilities and legal problems due to licensing issues (Decan, Mens, and Constantinou (2018); Inoue, Sasaki, Xia, and Manabe (2012); Lim (1994); Zapata et al. (2018)). Lately in 2016, removal of "left-pad", an 11 line source code, package from *npm* affected popular websites like Facebook, NetFlix, Airbnb. This incident triggered a debate regarding using small *npm* packages. Some people were critical about reusing small packages to the extent that they even questioned the programming competence of developers who use these small packages (Haney (2016)). Prior work (Abdalkareem, Nourry, et al. (2017); Abdalkareem, Oda, Mujahid, and Shihab (2019)) evaluated the rationale of developers regarding using these small packages, known as "trivial" packages. They defined trivial packages based on size and complexity, and they observed that approximately 17% of the packages in the *npm* ecosystem are trivial.

```
var toString ={}.toString;
module.exports = Array.isArray || function (arr) {
 return toString.call(arr) == '[object Array]';
};
};
```

Figure 1.1: The source code of trivial package `isarray`.

Although developers speculate that these packages are well maintained and tested, more than 50% of these packages have no test case. Moreover, developers claim that these packages improve their productivity and make the software applications less complex, more readable and performant.

We agree that trivial packages may be small in size and implement very specific functionality but the fact that they are so prevalent and popular warrants questioning their triviality. Therefore, the main objective of this thesis is to empirically examine the triviality of these packages from their usage in software applications.

1.1 Motivating Example

To highlight the popularity and prevalence of trivial packages, we describe three motivating examples. First, we examine the "isarray" package composed of only 3 lines of code shown in Figure 1.1. Our examination of this package shows that it is extremely popular and has more than 24 million downloads per week. One of the prominent reasons behind the large popularity of this package is backward compatibilities with browsers that do not support Array.isArray provided by JavaScript 1.8.5. For this reason, 214,330 (40%) other packages in the *npm* ecosystem directly or indirectly (transitive) depend on this package, which makes this package extremely important in this ecosystem. Second, the removal of the "left-pad", which contains only 11 lines of codes, package from the *npm* ecosystem had interrupted most popular websites like Facebook, Netflix, and Airbnb. Finally, we observe that the three of the top 5 most depended on packages in the *npm* ecosystem are small packages (Zimmermann, Staicu, Tenny, and Pradel (2019)).

It is apparent from the aforementioned examples that some trivial packages are very popular and heavily depended on by other packages. It is essential to evaluate whether these trivial packages play a trivial role in software applications that depend on them. Moreover, more than 50% of trivial

packages have no test cases, it is important to assess if these packages have an impact on the quality of software applications.

1.2 Thesis Statement

In this Master's thesis, we focus on understanding the use of trivial packages in software applications. We analyze the software applications that depend on these packages to understand two aspects. First, we evaluate if these packages are important in the scope of these software applications and in the software ecosystem to which they belong to. Second, we examine the quality impact of these packages on software applications. We formulate our research problem as follows:

Although prior work shows that the use of trivial packages has become increasingly common, little is known about their importance and impact on software applications. We want to investigate the triviality of trivial packages.

1.3 Thesis Overview

The body of the thesis is composed of two main chapters:

1.3.1 Chapter 3: Untriviality of Trivial Packages: An Empirical Study of the npm JavaScript Packages

Prior work has investigated the developer's rationale regarding using trivial packages. Although these trivial packages provide simple functionalities, they are popular and prevalent in the *npm* ecosystem. This popularity and prevalence of trivial packages piqued our interest in questioning their triviality. To understand the triviality of these packages, we mine a large set of NodeJs applications that use trivial *npm* packages and evaluate these packages relative importance by evaluating how these packages are used in these applications. Specifically, we evaluate the triviality of these trivial packages from two complementary aspects: based on application usage and ecosystem usage. Our result shows that:

- Files that have trivial package dependency are comparatively more ‘central’ within the scope of a software application than other files.

- In these JavaScript files, we found that a significantly higher percentage of API calls are attributed to trivial packages compared to non-trivial packages. Therefore, these packages play a significant role in these files.

- In the package dependency network, which consists of all direct and transitive dependency of the software applications in our dataset, trivial packages are statistically more central as these packages are more depended upon by other packages than non-trivial packages.

1.3.2 Chapter 4: An Empirical Study on the Impact of Using Trivial Packages on Software Quality

Motivated by the findings of the previous study and by the fact that more than 50% of the trivial packages have no test case, we empirically examine the impact of using these packages on software quality. Additionally, we categorize trivial packages based on their functionality. Our analysis shows that:

- Trivial packages render a wide variety of functionalities ranging from simple string modification to server management or providing security.

- Applications that use trivial packages tend to have a higher percentage of bug-fixing commits compared to the applications that do not have any trivial package dependency.

- Files that depend on trivial packages are statistically more buggy than files that do not depend on trivial packages.

- The commits that introduce trivial packages in JavaScript files are statistically more fix-inducing than other commits, which makes these changes risky.

1.4 Thesis Contributions

The contributions of this thesis are as follows:

- We provide a novel approach to evaluate how important trivial *npm* packages are by extensively analyzing their usage from applications and ecosystems perspective.
- We formulate various metrics to understand the importance of a package in a dependency network of the *npm* ecosystem.
- We provide an extensive insight on the development activities that introduce trivial packages in to an application and the functionalities trivial packages provide.
- We conducted a large scale empirical study to examine the quality impact of using trivial packages in JavaScript applications.

1.5 Related Publications

Earlier versions of the work presented in this thesis have been previously presented or submitted to different renowned software engineering events:

- A. Chowdhury, "On the Untriviality of Trivial Packages: An Empirical Study of the npm JavaScript Packages", Poster Presented at Consortium for Software Engineering Research (CSER), Fall 2016 Meeting, Markham, Ontario, Canada, 2018.
- A. Chowdhury, R. Abdalkareem, E. Shihab, and B. Adam "On the Untriviality of Trivial Packages: An Empirical Study of the npm JavaScript Packages", Under Submission to the IEEE Transactions on Software Engineering (TSE), 16 pages (2019).
- A. Chowdhury, R. Abdalkareem, E. Shihab and S. McIntosh "An Empirical Study on the Impact of Using Trivial Packages on Software Quality", In Preparation to be Submitted to the Journal of Empirical Software Engineering (EMSE), 29 pages (2019),

Chapter 2

Related Work

The work that is most related to our study falls into three main categories: third-party package usage, software ecosystems, and software quality.

2.1 Third-party Package Usage

The increasing trend of using third-party packages in software applications has motivated researchers to analyze why and how these packages are created and maintained. Xu *et al.* (Xu et al. (2019)) studied the reason behind the reuse and re-implement of external packages in software applications. Developers often replace their self-implemented methods by external libraries because they were initially unaware of the library or it was unavailable back then. Later on, when they become aware of a well maintained and tested package, they replace their own code by that package. Although developers prefer to reuse code than re-implementing, they replace the external package by implementing the methodology themselves when they become aware that they only use a small number of functionalities of a heavy package or if the package methods become deprecated. This study encourages package developers to produce lightweight packages e.g., trivial packages. Abdalkareem *et al.* (Abdalkareem, Nourry, et al. (2017)) studied an emerging code reuse practice in the form of lightweight packages in the *npm* ecosystem. Abdalkareem *et al.* (Abdalkareem, Nourry, et al. (2017)) studied various aspects of trivial packages. They defined trivial packages based on the size and complexity of these packages and we adopt this definition in our study. Their study

was conducted upon understanding why developers use trivial packages. Kula *et al.* (Kula, Ouni, Germán, and Inoue (2017)) also study small packages in the *npm* ecosystem. Their study shows that these small packages either provide trivial functionalities or they act as a facade to load other external packages. Therefore, these packages, when act as a facade, have long dependency chains. Both of these studies evaluate small or micro packages as standalone units, our study examines the importance and quality impact of these trivial packages where they are used. Trivial packages are not only available in the *npm* ecosystem. In another study, Abdalkareem *et al.* (Abdalkareem et al. (2019)) observed that these packages are also prevalent in PyPI (Python Package Index) but 70.3% of the developers consider using these packages in software applications a bad practice. Therefore, perception of package use varies across software ecosystems.

2.2 Software Ecosystems

The software applications that belong to the same software ecosystem was a research interest. Several studies examine software ecosystems to understand their characteristics and evolution (e.g., Bavota, Canfora, Penta, Oliveto, and Panichella (2013); Bloemen, Amrit, Kuhlmann, and Ordóñez Matamoros (2014); Decan, Mens, Claes, and Grosjean (2016); German, Adams, and Hassan (2013); Kabbedijk and Jansen (2011); Manikas (2016)).

Several studies examined direct and transitive dependencies of software applications. Wittern *et al.* (Wittern, Suter, and Rajagopalan (2016)) examined packages in the *npm* ecosystem and observed that 32.5% of the packages have 6 or more dependencies. Moreover, 27.5% of the packages in *npm* are core packages as they are largely dependent on by other packages. Fard *et al.* (Fard and Mesbah (2017)) evaluated changeability in *npm* applications and showed that the average number of dependencies in these applications is six and the number is always in the growing trend. Kikas *et al.* analyzed the dependency network structure and evolution of JavaScript, Ruby, and Rust ecosystems and showed that the number of transitive dependencies of the packages in these ecosystems is 10 times higher than the number of direct dependencies and this scenario is growing exponentially (Kikas, Gousios, Dumas, and Pfahl (2017)). Zimmermann *et al.* (Zimmermann et al. (2019)) systematically examine dependencies between packages, the maintainers responsible for packages

in the *npm* while focusing on security issues. Their results show that individual packages could impact large parts of the *npm* ecosystem. They also reported that a very small number of developers are responsible for a large number of *npm* packages.

In our study, we also see that direct dependencies are only the tip of the iceberg, whereas indirect dependencies make up the largest portion of a package dependency network. We found 10,507 distinct packages as direct dependencies to these applications whereas the package dependency network, which has direct and transitive dependencies of these software applications, has 32,319 packages.

Researchers also investigated the developers' rationale for selecting a package for their software application. Haenni *et al.* found that developers generally do not apply any logical reasoning when selecting packages, they just use them to accomplish their task (Haenni, Lungu, Schwarz, and Nierstrasz (2013)). Abdalkareem *et al.* (Abdalkareem, Nourry, et al. (2017)) found that developers have biased perception about trivial packages, thinking that these packages are well tested. After including third-party packages, developers are often too reluctant to updates their dependencies. New versions of the packages improve functionalities and fixe security issues or bugs. Kula *et al.* (Kula, Germán, Ouni, Ishio, and Inoue (2018)) observed that 81.5% of their studied applications have outdated dependencies, although these applications heavily depend on external packages. Their interviewing of developers revealed that they were often unaware of the security vulnerabilities of underlying dependencies and perceived updating dependencies not a necessity but additional work. The study of Wittern *et al.* (Wittern et al. (2016)) showed that the package version number is not a good predictor of a package's maturity. To assist developers in updating dependencies, evaluating four software packaging ecosystems (Cargo, npm, Packagist, and Rubygems), Decan *et al.* (Decan and Mens (2019)) proposed an evaluation based on the "wisdom of the crowd" to select appropriate semantic versioning constraints for their dependencies. These types of ecosystem-wide studies helped to clarify various general misconceptions and mitigate bad practices in ecosystems.

Lertwittayatri *et al.* (Lertwittayatrai et al. (2017)) analyzed *npm* ecosystem topology by using network analysis technique to extract patterns of existing libraries by studying its localities. Mens (Mens (2016)) discussed the socio-technical aspects of software maintenance and evolution. He emphasizes on studying both technical and social factors while analyzing software ecosystems.

We utilized his proposed metrics to evaluate the effect of the removal of a package from a software ecosystem.

Other studies examine the API usage of external packages. Mileva *et al.* (Mileva, Dallmeier, and Zeller (2010)) studied API usage patterns of external libraries to examine the popularity of their API. They used this popularity metric to determine if a package is successful or not. Holmes *et al.* (Holmes and Walker (2007)) quantitatively analyzed how APIs are used. They consider the frequency of API use as the popularity and importance of that API. We determine the importance of an external package by analyzing the percentage of its API calls in the files that depend upon those packages.

Overall, this thesis examines software applications that depend on at least one trivial package from the *npm* ecosystem. This thesis is focused on the characteristics of software applications that use external packages from *npm* ecosystem. This categorization helps understand the ecosystem better and adhere to good practices and mitigate bad practices ecosystem-wide.

2.3 Software Quality

Quality assessment and bug prediction is one of the most important domains in software engineering research. In our study, similar to other studies (Kim, Whitehead, and Zhang (2008); McIntosh, Kamei, Adams, and Hassan (2016); Śliwerski, Zimmermann, and Zeller (2005); Wehaibi, Shihab, and Guerrouj (2016)), we use a keyword-based approach to recognize commits that fix some bugs. Abdalkareem *et al.* (Abdalkareem, Shihab, and Rilling (2017)) analyzed the quality of software applications that reuse code from StackOverflow. We analyze the quality of software applications that do not source raw source code but rather reuse code in the form of external packages. Similar to Abdalkareem*et al.*, we evaluate the bugginess of software applications before and after code from the external source is reused in the applications.

Prior studies evaluated changes that induce future bugs in software applications. Śliwerski *et al.* (Śliwerski et al. (2005)) introduced the SZZ technique to locate fix-inducing changes by checking the version control system and bug database. Several other studies enhanced the SZZ algorithm (da Costa et al. (2017); Kim, Zimmermann, Pan, and Jr. Whitehead (2006); Mizuno and Hata (2013);

Williams and Spacco (2008)). Our study leverages the SZZ technique to examine the riskiness of the commits that introduce trivial packages into software applications. We utilize Commit guru (**?**) for analyzing fix-inducing changes.

2.4 Summary

This chapter surveyed prior studies that are most related to this thesis. Specifically, it discussed work related to the third-party package creation and usage in software applications and ecosystems and how they are related to the impact of overall software quality. Our literature review showed that the trend of using trivial packages is popular in different software ecosystems that include for example *npm* and PyPI. However, most of the prior work assumes that these trivial packages are by definition small in size and provide simple functionalities and that their impact of the overall software applications and ecosystems can be neglected. To fill this gap in the following two chapters (Chapters 3 and 4), we describe two empirical studies on the triviality of trivial packages and how trivial packages can impact software quality.

Chapter 3

Untriviality of Trivial Packages: An Empirical Study of the npm JavaScript Packages

3.1 Introduction

The use of third-party packages is becoming increasingly popular since it allows teams to reduce development time and costs and increase productivity (Abdalkareem, Nourry, et al. (2017); Murphy-Hill et al. (2019); Wagner and Murphy-Hill (2019)). A major enabler for the use of third-party packages (hereafter referred to as packages) is the capability for developers to easily share their code through software packages on dedicated platforms, known as software package managers (e.g. Node Package Manager (*npm*) and Python Package Index (PyPI)). Entire ecosystems have been created around these package managers, e.g., the Node.js ecosystems are largely supported by *npm* (Cox (2019)).

Despite the many benefits and wide popularity of using software packages, they also pose some major drawbacks such as increased maintenance costs, and increased risk of exposure to vulnerabilities and even legal issues (Decan et al. (2018); Inoue et al. (2012); Lim (1994); Zapata et al.

(2018)). One specific incident, the *left-pad* incident (Abdalkareem, Nourry, et al. (2017); Macdomald (2016)), triggered a large debate on whether developers should be reusing packages for "trivial tasks"[1]. Since then a number of studies focused on the topic of "trivial packages" and found that indeed, the left-pad incident is not isolated, and that trivial packages account for more than 17% of the 800,000 packages on *npm* (Abdalkareem, Nourry, et al. (2017); Kula, Ouni, German, and Inoue (2017)). In addition, these packages tend to be heavily used, with some trivial packages (e.g., `escape-string-regexp`) being downloaded more than 11 million times per week (npm search (2018)).

The fact that these trivial packages play such a central role made us ask the question **are trivial packages really trivial?** Although we do agree that these packages may be small in size and implement very specific functionality, their prevalence warrants questioning their triviality. In this chapter, we examine the triviality of trivial packages based on their *usage*. We focus on the usage of trivial packages in (1) the applications that use them (**application usage**) and (2) the role they play in the ecosystem they belong to (**ecosystem usage**).

We perform an empirical study analyzing more than 15,000 JavaScript applications, of which 3,965 depend on trivial packages. To examine **application usage**, we use static analysis to determine the importance of the files that use trivial packages and analyze how widely the trivial packages are used in these files. To examine **ecosystem usage**, we leverage network analysis to examine the role of trivial packages in the ecosystem's dependency network. Our study is formalized through three Research Questions (RQs):

- **Application usage. RQ1: Are trivial packages used in important parts of JavaScript applications?** To better understand how applications use trivial packages, we examine their role in the source code files of the applications that depend on them. Using the call graph, we find that files that depend on trivial packages are important in their respective applications. This finding indicates that trivial packages may not be so trivial after all, because they are used in important parts of the applications that depend on them.

- **Application usage. RQ2: How widely used are trivial packages in JavaScript Applications?**

[1]The left-pad incident refers to a 11-line package that implements simple string manipulation. This package was used by Babel, a package that is used by the most major website, including Facebook, Netflix, and Airbnb.

In addition to knowing if the trivial packages are used in important parts of the application, we study if the trivial packages are widely used (i.e., are they only used in one important part or throughout the applications). Again, we use static source code analysis to determine the percentage of API calls that are made to trivial packages. Also, we measure the entropy of packages to determine the spread of their use. We find that trivial packages are at least as widely used as non-trivial packages, indicating that they may not be so trivial.

- Ecosystem usage. **RQ3: Do trivial packages play an important role at the ecosystem level?** To complement our analysis in RQs 1 and 2, which focus on application-level usage, we examine the importance of trivial packages within the ecosystem. We study the package dependency network for both direct and transitive dependencies of the studied applications. We find that trivial packages are more important to the ecosystem than non-trivial packages. Moreover, we find that removing certain trivial packages from the ecosystem may impact up to 30% of other packages in the ecosystem. Our result shows that trivial packages are important building blocks in the ecosystem, and hence that their role is not trivial.

Our study makes the following contributions:

- To the best of our knowledge, this is the first in-depth study that examines the importance and role of trivial packages to applications using them and to the ecosystem to which they belong.
- The findings of this chapter are based on an extensive analysis, which includes a large dataset of JavaScript applications that depended on trivial packages and the use of a state-of-the-art technique that include dependency network analysis.
- To encourage replication and further study on the use of trivial packages, we disclosed our dataset and source code for our analysis in our replication package.

Chapter organization: Section 3.2 presents our study design and approach. We describe our results in Section 3.3. We discuss the implications of our study in Section 3.4. We discuss threats to validity in Section 3.5. Finally, Section 3.6 concludes the chapter.

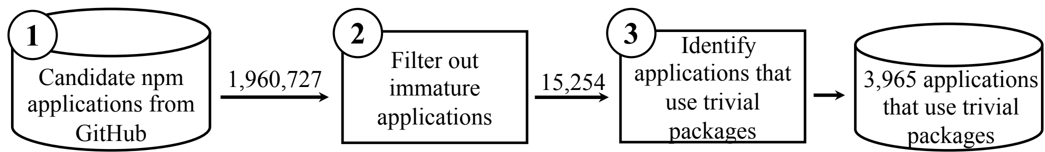

Figure 3.1: An overview of our data collection approach.

3.2 Case Study Design

To investigate the role of trivial packages in software applications, we study a large dataset of JavaScript software applications that depend on at least one trivial package. Figure 4.1 shows an overview of our general approach. We describe each step in our approach below.

3.2.1 Dataset of Candidate Applications

Our analysis focuses on understanding the role of trivial packages in software applications that use them, we must study a diverse and sufficiently large number of JavaScript applications that depend on trivial packages.

To acquire our dataset, we resort to the public GHTorrent dataset (Gousios (2013)) to extract information about all the JavaScript applications hosted on GitHub. We extract the data pertaining to 7,863,361 JavaScript applications that are hosted on GitHub, as of 15th March 2019. We then filter out applications that do not use *npm* as their package management system. We found 2,289,130 applications use *npm* as their package management system (i.e., applications have `package.json` file, which is the configuration file for *npm* applications). Moreover, some *npm* packages use GitHub as their code repository, we exclude these *npm* packages from our list by crosschecking our list of URLs and GitHub URLs of all the *npm* packages.We exclude *npm* package repositories from our dataset so we do not analyze them as standalone JavaScript applications. We identify 3,28,343 *npm* packages in our list of candidate applications and we filter these packages out.

3.2.2 Pruning List of Applications

As recommended in prior work (Abdalkareem, Nourry, et al. (2017); Kalliamvakou et al. (2014)), we perform extra steps to eliminate immature applications from our candidate dataset. We adopt similar filtering criteria that were used in prior work. We choose to select applications that are

Table 3.1: Filtering steps of the studied JavaScript applications.

Filtering Step	**# Applications**
JavaScript applications in GitHub	7,863,361
npm Applications in GitHub	2,289,130
JavaScript applications that are not *npm* packages	1,960,787
Filtering out immature and/or inactive applications	15,254

non-forks, have more than 100 commits by more than one contributor and have a community interest (i.e., applications that have at least one star and a watcher on GitHub). Finally, we select applications that have at least one external *npm* package dependency. These filtering steps allow us to extract a list of 15,254 JavaScript applications that are the client of *npm* packages (Step 2 Figure 4.1). Table 3.1, shows the steps and number of applications after each step in the dataset acquisition process. Table 3.2 shows the summary statistics for different metrics of the selected JavaScript applications in our candidate dataset. Our dataset contains a good distribution of applications in terms of developers, commits, watchers and stars.

3.2.3 Identifying JavaScript Applications that Use Trivial Packages

The goal of this study is to understand the role of trivial packages in JavaScript applications, we must identify applications that depend on trivial *npm* packages in the selected candidate applications. We start by cloning the selected 15,254 applications. Then, we analyze them following a four-step approach (step 3 in Figure 4.1) to identify applications that use trivial packages.

First, we extract each application dependency information by examining the `package.json` file, which is the configuration file for *npm* applications. The package.json, among other configurations, specify the list of packages that the application depends on.

We extract the package name and its associated version for each dependency for each application in our 15,254 applications candidate dataset.

Once we have the list of dependencies for each application in our candidate dataset, we download these packages using the package name and related version information. We download the dependent packages by using the npm-pack command (*npm-pack* (2009)). The npm-pack command

Table 3.2: Summary of the number of developers, commits, watchers, and stars for 15,254 JavaScript projects.

Measurement	**Min.**	**Median**	**Mean**	**Max.**
Developers	2	5	6.74	69
Commits	100	271	669	97,504
Watchers	1	6	23.99	2,451
Stars	1	9	303.73	48,765

consults with the *npm* registry (*npm-registry* (2009)) and resolves the semantic version and downloads the appropriate 'tar' file that contains the source code of the package for each dependency-version pair.

Third, once we have the 'tar' file for each *npm* package, we analyze them to identify trivial packages. We extract the 'tar' file and analyze if the package is trivial or not by leveraging the definition proposed by Abdalkareem *et al.* (Abdalkareem, Nourry, et al. (2017)), which categorize a package as trivial if its number of JavaScript "Line of code (LOC)" $\leq$ 35 and "Cyclomatic Complexity" $\leq$ 10. We analyze all the packages using the Understand tool (*SciTools.com* (1996)). Understand is a static analysis tool that provides, amongst other metrics, Line of Code (LOC) and Cyclomatic complexity measures for the packages.

Forth, we categorize applications that are trivial package dependent. We used the depchecker (*depcheck-npm* (2013)) tool to extract the external packages that are used in JavaScript files. For each file in the studied JavaScript applications, we extract the number of dependent packages, and how many of these dependent packages are trivial. If a file depends on one or more trivial packages, we consider that file as a trivial dependent file, otherwise a non-trivial dependent file. If an application has at least one trivial dependent file then we flag it as a trivial dependent application.

According to this approach, in our candidate dataset, among the 15,254 JavaScript applications that we analyze, 26% (3,965) of the applications are trivial dependent. We want to analyze the role

Table 3.3: Distribution of number of *npm* packages in all the JavaScript applications in our dataset.

Type of packages	**Min.**	**Median**	**Mean**	**Max.**
trivial	1	2	2.34	31
non-trivial	1	16	19.69	106

of trivial packages in JavaScript applications, we conduct our analysis on these 3,965 JavaScript applications dataset that use at least one trivial *npm* package. Table 3.3 shows the distribution of trivial and non-trivial packages in the applications in our dataset.

3.3 Case Study Result

This section presents the results of our three RQs. For each RQ, we provide motivation, describe the approach used, and present our results.

3.3.1 RQ1: Are trivial packages used in important parts of JavaScript applications?

Motivation: Previous work showed that trivial *npm* packages are widespread, and has arguably some negative impact on software applications (Abdalkareem, Nourry, et al. (2017)). However, these packages are small in size and complexity, one may expect that they are used in the unimportant part of software applications. To understand how applications use trivial packages, we examine their role in the source code files of the dependent applications. For example, if a trivial package is used in an isolated part (i.e., file) in an application then its impact on that application can be neglected. Answering this question will help us understand the relative importance of trivial packages in the software applications that use them.

Approach: To examine a trivial package's importance in a JavaScript application, we identify the files that use trivial packages since they provide a direct link between trivial packages and their importance in an application. In this analysis, a trivial dependent file is a file that uses at least one trivial package, whereas, a non-trivial dependent file is a file that does not use any trivial packages.

We examine the importance of trivial dependent files by analyzing the dependency graph among the files of an application and measure the centrality score (Freeman (1978)) of trivial dependent and non-trivial dependent files.

Table 3.4: Distribution of number of files in projects in our dataset.

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
10	26	54	115.1	109	5921

Table 3.5: The distribution of the degree centrality of *trivial* and *non-trivial dependent* JavaScript files.

File Type	**Min.**	**1st Qu.**	**Median**	**Mean**	**3rd Qu.**	**Max.**
Trivial	0.00	0.003	0.022	0.061	0.070	1
Non-Trivial	0.00	0.000	0.003	0.021	0.019	1

To identify the JavaScript files that are more central in a software application, we apply network analysis on the call graph of each application and measure the centrality score. The centrality score of a node in a network reflects how important that node is in the network (Cadini, Zio, and Petrescu (2009); Qi, Fuller, Wu, Wu, and Zhang (2012); White and Smyth (2003)). In scientific literature, network analysis is a popular measure in social sciences, which studies networks between humans (actors) and their interactions (ties). In our context, the JavaScript files are the actors and their inter-dependencies are the ties. For each JavaScript file within an application, we extract information on which other files the concerned file depends (out-degree) and by which other files the concerned file is being dependent upon (in-degree). Then, we calculate the degree centrality score (Freeman (1978)) for each file of an application in our dataset. The degree centrality score is a measure of the number of in-degree and out-degree for a JavaScript file within an application. This degree centrality score is normalized by dividing by $n-1$, the maximum possible degree in a graph that has n total nodes in that graph. The degree centrality of a node V_i is given by:

$$Degree\ Centrality\ (V_i) = \frac{|N\,(V_i)|}{n-1} \tag{1}$$

Where the $|N(V_i)|$ is the number of nodes (files in our case) that are connected to the node V_i (i.e., file under examination). The degree centrality score has a value in [0, 1], where 1 means that the node is in the center of the network (i.e., connected to all other nodes) and zero indicates that the node is isolated.

To calculate the degree centrality of trivial and non-trivial dependent files in each application in our dataset, we start by generating a call graph representation of files in every application. We use the `madge` tool to generate the call graphs (Henningsson (2014)). The output of this tool is a call graph that shows each file in a software application and a list of files it depends on. We then

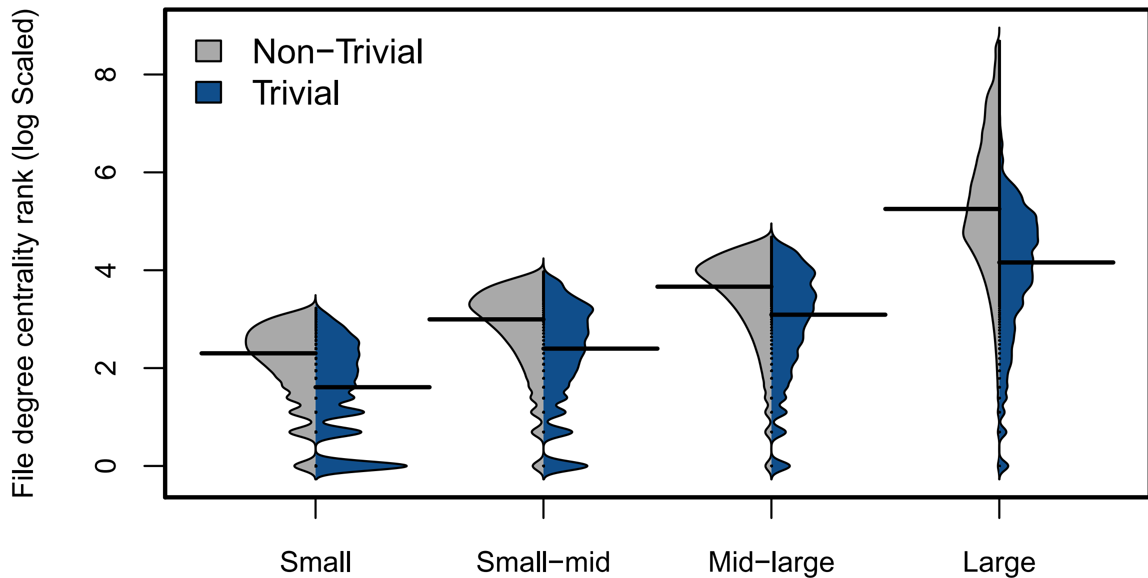

Figure 3.2: Distribution of degree centrality rank of trivial dependent and non-trivial dependent files in different project groups based on number of files.

run the networkx tool (Aric Hagberg and Swart (2005)) on the generated call graph, to calculate the centrality score of every file in the graph. The networkx tool is a well-known tool for analyzing and visualizing social network data. Finally, to put our results in perspective, we compare and contrast the degree centrality score for trivial and non-trivial dependent files.

In addition, to get a more detailed understanding of the JavaScript file's relative importance within a software application, we rank the files based on their degree centrality score, e.g., JavaScript file with highest degree centrality score is ranked 1 and the rank increases with decreasing degree centrality values. The trivial dependent applications in our dataset vary in the number of JavaScript files, we segment the applications into four groups (based on the quartile they fall) namely small, small-mid, mid-large, and large applications based on the distribution of the number of JavaScript files in the applications. From the distribution of the number of files in the studied applications, shown in Table 3.4, we group applications having #files $<$ 1st Qu. into small applications; 1st Qu. $\geq$ #files $<$ median into small-mid applications; median $\geq$ #files $<$ 3rd Qu. into mid-large applications, and #files $\geq$ 3rd Qu. into large applications. We compare the distribution of degree centrality rank for trivial dependent and non-trivial dependent files in each group of applications.

Results: Table 3.5 shows the summary distribution of the degree centrality score for trivial and

non-trivial dependent files in our dataset. We observe that overall the degree centrality values for trivial dependent files are higher than that of non-trivial dependent files. The table shows that the median/mean degree centrality values are 0.022/0.061 and 0.003/0.021 for trivial and non-trivial dependent files, respectively. To test if the difference is statistically significant between the two result sets, we applied the nonparametric Wilcoxon rank-sum test (*wilcox.test function* (2010)). We determine if the difference is statistically significant at the customary level of 0.01. We also estimated the magnitude of the difference between datasets using the Cliff's Delta (*cliff.delta function* (2010)) (or d). Cliff's Delta is a non-parametric effect size measure for ordinal data. We consider the effect size values: negligible for $|d| < 0.147$, small for $0.147 \leqslant |d| < 0.330$, medium for 0.330 $\leqslant |d| < 0.474$ and large for $|d| \geqslant 0.474$. We found that the results is statistically significant (p-value $< 2.2e\text{-}16$) with medium effect size ($d = 0.3471$).

Figure 3.2 shows a beanplot distribution of the degree centrality rank of trivial dependent and non-trivial dependent files for the four groups of applications. We observe that for each group of applications, the trivial dependent files have a lower degree centrality rank than that of non-trivial dependent files, which indicate that trivial packages are used in an important part of the applications. Also, the results for each segment is significant(p-value $< 2.2e\text{-}16$). We also measured the effect size and observed -0.3853 (medium), -0.2397 (small), -0.3355 (medium) and -0.5040 (large) Cliff's delta value for small, small-mid, mid-large and large applications respectively. Overall, these results highlight that trivial packages are used in files that are more central in the studied JavaScript applications.

Our findings indicate that trivial packages are used in more important and central parts of software applications compared to non-trivial packages. In our dataset, trivial dependent files have on median degree centrality value of 0.022 while it is 0.001 for non-trivial dependent files. This difference is statistically significant.

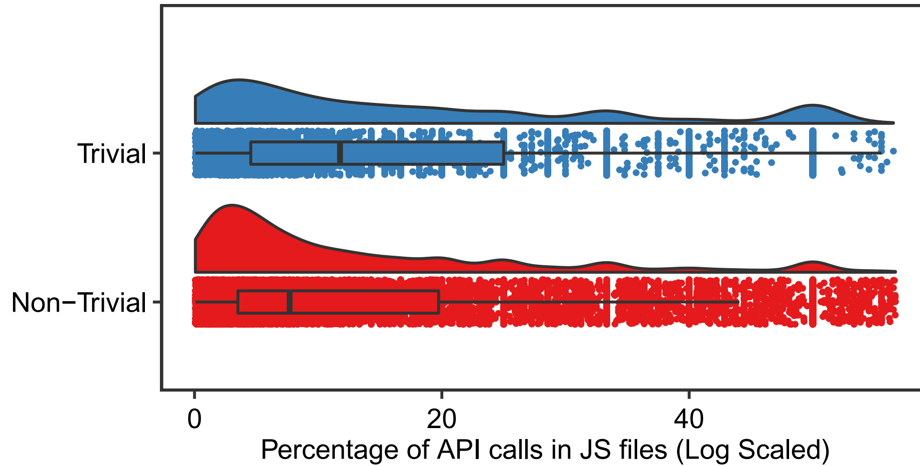

Figure 3.3: Distribution of percentage of API calls for trivial and non-trivial packages in JS files.

3.3.2 RQ2: How widely used are trivial packages in JavaScript Applications?

Motivation: We saw that trivial packages are used in important parts of the applications that depend on them. Next, we want to examine the diffusion of a used package across the applications. In other word, we want to examine if trivial packages are used only in important parts of the applications or their usage is dispersed across different parts of the applications. For example, prior work showed that if the Application Programming Interfaces' (API) of a package Pkg_A are invoked less than APIs' of another package Pkg_B in a software application then this is a clear indication that Pkg_B is more important than Pkg_A in that specific application (Holmes and Walker (2007)). Thus, low usage of trivial package APIs' in a JavaScript file suggests that, even if these packages are used in more important files, these package's importance within that file is low. Therefore, we investigate how heavily a trivial package's APIs are used within a JavaScript file to determine these package's importance within the trivial dependent JavaScript files.

Approach: To determine how widely used trivial packages are within an application, we again perform a two-way analysis. First, we measure the percentage of each package's application programming interface calls in a file that depends on an external package in our dataset. Then, we examine how widespread the use of a package is in each application. We use static code analysis and calculate the following two measures:

Percentage of trivial package API calls in a trivial dependent file: Although, based on our definition,

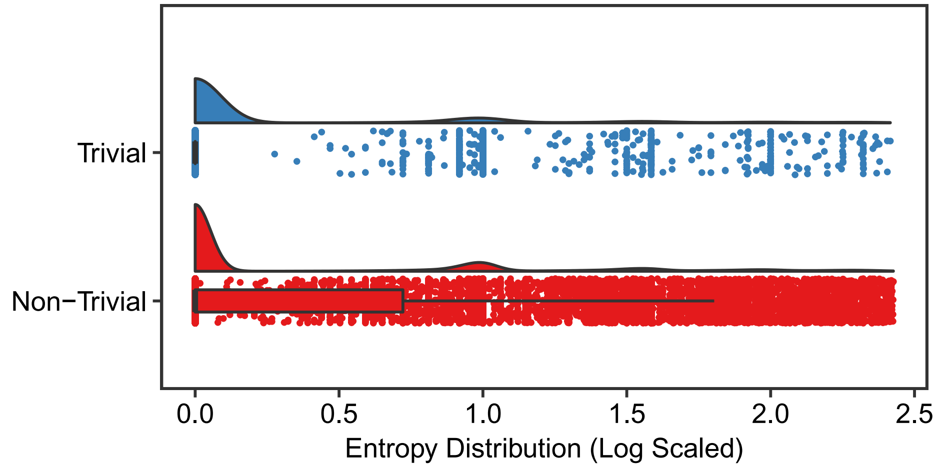

Figure 3.4: Distribution of trivial and non-trivial package API entropy.

a trivial dependent file has at least one trivial package dependency, in fact, it can have any number of non-trivial package dependencies. In our dataset, the median number of trivial and non-trivial packages in trivial dependent files are 1 and 3, respectively. Therefore, these files have a lower number of trivial package dependencies, we want to understand what percentage of total API calls in a trivial dependent file are associated with trivial packages. We use a static source code analysis tool (Understand tool (*SciTools.com* (1996))) to extract and measure all the occurrences of external package API calls in JavaScript files. Then, we calculate the percentage of a package's API calls within a JavaScript file by accounting all the API calls in that file.

External package entropy: We again use the extracted information about the API calls of external packages to compute the entropy of the packages. The entropy of a package shows how widely the package is used in an application. The higher the entropy of a package (i.e., API usage spread across files.), the more difficult it gets to uproot the package from the application. Similar to prior work (Hassan (2009); Kamei et al. (2013)), we define the entropy of an external package as the distribution of API calls of that package across files. For example, in a JavaScript application, the package Pkg_x's APIs are called 10 times in file $F1$, 15 times in file $F2$, and twice in file $F3$, we calculate the entropy of the package Pkg_x as $(-\frac{10}{27}\log_2\frac{10}{27} - \frac{15}{27}\log_2\frac{15}{27} - \frac{2}{27}\log_2\frac{2}{27})$, which equal to 1.28. IHigher the entropy value, the more widespread is the usage of the package in a JavaScript application and if a package is used only in a single file then its entropy is zero.

Result: Figure 3.3 shows the distribution of percentage of API calls for trivial packages and non-trivial packages within the trivial dependent files. We observe that the median value of the percentage of API calls for trivial packages within trivial dependent files is higher than that of non-trivial packages with a median of 11.76% and 7.69% calls, respectively. We examine whether the result is statistically significant and calculate the effect size. We found that the results are statistically significant (p-value $< 2.2\text{e-}16$) and the effect size is small (Cliff's $delta$ estimate = 0.25). This API call analysis of trivial dependent files shows that trivial packages play an important role in these files.

In the second part of this research question, we investigate the distribution of API calls of a trivial package across the application by computing its entropy. Figure 3.4 shows a bean-plot distribution of entropy scores for trivial and non-trivial packages. We observe that trivial and non-trivial packages have similar entropy score distribution with median entropy scores equal to zero for both types of packages. Most of the packages (68.067%) in our dataset have zero entropy scores, which suggests that these packages are used in only a single JavaScript file in the studied JavaScript applications. This result is statically significant with p-value $< 1.789\text{e-}05$ but the effect size is negligible (Cliff's $delta$ estimate: -0.1119). The entropy score distribution of trivial and non-trivial packages indicates that trivial and non-trivial packages tend to be used in the same way thus these two types of packages are equally important in software applications.

> ***A higher percentage of total API calls of JavaScript files are associated with trivial packages (11.76% and 7.69% for trivial and non-trivial packages) and thus these packages are important within these files. Moreover, the entropy distribution of trivial and non-trivial packages shows both types of packages are equally important in software applications.***

3.3.3 RQ3: Do trivial packages play an important role at the ecosystem level?

Motivation: In previous research questions, we found that trivial packages are important components for the JavaScript applications that directly depend on them. However, *npm* packages, trivial or non-trivial, do not exist in isolation, they interconnect with other packages and they form what is

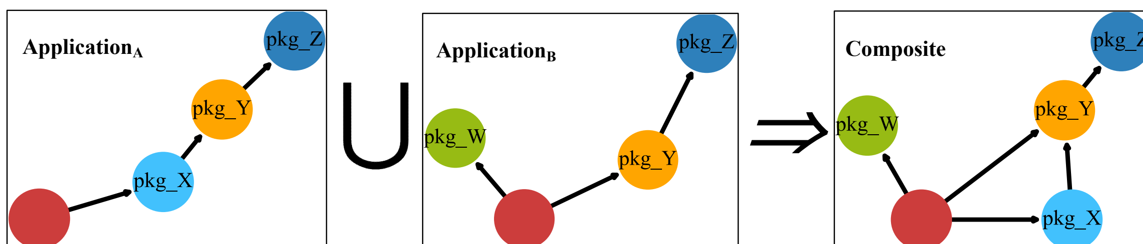

Figure 3.5: Composite Dependency Network

known as the *npm* ecosystem. Examining how important trivial packages are in the software ecosystem they belong provide a general understanding of their importance. Thus, we seek to understand the importance of a trivial package in the dependency network of *npm* ecosystem, which consists of all direct and indirect dependencies of the studied applications.

Approach: To examine the importance of trivial packages from the *npm* ecosystem perspective, we extract all the dependencies (direct and indirect) for each JavaScript application in our dataset and construct its dependency network graph. To extract this package dependency graph, initially, we install and clone the applications' dependencies by using the `npm install` command, which installs the package version specified in package.json file. Thus, all the direct and indirect dependencies of every application in our dataset are saved locally in the application's home directory in a folder named "node_modules". Then, we use the `npm-ls` (*npm-ls* (2010)) to list installed package and their inter-dependencies in *json* format. Subsequently, we merge all the dependency network graphs of all the applications in our dataset and compile a composite dependency network at a given point in time. Figure 3.5 depicts an example of the process of merging the dependency network graphs of two JavaScript applications ($Application_A$ and $Application_B$). In our example, $Application_A$ is directly dependent on pkg_X, which in turn depends on pkg_Y whereas pkg_Y depends on pkg_Z. $Application_B$ has two direct dependencies and one transitive dependency. Here, in the composite dependency network, the dependency hierarchy is preserved while accommodating all the dependencies of both applications. We recursively apply this merging process on all the dependency network of all the applications in our dataset. As a result of this merging process, we get a composite package dependency network that consists of 32,319 connected packages. We

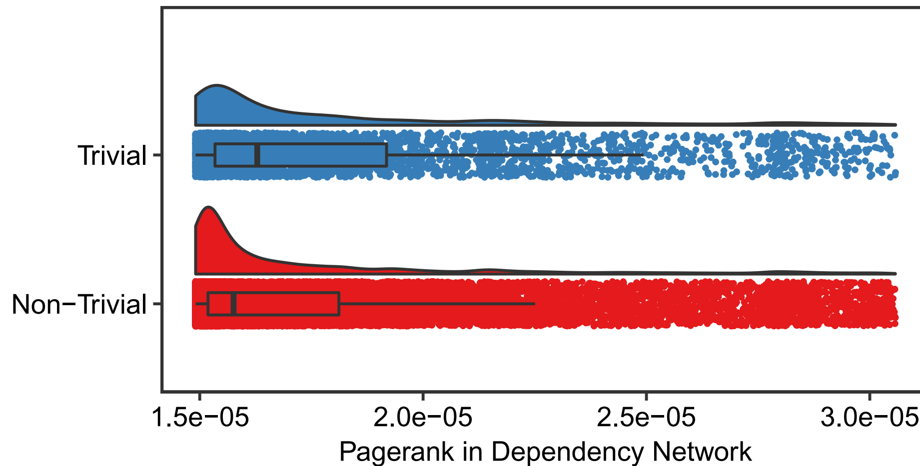

Figure 3.6: Distribution of PageRank values for trivial and non-trivial packages.

analyze the source code of each package in the constructed dependency network and identify trivial and non-trivial packages. We use the composite packages dependency network to examine the importance of trivial packages in two complementary measures. First, we measure the importance of trivial packages within this dependency network using the PageRank algorithm (Brin and Page (1998)). Second, we study the importance of the trivial packages by measuring the *Technical Bus Factor (TBF)* of these packages. Similar to the idea of *social bus factor*, which measures the effect of removal of a developer from a project, the *TBF* measures the effect of the removal of a package from a dependency network (Mens (2016)). In the following subsection, we describe how we measure these values for every package in our constructed graph.

PageRank of External Packages: PageRank score (Brin and Page (1998)) of a node (packages in our case) indicates the importance of the node in a network. The more dependent on a node in a network the higher is its PageRank score. PageRank has a value in [0, 1]. We calculated the PageRank score of every package (trivial and non-trivial) in our composite package dependency network. We again use the network analysis tool called *networkx* tool (Aric Hagberg and Swart (2005)). Then, we compare the PageRank score of trivial and non-trivial packages.

Technical Bus Factor (TBF): To understand the effect of removing one trivial package from the package dependency network, we calculate *TBF*, which simulates the removal of a package from our constructed composite network. We then evaluate how many other packages, directly or indirectly dependent on the removed package, are affected. We calculate what percentage of 32,319 packages,

which is the total number of packages in our dependency network, are affected by the removal of one package from the package dependency network. The higher a package's *Technical Bus Factor (TBF)* value; the more vital that node is in the package dependency network.

Result: Figure 3.6 shows PageRank score distribution for trivial and non-trivial packages. We notice that the median PageRank score of trivial packages (1.71e-05) is higher than that of non-trivial packages (1.61e-05). This result is significant (p-value $<$ 2.2e-16) and effect size is small (Cliff's delta estimate: 0.1578). This result shows that many packages are dependent upon trivial packages which makes trivial packages vital nodes in the ecosystem that they belong to.

Table 3.6 shows the statistical summary of the distribution of *technical bus factor (TBF)* of the trivial and non-trivial packages. We see that removing a trivial package from our composite dependency network has a much larger impact than that of non-trivial package removal. We see that the median *TBF* values for trivial packages is 0.0155 while it is 0.0093 for non-trivial packages. We observe that this result is a statistically significant with p-value $<$ 2.2e-16 and small effect size (Cliff's delta estimate: 0.1525).

We manually analyze top twenty trivial packages, based on *TBF* values, to understand characteristics of these packages. Table 3.7 shows the name, *TBF* value, its rank in dependency network based on *TBF* and the description of the functionalities of the top trivial packages. From Table 3.7, we see that these trivial packages have *TBF* values ranges between 36.82 and 28.91, which means that trivial packages in the list based on the *TBF* value can affect approximately 29% of all packages in the dependency network when any one of these is removed. We rank these packages in dependency network based on their *TBF* where package with highest *TBF* is ranked 1 and rank increases with decreasing *TBF*.

Based on our manual examination of these trivial packages, we found that these packages

Table 3.6: The statistical summary of the distribution of technical bus factor (TBF) for the trivial and non-trivial packages in our composite dependency network.

File Type	**Min.**	**1st Qu.**	**Median**	**Mean**	**3rd Qu.**	**Max.**
Trivial	0.00	0.0031	0.0155	3.5324	0.1918	36.8174
Non-Trivial	0.00	0.0031	0.0093	1.9480	0.0495	34.9485

Table 3.7: The top-20 most impactful trivial packages measured by *Technical Bus Factor (TBF)*.

Packages	**TBF**	**Rank**	**Functionality**
inherits	36.82	1	Inherits one constructor's prototype to another constructor.
isarray	35.43	2	Checks if the object in the argument is an array.
process-nextick-args	34.15	10	Amends the functionality of process.nextTick, which defers a callback function until next eventloop, by enabling it to accept arguments.
debuglog	34.13	11	Shows degugging information in stderr.
escape-string-regexp	32.26	14	Escapes special characters.
ansi-regex	32.00	18	Matches ANSI escape codes.
object-assign	31.90	22	Assigns values to objects.
strip-ansi	31.89	24	Removes ANSI escape codes from a string.
indexof	31.14	49	Returns index of an object in an array.
foreach	30.87	59	Iterates over the key value pairs of either an array or a dictionary like object.
pinkie-promise	30.54	63	Returns JavaScript promise object
is-object	30.20	64	Checks if the argument is an object.
get-stdin	30.10	65	Get standard input as a string or buffer.
xtend	30.03	68	Extends an object by appending all of the properties from each object in a list.
has-flag	29.77	70	Checks if function argument has a specific flag.
has-color	29.67	73	Detects whether a terminal supports color.
once	29.65	74	Restricts a function to be called only once.
graceful-readlink	29.02	79	Returns a file's symbolic link.
number-is-nan	28.91	82	Checks whether the value in the argument is undefined and its type is Number

provide popular utility functions, enhancement of JavaScript standard functionalities, and cross-platform compatibility features.

First, the examined trivial packages provide some popular utility functions like checking objects, e.g., `has-flag`, `has-color`, `is-object`, `number-is-nan`; string operations, e.g., `ansi-regex`, `strip-ansi`; and object manipulation, e.g., `xtend`, `foreach`. The second group of the examined trivial packages is used to enhance the existing native functionality of the JavaScript engine. For example, `process-nextick-args` (Metcalf (2015)) extends the capability of *process-nextick* by enabling this function to accept arguments. Finally, we found some trivial packages provide functionalities that help developers to deal with cross-platform compatibility. JavaScript code can be run on different types and versions of web browsers, these packages

provide backward and forward compatibility. For example, `isarray` (Gruber (2013)) is a well-known package and in the dependency network it is ranked 2nd based on it's *TBF*. It provides same functionality like the native *Array.isArray*. *Array.isArray* is supported by browsers with newer version, e.g. IE9+, Chrome 5+, Firefox 4+, Opera 10.5+ and Safari 5+. However, as this function is not supported in older versions of browsers, *isarray* is used to support older browser versions that are not compatible with ECMAScript 5 or later. These types of packages that provide cross-platform compatibility are known as *ponyfills* and *polyfills* (Sorhus (2016)). *polyfills* are prone to unexpected bugs as these pollute the global scope. *ponyfills* is the smarter alternative as it exports functionalities as a module without exploiting global scope. 25% of top 20 trivial packages e.g isarray, debuglog, object-assign, pinkie-promise, number-is-nan, are *ponyfills*. Furthermore, 37.97% of all *ponyfill* solutions in *npm* are trivial packages (*npms-ponyfill* (2009); Sorhus (2016)). From this analysis, we see that trivial packages are often the byproduct of compatibility efforts.

Additionally, this analysis of the top 20 trivial packages revealed that some developers have a proclivity of publishing trivial packages. For example, Sindre Sorhus (Sorhus (2013)), a famous open-source developer, who created Yeoman (*Yeoman* (2012)) and Awesome Project (Sorhus (2014)), collaborated 7 of the top 20 trivial packages. We examined all of his 1,148 packages in *npm* and surprisingly 55.14% of his published packages are trivial packages.

Trivial packages are vital nodes in the package dependency network (i.e., ecosystem). In fact, our results show that 16.19% of trivial packages and only 9.27% of non-trivial packages have a TBF value grater than 15%.

3.4 Discussion

Our results were presented on a specific snapshot of the applications and their dependencies. Hence, we further investigate the validity of our findings over time.

Re-examining the Role of Trivial Packages Overtime

In research questions 1 and 2, we focus on studying the importance of trivial packages from the perspective of how they are used. We examine the current snapshot of the studied applications[2]. Now, we examine the role of trivial packages in the studied applications over time. We believe that examining the usage of the trivial package over time will provide us with a general overview of the usage of trivial packages compare to only examine the current snapshot. Also, an increment in the number of trivial dependent files overtime in a software application suggests these packages' importance and developer's reliance on these packages whereas decrement suggests otherwise.

First, we examine the evolution of the number of trivial dependent files over their development timespan of an application. Second, we analyze the evolution of the percentage of trivial package API calls in trivial dependent files over the development timespan of software applications To identify the development period in which an application has some trivial package dependency, we need to know the commit that introduced the first trivial package in an application. This commit is either the first commit in a software application or before this commit the application was non-trivial dependent. All the applications in our dataset use git as their source control system, we iterate each commit starting from the initial commit of a software application to check if the commit is adding any trivial package into a JavaScript file. When we encounter such commit, we break the iteration and mark and register that commit as a trivial introducing commit for that software application.

Trivial dependent applications in our dataset start being trivial dependent from the trivial introductory commit. We consider the development timespan of an application, which ranges from first trivial introductory commit till the latest commit as trivial dependent development timespan (TDDT). We segment this TDDT into 10 equal parts by the means of the total number of commits in this period. For each application, we count the total number of commits in its TDDT and take a snapshot at each 10th percentile commit. Therefore, this segmentation process provides 11 snapshot points for each application, which are at: first trivial introductory commit, 10% commit, 20% commit, 30% commit, 40% commit, 50% commit, 60% commit, 70% commit, 80% commit, 90% commit and latest commit. As module growth is a predicted phenomenon in the software development lifecycle (Godfrey and Qiang Tu (2000); Lehman (1980); Xie, Chen, and Neamtiu (2009)),

[2] In our study, the current snapshot of an application refers to the date when we collected application in our dataset.

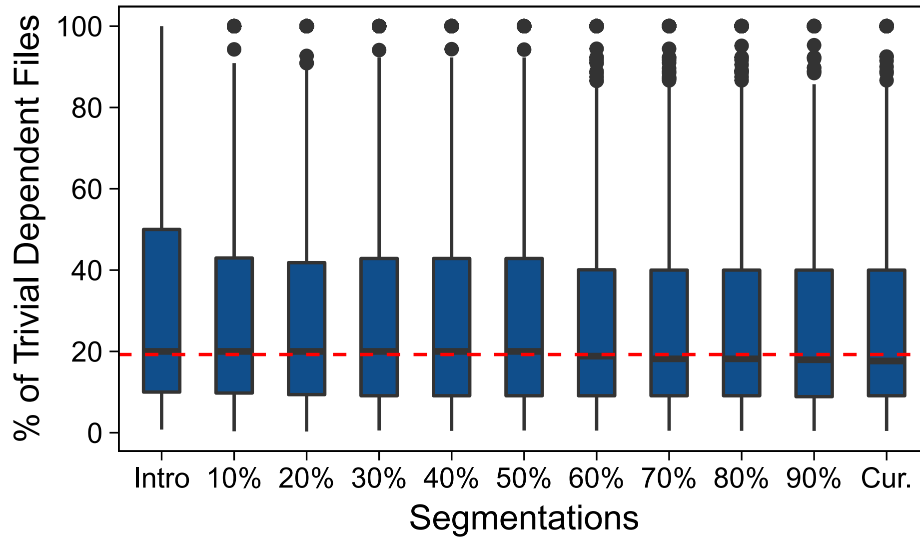

Figure 3.7: The distribution of the percentage of trivial dependent files in all the studied applications based on TDDT segmentations. Dotted horizontal line present overall median.

Table 3.8: The statistical summary of the distribution of external package API call percentage in JavaScript files throughout application's development lifespan. The table shows the distribution for trivial packages (TP) and non-trivial packages (NPT).

Segments	**Intro**		**10%**		**20%**		**30%**		**40%**		**50%**		**60%**		**70%**		**80%**		**90%**	
	TP	**NTP**	**TP**	**NTP**	**TP**	**NTP**	**TP**	**NTP**	**TP**	**NTP**	**TP**	**NTP**	**TP**	**NTP**	**TP**	**NTP**	**TP**	**NTP**	**TP**	**NTP**
Min.	0.1	0.1	0.1	0.1	0.2	0.2	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Median	16.7	9.1	16.7	9.1	16.7	9.1	16.7	9.1	16.7	9.1	16.7	8.6	16.7	8.7	16.7	8.3	16.7	8.3	16.7	8.3
Mean	30.1	16.4	30.4	16.5	30.0	16.4	30.5	16.3	30.2	16.3	30.7	15.9	30.7	16.0	30.8	15.9	30.8	15.9	30.3	15.8
Max.	100	96.6	100	96.5	100	96.6	100	96.8	100	96.9	100	97.1	100	96.9	100	97.3	100	97.5	100	97.6
p**-value**	<2.2e-16		<2.2e-16		<2.2e-16		<2.2e-16		<2.2e-16		<2.2e-16		<2.2e-16		<2.2e-16		<2.2e-16		<2.2e-16	
Cliff's d	0.2434		0.2540		0.2453		0.2626		0.2589		0.2641		0.2635		0.2661		0.2760		0.2543	

we measure the percentage of trivial dependent files to all files across an application's TDDT, not the raw number.

Figure 3.7 shows box-plots of the percentage of the number of trivial dependent files in all the studied applications in our dataset for the 11 snapshot points in the applications' TDDT. Here, we observe that the percentage of the number of trivial dependent files remains almost constant over time with approximately the median percent of trivial dependent files equal to 20%. These results reflect the importance of and developer's reliance on these trivial packages in software applications.

We further investigate the percentage of trivial packages' API calls in trivial dependent files throughout the concerned application's TDDT. Table 3.8 the shows percentage of package's API calls distribution in these files for each application across its TDDT. Once again, to put our analysis

in perspective, for every TDDT segment, the table shows the percentage of trivial packages (TP) and non-trivial packages' API calls.

From Table 3.8, we observe that the percentage of trivial package's (TP) API calls is higher than that of non-trivial packages (NTP) API call at each snapshot point in the applications development timespan. For example, at 30%'s TDDT, we see that trivial packages' API calls are higher (with mean=30.5 and median = 16.7) that the percentage of API calls for the non-trivial packages (with mean = 16.3 and median = 9.1). We see similar results at the late of the development lifespan of the studied applications. At 90%'s TDDT, we see that with 30.3/16.7 mean/median of API calls for trivial packages is higher than the ones for the non-trivial packages (15.8/8.3).

To examine whether the results are statistically significant, we perform the Wilcoxon rank-sum test and the Cliff's Delta effect size test on the data from each segment. The last two rows of Table 3.8 shows p-value and the effect size between the percentage of trivial and non-trivial packages' API calls for every TDDT. We see that these results are statistically significant and have small effect sizes in all the snapshot points. For example, at 30% TDDT, we found that the difference between the percentage of the API calls for trivial and non-trivial packages are statically significant (p-value = $<$2.2e-16) and the effect size is small. This analysis shows that the percentage of API calls for trivial packages within trivial dependent files remains higher throughout the development timespan of the concerned software applications.

3.5 Threats to validity

In this section, we discuss the threats of validities related to our study.

3.5.1 Construct validity

Construct validity considers the relationship between theory and observation, in case the measured variables do not measure the actual factors. In our study, we used several in-house and state-of-the-art tools and techniques. We used the Depchecker (*depcheck-npm* (2013)) tool to extract file-level dependencies, the madge tool (Henningsson (2014)) for generating call graph, and the Understand tool (*SciTools.com* (1996)) for static analysis. Hence, we are limited by the accuracy of

these tools. Our study consists of four million JavaScript files. Thus, it is time and resource consuming to manually check each file and these tools' results. To mitigate the threats related to using these state-of-the-art tools, we randomly selected five applications from our dataset and manually cross-checked the output of these tools and in all cases, the tools produce the correct results. We also use the networkx (Aric Hagberg and Swart (2005)) tool to generate the dependency graph of files of every JavaScript application. Again, our graph dependency network analysis may influence the accuracy of the generated graph. To alleviate these issues, we manually examine the generated call graphs for five applications in our dataset and found that these graphs represent the dependency structure between files in these applications.

To answer our second research question, we only captured the direct usage of external packages in our static code analysis. For example, a package "X" is imported (e.g require statement) and assigned it to a variable "a" and later "a" is assigned to another variable "b". We only tracked the external package usage with variable "a" and did not track "b". We decide to examine the direct usage of these packages for two main reasons. First, this type of transitive assignment of a variable is very rare in JavaScript code as other work shows (Feldthaus, Schfer, Sridharan, Dolby, and Tip (2013)). We believe that this shortcoming does not significantly impact our finidings. Second, if we miss some of the usages of external packages, we missed both trivial and non-trivial packages. As we contrast trivial and non-trivial package usage, this effect will not affect the result of the comparison.

3.5.2 External validity

Our dataset only consists of JavaScript applications, which use *npm* as their package manager, hence our findings may not hold for applications written in other programming languages or use different package manager. However, *npm* mainly supports JavaScript applications and it is one of the largest and most rapidly growing software ecosystem (Decan, Mens, and Grosjean (2019)). In addition, our dataset presents only open source application hosted on GitHub that may do not reflect proprietary applications. Also, our initial dataset size is 15,254 JavaScript applications that use the *npm* package manager, which may not represent the whole population of JavaScript applications.

3.6 Chapter Summary

Code reuse in the form of small/trivial packages became prevalent in software development Abdalkareem, Nourry, et al. (2017); Abdalkareem et al. (2019). We observe that these trivial packages, being small in size and complexity, provide various functionalities ranging from string manipulation to security. Thus it is important to understand whether these packages are trivially used or their usage in software applications transcends their triviality. We empirically examine trivial packages relative importance their use cases from two point of views; from the applications usage and ecosystem usage. We analyze a large dataset of open-source JavaScript applications that depend on at least on trivial package.

We observe that trivial packages are used in important part of the examined software applications compare to non-trivial packages. Our results show that trivial dependent files have on median 0.022 degree centrality value while it is 0.001 for non-trivial dependent files. We also, found that trivial packages have a higher percentage of total API calls of JavaScript files (11.76% and 7.69% for trivial and non-trivial packages). As for the ecosystem usage, we examine the relative importance of trivial packages in the ecosystem they belong to where we analyze the dependency graph of the direct and transitive dependencies of software applications in our dataset. We observe that trivial packages are highly dependent upon packages in the *npm* ecosystem, which makes trivial packages salient in the ecosystem. In some case removing one trivial package from the *npm* ecosystem could effect up to 30% of the whole *npm* ecosystem.

In the next chapter, we focus on studying the impact of using trivial packages on software quality. We first examine the functionalities that trivial packages provide and the development activities that introduce trivial packages to software applications. We then focus on examining the impact of using trivial packages on the files- and applications-levels.

Chapter 4

An Empirical Study on the Impact of Using Trivial Packages on Software Quality

4.1 Introduction

Nowadays, software applications heavily depend on reusing other source code in the form of external packages that are generally available in package manager platforms (e.g., *npm*, RubyGems, Maven, PyPi, NuGet.). The availability of large amounts of these tailored third-party packages facilitates and accelerates software development and its evolution. Thus, it becomes a broadly adopted practice in software development (Inoue et al. (2012); Mockus (2007)).

Despite ubiquitous usage, whether source code reuse practice is healthy or not is subject to debate among researchers. For example, prior work showed that code reuse can reduce time-to-market and speed up overall productivity (Basili, Briand, and Melo (1996); Lim (1994); Mohagheghi, Conradi, Killi, and Schwarz (2004)). Additionally, using third-party packages enhances developer's productivity Wagner and Murphy-Hill (2019), therefore, companies encourage using these packages to gain initial momentum of a software application (Haefliger, von Krogh, and Spaeth (2008)). Conversely, code reuse may lead to an increase in maintenance costs (Lim (1994)) in the long run

and even expose an organization to legal issues (Abdalkareem, Shihab, and Rilling (2017); Inoue et al. (2012)). Because of these confounding factors, the study of different types of code reuse and their impact on software quality became a popular research interest (Abdalkareem, Shihab, and Rilling (2017); Basili et al. (1996); Bavota et al. (2013); McCamant and Ernst (2003); Mohagheghi et al. (2004)).

In a contemporary study, Abdalkareem *et al.* (Abdalkareem, Nourry, et al. (2017)) identified a specific genre of code reuse practice where developers tend to use packages that implement simple and trivial tasks. Developers of the applications that depend on trivial packages perceive that these packages are well-tested but in reality, more than 50% of these packages do not have any test code written. However, this prior work examines the use of trivial packages from developers' perspective and there is no empirical evidence on how the usage of trivial packages may impact the quality of the applications that depend on them.

To that end, we empirically examine a dataset consisting of 5,757 JavaScript applications to understand the quality impact of using trivial packages. We analyze these applications and identified 3,158 applications that use at least one trivial package. First, we examine to understand the development scenario which introduces these trivial packages into software applications. In addition, we analyze what kind of functionalities trivial packages provide. To do so, we analyze and categorize the commits that introduce first trivial packages into software applications and the functionalities that these packages provide. Second, we examine the quality impact of using trivial packages on the file- and application-levels. To examine the quality impact of trivial packages and similar to previous work (Abdalkareem, Shihab, and Rilling (2017); Foucault, Palyart, Blanc, Murphy, and Falleri (2015); Kim et al. (2008); McIntosh, Kamei, Adams, and Hassan (2014); Śliwerski et al. (2005); Wehaibi et al. (2016)), we use the number of bug-fixing commits as a proxy for software quality. More specifically, we ask the following research questions:

- **RQ0: In which context trivial packages are introduced into a software application and what types of functionalities trivial packages provide?** We identify 11 types of development activities that are responsible for introducing trivial packages into software applications. We found that 'Modifying Functionalities', 'Building', 'Refactoring', and 'Improving Performance' are the

most frequent development activities that introduce trivial packages into software applications. Additionally, our results show that trivial packages render a wide variety of functionalities ranging from simple string modification to server management or providing security.

- **RQ1: Does using trivial JavaScript packages impact the overall quality of applications?** Our results show that JavaScript applications that use trivial packages tend to have a higher percentage of bug-fixing commits compared to the applications that do not have trivial package dependency.

- **RQ2: What is the impact of trivial packages on the quality of the files?** Based on our examination of the percentage of bug-fixing commits, which is a proxy to software quality, in JavaScript files, we observe that files which depend on trivial packages tend to have significantly more bug-fixing commits.

- **RQ3: Are commits that introduce trivial packages in JavaScript files risky?** Our study reveals that the commits that introduce trivial packages in JavaScript files are significantly more fix-inducing than other commits, which makes these changes risky (Shihab, Hassan, Adams, and Jiang (2012)). Moreover, we observe that the commits that introduce trivial packages of utility category are most risky as 37% of all risky commits are attributed to this category.

Our work makes the following contributions:

- We manually categorize more than 1,900 trivial packages based on their functionalities. Therefore, our study amends the definition of trivial packages by adding what kind of functionalities trivial packages provide.
- We conduct qualitative analysis to understand the development scenario which ships these trivial packages into software applications.
- We studied more than 5,000 JavaScript applications and applying different MSR technique to examine the impact of using trivial packages on software quality.

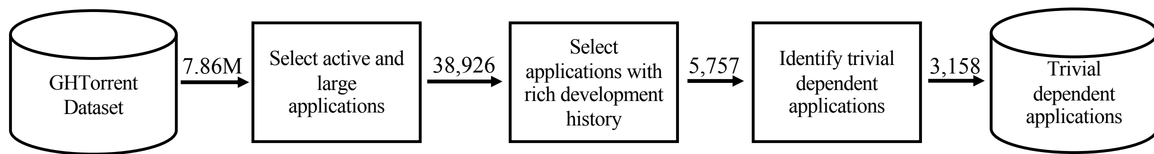

Figure 4.1: Overview of the dataset selection process.

Chapter organization: Section 4.2 presents our study design and approach. We describe our results in section 4.3. Threats to validity is shown in Section 4.4. Finally, Section 4.5 concludes the chapter.

4.2 Case Study Design

The *goal* of our study is to understand the quality impact of using trivial packages on software applications. To do so, we study a large dataset of JavaScript applications. In the following sections, we describe the selection process of the applications in our dataset, and identification of applications and JavaScript files that depend on trivial packages. Figure 4.1 provides an overview of our study design. We describe each of the steps in the approach below.

4.2.1 Dataset of candidate applications

To examine the impact of using trivial packages in JavaScript applications that use *npm* as their package management system, we need to examine a sufficient number of JavaScript applications that depend on trivial packages. It is important to study a large and diverse set of JavaScript applications to conduct a generalized experiment and provide confidence in our results. For dataset acquisition, we query the public GhTorrent dataset (Gousios (2013)) and get the list of 7.86 million JavaScript applications hosted on Github, as of March 2019. Out of these JavaScript applications 2.29 million applications have `package.json` file, which is the configuration management file for Node.js applications. Since some *npm* packages are hosted on GitHub as well as JavaScript applications and we want to examine the applications that depend on *npm* packages but not the *npm* packages themselves, we distinguish between *npm* packages and JavaScript applications. We extract the Github URLs of the *npm* packages from the *npm* registry (*npm-registry* (2009)). We then crosscheck

with the URLs we extracted from the GhTorrent dataset. We exclude the URLs that are common in both sources. After this filtration process, we obtain 1.96 million JavaScript applications.

4.2.2 Select active and large JavaScript applications

Since some applications on GitHub are immature (Kalliamvakou et al. (2014)), we perform extra steps to eliminate such immature applications by adopting similar filtering criteria that were used in prior work (Abdalkareem, Nourry, et al. (2017); Kalliamvakou et al. (2014)). We select applications that are non-forked, have more than 100 commits by more than one contributor, and have more than a year of development lifespan. Additionally, we select those applications, which have at least one external package dependency. These filtering steps allow us to extract a list of 38,962 JavaScript applications that are the client of *npm* packages.

4.2.3 Select applications with rich development history

To eliminate application with little development history, we count the number of commits in each application. We then filter out applications that have less than or equal to 255 commits, which is the median number of commits in the applications of our dataset. This filtering step narrows our dataset to 5,757 applications for further analysis. It is important to note that this filtering step is essential since it allows us to 1) filter out applications that do not have enough software development history, and 2) study a sufficiently large number of applications, but at the same time manageable in size since we will perform some manual analysis on these applications.

4.2.4 Identify applications that use trivial packages

To identify trivial packages and to distinguish files and applications that depend on these trivial packages, we follow a three-step approach.

First, we extract all the packages that an application depends on by looking into `package.json`, which is the configuration file for *npm* applications that contains, among other configurations, a list of all used *npm* packages along with their versions. Upon the completion of this step, we get the dependent packages names and associated versions information for each application of the 5,757 applications dataset.

Second, we download all the dependencies of each application in our dataset using npm-pack command (*npm-pack* (2009)). npm-pack command consults with *npm* registry (*npm-registry* (2009)) and resolves the semantic version and downloads appropriate *tar* file for each dependency-version pair. We extract the *tar* file and analyze if the package is trivial or not by leveraging the definition proposed by Abdalkareem *et al.* (Abdalkareem, Nourry, et al. (2017)), which categorizes a package as trivial if it has "Line of code (LOC)" ≤ 35 and "Cyclomatic Complexity" ≤ 10. To measure the number of lines of code and the Cyclomatic complexity of each package's source code, we use the Understand tool (*SciTools.com* (1996)). Understand is a static analysis tool that provides, amongst other metrics, Line of Code (LOC) and Cyclomatic complexity measures.

Third, we categorized applications and files that are trivial package dependent. We used the depchecker (*depcheck-npm* (2013)) tool to extract which dependent packages are used in which JavaScript file. From the previous step, we label packages as trivial and non-trivial. If a file depends on one or more trivial packages then we consider that file as a trivial dependent file otherwise it is considered as a non-trivial dependent file. In the same way, we consider applications that have at least one trivial dependent file as trivial dependent applications otherwise they are considered as non-trivial dependent applications.

According to this approach, in our dataset, among 5,757 applications, 3,158 are trivial dependent applications and 2,599 are non-trivial dependent. Table 4.1 shows a summary statistic for all the applications in our dataset and the trivial dependent applications. It shows the number of commits, number of developers, development age in days and number of application's dependencies for all applications in our dataset. From Table 4.1, we observe that overall, the mean number of commits per application is 1,123 and the mean number of developers is 39.73. However, when we look at the applications that depend on at least one trivial package (3,158 applications), we see that this mean number of developers is 37 and these have 1,282 commits on average. We also observe that the applications in our dataset have on average more than 3 years of development history. This analysis shows that our dataset contains a mature and representative sample of JavaScript applications that are hosted on GitHub.

Table 4.1: Summary statistic of the studied dataset.

Measure	All Applications				Trivial Dependent Applications			
	Min.	Median	Mean	Max.	Min.	Median	Mean	Max.
Commits	256	566	1,123	62,183	256	587	1,282.1	62,183
Developers	2	12	39.73	1,432	2	12	37.64	1,432
Development age in days	365	1127	1232	17541	365	1072	1205	17541
Dependencies	1	18	25.7	228	1	29	33.68	228

4.3 Case Study Result

Since prior work investigated the reasons behind using trivial JavaScript packages (Abdalkareem, Nourry, et al. (2017)) and it shows that more than 50% of the trivial packages have no test case, the main *goal* of our study is to examine the impact of using trivial packages on the quality of software applications that use them. To that end, we conduct an empirical study to understand the impact of trivial packages on software application's quality. In this section, we answer our research questions. For each research question, we motivate the question, describe our approach to answers the question, and present the result.

4.3.1 RQ0: In which context trivial packages are introduced into a software application and what types of functionalities trivial packages provide?

Motivation: Previous work showed that the use of trivial packages is very common and has arguably some advantage (Abdalkareem, Nourry, et al. (2017)). We know that trivial packages are smaller in size and complexity. Therefore, the general speculation is that most of the functionalities that trivial packages render can be implemented with ease by application developers themselves rather than depending on some external packages (Haney (2016)). So, under what circumstances a developer imports an external package that is small in size is worth investigating. Additionally, although we know the structural definition of trivial packages, what type of functionalities trivial packages render is yet unknown. Thus, in this research question, first, we investigate the context in which trivial packages are introduced into a software application and second, we classify functionalities that trivial packages render.

Approach: To answer this research question, we examine the trivial packages in two ways. First,

Table 4.2: Type of development activities associated with introducing trivial packages into JavaScript applications.

Objectives	**%**	**Brief Description**	**Example Commit Message**
Modifying functionalities	41.3	These activities are involved with managing application dependencies, modifying front-end and back-end functionalities.	*"add library;Update to mathoid-mathjax-node 0.7"*
Building	29.2	These commits manage build infrastructure and modify build tasks.	*"chore(all): new build, contrib and lint"*
Refactoring	18.7	Refoctoring codebase for maintainability.	*"Refactor structure to modularize config versus non config items"*
Performance improving	13.1	Improve response time and scalability	*"Bug 886446 - [email] Partition card loading, cache initial card HTML for fast startup"*
Testing Code	12.8	These commits modifies test code.	*"Added unit tests and basic service registry impl "*
Improving code readability	6.1	Keep code precise and readable.	*"Clean up gruntfile, remove unused dependencies"*
Networking	5.8	Change related to routing and connection to external resources.	*" initial setup: app structure, dependencies, readme, testing tools, build tools, developer tools, basic routing #11, Travis, license"*
Project designing	4.4	These commits involves scaffolding application structure,adding middleware.	*"JHK: github auth enabled, yo mean scaffolding"*
Security	2.9	These changes modify security related features.	*"Added oauth2 foursquare passport strategy "*
Documentation	1.8	Change related to documentation and code comments.	*"moved swagger UI, updated some docs"*
Other tasks	20.7	These commits comprises of initail commits, merging files, adding schemas.	*"initial application commit"*

we examine the development history of the applications to understand the development context that introduces these packages into software applications. Second, we analyze the functionalities that trivial packages render.

To understand the context in which trivial packages are introduced in JavaScript applications, we study the development history of trivial dependent applications. In doing so, we identify the commits that introduce trivial packages into the applications. Since all the applications in our dataset use `git` as their version control system, we analyze each commit of an application that touches a JavaScript file to determine the commit that first introduced trivial packages in the application. Then, we perform a manual examination of these commits. Since our dataset has 3,158 applications that use trivial packages, it is arduous and error-prone to manually examine all of these applications. Thus, we draw a statistically significant sample with a 95% confidence level and confidence interval of 5% similar to prior work (Mujahid, Sierra, Abdalkareem, Shihab, and Shang (2018)). This sampling process resulted in randomly selected 343 trivial dependent applications from our dataset. Therefore from this sample, we found 343 commits that introduced first trivial packages into these applications. We manually examine these commits to comprehend the development context that introduces trivial packages to an application. To categorize these commits, the first two authors read the commit messages and examine the related source code changes and come up with several categories for the commit activities. Then, the two authors discuss the extracted classifications and agreed on the obtained categories. Therefore, they tag each commit with one or more categories. To measure the agreement between these two authors regarding the categorization, we use Cohen's Kappa coefficient (J. Fleiss, Levin, and Paik (2013)). Cohen's Kappa coefficient is a widely used statistical method that evaluates the inter-rater agreement level for categorical scales (J. Fleiss et al. (2013)). This coefficient has a value range between -1.0 and 1.0; where -1 means negative agreement, 0 indicates no agreement and 1 indicates full agreement. In our analysis, we found the level of agreement between the authors is 0.75, which is a moderate inter-rater agreement (J. L. Fleiss and Cohen (1973)).

In the second part of this research question, we investigate the functionalities that trivial packages provide. First, since the studied applications in our dataset have a large number of trivial

packages (1,958), we take a statistically significant sample having a 95% confidence level and confidence interval of 5% and thus we get a sample size of 459 trivial packages. For each of the selected 459 trivial packages, we retrieve their 'readme' files from the *npm*[1] website for the packages. We followed the same aforementioned process to categorizing trivial packages based on their functionality. We run a two-phase iterative process (Seaman (1999)). The first two authors examine the source code and read the 'readme' file of the trivial packages and come up with some categories. Then, they discussed the extracted categories and finalized the categories. In the second phase, they apply the functional categories in the whole population (1,958) but the 'readme' file for 43 packages was missing in the *npm* website. Once again, we examine the agreement between the two authors using Cohen's Kappa coefficient (J. Fleiss et al. (2013)) and found the level of agreement measured between the authors is 0.83, which is considered to be an excellent inter-rater agreement.

Results: Table 4.2 shows the eleven development activity types that introduce trivial packages in JavaScript applications. For each category, we provide the percentage of commits in that category for applications in the sample, a brief description of the category, and an example commit message related to the introduction of the trivial package. Since some commits perform multiple development activities e.g. add new functionality and refactor in the same commit, hence, the commit can be mapped to more than one category. Thus the percentage of categories may sum up to more than 100%.

From Table 4.2, we observe that JavaScript developers tend to introduce trivial packages while performing diverse development activities such as refactoring, modifying functionalities, building, performance improving, and testing. As Table 4.2 shows, 'Modifying functionalities', 'Building', 'Refactoring', and 'Performance improving' are the most frequent development activities that introduce trivial packages. It is also worth mentioning that during our qualitative analysis, we find in 64 instances where, in a commit message, trivial package names are explicitly mentioned e.g., the commit that introduces "load-grunt-tasks" dependency in the application has *"Add load-grunt-tasks"* in the header of the commit. This shows that these trivial packages are the protagonist in those commits. In some cases, the commits were not descriptive enough for a clear classification, for example, commit messages *"Just Wow"*; *"Initial Commit"*. We categorize such a trivial package

[1] https://www.npmjs.com/

Table 4.3: Categories of trivial packages based on functionality.

Category	%	Brief Description	Example of Trivial Packages
Utility	31.07	Trivial packages in this category provides functionalities like string manipulation, stream operation, configuration management, file operations, various data related operations and conversions.	capitalize
Datastructure, numerical and logical operation	11.75	These trivial packages provides functionalities regarding different types of data structure manipulation; numerical, geometric and logical functionalities.	reduce
Build	10.18	These trivial packages help building the application.	grunt-bell
Front-end development	7.94	This category of trivial packages provide CSS functionalities, various front-end modules and DOM manipulation functionalities.	dom-select
Extend or optimize functionality	7.42	These trivial packages are patches that help to extend the scope of another function or module or optimize the function usages.	middleware-responder
Error handle, debug and testing	6.89	These trivial packages provide error management, testing code and debugging functionalities.	debuglog
Security	4.60	Trivial packages that provide security, authentication, encryption and decryption related functionalities falls into this category.	md5-hex
Communication, networking and RPC	4.18	These packages provide remote process or service communication, location discovery, path management related services.	connections
Compiling, parsing and compression	3.66	These trivial packages help compilation, parsing and minification of JavaScript, CSS and html	koa-html-minifier
Concurrency control	2.30	These trivial packages provide asynchronous call managements, callback and promise related functionalities.	promise-all
Dependency management	2.25	These packages manage dependencies, loading preset of packages for the application.	static-reference
Performance optimization	2.14	The main functionalities of these trivial packages are caching, load management, benchmarking, manage cookies, watching memory, checking HTTP response freshness etc.	nocache
Event handle	1.72	These package provides smoother user interaction facilities.	add-event-handler
Project design	1.46	Scaffolding application structure, templating, generating default configurations are the functionalities that these packages provide.	hapi-dust
Wrapper	1.15	Packages in this category encapsulates some function, package or module so that access to these functional blocks can be generalized.	karma-es6-shim
Process and server management	0.68	Packages help to manage various process and handle different server like database servers.	server-destroy
Documentation	0.63	These packages helps generating and archiving various application related informations	source-map-to-comment

introducing activity as 'Other tasks'.

In the second part of this research question, we categorized trivial packages based on their functionalities. Table 4.3 shows the 17 category of functionalities that the trivial packages in our dataset render. Once again, for each category of used trivial packages, Table 4.3 shows the percentage of packages, a brief description, an example package for each category. Here we see that trivial packages provide a variety of functionalities. We observe that 'Utility', 'Datastructure, numerical and logical operation' and 'Build' are the categories that consist of more than have (51.79%) of packages in our trivial package dataset. Trivial packages in the 'Utility' category provide functionalities like string manipulation, stream operations, file operations, configuration and data conversion. In the 'Datastructure, numerical and logical operation' categories trivial packages facilitate various data structure and mathematical operations while 'Build' packages help to build the applications. 'Compiling, parsing and compression' consists only 3.66% packages in our dataset but these trivial packages are download most on average.

We observe that 'Modifying functionalities' and 'Building' are the top two activities that introduce trivial packages into software applications. On the other hand 'Utility', 'Datastructure, numerical and logical operation' , 'Build', 'Front-end development' and 'Extend or optimize functionality' are the top functionalities that trivial package provides. From the descriptions of the aforementioned categories of trivial introductory activity and functionalities of trivial packages, as well as from explicit mentioning of trivial package names in 64 instances of trivial introductory commits, it can be deduced that trivial packages play a significant role in the commits that introduce trivial packages into an application.

Observation - 'Modifying functionalities', 'Building' and 'Refactoring' are the three most frequent activities that introduce trivial packages in an application. In addition, our analysis shows that trivial packages provide developers with a wide range of functionalities but 'Utility', 'Datastructure, numerical and logical operation' and 'Build' are the most prominent categories as these categories constitute 53% of the packages.

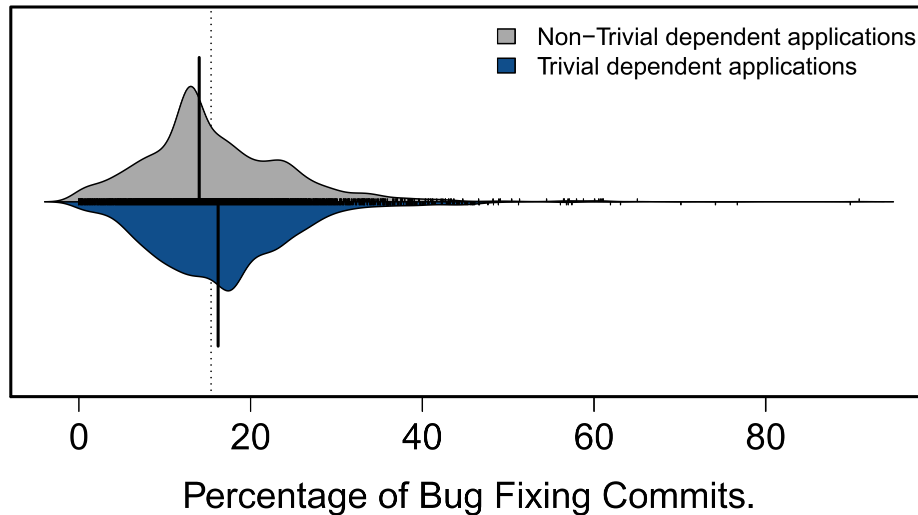

Figure 4.2: Distribution of percentage of bug-fixing commits in *trivial dependent* and *non-trivial dependent* applications. The solid horizontal lines represent the medians of the distribution. The dotted horizontal line is the overall median.

4.3.2 RQ1: Does using trivial JavaScript packages impact the overall quality of applications?

Motivation: Prior work showed trivial packages add dependency overhead and more than 50% of these packages have no test case written (Abdalkareem, Nourry, et al. (2017)). Therefore, we suspect that applications that depend on trivial packages may suffer from quality issues. Thus, in this research question, we investigate to determine the quality impact of trivial package dependencies in overall application-level granularity. We believe that answering this question will provide us with an overall glance at these package's impact on software quality.

Approach: To have a general overview of the quality of applications that depends on trivial packages, we compare these application's quality with that of the applications that do not have any trivial package dependency. We use the number of bug-fixing commits as a proxy for an application's quality, similar to previous studies (Kim et al. (2008); Śliwerski et al. (2005); Wehaibi et al. (2016)). We flag bug-fixing commits by leveraging a well-known approach that examines the appearance of pre-defined set of keywords that include "*bug*", "*fix*", "*defect*", "*patch*","*error*","*issue*", "*problem*", "*fail*", "*crash*" and their variants in commit messages (Abdalkareem, Shihab, and Rilling (2017); Eyolfson, Tan, and Lam (2011); Mockus and Votta (2000); Śliwerski et al. (2005)). We examine

the impact of trivial packages on the application level in a two-fold complementary approach. First, we compare the distribution of bug-fixing commits in applications that depend on trivial packages and applications that do not depend on trivial packages. Second, for each software application that has any trivial package dependency, we compare the distribution of bug-fixing commits before and after the first trivial introductory commit in that application. In both of these analyses, we use the percentage of bug-fixing commits instead of the raw number since these applications vary in the number of commits.

To understand the quality of a software application before and after a trivial package introduction, we identify the commit that introduces the first trivial package into a software application. This trivial package introducing commit is either the first commit in an application or before this commit, the application did not have any trivial package dependency. Since the applications in our dataset use `git` as the version control system, we iterate through the development history of a software application starting from the initial commit and analyze if that commit introduces any external packages of trivial nature. When we find such a commit, we break the iteration and register that commit as the first trivial package introducing commit for that application. For each trivial dependent application, this trivial introductory commit split the development period of a software project into two parts and we calculate and compare the percentage of bug-fixing commits for both segments.

Results: Figure 4.2 shows the distribution of bug-fixing commits in trivial dependent and non-trivial dependent applications. We observe that applications that are using trivial packages have a higher percentage of bug-fixing commits (median=16.22) than their counterparts (median=14.01). To examine whether the result is statistically significant, we applied the nonparametric Wilcoxon rank-sum test (*wilcox.test function* (2010)). We determine if the difference is statistically significant at p-value $<$ 0.05. We estimate the magnitude of the difference between datasets using Cliff's Delta (*cliff.delta function* (2010)) (or d). Cliff's Delta is a non-parametric effect size measure for ordinal data. We consider the effect size values: negligible for $|d| < 0.147$, small for $0.147 \leqslant |d| < 0.33$, medium for $0.33 \leqslant |d| < 0.474$ and large for $|d| \geqslant 0.474$. We found that the result is significant as p-value $<$ 5.479e-07, which is less than 0.05, with negligible effect size (Cliff's delta value is 0.0767).

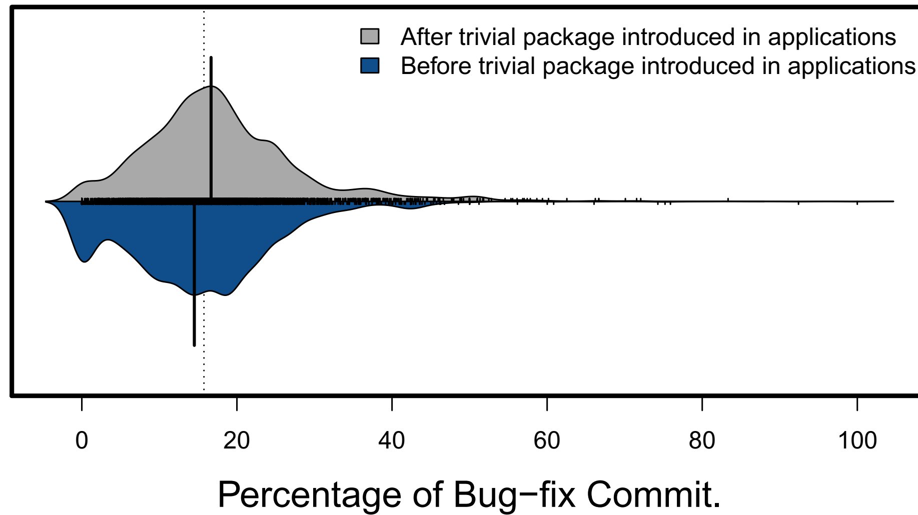

Figure 4.3: Distribution of percentage of bug-fixing commits before and after trivial package introduction in applications. The solid horizontal lines represent the medians of the distribution. The dotted horizontal line is the overall median.

We also examine the percentage of bug-fixing commit before and after introducing trivial packages to the studied applications. The beanplots in Figure 4.3 shows the bug-fixing commit distribution before and after trivial package introduction into the applications in our dataset. In our dataset, among 3,158 trivial dependent applications, 92.02% of the applications introduce trivial packages in a commit later than the initial commit of the applications and trivial packages were introduced with the initial commit in rest of the applications. Figure 4.3 shows the distribution for the applications that did not have a trivial dependency in their initial commit.

Here, we observe that the distribution of the percentage of bug-fixing commits is higher after the introduction of a trivial package in the trivial dependent applications. We see that the median of the percentage of bug-fixing commits before trivial package introduction in a trivial dependent application is 14.51 whereas after trivial introduction the value is 16.67. Once again, we examine whether the result is statistically significant. We found that the result is statistically significant (p-value $<$ 2.2e-16) although the effect size is small (cliff's delta = 0.1516).

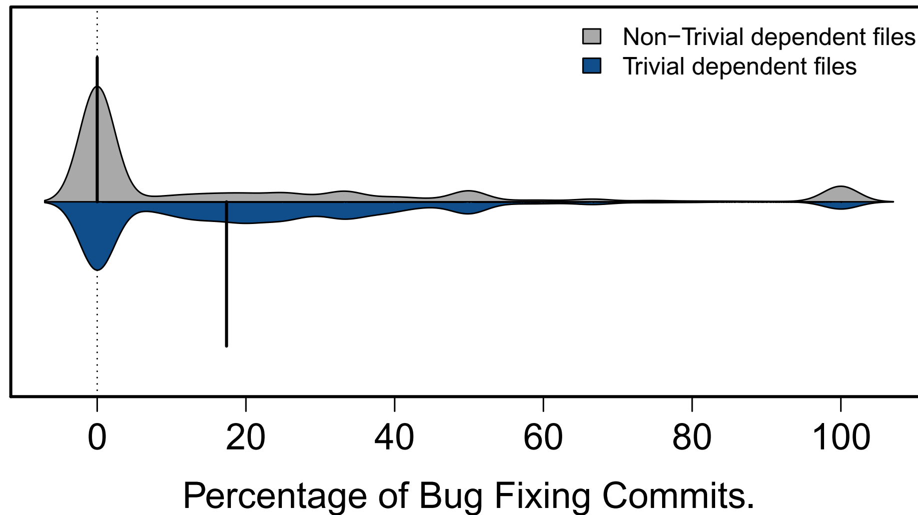

Figure 4.4: The distribution of percentage of bug-fixing commits in *trivial dependent* and *non-Trivial dependent* files. The solid horizontal lines represent the medians of the distribution. The dotted horizontal line is the overall median.

Observation - Our results show that the distribution of percentage of bug-fixing commits is higher in trivial dependent applications than in non-trivial dependent applications. Moreover, we observe that trivial dependent applications have higher bug-fixing commits after the introduction of trivial packages.

4.3.3 RQ2: What is the impact of trivial packages on the quality of the files?

Motivation: From the analysis in the overall application level, we observe that trivial package dependent applications have a higher percentage of bug-fixing commits. Now, we want to conduct a deeper analysis by examining the quality impact of trivial packages in JavaScript files. We examine JavaScript files since they are the building blocks of an application and trivial packages are instantiated and its APIs are invoked from within JavaScript files. Therefore, examination at the JavaScript files level provides us with more insight into the quality impact of trivial packages.

Approach: To answer this research question, we focus on analyzing the 3,158 applications that have at least one trivial dependent file. We analyze each dependency of each file. We flag files that use at least one trivial package as a trivial dependent file. We then analyze each commit of these applications and extract files that are being modified in the commit. Therefore, we know in a commit which files are being modified and if these files are trivial dependent files or not. Moreover, we

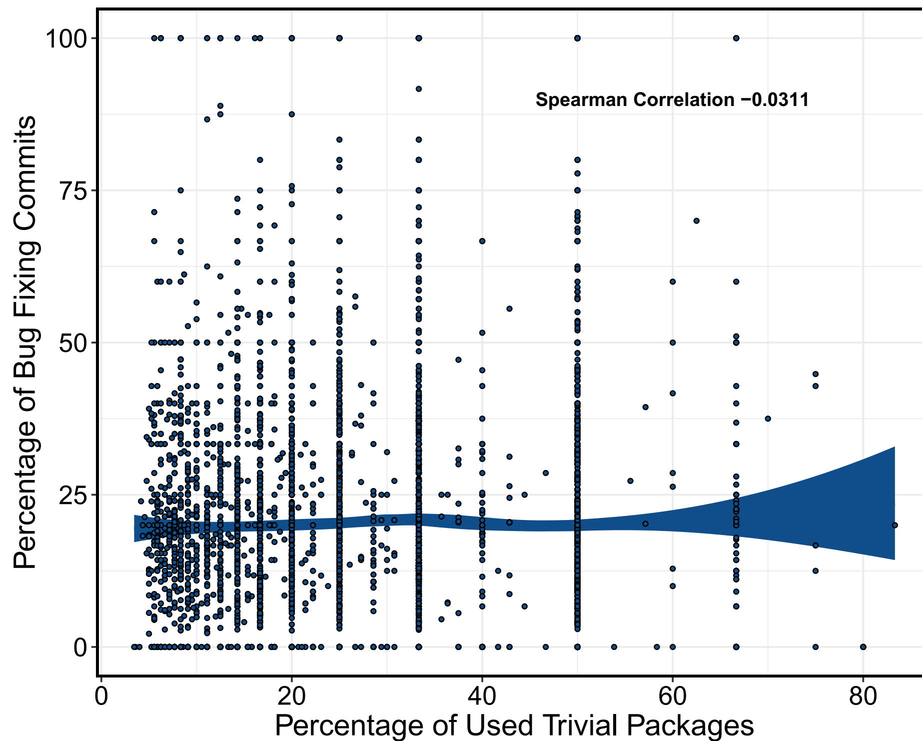

Figure 4.5: The correlation between the percentage of number of trivial package used and percentage of number of the bug-fixing commits in file.

analyze to check if the commit itself is bug-fixing commit or not. To understand the quality impact of trivial packages on the file level, we again conduct the two-fold complimentary analysis. First, we examine the distribution of bug-fixing commits on a file level. In trivial dependent applications, some files depend on trivial packages and some do not. We compare the distribution of the percentage of bug-fixing commits between these two types of files. Percentage of bug-fixing commits in a JavaScript file is calculated by accounting all the commits that touch the file and examining how many of these commits are bug-fixing. Additionally, in the trivial dependent files, we examine the relationship between the percentage of bug-fixing commits in the files and the percentage of trivial packages used in that file. We calculate the percentage of the trivial packages used by considering the total number of third *npm* packages used in the observed file and out of those packages how many are trivial. To determine the statistical measure of the strength of a linear relationship between the percentage of bug-fixing commits and the percentage of trivial packages used in files, we use Spearman rank correlation tests (p) (*Spearman function — R Documentation* (2010)). The reason

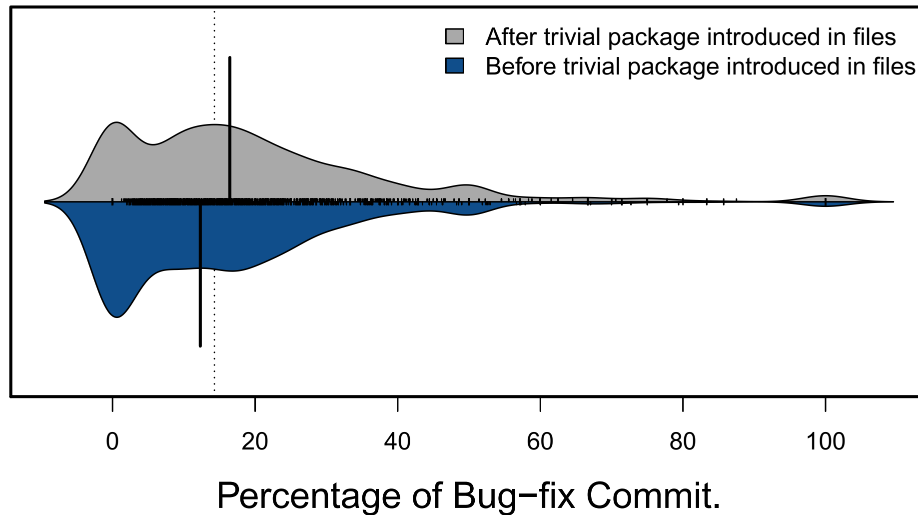

Figure 4.6: The distribution of bug-fixing commits before and after trivial package introduction in files that are converted from *non-trivial dependent* to *trivial dependent* in that commit. The solid horizontal lines represent the medians of the distribution. The dotted horizontal line is the overall median.

for using this rank correlation in preference to other types of correlation (e.g., Pearson, Kendall) since our data is not normally distributed (Bonett and Wright (2000); Fieller, Hartley, and Pearson (1957)). Second, we calculate the percentage of bug-fixing commits before and after a trivial package introduction in trivial dependent files. The first trivial package may be introduced to a file, as a dependency, in the commit that created the file or in a later commit that transforms the file from non-trivial package dependent to trivial package dependent. For the files that are created with a trivial package dependency, we analyze its percentage of the bug-fixing commit. On the other hand, if a file is transformed from non-trivial dependent to trivial dependent then we compare the percentage of bug-fixing commits before and after its trivial introductory commit.

Results: Figure 4.4 shows the comparison of the distribution of bug-fixing commits in trivial dependent and non-trivial dependent files. Here, we observe that the median value of the percentage of bug-fixing commits for trivial dependent files is 17.33 whereas the value for non-trivial dependent files is zero. We observe that the result is statistically significant(p-value = 2.2e-16) and the effect size is small (Cliff's Delta value is 0.3273). From this analysis, we observe that files using trivial packages are involved in more bug-fixing commits compared to files that do not depend on trivial packages. Interestingly, Figure 4.4 shows a small number of JavaScript files that have a large

percentage of bug-fixing commits. To understand this phenomenon, we examined the files having more than 50% of bug-fixing commits and found that these files consist of only 11.5% of files in our dataset and these files have a very low number of changes (median=3). Besides, we examine if there is a relation between the percentage of bug-fixing commits and the percentage of trivial packages used in trivial dependent files. Figure 4.5 shows the correlation between the percentage of bug-fixing commits and the percentage of trivial packages used in trivial dependent files in our dataset. Here we see that the value of Spearman rank correlation is -0.0311, which suggests a weak correlation, which potentially indicates that there is no correlation between percentage of used trivial packages and percenatge of bug-fixing commits in these files.

In the second part answering to this research question, we compare the percentage of bug-fixing commits before a trivial package being introduced in a trivial dependent file with that after a trivial package is introduced in that file. Since some trivial dependent files are created with trivial dependency and others are converted from non-trivial dependent files to trivial dependent files at some later commits, we first want to examine the distribution of these different files. In our dataset, 61.99% of trivial dependent files are created with a trivial package dependency and 31.01% files imported trivial dependency later in their development. Since we want to examine the quality of the JavaScript files before and after the use of trivial packages in term of percentage of bug-fixing commit, we focus on the 31.01% files imported trivial dependency later in their development. Figure 4.6 shows the distribution of bug-fixing commits in these files before and after they became trivial dependent files. From Figure 4.6, we see that, before trivial packages introduction into a trivial dependent file the median value of percentage of bug-fixing commits is 12.31% and after trivial introduction the value is 16.46%. We observe that the result is statistically significant(p-value $=$ 2.2e-16) with small effect size (Cliff's Delta value is 0.1473).

Observation - Our findings shows that files that depend on trivial packages tend to be more buggy compared to files that do not use trivial packages. In addition, we observe that JavaScript files have more bug-fixing after they depend on trivial packages.

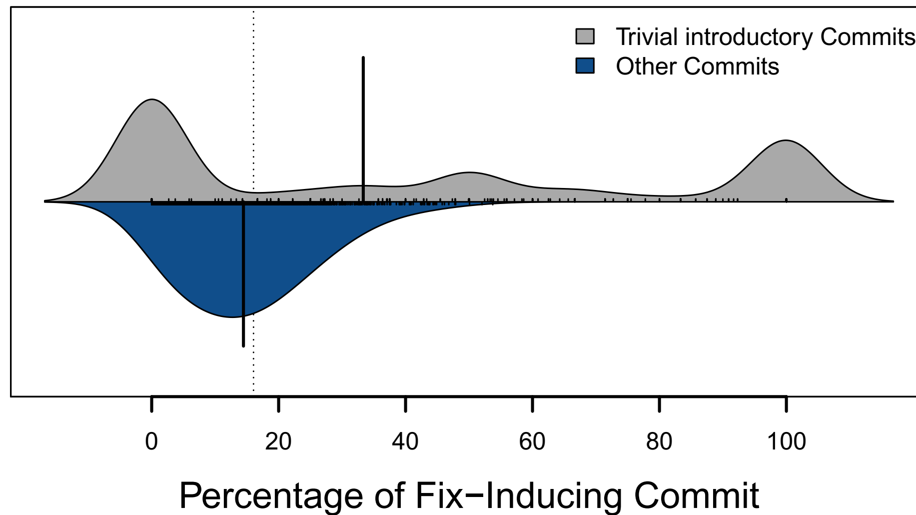

Figure 4.7: Distribution of percentage of fix-inducing commits in trivial introduction commits in files and other commits. The solid horizontal lines represent the medians of the distribution. The dotted horizontal line is the overall median.

4.3.4 RQ3: Are commits that introduce trivial packages in JavaScript files risky?

Motivation: Thus far, we saw that overall projects that use trivial packages tend to have a higher percentage of bug-fixing commits compare to project that do not use trivial packages. However, up to know To better understand the role of the these trivial packages in the introduction of bugs into the studied applications, we examine whether commits that introduce trivial package are the ones that introduce the bug to the projects. In addition, we want to examine which the type of trivial packages are introduced in these commits .

Approach: To examine whether the trivial introducing commits are risky changes, first, we identify the commits that introduce trivial packages into a trivial dependent file. To do that, we analyzed the development history of trivial dependent files and register the commit that introduces first trivial packages into a file. Furthermore, we categorize these commits based on the type of trivial package they introduce. Then we utilize the well-known SZZ algorithm (Williams and Spacco (2008)) to analyze the development history of the applications and identify the commits that are fix-inducing (Misirli, Shihab, and Kamei (2016); Shihab et al. (2012); Śliwerski et al. (2005)). The main goal of the SZZ algorithm is to detect the commit that induces future fixes in software applications. This algorithm identifies the commit that fixes a defect and then, it detects the lines that

Table 4.4: The percentage of fix-inducing commits for each of the different type of functionalist that trivial packages provide. *The percentage of fix-inducing commits all commits for each of the different type of functionalities. **The percentage of fix-inducing commits for each type to all fix-inducing commits related to trivial packages.

Category	#Fix-inducing commits	*% to trivial inducing	**% to All Fix-inducing
Utility	789	47.13	37.04
Data structure, numerical and logical operation	207	47.70	9.72
Error handle, debug and testing	168	45.16	7.89
Project design	137	57.56	6.43
Compiling, parsing and compression	121	41.02	5.68
Dependency management	120	31.33	5.63
Extend or optimize functionality	117	45.17	5.49
Communication, networking and RPC	83	36.89	3.90
Concurrency control	78	36.79	3.66
Security	75	40.54	3.52
Front-end development	72	36.18	3.38
Build	61	32.33	2.86
Performance optimize	49	46.23	2.30
Wrapper	26	54.17	1.22
Event handle	19	50.00	0.89
Process and server management	5	35.71	0.23
Documentation	3	30.00	0.14

are modified by that fixing commit. Finally, it tracks back the development history to identify the commit in which the line was changed and identify this commit as a fix-inducing commit. To extract fix-inducing commits, we use the SZZ implementation provided by the CommitGuru tool (Rosen, Grawi, and Shihab (2015)). Since analyzing all the applications in our dataset using the SZZ algorithm is prohibitively expensive, we run the techniques on a randomly selected sample of 1,364 applications from the 3,158 trivial dependent applications in our dataset. This sample represents a statistically significant sample with 95% confidence level and confidence interval of 2. Once we get the list of fix-inducing commits and trivial inducing commits for an application, we crosscheck them and flag the commits that introduce trivial packages and are fixing-inducing. We consider these commits as risky trivial introducing commits. To put our analysis in perspective, we compare riskiness of trivial introducing commits with that of all other commits in trivial dependent files. Besides, we categorize trivial introducing commits based on the type of trivial packages they introduce and compare riskiness in these commit categories.

Results: Figure 4.7 shows the percentage of risky commits that introduce trivial packages in trivial dependent files. We observe that trivial inducing commits tend to be more risker commits compared to other commits in the trivial dependent files. Figure 4.7 shows that median value for percentage of risky changes for the commits that introduce trivial packages is 33.33% whereas for all other commits that touch trivial dependent files this value is 14.46%. We found that this result is statistically significant (p-value = 1.808e-07) with negligible effect size (d = -0.1209313).

We categorize trivial package introduction commits based on what type of trivial package they introduce and examine risky commits for each of these categories. Table 4.4 shows the percentage of risky commits for different types of trivial package introduction commits. Here, the second column shows the number of risky trivial introduction commits for each type of trivial package introductory commit categories whereas the third column shows what percentage of these commits are risky (fix-inducing). The fourth column shows what percentage of the total risky commit is associated with each category of trivial introduction commits.

From table 4.4, we observe that trivial packages in the Utility category are the largest contributors in term of risky commits, almost one-third of fix-inducing commits are attributed to this category. Moreover, we observe that roughly one out of two commits that introduce trivial packages in the "Project design", "Wrapper", "Error handle", "Datastructure, numeric and logical operation" and "Utility" categories are risky.

Observation - Commits that introduce trivial packages in JavaScript files are more risky compared to other commits. Also, our result shows that on median 33.33% of the commits that introduce trivial packages requires future fixes.

4.4 Threats to validity

In this section, we discuss the threats of validity related to our study.

4.4.1 Internal validity

Internal validity concerns factors that could have influenced our results. Our study heavily depends on the use of *Depchecker* tool (*depcheck-npm* (2013)) that extracts the used dependencies in the studied JavaScript applications. Thus, we are limited to the accuracy of the *Depchecker* tool. To mitigate this threat, we randomly selected 10 applications from our dataset and manually examine the output of the tool. We found that in all the examined applications the *Depchecker* tool correctly identified the used dependencies. To investigate the type of functionalities trivial packages provide, the two authors perform a manual examination of all the used trivial packages in our study. This process is subject to human bias and to mitigate this threat, the first two of the authors separately classify trivial packages and measure the inter-agreement between them. We found their agreement to be excellent (Cohen's Kappa value of 0.75).

To examine the impact of using trivial packages in an application, we use bug-fixing commits as a proxy for application quality. It can be debated if bug-fixing can be used as a proxy for quality, however, several prior studies adopted this approach (Abdalkareem, Shihab, and Rilling (2017); Foucault et al. (2015); Kim et al. (2008); McIntosh et al. (2014); Śliwerski et al. (2005); Wehaibi et al. (2016)). Bug-fixing commits are more certain and can be pointed out specifically with minimum error. Prior studies suggest that the automatic extraction process of bug-fixing commits introduces some errors (Bird et al. (2009); Bissyande et al. (2013); Herzig, Just, and Zeller (2013)) but as our dataset is fairly big and have rich development history thus manually extract bug-fixing commit is not possible. To mitigate this threat, we first adopted well-accepted methods to automatically select bug-fixing commits based on keywords that is used in prior work (Abdalkareem, Shihab, and Rilling (2017); Eyolfson et al. (2011); Mockus and Votta (2000); Śliwerski et al. (2005)). Second, we took a statistical sample of 384 bug-fixing commits with 95% confidence level and confidence interval of 5. We then manually analyzed each of the selected identified bug-fixing commits and found that there is only 2.84% of these commits that may not introduce bug-fixes.

4.4.2 External validity

Threats to external validity concern the generalization of our findings. In this study, we focus on JavaScript packages published on *npm*, which is one of the most popular and largest package managers for developers; hence, our results may not generalize to other package manager platforms. To alleviate this threat, more suggest that another package manager should be studied in the future. The examined applications in our study present only open-source applications hosted on GitHub that do not reflect proprietary applications and applications from another hosting platform such as GitLab.

4.5 Chapter Summary

In this chapter, we investigate the impact of using trivial packages on the quality of JavaScript applications that depend on them. We conduct an empirical study through analyzing more than 3,000 open-source JavaScript applications that use at least one trivial *npm* package. We start first by examining the type of development activities that introduce the trivial packages to the studied applications and the functionalities that trivial packages provide. We find that modifying functionalities, building, and refactoring are the most development activities related to the use of trivial packages. We observe that trivial package provides variate of functionalities such as utility and data structure, numeral and logical operation.

Second, we investigate the impact of using trivial packages on the file- and application-level in terms of the percentage of bug-fixing commits. We find that files and applications that use trivial packages tend to have a higher percentage of the bug-fixing commit. Finally, we examine the riskiness of the commits that introduce trivial packages into JavaScript files in our studied JavaScript applications. We notice that the commits that introduce trivial packages in JavaScript files are riskier compared to other commits. Also, our result shows that on median 33.33% of the commits that introduce trivial packages require future fixes.

Chapter 5

Summary, Contributions and Future Work

This chapter concludes the thesis. We present a summary of the results presented throughout this thesis. Then, we discuss possible directions for future work.

5.1 Summary of findings

This thesis focuses on trivial packages not as standalone units but assesses their contribution as building blocks in software applications. In this thesis, first, we conduct an empirical study to understand how these packages are used in software applications and evaluate their relative importance. Then, we evaluate the impact of using trivial packages on software quality. The following are the summaries of this thesis chapters.

Chapter 3 examines the use of trivial packages in software applications. In this chapter, we examine the relative importance of trivial packages. In our empirical analysis, we observed that: 1) trivial packages are used in important parts of a software application. 2) these packages are very important within JavaScript files as a significant percentage of API calls in these files are attributed to trivial packages. 3) trivial packages are also vital in the package dependency network. The packages dependency network is composed of direct and transitive dependencies of software applications. Therefore, our overall analysis shows that these packages are vital building blocks in

modern software development.

Chapter 4 assesses the impact of using trivial packages on software quality. From the previous chapter, we observe that trivial packages are vital and are used in important parts of software applications. Moreover, from the previous study by Abdalkareem et al. (Abdalkareem (2017); Abdalkareem, Nourry, et al. (2017)) it is evident that more than 50% of trivial packages do not have any test case. Therefore, in this chapter, we looked into the quality impact of trivial packages in software applications. Additionally, we qualitatively and quantitatively analyze the commits that introduce trivial packages in software applications in order to fathom the development scenario that ships these small packages into these matured software applications. Our analysis shows that: 1) trivial packages are introduced in software applications while developers are doing various development activities like "Building", "Refactoring", "Improving Performance" etc. 2) Software applications and files that depend on trivial packages are significantly buggier than the applications and files that do not depend on trivial package. 3) We found that commits that introduce these packages in JavaScript files are significantly riskier than other commits as they often induce future fixes.

5.2 Contribution

The contributions of this thesis are as follows:

- We provide a novel approach to evaluate how important trivial *npm* packages are by extensively analyzing their usage from applications and ecosystems perspective.
- We formulate various metrics to understand the importance of a package in a dependency network of the *npm* ecosystem.
- We provide an extensive insight on the development activities that introduce trivial packages in to an application and the functionalities trivial packages provide.
- We conducted a large scale empirical study to examine the quality impact of using trivial packages in JavaScript applications.

5.3 Future Work

We believe that our thesis makes a positive contribution towards the goal of understanding the use of trivial packages in software applications. However, there are still many open challenges that need to be tackled to improve the development practice of using trivial packages. We now highlight some avenues for future work.

5.3.1 Detecting Trivial Packages That Provide Similar Functionalities:

During our studies and throughout our manual examination of trivial *npm* packages, we observed that there are several trivial packages in the *npm* ecosystem that provide similar functionality. For example, filereader, file-reader, and @tanker/file-reader packages offer the functionality of reading files from the browser. We believe developing an automated technique to identify these trivial packages would increase the maintainability of the *npm* ecosystem and improve overall quality.

5.3.2 Generate Automated Test Cases for the Packages:

The previous study by Abdalkareem *et al.* (Abdalkareem, Nourry, et al. (2017)) points out that more than 50% of the trivial packages do not have any test case. Our analysis of the top 1000 most depended on trivial packages also reveals that some of these packages have poor test score. Techniques should be developed to generate automated tests so that the test score for those trivial packages can be increased and thus reduce the chances of packages breakage.

5.3.3 Automate the Evaluation of Ecosystem Health:

In this thesis, we observed that some packages are vital for the stability of the *npm* ecosystem as a large number of other packages depend on them. We preliminarily evaluated the top 1000 most depended on trivial packages but they have health issues like less test coverage, use outdated or vulnerable packages as their dependency, less maintained, etc. Scrutiny and improving the health of the packages that are vital for the *npm* ecosystem should be of high priority.

5.3.4 Automatically Generate Smaller Packages:

During our studies, we observed that JavaScript developers proclaim that trivial packages are analogous to "Lego Blocks", where large complex systems can be built without having to know every single detail of how everything works (Sorhus (2018)). There are some popular packages in *npm* that are very well accepted and of high quality such as react, babel. However, these packages are huge in size and provide so many functionalities that developers of an applications may not need all their functionalities. We believe that one future work and be done to analyze those packages and develop an automatic technique to segment those packages into smaller more usable packages.

References

Abdalkareem, R. (2017). Reasons and drawbacks of using trivial npm packages: The developers' perspective. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering* (pp. 1062–1064). ACM.

Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., & Shihab, E. (2017). Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering* (pp. 385–395). ACM.

Abdalkareem, R., Oda, V., Mujahid, S., & Shihab, E. (2019, may). *On the Impact of Using Trivial Packages: An Empirical Case Study on npm and PyPI.* Zenodo.

Abdalkareem, R., Shihab, E., & Rilling, J. (2017). On code reuse from stackoverflow : An exploratory study on android apps. *Information and Software Technology*, *88*(C), 148–158.

Aric Hagberg, D. S., & Swart, P. (2005, July). *Networkx - network graph analysis.* https://networkx.github.io/. ((Accessed on 02/17/2019))

Basili, V. R., Briand, L. C., & Melo, W. L. (1996, October). How reuse influences productivity in object-oriented systems. *Communications of the ACM*, *39*(10), 104–116.

Bavota, G., Canfora, G., Penta, M. D., Oliveto, R., & Panichella, S. (2013). The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *Proceedings of the 2013 ieee international conference on software maintenance* (pp. 280–289). IEEE Computer Society.

Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., & Devanbu, P. (2009). Fair and balanced?: Bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the european software engineering conference and the acm sigsoft symposium on the foundations of*

software engineering (pp. 121–130). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1595696.1595716 doi: 10.1145/1595696.1595716

Bissyande, T. F., Thung, F., Wang, S., Lo, D., Jiang, L., & Reveillere, L. (2013). Empirical evaluation of bug linking. In *Proceedings of the 2013 17th european conference on software maintenance and reengineering* (pp. 89–98). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dx.doi.org/10.1109/CSMR.2013.19 doi: 10.1109/CSMR.2013.19

Bloemen, R., Amrit, C., Kuhlmann, S., & Ordóñez Matamoros, G. (2014). Gentoo package dependencies over time. In *Proceedings of the 11th working conference on mining software repositories* (pp. 404–407). ACM.

Bonett, D. G., & Wright, T. A. (2000, Mar 01). Sample size requirements for estimating pearson, kendall and spearman correlations. *Psychometrika*, *65*(1), 23–28. Retrieved from https://doi.org/10.1007/BF02294183 doi: 10.1007/BF02294183

Brin, S., & Page, L. (1998). *The anatomy of a large-scale hypertextual web search engine* (Vol. 30) (No. 1-7). Elsevier.

Cadini, F., Zio, E., & Petrescu, C.-A. (2009). Using centrality measures to rank the importance of the components of a complex network infrastructure. In *Proceedings of the critical information infrastructure security* (pp. 155–167). Springer Berlin Heidelberg.

cliff.delta function. (2010). https://www.rdocumentation.org/packages/effsize/versions/0.6.4/topics/cliff.delta. ((Accessed on 06/05/2019))

Cox, R. (2019, August). Surviving software dependencies. *Commun. ACM*, *62*(9), 36–43.

da Costa, D. A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., & Hassan, A. E. (2017, July). A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, *43*(7), 641-657.

Decan, A., & Mens, T. (2019). What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*, 1-1. doi: 10.1109/TSE.2019.2918315

Decan, A., Mens, T., Claes, M., & Grosjean, P. (2016). When github meets cran: An analysis of inter-repository package dependency problems. In *2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner)* (Vol. 1, p. 493-504). IEEE Computer

Society.

Decan, A., Mens, T., & Constantinou, E. (2018, May). On the impact of security vulnerabilities in the npm package dependency network. In *2018 ieee/acm 15th international conference on mining software repositories (msr)* (p. 181-191).

Decan, A., Mens, T., & Grosjean, P. (2019). An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Softw. Engg.*, *24*(1), 381–416.

depcheck-npm. (2013). https://www.npmjs.com/package/depcheck. ((Accessed on 06/05/2019))

Eyolfson, J., Tan, L., & Lam, P. (2011). Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th working conference on mining software repositories* (pp. 153–162). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1985441.1985464 doi: 10.1145/1985441.1985464

Fard, A. M., & Mesbah, A. (2017). Javascript: The (un)covered parts. *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 230-240.

Feldthaus, A., Schfer, M., Sridharan, M., Dolby, J., & Tip, F. (2013). Efficient construction of approximate call graphs for javascript ide services. In *Proceedings of the 2013 35th international conference on software engineering (icse)* (p. 752-761). ACM.

Fieller, E. C., Hartley, H. O., & Pearson, E. S. (1957). Tests for rank correlation coefficients. i. *Biometrika*, *44*(3/4), 470–481. Retrieved from http://www.jstor.org/stable/2332878

Fleiss, J., Levin, B., & Paik, M. (2013). *Statistical methods for rates and proportions*. Wiley. Retrieved from https://books.google.ca/books?id=9VefO7a8GeAC

Fleiss, J. L., & Cohen, J. (1973). The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability. *Educational and psychological measurement*, *33*(3), 613–619.

Foucault, M., Palyart, M., Blanc, X., Murphy, G. C., & Falleri, J.-R. (2015). Impact of developer turnover on quality in open-source software. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering* (pp. 829–841). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2786805.2786870 doi: 10.1145/2786805

.2786870

Freeman, L. C. (1978). Centrality in social networks conceptual clarification. *Social networks*, *1*(3), 215–239.

German, D. M., Adams, B., & Hassan, A. E. (2013). The evolution of the r software ecosystem. In *Proceedings of the 2013 17th european conference on software maintenance and reengineering* (p. 243-252). IEEE.

Godfrey, & Qiang Tu. (2000, Oct). Evolution in open source software: a case study. In *Proceedings 2000 international conference on software maintenance* (p. 131-142). IEEE.

Gousios, G. (2013). The ghtorrent dataset and tool suite. In *Proceedings of the 10th working conference on mining software repositories* (pp. 233–236). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2487085.2487132

Gruber, J. (2013, May). *isarray-npm.* https://www.npmjs.com/package/isarray. ((Accessed on 07/24/2019))

Haefliger, S., von Krogh, G., & Spaeth, S. (2008, January). Code reuse in open source software. *Manage. Sci.*, *54*(1), 180–193. Retrieved from http://dx.doi.org/10.1287/mnsc.1070.0748 doi: 10.1287/mnsc.1070.0748

Haenni, N., Lungu, M., Schwarz, N., & Nierstrasz, O. (2013). Categorizing developer information needs in software ecosystems. In *Proceedings of the 2013 international workshop on ecosystem architectures* (pp. 1–5). ACM.

Haney, D. (2016, March). *Npm & left-pad: Have we forgotten how to program? - blogging my experiences as a developer and engineering manager.* https://www.davidhaney.io/npm-left-pad-have-we-forgotten-how-to-program/. ((Accessed on 09/30/2019))

Hassan, A. E. (2009, May). Predicting faults using the complexity of code changes. In *Proceedings of the 2009 ieee 31st international conference on software engineering* (p. 78-88).

Henningsson, P. (2014, April). *madge-npm.* https://www.npmjs.com/package/madge. ((Accessed on 06/05/2019))

Herzig, K., Just, S., & Zeller, A. (2013). It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 international conference*

on software engineering (pp. 392–401). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2486788.2486840

Holmes, R., & Walker, R. J. (2007). Informing eclipse api production and consumption. In *Proceedings of the 2007 oopsla workshop on eclipse technology exchange* (pp. 70–74). ACM.

Inoue, K., Sasaki, Y., Xia, P., & Manabe, Y. (2012). Where does this code come from and where does it go? - integrated code history tracker for open source systems -. In *Proceedings of the 34th international conference on software engineering* (pp. 331–341). IEEE Press.

Kabbedijk, J., & Jansen, S. (2011). Steering insight: An exploration of the ruby software ecosystem. In *International conference of software business* (pp. 44–55).

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., & Damian, D. (2014). The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories* (pp. 92–101). ACM.

Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., & Ubayashi, N. (2013, June). A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, *39*(6), 757-773.

Kikas, R., Gousios, G., Dumas, M., & Pfahl, D. (2017). Structure and evolution of package dependency networks. In *Proceedings of the 14th international conference on mining software repositories* (pp. 102–112). IEEE Press.

Kim, S., Whitehead, E. J., Jr., & Zhang, Y. (2008, March). Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, *34*(2), 181–196.

Kim, S., Zimmermann, T., Pan, K., & Jr. Whitehead, E. J. (2006, Sep.). Automatic identification of bug-introducing changes. In *21st ieee/acm international conference on automated software engineering (ase'06)* (p. 81-90). doi: 10.1109/ASE.2006.23

Kula, R. G., Germán, D. M., Ouni, A., Ishio, T., & Inoue, K. (2018). Do developers update their library dependencies? - an empirical study on the impact of security advisories on library migration. *Empirical Software Engineering*, *23*(1), 384–417. Retrieved from https://doi.org/10.1007/s10664-017-9521-5 doi: 10.1007/s10664-017-9521-5

Kula, R. G., Ouni, A., Germán, D. M., & Inoue, K. (2017). On the impact of micro-packages: An empirical study of the npm javascript ecosystem. *CoRR*, *abs/1709.04638*. Retrieved from

http://arxiv.org/abs/1709.04638

Kula, R. G., Ouni, A., German, D. M., & Inoue, K. (2017). *On the impact of micro-packages: An empirical study of the npm javascript ecosystem.*

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, *68*(9), 1060-1076.

Lertwittayatrai, N., Kula, R. G., Onoue, S., Hata, H., Rungsawang, A., Leelaprute, P., & Matsumoto, K. (2017, Dec). Extracting insights from the topology of the javascript package ecosystem. In *2017 24th asia-pacific software engineering conference (apsec)* (p. 298-307).

Lim, W. C. (1994). Effects of reuse on quality, productivity, and economics. *IEEE Software*, *11*, 23-30.

Macdomald, F. (2016, March). *A programmer almost broke the internet last week by deleting 11 lines of code - sciencealert.* http://www.sciencealert.com/how-a-programmer-almost-broke-the-internet-by-deleting-11-lines-of-code. ((accessed on 06/03/2016))

Manikas, K. (2016, jul). Revisiting software ecosystems research: A longitudinal literature study. *Journal of Systems and Software*, *117*, 84–103.

McCamant, S., & Ernst, M. D. (2003). Predicting problems caused by component upgrades. In *Proceedings of the 9th european software engineering conference held jointly with 11th acm sigsoft international symposium on foundations of software engineering* (pp. 287–296). ACM.

McIntosh, S., Kamei, Y., Adams, B., & Hassan, A. E. (2014). The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th working conference on mining software repositories* (pp. 192–201). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2597073.2597076 doi: 10.1145/2597073.2597076

McIntosh, S., Kamei, Y., Adams, B., & Hassan, A. E. (2016). An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. *Empirical Software Engineering*, *21*(5), 2146-2189.

Mens, T. (2016). An ecosystemic and socio-technical view on software maintenance and evolution. In *Proceedings of the 2016 ieee international conference on software maintenance and evolution (icsme)* (Vol. 00, p. 1-8). IEEE.

Metcalf, C. (2015, May). *process-nextick-args - npm*. https://www.npmjs.com/package/process-nextick-args. ((Accessed on 07/24/2019))

Mileva, Y. M., Dallmeier, V., & Zeller, A. (2010). Mining api popularity. In *Proceedings of the 5th international academic and industrial conference on testing - practice and research techniques* (pp. 173–180). Springer-Verlag.

Misirli, A. T., Shihab, E., & Kamei, Y. (2016, Apr 01). Studying high impact fix-inducing changes. *Empirical Software Engineering*, *21*(2), 605–641. Retrieved from https://doi.org/10.1007/s10664-015-9370-z doi: 10.1007/s10664-015-9370-z

Mizuno, O., & Hata, H. (2013). A metric to detect fault-prone software modules using text filtering. *International Journal of Reliability and Safety*, *7*(1), 17-31.

Mockus, A. (2007). Large-scale code reuse in open source software. In *Proceedings of the first international workshop on emerging trends in floss research and development* (pp. 7–). IEEE Computer Society.

Mockus, A., & Votta, L. G. (2000). Identifying reasons for software changes using historic databases. In *Proceedings of the international conference on software maintenance (icsm'00)* (pp. 120–). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=850948.853410

Mohagheghi, P., Conradi, R., Killi, O. M., & Schwarz, H. (2004). An empirical study of software reuse vs. defect-density and stability. In *Proceedings of the 26th international conference on software engineering* (pp. 282–292). IEEE Computer Society.

Mujahid, S., Sierra, G., Abdalkareem, R., Shihab, E., & Shang, W. (2018, Dec 01). An empirical study of android wear user complaints. *Empirical Software Engineering*, *23*(6), 3476–3502. Retrieved from https://doi.org/10.1007/s10664-018-9615-8 doi: 10.1007/s10664-018-9615-8

Murphy-Hill, E., Jaspan, C., Sadowski, C., Shepherd, D., Phillips, M., Winter, C., … Jorde, M. (2019). What predicts software developers' productivity? *IEEE Transactions on Software Engineering*.

npm-ls. (2010). https://docs.npmjs.com/cli/ls.html. ((Accessed on 04/21/2019))

npm-pack. (2009). https://docs.npmjs.com/cli/pack/. ((Accessed on 06/05/2019))

npm-registry. (2009). https://docs.npmjs.com/misc/registry/. ((Accessed on 06/05/2019))

npm search. (2018, July). *escape-string-regexp - npm.* https://www.npmjs.com/package/escape-string-regexp. ((accessed on 10/02/2019))

npms-ponyfill. (2009). https://npms.io/search?q=keywords%3Aponyfill. ((Accessed on 07/05/2019))

Qi, X., Fuller, E., Wu, Q., Wu, Y., & Zhang, C.-Q. (2012). Laplacian centrality: A new centrality measure for weighted networks. *Information Sciences*, *194*, 240 - 253.

Rosen, C., Grawi, B., & Shihab, E. (2015). Commit guru: Analytics and risk prediction of software commits. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering* (pp. 966–969). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2786805.2803183 doi: 10.1145/2786805.2803183

Scitools.com. (1996). https://scitools.com/. ((Accessed on 06/05/2019))

Seaman, C. B. (1999, 07). Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, *25*, 557-572. Retrieved from doi.ieeecomputersociety.org/10.1109/32.799955 doi: 10.1109/32.799955

Shihab, E., Hassan, A. E., Adams, B., & Jiang, Z. M. (2012). An industrial study on the risk of software changes. In *Proceedings of the acm sigsoft 20th international symposium on the foundations of software engineering* (pp. 62:1–62:11). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2393596.2393670 doi: 10.1145/2393596.2393670

Śliwerski, J., Zimmermann, T., & Zeller, A. (2005). When do changes induce fixes? In *Acm sigsoft software engineering notes* (Vol. 30, pp. 1–5).

Sorhus, S. (2013). *npm.* https://www.npmjs.com/~sindresorhus. ((Accessed on 07/05/2019))

Sorhus, S. (2014, July). *sindresorhus/awesome: awesome lists about all kinds of interesting topics.* https://github.com/sindresorhus/awesome#readme. ((Accessed on 07/05/2019))

Sorhus, S. (2016, September). *like polyfill but with pony pureness.* https://github.com/

sindresorhus/ponyfill. ((Accessed on 07/05/2019))

Sorhus, S. (2018, October). *Small focused modules.* https://blog.sindresorhus.com/small-focused-modules-9238d977a92a. ((Accessed on 10/15/2019))

Spearman function — r documentation. (2010). https://www.rdocumentation.org/packages/SuppDists/versions/1.1-9.4/topics/Spearman. ((Accessed on 06/05/2019))

Wagner, S., & Murphy-Hill, E. (2019). Factors that influence productivity: A checklist. In C. Sadowski & T. Zimmermann (Eds.), *Rethinking productivity in software engineering* (pp. 69–84). Berkeley, CA.

Wehaibi, S., Shihab, E., & Guerrouj, L. (2016, March). Examining the impact of self-admitted technical debt on software quality. In *2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner)* (Vol. 1, p. 179-188). IEEE.

White, S., & Smyth, P. (2003). Algorithms for estimating relative importance in networks. In *Proceedings of the ninth acm sigkdd international conference on knowledge discovery and data mining* (pp. 266–275). ACM.

wilcox.test function. (2010). https://www.rdocumentation.org/packages/stats/versions/3.5.1/topics/wilcox.test. ((Accessed on 06/05/2019))

Williams, C., & Spacco, J. (2008). Szz revisited: Verifying when changes induce fixes. In *Proceedings of the 2008 workshop on defects in large software systems* (pp. 32–36). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1390817.1390826 doi: 10.1145/1390817.1390826

Wittern, E., Suter, P., & Rajagopalan, S. (2016, May). A look at the dynamics of the javascript package ecosystem. In *2016 ieee/acm 13th working conference on mining software repositories (msr)* (p. 351-361). doi: 10.1109/MSR.2016.043

Xie, G., Chen, J., & Neamtiu, I. (2009, Sep.). Towards a better understanding of software evolution: An empirical study on open source software. In *2009 ieee international conference on software maintenance* (p. 51-60).

Xu, B., An, L., Thung, F., Khomh, F., & Lo, D. (2019, Sep 05). Why reinventing the wheels? an empirical study on library reuse and re-implementation. *Empirical Software Engineering*.

Yeoman. (2012). https://yeoman.io/. ((Accessed on 07/05/2019))

Zapata, R. E., Kula, R. G., Chinthanet, B., Ishio, T., Matsumoto, K., & Ihara, A. (2018). Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages. In *Proceedings of the 2018 ieee international conference on software maintenance and evolution (icsme)* (p. 559-563).

Zimmermann, M., Staicu, C., Tenny, C., & Pradel, M. (2019). Small world with high risks: A study of security threats in the npm ecosystem. *CoRR*, *abs/1902.09217*.