Study on Reuse Issues in Open Source Software

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2021

Shi QIU

Abstract

Open source software (OSS) is software whose source code can be reused under some particular terms and conditions. Reusing OSS in software development has long been proved to be a good method to increase software productivity. Although OSS reuse can help developers create software products more efficiently, it also comes with some reuse issues needed to pay attention to. One important aspect is software license, without which under what conditions OSS can be redistributed, reused, and modified can not be understood correctly. Another aspect is software copyright, without which the copyright owner who owns the right to determine these conditions can not be identified clearly. Furthermore, understanding the popularity growth of OSS is also important for maintenance and reuse.

In the first part of this dissertation, we deal with the issue of software copyright. Software copyright claims an exclusive right for the software copyright owner to determine whether and under what conditions others can modify, reuse, or redistribute this software. For OSS, it is very important to identify the copyright owner who can control those activities with license compliance. In this part, as the first step of understanding copyright owner, we will explore the situation of the software copyright in the Linux kernel, a typical example of OSS, by analyzing and comparing two kinds of datasets, copyright notices in source files and source code contributors in the software repositories. The discrepancy between the two kinds of analysis results is defined as copyright inconsistency. The analysis result has indicated that copyright inconsistencies are prevalent in the Linux kernel. We have also found that code reuse, affiliation change, refactoring, support function, and others' contributions potentially have impacts on the occurrence of the copyright inconsistencies in the Linux kernel. This study exposes the difficulty in managing software copyright in OSS, highlighting the usefulness of future work to address software copyright problems.

In the second part of this dissertation, we deal with the difficulty of copyright notice identification of source files. For OSS, identifying copyright notices is important. The copyright owner is allowed to change its license or to grant a commercial one to a third party and start legal proceedings to enforce its license. Several licenses also require that the copyright owner

of OSS projects being reused should be recorded explicitly. However, both the collaborative manner of OSS project development and the large number of source files increase its difficulty. In this part, we aim at automatically identifying the copyright notices in source files based on machine learning techniques. The evaluation experiment shows that our method outperforms FOSSology, the only existing method based on the regular expression.

In the third part of this dissertation, we deal with the issue of software license. Software license is a written text that claims under what conditions others can modify, reuse, or redistribute the software. A license may violate another one according to the terms and conditions. Making software by reusing OSS as dependency may cause dependency-related license violation if the developers overlook the license of the dependency. In this part, we first conduct an empirical study on `npm` - a JavaScript-based software ecosystem - to study the prevalence of dependency-related license violations. The result suggests that only a few packages (0.644%) in `npm` have dependency-related license violations. However, we also observe that including the packages licensed under copyleft licenses in the dependency network highly potentially causes dependency-related license violations. We then conduct a preliminary questionnaire on the authors of packages detected as having dependency-related license violations to study the developers' attitudes. The results reveal: 1) the developers' overlooking and misunderstanding of the dependency-related license violations; 2) the difficulties in managing dependency-related license violations and the developers' demands for help.

In the fourth part of this dissertation, we deal with the issue of software popularity. In OSS ecosystems, software popularity is valuable information to developers because they continually want to know whether their software is attracting and gaining acceptance. Software popularity is also an important indicator to suggest if a software is beating its competitors in an OSS ecosystem. Meanwhile, the software with the rapid growth of popularity is a double-edged sword for OSS ecosystem and its reusers. Accordingly, it is important to understand the popularity growth of packages (i.e., how fast packages become popular). In this part, we conduct an exploratory study on packages in the node package manager (`npm`) to understand: (1) the characteristics of popularity growth, and (2) the factors that could affect popularity growth. We propose a method to model popularity growth as a curve and find that popularity growth mathematically follows three models — accelerated growth model (i.e., quadratic model), steady growth model (i.e., linear model), and decelerated growth model (i.e., square root model). The results show that 51.56% of the studied packages depict steady growth model, followed by accelerated growth model and decelerated growth model, 40.02% and 7.20% respectively. Furthermore, we reveal that factors including age, dependents, new features, and functionalities have impacts on

popularity growth. Our study shows potential tips for helping practitioners in developing and evolving packages in a competitive OSS ecosystem.

The findings in this dissertation will help practitioners who reuse OSS in practice and researchers who are to create a better platform for OSS reuse.

List of Publications

1. Shi Qiu, Daniel M. German, Katsuro Inoue. "An Exploratory Study of Copyright Inconsistency in the Linux Kernel." IEICE TRANSACTIONS on Information and Systems, 2021 (to appear).

2. Shi Qiu, Daniel M. German, Katsuro Inoue. "A Machine Learning Method for Automatic Copyright Notice Identification of Source Files." IEICE TRANSACTIONS on Information and Systems, Vol.E103-D, No.12, pp.2709-2712, Dec. 2020.

3. Shi Qiu, Raula Gaikovina Kula, Katsuro Inoue. "Understanding Popularity Growth of Packages in JavaScript Package Ecosystem." The 3th IEEE/ACIS International Conference on Big Data, Cloud Computing, and Data Science Engineering (BCD 2018), pp.55-60, Yonago, Japan, Jul. 2018.

4. Shi Qiu, Daniel M. German, Katsuro Inoue. "Empirical Study on Dependency-related License Violation in the JavaScript Package Ecosystem." Journal of Information Processing, 2021 (under the final review).

Acknowledgement

First of all, I wish to express my deepest gratitude to my supervisor Professor Katsuro Inoue, for his continuous support and insightful advice throughout these years, no matter in research or my personal life. It would be impossible for me to complete this work without his guidance. It is my fortune to have the opportunity to have been supervised by him and have been inspired by his immense knowledge and enthusiasm toward research. I do not doubt that all the things he taught me would be my precious wealth throughout my life.

I would like to thank my co-supervisors: Professor Daniel M. German at the University of Victoria. He gave me a lot of motivating research ideas and inspiring comments on my work. I am grateful for his guidance and advice.

Besides my supervisors, I would like to thank the rest of my thesis committee: Professor Shinji Kusumoto and Professor Fumihiko Ino, for their valuable comments regarding my research.

I would like to thank Associate Professor Takashi Ishio and Assistant Professor Raula Gaikovina Kula at Nara Institute of Science and Technology, and Assistant Professor Yuki Manabe at the University of Fukuchiyama for their kindly guidance and insightful comments on my research. In particular, thanks to Yuhao Wu at Hitachi, Ltd. He was like an elder brother to me providing advice and encouragement continuously. I am very thankful for building a true friendship with him.

I would also like to express my gratitude to Associate Professor Makoto Matsushita, Assistant Professor Tetsuya Kanda, Specially Appointed Professor Shusuke Haruna, Associate Professor Norihiro Yoshida at Nagoya University, Assistant Professor Eunjong Choi at Kyoto Institute of Technology, Ali Ouni at the University of Quebec, Dr. Xin Yang, and Ms. Kate Stewart at The Linux Foundation, for their support and advice.

I would like to thank all the members of Inoue Laboratory, creating a friendly environment for studying and researching. Special thanks to our lab secretary Ms. Mizuho Karube, for her continuous support and assistance in the past 6 years. I would never forget the joyful time I had in Inoue Laboratory.

Thanks are also due to all the friends I made in Japan. They helped me in many aspects besides my research and made my life enjoyable here.

Last but not least, I am deeply indebted to my parents, Junying Qiu and Yan Qin, for their spiritual support throughout my studies and unconditional love throughout my life.

Contents

1 Introduction **1**
1.1 OSS and OSS Reuse 1
1.1.1 Open Source Software 1
1.1.2 OSS Reuse 2
1.2 The Reuse Issues in OSS 3
1.2.1 Copyright Inconsistency 4
1.2.2 Copyright Notice Identification 6
1.2.3 Dependency-related License Violation 6
1.2.4 Popularity Growth of OSS 7
1.3 Overview of the Dissertation 8

2 Study on Copyright Inconsistency in the Linux Kernel **9**
2.1 Introduction 9
2.2 Background 12
2.2.1 Copyright Notices in FOSS Projects 12
2.2.2 Source Code Committers in FOSS Projects 13
2.3 Copyright Inconsistency 14
2.3.1 Definition 14
2.3.2 Categorization 14
2.4 Empirical Study 16
2.4.1 Research Questions 16
2.4.2 Dataset Construction 16
2.4.3 Analysis Method 19
2.5 Results 20
2.5.1 RQ1: How Prevalent Are Copyright Inconsistencies? 20
2.5.2 RQ2: What Caused the Copyright Inconsistencies? . 22
2.6 Threats to Validity 24
2.7 Related Work 28
2.7.1 Software Ownership 28
2.7.2 OSS Contributor 28
2.7.3 Software License 29
2.8 Conclusion of This Chapter 29

3 A Machine Learning Method for Automatic Copyright Notice Identification of Source Files 31

3.1 Introduction . . . 31
3.2 Machine Learning Method . . . 32
3.3 Comparison of Four Supervised Classifiers . . . 33
3.3.1 Dataset Construction . . . 34
3.3.2 Experiment and Results . . . 34
3.4 Comparison to FOSSology . . . 35
3.4.1 Dataset Construction . . . 35
3.4.2 Experiment and Results . . . 36
3.5 Conclusion of This Chapter . . . 36

4 Study on Dependency-related License Violation in the JavaScript Package Ecosystem 37

4.1 Introduction . . . 37
4.2 Background . . . 39
4.2.1 License Violation . . . 39
4.2.2 Package Dependency . . . 40
4.2.3 Dependency-related License Violation . . . 41
4.3 The Prevalence of Dependency-related License Violation . . . 43
4.3.1 Data collection . . . 43
4.3.2 Method . . . 43
4.3.3 Results and discussion . . . 49
4.4 Preliminary Questionnaire . . . 51
4.4.1 Questionnaire Design . . . 51
4.4.2 Data collection . . . 51
4.4.3 Results and discussion . . . 52
4.5 Threats to Validity . . . 54
4.6 Related Work . . . 55
4.6.1 Software License . . . 55
4.6.2 License Compliance . . . 56
4.7 Conclusion of This Chapter . . . 56

5 Study on Popularity Growth of Packages in the JavaScript Package Ecosystem 59

5.1 Introduction . . . 59
5.2 Modeling Popularity Growth as a Curve . . . 61
5.3 Empirical Evaluation . . . 63
5.3.1 Research Question 1 . . . 63
5.3.2 Research Question 2 . . . 65
5.4 Discussion . . . 70
5.5 Conclusion of This Chapter . . . 71

6 Conclusion and Future Work **73**
6.1 Conclusion 73
6.2 Future Directions 74

List of Figures

2.1 The impact of the selection of the minimum threshold percentage on the results of the detection of the committer-not-holder inconsistency. 26

4.1 The examples of direct dependency and indirect dependency. 41
4.2 A part of the dependency network of `cstar` package in `npm`. 41
4.3 License compatibility network. 42

5.1 The popularity growth for three packages representing the three proposed models. In each plot, the blue curve is the one created with the original data while the red curve is the model that fits best. Note that Figure 5.1(a) represents the accelerated growth model (`grunt`), Figure 5.1(b) represents the steady growth model (`wcsize`) and Figure 5.1(c) illustrates the best-fits for decelerated growth model (`active`). . 61
5.2 The effect of dependencies on downloads in `npm`. When package a is installed, package b, c and d is downloaded at the same time. 62
5.3 The word-cloud graph using the keywords of every package fitted by each growth models. 68

List of Tables

2.1 Summary of the roles introduced in this chapter. 11
2.2 Result of using `cregit` on generic.h. 15
2.3 Summary of the target version of the Linux kernel. 16
2.4 A part of the built organization dictionary. 19
2.5 The number of source files with different types of copyright inconsistencies and the ratios. Ratio to ① means the ratio of the source files detected as having this type of inconsistency to all source files in the sample dataset. Ratio to ④ means the ratio of the source files detected as having this type of inconsistency to the source files detected as having any type of inconsistency. 20
2.6 The reasons why holder-not-committer inconsistencies happened in the Linux kernel version 4.14. 23
2.7 The reasons why committer-not-holder inconsistencies happened in the Linux kernel version 4.14. 24
2.8 The reasons why holder-not-committer inconsistencies happened in the Linux kernel version 5.80. 27
2.9 The reasons why committer-not-holder inconsistencies happened in the Linux kernel version 5.80. 28

3.1 Examples of the identified copyright notices using FOSSology for bonito64.h. 32
3.2 The word categorization and the tokens we use to replace words. 33
3.3 Summary of the target version of the Linux kernel. 34
3.4 Comparison of four classifiers. 35
3.5 Evaluation of the proposed method. 36

4.1 The list of the selected licenses. 45
4.2 The rules of how changes should be handled in `npm`. 47
4.3 The examples of how to specify the ranges. 48
4.4 The historical meta-data dataset constructed for `cstar` package in `npm`. 48

4.5 The attached information of version and license for the dependencies in the dependency network constructed for `cstar` package. 49
4.6 The top 10 dependency-related license violations. 50
4.7 The proportion of the selected licenses in `npm`. 50

5.1 Mathematical models 62
5.2 Summary Statistics of the collected dataset 64
5.3 Best Fitting Results for the 102,341 target packages. 64
5.4 Summary Statistics of the 101,088 fitted packages. 66

Chapter 1

Introduction

Software is playing an important role in society nowadays. A lot of approaches have been proposed to support software development. Among them, software reuse has been proved to be an efficient and effective way to increase software productivity and improve the quality of software[11, 56, 64], especially open source software (OSS) reuse. OSS is software whose source code can be reused under some particular terms and conditions. OSS reuse makes valuable contributions in helping developers create new software products in modern software development [63]. However, there are also some reuse issues in OSS. In this dissertation, we deal with these issues. We first introduce OSS and OSS reuse in Section 1.1. We then introduce the reuse issues in OSS in Section 1.2 respectively.

1.1 OSS and OSS Reuse

In this section, we provide a brief background of OSS and OSS reuse respectively.

1.1.1 Open Source Software

OSS could be defined as software whose source code can be reused under some particular terms and conditions with the help of open source license. Open source license allows the software to be freely used, modified, and redistributed by anyone, as long as the conditions of its license are satisfied. OSS employs new types of development activities, community networking, and organization structure [61]. Different from proprietary software, OSS is developed in a collaborative manner, receiving contributions from a large number of people from different countries/regions, or different organizations. OSS supports our daily life in a wide variety of aspects: mobile phone, web service, and IT system infrastructure all rely on OSS. Hundreds of OSS are in use by thousands to millions of users nowadays, some

of which have millions of lines of source code. For example, the Linux kernel is contributed by 21,074 different contributors at the end of 2019 [51]. OSS has become an important asset in software development. The famous examples of OSS projects are Linux kernel[1], Apache[2], Mozilla[3], etc.

There are different terms for describing the OSS phenomenon: open source software (OSS), free software, and free open source software (FOSS), etc. Although there are differences between them with regards to the assigned licenses, they are often treated as the same thing [43, 61]. We use open source software (OSS) as the general term because the difference of these terms does not affect the main goal of this research.

OSS projects are usually created by an individual or a group motivated by a specific need [68]. With the development of OSS, more and more contributors join and contribute the source code to the project, forming a broad community known as the OSS community. Generally speaking, there are two types of contributors: individuals and organizations, motivated by different types of motivation. They could participate in OSS projects as many roles, such as contributing source code, testing software, reviewing the source code, reporting and fixing bugs, writing documentation, and managing the development. OSS community has become the most important foundation of OSS, guaranteeing the development, support, and maintenance of OSS [13].

Furthermore, some OSS projects are developed and evolve together in a shared environment, known as the OSS ecosystem [54]. To develop a new OSS project, other OSS projects in the OSS ecosystem could be reused easily. OSS ecosystem has emerged in recent years as an efficient and effective way to increase the productivity of OSS and the activity of the OSS community. Furthermore, OSS ecosystems are increasingly popular for their economic, strategic, and technical advantages and have also become an emergent field of research that has been addressed from various perspectives [9]. The famous examples of OSS ecosystems are npm[4], RubyGems[5], etc.

1.1.2 OSS Reuse

Software reuse is the process of creating software systems from existing software rather than building software systems from scratch [30, 45]. Software reuse has long been advocated and proved as a good practice to save time, reduce cost, and increase quality. There are four types of software reuse: data reuse, architecture reuse, design reuse, code reuse, and module

[1]https://www.kernel.org/
[2]https://httpd.apache.org/
[3]https://www.mozilla.org/
[4]https://www.npmjs.com/
[5]https://rubygems.org/

reuse [40]. Developers could benefit from software reuse on many aspects such as code quality, coding efficiency, maintenance, and quality.

OSS reuse is one of the most common practices of software reuse. With the rapid growth of OSS, OSS reuse is playing a more and more important role in software development [63]. For example, almost half of the Norwegian software companies reuse OSS in their products [37], while 30% of the functionality of OSS projects in general reuse existing components [63]. OSS developers are willing to share their own source code and they are also glad to reuse others'. On one hand, reusing OSS could save development time and reduce the cost compared to writing the software from scratch [5]. On the other hand, the source code of OSS can be accessed publicly and often be widely tested by many other developers, because of which the quality can also be improved and ensured [6]. The rapid development of software forges like GitHub[6] and Google Code[7] also create a convenient environment for developers to share their OSS, reuse others' OSS, and collaborate with others.

1.2 The Reuse Issues in OSS

Although the developers benefit a lot from OSS reuse, OSS reuse also comes with many reuse issues needed to be taken special care of. On one hand, OSS reuse may have an important side-effect on software quality. For example, although more than 80% of the systems depended on outdated external libraries, developers were unaware of any security risks that were introduced into the system [48]. On the other hand, OSS reuse may have reusers face lawsuits and potential monetary penalties. If OSS is not reused according to the conditions and terms described by its licenses properly, the copyright owner could start legal proceedings against the reusers to enforce its license. For example, in the legal dispute between Oracle and Google, Oracle claimed for a penalty of $8.8 billion from Google [76]. Overlooking the copyright notices will also have the reusers be involved in the legal disputes with the original copyright owners. For example, Patrick McHardy - a developer of the Linux kernel - has sued some companies by claiming he owns the copyright notice of some components in the Linux kernel [57, 74].

There are much of reuse issues in OSS still needed to be addressed: the inconsistency between the declared holders in the copyright notices and the contributors of the source code (i.e. *copyright inconsistency*) increases the difficulty of establishing copyright ownership; copyright notice identification of source files is difficult; including other OSS with an incompatible license as dependency results into dependency-related license violation easily; developers have no knowledge of the popularity growth of OSS. In this

[6]https://github.com/
[7]https://code.google.com/

dissertation, we deal with these reuse issues to support OSS reuse. We introduce each of them respectively.

1.2.1 Copyright Inconsistency

Software copyright claims an exclusive right to determine whether and under what conditions this software can be modified, reused, or redistributed. Identifying who owns the software copyright is important because they have the right to change the license, granting a commercial one to a third party, or start legal proceedings to enforce the license.

The individuals or organizations explicitly declared in copyright notices (i.e. *holders*) and the individuals or organizations who actually contribute the source code (i.e.*contributors*) are the information that is usually used to establish the copyright ownership of an OSS project. However, they are not always consistent. In other words, the copyright notices recorded in the source files do not always reflect the actual contributors. We define the inconsistency between the holders in the copyright notices and the contributors of the source code as copyright inconsistency.

As an example of copyright inconsistency, we observed a file named hid-cypress.c in the Linux kernel. Before the commit whose id is 2f43f8749ebaeb4934f73a6f864fcbb60ce9f48a, the comments in the header part of source file are as follow:

```
/*
 *  HID driver for some cypress "special" devices
 *
 *  Copyright (c) 1999 Andreas Gal
 *  Copyright (c) 2000-2005 Vojtech Pavlik <vojtech@suse.cz>
 *  Copyright (c) 2005 Michael Haboustak <mike-@cinci.rr.com>
 *  for Concept2, Inc
 *  Copyright (c) 2006-2007 Jiri Kosina
 *  Copyright (c) 2007 Paul Walmsley
 *  Copyright (c) 2008 Jiri Slaby
 */

/*
 * This program is free software; you can redistribute it and
 * /or modify it under the terms of the GNU General Public
 * License as published by the Free Software Foundation;
 * either version 2 of the License, or (at your option) any
 * later version.
 */
```

It is easy to know that this file is licensed under GPL 2.0+ license, attached with 6 copyright notices. But in the commit mentioned above,

copyright notice *Copyright (c) 2007 Paul Walmsley* is removed from the comments. The reason is clarified in the commit log of this commit:

```
Paul Walmsley has implemented dynamic quirk handling back in
2007 through commits:

2eb5dc3 ("USB HID: encapsulate quirk handling into
hid-quirks.c")
8222fbe ("USB HID: clarify static quirk handling as
squirks")
8cef908 ("USB HID: add support for dynamically-created
quirks")
876b927 ("USB HID: add 'quirks' module parameter")

and as such, his copyright rightly belongs to
drivers/hid/usbhid/hid-quirks.c file.

However when generic HID code has been converted to bus and
individual quirks separated out to individual drivers on the
bus, the copyright has been blindly transferred into all the
tiny drivers, which actually don't contain any of Pauls'
copyrighted code.

Remove the copyright from those sub-drivers.

Signed-off-by: Jiri Kosina <jkosina@suse.cz>
Acked-by: Paul Walmsley <paul@pwsan.com>
```

As we can see, during the software evolution, copyright notice hold by Paul Walmsley has been wrongly imported into this file. However, Paul Walmsley does not own any source code in this source file. We also confirm this by tracking the history of the source file. No source code is contributed by Paul Walmsley while his copyright notice is recorded. A copyright inconsistency occurs in such a situation.

Copyright inconsistency in the industry can bring a large amount of damage to a company. For example, Patrick McHardy - a developer of the Linux kernel - has sued some companies by claiming his the authorship of some components of the Linux kernel [57, 74]. As a first step to deal with the issue of copyright inconsistency, we aim to investigate the prevalence of copyright inconsistencies and the reasons causing their occurrence in this research.

We will discuss this issue in detail in Chapter 2.

1.2.2 Copyright Notice Identification

Copyright notice claims who has the right to determine under what conditions this software can be redistributed, reused, and modified with the help of software licenses. Generally, a copyright notice is a sentence including the explicitly declared holders, the word "Copyright" or a copyright sign, and the valid year, written with the software license. An example of the copyright notice is as follows:

```
#
# Copyright (C) 2015-2021 Osaka Univ.
#
# This program is free software; you can
# redistribute it and/or modify it under the
# terms of the GNU General Public License as
# published by the free Software Foundation;
# either version 2, or (at your option)
# any later version.
#
```

This copyright notice declares that Osaka University owns the copyright of this software.

For OSS, it is very important to identify the copyright notices in source files since the individuals or organizations declared in the copyright notices can control those activities with license compliance. However, identifying copyright notice of source files is difficult. Both the collaborative manner of FOSS project development and the large number of source files increase the difficulty.

In this research, we aim at automatically identifying the copyright notices in source files based on machine learning. We first proposed a method to vectorize the copyright notice using the extracted natural language features. We then optimized and compared the accuracy of four supervised classifiers to address our research goals, followed by an evaluation experiment in which our method outperforms FOSSology, a state-of-the-art approach based on regular expression matching.

We will discuss this issue in detail in Chapter 3.

1.2.3 Dependency-related License Violation

Software license grants the permissions of reusing and redistributing the software to its users. Generally, it is a text written in the header part of the source files. Open source license is software license approved by Open Source Initiative, describing under what the terms and conditions OSS could be used, modified, and shared. Generally speaking, there are two types of open source licenses: the permissive license and copyleft license.

A license may violate another one according to the terms and conditions. License violation may cause potential legal risks [80]. Usually, a permissive license violates a copyleft one.

Here is an example: The following texture is a part of the Apache-2.0 license:

```
[...]
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at
http://www.apache.org/licenses/LICENSE-2.0
Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
[...]
```

While GPL-2.0+ license says:

```
[...]
This program is free software; you can redistribute it
and/or modify it under the terms of the GNU General Public
License as published by the Free Software Foundation; either
version 2, or (at your option) any later version.
[...]
```

According to the terms and conditions of the two licenses, it is easy to know that the Apache-2.0 license violates the GPL-2.0+ license.

In the user-contributed OSS ecosystem, a package usually includes other packages as dependencies. This package's license may violate the license of its dependencies. We define this violation as a dependency-related license violation. In this research, we aim to investigate the prevalence of dependency-related license violation by conducting an empirical study on npm. We also design a preliminary questionnaire aiming to find out the developers' attitude.

We will discuss this issue in detail in Chapter 4.

1.2.4 Popularity Growth of OSS

Software popularity is valuable information to developers especially for software in the OSS ecosystem. It is an important clue to know whether a package is successful or not in the software ecosystem [12]. Meanwhile, the software with the rapid growth of popularity is a double-edged sword for the OSS ecosystem and its reusers. On one hand, the vulnerabilities or defects involved in this software will spread at the same time. On the other hand, if this software is suddenly removed from the OSS ecosystem, much

other software that depends on this software will stop work as well. Accordingly, it is important to understand the popularity growth of packages. To understand how fast software in the OSS ecosystem become popular, we conduct an empirical study on packages in `npm` - a large repository of JavaScript-based software packages - to investigate the characteristics of popularity growth and the factors that could affect popularity growth. The study on the characteristics of popularity growth is conducted based on the proposed method modeling popularity growth as a curve. The study on the factors that could affect popularity growth is conducted relying on statistic analysis examining whether or not there is a strong relationship between them and the proposed growth models.

We will discuss this issue in detail in Chapter 5.

1.3 Overview of the Dissertation

The rest of this dissertation is organized as follows:

Chapter 2 reports our work on analyzing copyright inconsistency and the reasons for its occurrence based on an empirical study of the Linux kernel.

Chapter 3 reports our work on proposing a method that automatically identifies the copyright notices in source files based on machine learning techniques.

Chapter 4 reports our work on investigating the prevalence of dependency-related license violations. We also conduct a preliminary questionnaire to study the developers' attitudes.

Chapter 5 reports our work on studying the popularity growth of packages in `npm` to understand the characteristics of popularity growth, and the factors that could affect popularity growth.

Chapter 6 concludes this dissertation and shows directions for future work.

Chapter 2

Study on Copyright Inconsistency in the Linux Kernel

2.1 Introduction

Software copyright grants the copyright owner a legal right to determine under what conditions this software can be redistributed, reused, and modified with the help of software licenses. Different from proprietary software, FOSS projects are developed in a collaborative manner, receiving contributions from a large number of people, named *contributors*, from different countries or regions, or different organizations. For example, the Linux kernel is contributed by 21,074 different contributors at the end of 2019, and those contributors have the potential to claim the copyright of the contributed software [51]. In this chapter, we name the individuals or organizations explicitly declared in copyright notices as the *holders*, and the individuals or organizations who actually own the copyrights as the *copyright owners*.

In general, the holders are seen as the copyright owners. However, the holders may not cover all contributors who could potentially become the copyright owners. So identifying the copyright owners of FOSS only by the holders is insufficient. Therefore, an important question in the copyright ownership problem of FOSS is: "Who are the actual copyright owners?" Identifying the copyright owner of FOSS is important for several reasons: a) only the copyright owner of FOSS is allowed to change its license or granting a commercial one to a third party, such as Oracle grants commercial licenses to MySQL[1] [33]; b) only the copyright owner of FOSS is allowed to start legal proceedings to enforce its license; c) several FOSS licenses (e.g. the

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

BSD family of licenses) require that the copyright owner of FOSS projects being reused be acknowledged in the documentation and other materials of the system that reuses it.

In recent years, some developers who once contributed to large FOSS projects have enforced the open source license against the distributors who use the FOSS. These charges are based on the terms of the highly restrictive open source licenses. These distributors may face lawsuits and potential monetary penalties if they ignore or violate the terms of these restrictive licenses. For example, Patrick McHardy - a developer of the Linux kernel - has sued some companies, claiming that he shares a great part of the authorship of the Linux kernel [57, 74]. While it is easy to find his copyright notice existing in some source files, it is not easy to identify what portions of his contributions still remain in the current kernel. In other words, his declaration of the copyright notice may be inconsistent with his remained contributions. Therefore, the question arises: "Are these developers enforcing the open source license against the distributors the actual copyright owners of redistributed FOSS projects?" To address this problem, both the open source community and the industry invested a great amount of effort. For example, an open source license compliance software system and toolkit called FOSSology[2] have been developed to detect the copyright notice buried in the source code [32]. BlackDuck[3] also provides a service of assessing the legal risks of software copyright to those who want to commercially reuse FOSS.

However, establishing copyright ownership in large FOSS projects and verifying whether the statement of a copyright notice is correct are complicated. On one hand, as we mentioned below, the contributors, who could potentially become the copyright owners as well, can not be totally covered by the holders. On another hand, since the large FOSS projects are usually co-developed by a large number of developers during a long period, the copyright notices potentially risk the poor management and can not refer to the actual copyright ownership. For example, a copyright notice can be added when a contributor contributes source code to a FOSS project, but if the contributed source code is totally deleted from this FOSS project, the added copyright notice is possibly left in the source file since the developer who did this deletion may not know which part of source code is governed by this copyright notice. In these cases, the copyright notices will denote copyright ownership incorrectly. The inconsistency between the holders in the copyright notices and the contributors of the source code (called *copyright inconsistency* or simply *inconsistency*) plays an important role in the difficulty of establishing copyright ownership in large FOSS projects.

[2]https://www.fossology.org/

[3]https://www.blackducksoftware.com/

Table 2.1: Summary of the roles introduced in this chapter.

Role	Definition
Copyright owner	Who actually owns the copyrights.
Holder	Who is explicitly declared in copyright notices.
Contributor	Who actually writes the source code.
Committer	Who commits the source code to repositories.

In this chapter, as a first step of understanding copyright ownership, we analyze copyright inconsistencies of the Linux kernel, which is a typical example of FOSS. To the best of our knowledge, no work has been done to go deep into copyright inconsistency. Only a few works analyzed the relationship between the contributors and the holders [19], which are insufficient to reveal some important problems about this issue such as how prevalent copyright inconsistencies are in FOSS and why copyright inconsistencies occur. Also, we want to know whether the statements of copyright notices are accurate and truly denote the contributors' contributions to the source files that contain them.

To achieve our goal, we must first identify the holders and contributors. The holders in copyright notices can be detected from the copyright notices directly. While identifying the contributors is much more difficult since tracking the "original" authors of the source code is a difficult task in software engineering as well. Here we use the *committers* who can be tracked in software repositories as an indicator of the contributors. The difference between committer and contributor is that committer is the developer who commits the source code while the contributor is the developer who actually writes the source code. Both of them could be the potential copyright owner who actually owns the copyrights. Although the committers are not always consistent with the contributors, they are the only explicit information we can observe in software repositories directly. We will explore the cases that a developer commits source code written by others in our empirical study. Using the committer can help us to conduct a quantitative analysis to study the copyright inconsistency because the holders and contributors are two most explicit pieces of information we can rely on to establish the ownership of the source file.

Table 2.1 summarizes all the roles introduced in this chapter.

Based on these questions, we first define and categorize the copyright inconsistency formally. We then conduct an empirical study to study copyright inconsistency. Our research questions are set as follows:

RQ1: How prevalent are copyright inconsistencies?

RQ2: What caused the copyright inconsistencies?

The contributions of this chapter are as follows:

1. We have made the first study to focus on the prevalence of copyright inconsistencies, relying a proposed analysis method to detect and study them.

2. We have also conducted an empirical study on the Linux kernel to find the reasons.

This chapter is organized as follows. Section 2.2 first provides a brief background on copyright notices in FOSS projects, after which Section 2.3 proposes the definition of copyright inconsistency. Our empirical study on the Linux kernel is described in Section 2.4, followed by Section 2.5 with a discussion of the results. Section 2.6 describes threats to validity. After a description of related work in Section 2.7, Section 2.8 concludes this chapter.

2.2 Background

In this section, we have a look at the practical situation of copyright notices and committers in the FOSS project.

2.2.1 Copyright Notices in FOSS Projects

A copyright notice is a sentence to declare the holders explicitly. Generally, a copyright notice begins with the word "Copyright" or a copyright sign "(C)", followed by the names of the holders. The valid year of this copyright notice is usually stated in the copyright notice as well. An example of the copyright notice in FOSS is as follows:

```
#
# Copyright (C) 2011-2013 Free Software
# Foundation, Inc.
#
# This program is free software; you can
# redistribute it and/or modify it under the
# terms of the GNU General Public License as
# published by the free Software Foundation;
# either version 2, or (at your option)
# any later version.
#
```

This copyright notice declares that the Free Software Foundation owns the copyright of this software. A FOSS license is stated with the copyright notice as well.

Usually, these holders are seen as owning the copyright. Note that in this chapter, holders refer to the individuals or organizations explicitly declared in copyright notices. While copyright owners refer to the individuals

or organizations who did own the copyrights. The holders may not always be consistent with the copyright owners.

Similar to other types of properties, copyright can be sold and transferred. Different FOSS projects have different rules on copyright transfer. Some FOSS projects require their developers to transfer the copyright of their contributions. For example, Oracle - the copyright owner of MySQL - requires all contributors to transfer the copyright of their contributions when they make the contributions [58]. This strict requirement ensures Oracle to be the only copyright owner. Therefore, "© 2019, Oracle Corporation and/or its affiliates" is the only copyright notice of MySQL, declaring that the holder, also the copyright owner, is Oracle Corporation and/or its affiliates. By having this only ownership of MySQL, Oracle is also able to offer commercial licenses.

However, most FOSS projects do not force the copyright transfer. They receive contributions from a large number of contributors, which makes establishing the ownership of FOSS more complicated. Therefore, a FOSS project may be copyrighted under from one to hundreds of copyright notices. An example is the Linux kernel where a lot of copyright notices can be observed. Meanwhile, these copyright notices are usually declared using some particular pattern. The situations of many other FOSS projects are similar to the Linux kernel.

2.2.2 Source Code Committers in FOSS Projects

Modern software development is usually participated by a lot of developers. The large FOSS projects are also developed in a collaborative manner, receiving commits from a large number of committers. For example, in the case of the Linux kernel, the Linux foundation - the organization dedicated to maintaining the Linux and other related FOSS projects communities - reported that over 15,600 developers from more than 1,400 companies have contributed to the Linux kernel since tracking began 11 years ago [14]. Version control systems such as git[4], usually have a feature known as "blame", to track who nominally commits certain lines of code to the repositories. To track the contribution at a more granular level of code tokens, a tool named `cregit` is developed to track the committer, time, and commit log of each token in source file [29].

In our following study, we rely on `cregit` to construct the dataset. Table 2.2 shows an example of the extracted information using `cregit` to a source file named generic.h in the Linux kernel. It is easy to know the list of the committers, and the proportion of the contribution of each committer. For example, Arnd Bergmann contributed 170 tokens to the source file,

[4] https://git-scm.com/

accounting for 80.6% of all tokens, and committed thrice, accounting for 27.3% of all commits.

2.3 Copyright Inconsistency

In this section, we make a precise definition of copyright inconsistencies and categorize them.

2.3.1 Definition

For the purpose of this research, copyright inconsistency refers to the inconsistency between the holders and the committer. The holders refer to the individuals or organizations explicitly declared in copyright notices according to our definition. The committers refer to the individuals who committed the source code to repositories.

2.3.2 Categorization

In many FOSS projects, copyright inconsistencies can be observed easily. Here we still take the source file generic.h in Linux kernel, as the target source file to observe. The header comments of this source file are as follows:

```
/*
 * spear machine family generic header file
 *
 * Copyright (C) 2009-2012
 * ST Microelectronics
 * Rajeev Kumar <rajeev-dlh.kumar@st.com>
 * Viresh Kumar <vireshk@kernel.org>
 *
 * This file is licensed under the terms of
 * the GNU General Public License version 2.
 * This program is licensed "as is" without
 * any warranty of any kind, whether express
 * or implied.
 */
```

It is easy to know that the holders are ST Microelectronics, Rajeev Kumar, and Viresh Kumar. Note that possibly Rajeev Kumar is an employee of ST Microelectronics according to the domain of his email address. We then extract the committers relying on `cregit`. The committers and the proportion of their contributions to the source files are shown in Table 2.2.

We can observe that inconsistencies happened in two ways.

Table 2.2: Result of using `cregit` on generic.h.

Person	#Token	T.Prop	#Commit	C.Prop
Arnd Bergmann	170	80.6%	3	27.3%
Viresh Kumar	29	13.6%	4	36.3%
Russell King	5	2.4%	1	9.1%
Robin Holt	5	2.4%	1	9.1%
Shiraz Hashim	1	0.5%	1	9.1%
Masahiro Yamada	1	0.5%	1	9.1%
Total	211	100.0%	11	100.0%

Firstly, we can not find any hints about the ST Microelectronics and Rajeev Kumar - both seen as the holders- in the committers. Their actual contributions to this source files are not clear. It may be a possible reason that their copyright notices are wrongly declared or just out of date.

Secondly, a lot of committers- Arnd Bergmann, Russell King, Robin Holt, Shiraz Hashim, and Masahiro Yamada - did not declare their copyright notices in the source file. This resulted in a problem that the ownership of this source file can not be established since we have no knowledge about the contributions of the committers to this source file. Especially, the copyright notice of Arnd Bergmann, who committed more than 80% source code to the source file by tokens, is not declared in the source as well. Establishing the ownership of a source file without regarding a committer who committed a large part of source code such as Arnd Bergmann is not wise.

So establishing the ownership of a source file only by the copyright notices is not sufficient. Some may argue that the committer such as Shiraz Hashim or Masahiro Yamada committed too few source codes. Therefore, the proportion of their commits can not support them to become the qualified committers and declare their ownership of this source file. We will have a discussion on this proportion in our empirical study in Section 2.4. These two types of inconsistencies between the holders and the committers can be observed in many other observed source files as well.

Based on the observation of Linux kernel, we observed two types of copyright inconsistency. One is the situation that the holder is not the committer (i.e. *holder-not-committer inconsistency*). Another one is the situation that the committer is not the holder (i.e. *committer-not-holder inconsistency*).

Note that if the organizations the committer belongs to are declared in the copyright notices, we determine there is no inconsistency in this situa-

Table 2.3: Summary of the target version of the Linux kernel.

Version	4.14
Date	Nov 13, 2017
#File	45,477

tion. A quantitative and executable analysis of the copyright inconsistency could be conducted based on the proposed definition and categorization.

2.4 Empirical Study

The goal of this section is to introduce our analysis methods to answer research questions. To achieve this goal, we select the Linux kernel - the most popular and successful open-source operating system kernel - to analyze. Table 2.3 shows a summary of the target version of the Linux kernel.

2.4.1 Research Questions

This empirical study aims at addressing the following research questions:

RQ1: How prevalent are copyright inconsistencies? This research question investigates the prevalence of copyright inconsistencies in the Linux kernel. Copyright inconsistencies will be detected using the proposed method, followed by quantitative analysis.

RQ2: What caused the copyright inconsistencies? This research question aims at finding the reasons causing the copyright inconsistencies in the Linux kernel. The results reveal the reasons by a qualitative analysis manually tracking the historical commit logs of the source files.

2.4.2 Dataset Construction

To achieve our goal, we first construct the datasets for our empirical study. The dataset construction consists of three steps - sampling, building the committer dataset, and building the holder dataset.

– Sampling:

We first download the source code of the Linux kernel from Github[5]. Notice that we only collect the source files whose life cycle can be entirely traced in Github. Some source files - known as pre-git files - have been created and evolved before the source code was uploaded to Github. Those pre-git files are not our target source files. This step makes copyright notices trackable, which helps us in answering the research questions. We end

[5]https://github.com/torvalds/linux

up with 38,932 source files after removing pre-git files from the total downloaded source files. We aim to construct a sample dataset by randomly selecting source files from these 38,932 source files. We use a statistical method to determine the sample size needed in order to get results that reflect the target population as precisely as needed. The required sample size was calculated so that our conclusions would generalize to all 45,477 source files with a confidence level of 95% and a confidence interval of 5 [6]. The calculation of statistically significant sample sizes based on population size, confidence interval, and confidence level is well established. The calculated required sample size is 381. At last, we randomly select 500 source files from 38,932 non pre-git files to construct the sample dataset.

Among these 500 source files, a part of source files' histories can not be tracked because of the mechanism of git. Git keeps track of changes to files in the working directory of a repository by their names. When a file is moved or renamed, git sees it as a creation of a new file while the original file is deleted. Since our analysis required the traceability of the entire history of a source file, we only select the source files which are not moved or renamed before. By removing source files once moved or renamed, we end up with a sample dataset consist of 414 source files. This number is still larger than 381 - the minimum required number of statistically significant sample size.

– **Building the committer dataset:**

In this step, we build the committer dataset. For each source file in the sample dataset, we extract the full name, number of tokens they contributed, and the proportion of this contribution for each committer using `cregit`. Note that a committer may have multiple different accounts in GitHub. We rely on `cregit` to solve this problem. `cregit` extracts the committer 's full name and use it as the unified identifier to merge the multiple different accounts owned by the same committer. We then extract the e-mail address of each committer by tracking the historical commit logs of the source file.

Lastly, we identify the organizations the committers belong to by checking the domain of their e-mail addresses. We use a semi-automatic method to achieve this goal. A domain dictionary is built to map the domain of the e-mail addresses to the organizations. When we try to identify the organization of a committer, we first check if the domain of his or her e-mail address is in the domain dictionary or not. If so, we determine the corresponding organization as the organization the committer belongs to. Otherwise, we manually check this domain and identify the organization for this committer. The pair of the domain and the organization is added into

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

the domain dictionary at the same time. Note that here an organization is indicated by a uniform identifier.

In this way, we build the committer dataset. All source files are contained in this dataset. For each of them, we list his or her full name, the number of tokens he or she contributed, the proportion of the contribution, e-mail address, and the organization he or she belongs to. During this process, we successfully build a domain dictionary as well.

– **Building the holder dataset:**

In this step, we build the holder dataset for each source file in the sample dataset. Note that we have built the committer dataset containing the full names of all committers and the uniform identifiers to indicate the organizations they belong to. We first use FOSSology [32] to detect the copyright notices, after which we manually check all detected copyright notices to remove the wrongly detected ones.

Then next, we build an organization dictionary shown in Table 2.4. The index is the uniform identifiers of the organizations. Each of the uniform identifiers refer to a list of possible organization names found in the analysis. The organization names are the different ways one organization might be referred to in copyright notices. In the beginning, the list only contains one name that we find in building the committer dataset.

We then match the full name of each committer to each detected copyright notice. We achieve this by checking whether the full name is included in the detected copyright notice or not. Note that this check is not case-sensitive. If we find the full name of one committer in the detected copyright notice, we determine it as the holder of this copyright notice. At the same time, the type of this holder is determined as individual. For the remaining copyright notices, we try to match them to the organization names in all lists in the organization dictionary. If an organization name is matched, we determine the index - the uniform identifier refers to the list containing this organization name - as the holder. At the same time, the type of this holder is determined as organization.

After these two matches, we try to manually identify the holders and their types for the remaining copyright notices. If the manually identified organization name has got a uniform identifier in the organization dictionary, we determine this uniform identifier as the holder. Meanwhile, we add this manually identified organization name into the list that the uniform identifier refers to. If the manually identified organization name has not got a uniform identifier in the organization dictionary, we create a new uniform identifier as the index and add this manually identified organization name into the list the newly created uniform identifier refers to. Finally, we build a holder dataset, for each source file in which the holders and the infor-

Table 2.4: A part of the built organization dictionary.

Index	List
ibm	IBM Corporation IBM Corp. International Business Machines Corp.
amd	Advanced Micro Devices, Inc AMD, Inc
ti	Texas Instruments, Inc. TI, Inc

mation about their related copyright notices and their types (individual or organization) are summarized.

During this process, we successfully build an organization dictionary at the same time. Table 2.4 shows a part of the built organization dictionary constructed during the analysis of the Linux kernel.

2.4.3 Analysis Method

To answer RQ1, we detect the copyright inconsistencies based on the built committer dataset and holder dataset. Based on the definitions in Section 2.3.1, we detect two types of copyright inconsistencies respectively - the committer-not-holder inconsistency and the holder-not-committer inconsistency.

To achieve our goals, we propose two definitions of committer for the detection of two types of copyright inconsistency. *General committers* refer to all committers who once committed source code. This definition will be used in the detection of the holder-not-committer inconsistency, which ensures that the holder-not-committer inconsistency can only be detected when we can not find any information about the holder in the committer dataset. *Core committers* refer to the committers who contributed more than a minimum threshold percentage of contribution. Setting a minimum threshold percentage of the contribution could exclude the minor committers who only do some simple work, and determines the core committer from a general committer. We will set this minimum percentage as 14.9% because it is the least percentage of source code at the token granularity of the contributor who committed the highest percentage of the source code in the sample dataset. We will discuss the effectiveness of this threshold and how this threshold could impact the result of the detection in Section 2.6. This definition will be used in the detection of the committer-not-holder inconsistency.

Table 2.5: The number of source files with different types of copyright inconsistencies and the ratios. Ratio to ① means the ratio of the source files detected as having this type of inconsistency to all source files in the sample dataset. Ratio to ④ means the ratio of the source files detected as having this type of inconsistency to the source files detected as having any type of inconsistency.

No	Inconsistency	#Files	Ratio to ①	Ratio to ④
①	sample dataset	414	100.0%	
②	holder-not-committer	134	32.4%	51.1%
③	committer-not-holder	229	55.3%	87.4%
④	any type of inconsistency	262	63.3%	100.0%
⑤	both two types	101	24.4%	38.5%

We first detect the holder-not-committer inconsistency. For each holder in the holder dataset, we check whether this holder is a name of any general committer or an organization that any general committer belongs to. A holder-not-committer inconsistency is detected if a holder is neither a name of any general committer or an organization that any general committer belongs to.

We then detect the committer-not-holder inconsistency. For each core committer in the committer dataset, we check whether its name or organization is recorded in the holder dataset or not. A committer-not-holder inconsistency is detected if neither a core committer's name nor its organization is recorded in the holder dataset.

To answer RQ2, we try to find the reasons behind the occurrence of the holder-not-committer inconsistency and the committer-not-holder inconsistency respectively. For each inconsistency, we manually check the commit logs and the comments in the source code of the source files to find out the reasons. A reason is determined only when it is explicitly recorded in the commit logs or the comments in source code.

2.5 Results

In this section, we report the results and have a discussion on the results to address research questions in our empirical study in Section 2.4.

2.5.1 RQ1: How Prevalent Are Copyright Inconsistencies?

In this research question, we analyze to what extent copyright inconsistencies exist in source files in the Linux kernel.

As can be seen from Table 2.5, 262 source files are detected as having copyright inconsistencies, accounting for 63.3% of all 414 source files. As a popular and well maintained OSS project, the copyright ownership of the Linux kernel should be clear and well maintained as well. The general image of the Linux communities is that there is not too much copyright inconsistency in the Linux kernel. An ideal situation is expected that there is no inconsistency. However, this result is out of our expectation and do not accord with the general image of the communities, which suggests that copyright inconsistencies are prevalent in the Linux kernel.

We can see that 134 source files are detected as having the holder-not-committer inconsistency, accounting for 32.4% of all 414 source files in sample dataset, and 51.1% of 262 source files detected as having any type of inconsistencies respectively. We can also see that 229 source files are detected as having the committer-not-holder inconsistency, accounting for 55.3% of all 414 source files in sample dataset, and 87.4% of 262 source files detected as having any type of inconsistencies respectively. It can be noticed that the committer-not-holder inconsistency is more prevalent than the holder-not-committer inconsistency. A possible reason for the prevalence of committer-not-holder inconsistency may be that only a few developers declare their copyright notices when they contribute the source code. Therefore, a lot of the committers are not included in the holders. We aim at finding the reasons in detail in RQ2.

It is also easy to know that 101 source files are detected as having both of two types, accounting for 24.4% of all 414 source files in sample dataset, 38.5% of 262 source files detected as having any type of inconsistencies, 75.3% of 134 source files detected as having the holder-not-committer inconsistency, and 44.1% of 229 source files detected as having the committer-not-holder inconsistency respectively. The result suggests that if the holder-not-committer inconsistency exists in a source file, there is also a high possibility of the occurrence of the committer-not-holder inconsistency. Oppositely, if the committer-not-holder inconsistency exists in a source file, there is no such high possibility of the occurrence of the holder-not-committer inconsistency. It may be a possible reason that for the source files detected as having the holder-not-committer inconsistency, the committers added others' copyright notices. Another possible reason may be that after a long time of software evolution, a lot of new source code is committed, and the new source code replaced the old source code committed by the committers before. We will investigate the reasons in RQ2.

As an answer to RQ1, copyright inconsistencies are prevalent in the Linux kernel, among which, the committer-not-holder inconsistency is more prevalent than the holder-not-committer inconsistency.

2.5.2 RQ2: What Caused the Copyright Inconsistencies?

We aim to find the reasons why copyright inconsistencies happened. To achieve this goal, for the holder-not-committer inconsistency, we manually check the commit logs and the comments in the source code of all 134 source files detected as having the holder-not-committer inconsistency. If a reason is explicitly recorded in a sentence, we note down that sentence. After that, we categorized all sentences to summarize the reasons.

For the committer-not-holder inconsistency, we select some source files detected as having the committer-not-holder inconsistency as the target. These source files should also satisfy the condition that no holder-not-committer inconsistency is detected. We end up with 128 source files. For these files, all holders are the committers, so we plan to investigate the other committers who are not holders to find why they are not recorded in copyright notices. Different from the holder-not-committer inconsistency, holders who did not add their copyright notices usually did not explain their reasons explicitly. So we try to find some hints about why holders did not add their copyright notices no matter they are explicit or not. We then categorized all sentences to summarize the reasons as well.

Table 2.6 shows the summarized reasons why holder-not-committer inconsistencies happened. Among them, *code reuse* and *refactoring* are two common activities in software development. It is possible that copyright notices are not well managed during these activities.

To our surprise, *affiliation change* - the change of the companies or organizations the developers belong to - plays an important role in the occurrence of the copyright inconsistency, which reveals that the management of copyright notices may be overlooked by the developers as well. The "affiliation change" is found when we observed that although the developer's affiliation identified by e-mail address is not consistent with the one in the copyright notice currently, this developer's affiliation may be consistent with the one in the copyright notice before. Specifically, we observed four cases: (1) The developer switches to a different company or organization. (2) The developer belongs to more than one company or organization. (3) The company or organization is merged into another one. (4) The developer uses a personal e-mail address. All these cases can be found by checking the commit logs, the comments in the source code of the source files, and the profiles of the developers. We also try to search the related information on the Internet (e.g. the information of the developer on LinkedIn[7], the information of the company or organization in Wikipedia[8], etc.) to endorse our findings. Note that we do not explore the reasons why "affiliation change" happens and whether possible legal risks exist. Developers may just forget

[7] https://www.linkedin.com/

[8] https://en.wikipedia.org/

Table 2.6: The reasons why holder-not-committer inconsistencies happened in the Linux kernel version 4.14.

Categorization	#Source files	Proportion
Code reuse	37	27.6%
Affiliation change	19	14.2%
Refactoring	14	10.4%
Support function	13	9.7%
Others' contributions	13	9.7%
Typo	5	3.7%
All replaced	2	1.5%
None	31	23.2%
Total	134	100.0%

to update the copyright notices and the legal risks also vary across different countries or regions. However, we do observe that "affiliation change" is an important reason causing copyright inconsistency and needs to be noticed.

Support function and *others' contributions* are two reasons why developers add others' copyright notices when they committed source code. Another interesting case is *All replaced*, which is the situation that the source code the holder committed is totally replaced by the source code committed by the committers later, but the copyright notice is retained. *All replaced* also suggests that copyright notices are not well managed in the Linux kernel. We draw a conclusion that *code reuse*, *affiliation change*, *refactoring*, *support function*, and *others' contributions* are the main reasons why the holder-not-committer inconsistency occurred.

Table 2.7 shows the summarized reasons why committer-not-holder inconsistencies happened. The results are similar to the results of holder-not-committer inconsistency. *Code reuse*, *affiliation change*, *refactoring*, *support function*, and *others' contributions* are still the main reasons why the committer-not-holder inconsistency occurred.

Based on these findings, the following practical suggestions may help practitioners: (1) When developers reuse source code, the provenance of reused source code should be recorded properly as well for the traceability of the copyright notice. (2) The list of contributors should be well maintained as an evidence of copyright ownership. (3) The communities should propose a set of practical guidelines for managing the copyright notices. For example, when a copyright notice is added, modified, or deleted, the reason and the coverage of influence should be recorded. (4) The copyright

Table 2.7: The reasons why committer-not-holder inconsistencies happened in the Linux kernel version 4.14.

Categorization	#Source files	Proportion
Code reuse	30	23.4%
Affiliation change	12	9.4%
Refactoring	8	6.3%
Others' contributions	7	5.5%
Typo	1	0.8%
None	70	54.6%
Total	128	100.0%

ownership should be ascertained with a finer granularity such as line level or token level. The related tools are needed to be developed.

As an answer to RQ2, we draw a conclusion that *code reuse*, *affiliation change*, *refactoring*, *support function*, and *others' contributions* are the main reasons why copyright inconsistency occurred.

2.6 Threats to Validity

This section discusses the threats to the validity of our research. Threats to construct validity concern the relationship between theory and outcome, and relate to possible measurement imprecision when extracting data we used in this study. In mining the repositories to build the committer dataset, we first rely on `cregit` to summarize the names of the committers and their contributions. `cregit` measures the contribution in terms of tokens. Compared with the method measuring the contribution in terms of lines before, `cregit` is more precise and could avoid the case that developers doing simple code change are seen as the owner of the total lines. However, the precision of `cregit` is not proved by a large-scale test. To limit this problem, we randomly select some source files to check the precision of `cregit` at the same time when we do the empirical study. Another threat in using `cregit` is its method of merging the multiple different accounts owned by the same committer. `cregit` merges them based on the full name of the committer. This method will be ineffective for the case that a committer uses more than one name to commit source code to a single source file. However, this defect does not affect the results because we do not observe this case in our empirical study.

Another threat in building the committer dataset is that we identify the organizations the committers belong to by checking the domain of their

e-mail addresses. But the committers possibly use their personal e-mail addresses. Also, the committers may not change their e-mail addresses in Github when they change their organizations. For these cases, we can not rightly identify their organizations. Considering that we also track the change of their organizations in answering the RQ2, this threat might have an impact on our empirical study as well.

In mining the repositories to build the holder dataset, we first rely on FOSSology to extract the copyright notices. Although we manually check the copyright notices detected by FOSSology to remove the wrongly detected ones, FOSSology could have missed some copyright notices. We plan to use other methods or tools to do the detection in the future.

Another case worthwhile of being discussed is the minimum threshold percentage of contribution to define the committer used to detect the committer-not-holder inconsistency. We have set this minimum threshold percentage as 14.9%. The percentage of source files detected as having the committer-not-holder inconsistency will increase if we do not set it. In our calculation, if we would lift the threshold, 87.9% of all source files are detected as inconsistent, which proves the effectiveness of the minimum threshold percentage in excluding the minor committers who only do some simple work. Furthermore, we also investigate the impact of the selection of the minimum threshold percentage on the results of the detection of the committer-not-holder inconsistency. Figure 2.1 shows the result. The rapid decrease of the number of the source files detected as having the committer-not-holder inconsistency with the threshold increasing from 0% to 14.9% proves the effectiveness of the minimum threshold percentage as well. Some studies set the minimum threshold percentage as 5% to exclude the minor contributors [10, 22]. To what extent the contribution is needed to become a qualified contributor or state the copyright ownership is still a complicated problem under discussion.

Another threat requiring consideration is the effectiveness of the sampling. To achieve an accurate result, a lot of manual works are included in our proposed method to detect copyright inconsistency, for which we have introduced sampling. To validate the effectiveness of the sampling and the manual works, we have conducted a comparison experiment. We first designed a fully-automatic method without manual works to detect copyright inconsistency and then used it to detect copyright inconsistencies targeting all 38,932 source files in the Linux kernel and 500 source files the sample dataset respectively. For all source files in the Linux kernel, 19,261 source files out of 38,932 total source files are detected as having hold-not-committer inconsistency, accounting for 49.5%, and 24,143 source files are detected as having committer-not-hold inconsistency, accounting for 62.0%. For the sample dataset, 228 source files out of 500 files are detected as having hold-not-committer inconsistency, accounting for 45.6%, and 309 source

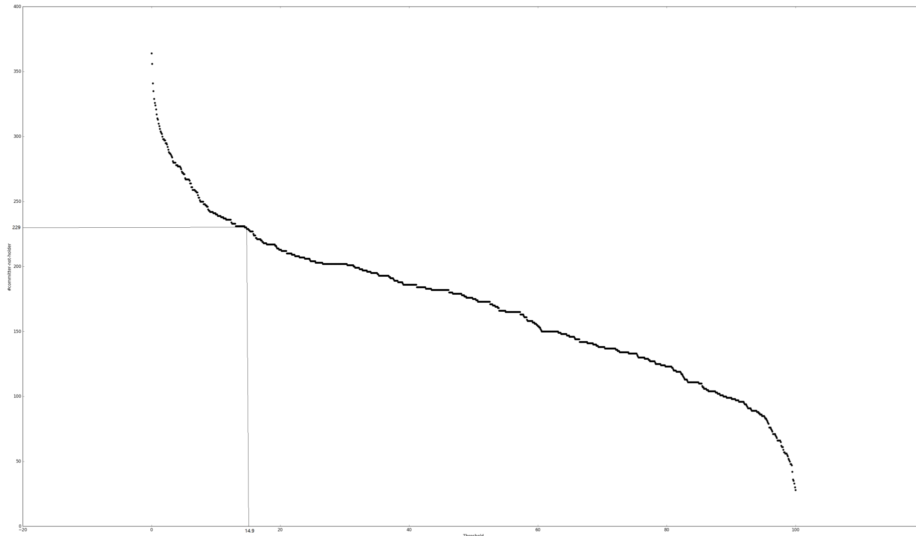


Figure 2.1: The impact of the selection of the minimum threshold percentage on the results of the detection of the committer-not-holder inconsistency.

files are detected as having committer-not-hold inconsistency, accounting for 61.8%. It is easy to find that the results are similar, which suggests that the sampling is effective. Also, this sampling method based on population size, confidence interval, and confidence level is well established. It is first proposed by Krejcie and Morgan in 1970 [44], and widely used and proven by other related works [15, 23, 36]. Then we have compared the results detected by the proposed method in Table 2.5 and the results detected by the newly designed fully-automatic method targeting the sample dataset. The proportions detected by the proposed method are smaller. Because of the manual works in the proposed method, the copyright inconsistencies detected by the proposed method are all actual ones while the copyright inconsistencies detected by the newly designed fully-automatic method are not. The results suggest that the manual works are effective and could improve accuracy.

Threats to internal validity concern factors internal to the study that could impact our results. Such a kind of threat does not affect exploratory study like the one in this chapter. The only case worthwhile of being discussed is our answering to RQ2, where we classify the reasons manually based on our knowledge.

Threats to external validity are related to the ability to generalize the finding in our study. Our empirical study is only conducted on the Linux kernel. Linux kernel is the most popular and successful open-source operating system kernel, receiving contribution from 13,500 developers from

Table 2.8: The reasons why holder-not-committer inconsistencies happened in the Linux kernel version 5.80.

Categorization	#Source files	Proportion
Code reuse	38	30.2%
Affiliation change	21	16.7%
Refactoring	12	9.5%
Support function	17	13.5%
Others' contributions	8	6.3%
Typo	2	1.6%
None	28	22.2%
Total	126	100.0%

more than 1,300 organizations. These features make Linux kernel a suitable target to study copyright inconsistencies. But according to the different requirements about the copyright, other open-source projects may have different results. We agree that it is necessary to replicate our empirical study on different projects.

Another case worthwhile of being discussed is the generalization of our findings in RQ2. To address this issue, we have repeated our experiment targeting another version of the Linux kernel and checked whether or not we can achieve similar results. The new experiment targeted version 5.80 of the Linux kernel, from which we randomly selected 500 source files to construct the sample dataset. After removing the source files once moved or renamed, we ended up with a sample dataset of 390 source files. We then repeated the detection. Among these 390 source files, 126 source files are detected as having hold-not-committer inconsistency, accounting for 32.3%, while 205 source files are detected as having committer-not-hold inconsistency, accounting for 52.6%. Among the 205 source files where committer-not-hold inconsistency are detected, 111 source files are detected as having only committer-not-hold inconsistency and no hold-not-committer inconsistency. Then we have used the same method to find reasons behind the occurrence of the copyright inconsistency. Table 2.8 and Table 2.9 show the results. The similar results suggest the ability to generalize the finding in RQ2.

Table 2.9: The reasons why committer-not-holder inconsistencies happened in the Linux kernel version 5.80.

Categorization	#Source files	Proportion
Code reuse	25	22.5%
Affiliation change	9	8.1%
Refactoring	6	5.4%
Others' contributions	2	1.8%
None	69	62.1%
Total	111	100.0%

2.7 Related Work

2.7.1 Software Ownership

Some studies in software engineering investigated software ownership. Girba et al. built the ownership based on the percentage of source code lines modified by contributors [31]. Tsikerdekis et al. proposed a code contribution ranking algorithm to build the ownership by tracking the survival of individual characters [66]. Bird et al. explored the effects of ownership on software quality[10]. Compared with their works, we use a more accurate and reasonable measure - token - to measure contributions. Especially, different from the above works, we also included copyright notice into consideration. Our work opens up a new way to study software ownership.

2.7.2 OSS Contributor

There are some studies that devoted to investigate OSS contributors and their contributions. German et al. studied the committers of the PostgreSQL project and found that apart from the core team, a large number of contributors sent source code patches to the project [24]. Hindle et al. discovered that the large commits including a large number of files are related to license or copyright owners [38]. Hammad et al. proposed two measures to measure the contribution of software developers in the evolved structural design of software systems [35]. Different from their works, we discussed the discrepancy between the contributors' contributions and the recorded copyright notices. Our work creates a possibility of importing the existing works on OSS Contributor into the software copyright management of the FOSS projects.

2.7.3 Software License

There are some studies that devote to identify licenses [27, 32]. Based on these studies, some researchers analyzed software licenses in open source projects and revealed some license issues. German et al. proposed a method to understand licensing compatibility issues in software packages [28]. Wu et al. proposed an approach to find license inconsistencies in similar files [78]. By investigating the revision history of these files, they summarized the factors that caused these license inconsistencies and tried to decide whether they are legally safe or not. Studies on software license are also closely related to this work. Many studies in software engineering investigated software license. However, to solve the legal risks in reusing FOSS, only the studies on software license are not sufficient. This work is a supplement to works on software license by studying the software ownership.

2.8 Conclusion of This Chapter

In this chapter, we first proposed the issue of copyright inconsistency, and then we defined copyright inconsistency and categorized different types of it. After that, we conducted an empirical study on the Linux kernel to study the prevalence of copyright inconsistency. To the best of our knowledge, this study is the first in this field to address this issue. We observed that the copyright inconsistency is prevalent in the Linux kernel. It suggests that the copyright notices recorded in the source files do not always reflect the actual contributors. To find how copyright inconsistency happens, we had a deeper look at the commit logs and the comments in source code to find the reasons why the copyright inconsistencies happened. We found that *code reuse*, *affiliation change*, *refactoring*, *support function*, and *others' contributions* are the main reasons.

The proposed method and results of this work can be reused in the following aspects: (1) The proposed method of detecting copyright inconsistency can be reused in other OSS projects to study the situation of copyright inconsistency in them. (2) Our findings on the prevalence of copyright inconsistency and the reasons causing them can be used as a benchmark to study the difference in the copyright-related issue among different OSS projects. (3) Our findings also provide a new perspective to study software development management, code reuse, and the organizational culture in OSS projects. (4) The datasets constructed in this work can be reused in other highly related works such as the participation of developers or companies in OSS projects and the collaboration between them.

In our future work, we will make some guidelines for developers to help them in dealing with copyright notices. We also aim to find a solution to manage the copyright notices in the FOSS projects.

Chapter 3

A Machine Learning Method for Automatic Copyright Notice Identification of Source Files

3.1 Introduction

Software copyright grants the copyright owner declared in the copyright notice a legal right to determine under what conditions this software can be redistributed, reused, and modified. Copyright notice is a few sentences mostly placed in the header part of a source file as a comment or in a license document in a FOSS project. Identifying the copyright notices of source files is important for several reasons: a) the copyright owner is allowed to change its license or to grant a commercial one to a third party; b) the copyright owner is allowed to start legal proceedings to enforce its license [33]; c) several FOSS licenses (e.g. the BSD family of licenses) require that the copyright owner of FOSS projects being reused should be acknowledged in the documentation and other materials of the system that reuses it [52].

However, copyright notice identification of source files is difficult. On one hand, different from proprietary software, FOSS projects are developed in a collaborative manner, receiving contributions from a large number of developers who potentially declare the copyright notice. On another hand, a large FOSS project usually consists of a large number of source files in which the copyright notices are buried.

To overcome these difficulties, a tool named FOSSology[1] has been developed to automatically identify copyright notice [32]. FOSSology identifies copyright notices in the comments of the source files using regular

[1] https://www.fossology.org/

Table 3.1: Examples of the identified copyright notices using FOSSology for bonito64.h.

1	copyright message in any source redistribution in whole or part.
2	Copyright (c) 1999 Algorithmics Ltd
3	Copyright (C) 2001 MIPS Technologies, Inc. All rights reserved.

expression-based matching. However, some sentences related to copyright could be wrongly identified as copyright notices by FOSSology. Table 3.1 shows examples of the identified copyright notices using FOSSology to a source file named bonito64.h in the Linux kernel. We can see that sentence 1 is identified as a copyright notice incorrectly. This is because the target sentence contained a keyword "copyright", and FOSSology's regular expression matches this keyword and all the following words in that sentence. To solve this problem, we propose a machine learning method in this chapter. The results of experiments suggest that Decision Tree and Random Forest perform best on automatic copyright notice identification of source files, and the proposed machine learning method outperforms FOSSology.

3.2 Machine Learning Method

In this section, we introduce a machine learning method for automatic copyright notice identification of source files. The proposed method consists of four steps.

1) Copyright-related sentence extraction and pre-processing: We first use a keyword-based method to extract copyright-related sentences from the comments in the source files. A sentence is extracted as a copyright-related sentence when it includes any copyright-related keywords and signals such as "copyright", "©", and "(C)". Some other potentially relevant words, such as "authored by", "written by", etc., are included as well. These heuristics are similar to FOSSology. The difference is that FOSSology uses the keywords to construct regular expressions. Although FOSSology may wrongly identify some copyright-related sentences as copyright notices, it still performs well in identifying all actual copyright notices with the help of these heuristics. Therefore, we use similar heuristics in this step to achieve a good recall. Our method will outperform FOSSology in achieving better precision by using the machine learning method. For each extracted copyright-related sentence, we tokenize it into words, lemmatize and convert each word to lowercase, and then remove punctuation.

2) Vectorization: We use the numbers of words of different categories as features to vectorize the copyright-related sentence. We categorize the

Table 3.2: The word categorization and the tokens we use to replace words.

Category	Token	Example
Copyright-related keyword	COPYRIGHT	copyright, author
Copyright-related signal	SIGNAL	©, (C), (c)
Year	YEAR	1991, 2002, 2013
E-mail address	EMAIL	addr@email.com
Others	OTHER	license, above

words into five categories, i.e. *copyright-related keyword*, *copyright-related signal*, *year*, *e-mail address*, and *others*. For each word in the copyright-related sentence extracted in Step 1, we replace it into a particular token. Table 3.2 shows the tokens we used to replace the words of different categories. After replacement, we count the number of tokens of each category and then construct a 5-dimension vector. We end up with a list of vectors representing the extracted copyright-related sentences.

3) Training Classifier: Four supervised classifiers are considered, i.e. Naive Bayes (NB), Decision Tree (DT), Random Forest (RF), and Support Vector Machine (SVM). These four classifiers are widely used in text classification tasks [21], [62] and related work addressing similar problem [65]. We implemented them with the Scikit-learn library[2] [60]. We train the classifiers with a manually labeled dataset. Each vector in this dataset is manually labeled as actual copyright notice (i.e. positive copyright notice) or false copyright notice (i.e. negative copyright notice). The task of the trained classifier is to classify a vector representing a copyright-related sentence as a positive copyright notice or a negative one. The details of how we train and evaluate the classifiers will be described in Section 3.3.

4) Copyright notice identification: For each vector, the trained classifier predicts whether it represents a copyright notice or not using the trained classifier. A copyright-related sentence is identified as a copyright notice if the corresponding vector is classified as representing a copyright notice. Finally, we identify all copyright notices of a source file.

3.3 Comparison of Four Supervised Classifiers

In this section, we first describe how we construct the datasets. We then aim to find which supervised classifier performs best by comparing the performance of the four classifiers.

[2] https://scikit-learn.org/stable/

Table 3.3: Summary of the target version of the Linux kernel.

Version	4.14
Date	Nov 13, 2017
#File	45,477

3.3.1 Dataset Construction

To achieve this goal, we choose the Linux kernel - the most popular and successful open-source operating system kernel - as the target dataset. The source code of the Linux kernel is downloaded from Github[3]. Table 3.3 shows a summary of the target version of the Linux kernel.

We first randomly select 2000 source files from 45,477 source files in the Linux kernel and extract copyright-related sentences using the keyword-based method we described in Step 1 in Section 3.2. We end up with 2,297 copyright-related sentences. Note that the duplicate sentences are removed here. It means that even if a sentence exists in two or more source files, we only record it once. For each sentence, we manually inspect and label it as a positive copyright notice or a negative one. As a result, 2,146 sentences are labeled as positive copyright notices, and 151 sentences are labeled as negative ones respectively. The positive copyright notices and the negative ones are unbalanced, so balancing techniques have to be applied [39], [7]. To address the issue of unbalanced data, we manually create the negative copyright notices by randomly replacing words in 151 found negative copyright notices. A similar manual method has been proven effective in handling imbalance datasets in other software engineering tasks [73]. Note that the words used to do replacement are from the words in all 2,297 copyright-related sentences. In this way, we successfully extend the number of negative copyright notices to 2,146. We finally construct a dataset consisting of 2,146 positive copyright notices and 2,146 negative ones.

3.3.2 Experiment and Results

To evaluate the performance of four supervised classifiers, we first randomly split the dataset into two parts, 17% for the testing dataset and 83% for the training dataset. To train the classifiers, 5-fold cross-validation is performed for the training dataset [59]. In 5-fold cross-validation, the training dataset is partitioned into five equal-sized subsets. Each subset has the same percentage of labels. Every time, four subsets are used to train the classifiers and the remaining one is used for validation. This process iterates

[3]https://github.com/torvalds/linux

Table 3.4: Comparison of four classifiers.

Classifier	Label	Precision	Recall	F1-score
NB	Positive	0.98	0.83	0.90
NB	Negative	0.85	0.98	0.91
DT	Positive	1.0	1.0	1.0
DT	Negative	1.0	1.0	1.0
RF	Positive	1.0	1.0	1.0
RF	Negative	1.0	1.0	1.0
SVM	Positive	1.0	0.98	0.99
SVM	Negative	0.98	1.0	0.99

5 times until every fold has been used for testing once. Hyperparameter tuning - the process of determining a good set of hyperparameters - is conducted here to achieve the best performance. We train the classifiers with the best hyperparameters and then evaluate their performance with the testing dataset. To evaluate the performance of four classifiers, we use the following metrics: (1) Precision, which refers to the ratio of the number of correct identification to the total number of identifications made of copyright notices; (2) Recall, which refers to the ratio of the number of correct identification to the total number of manually extracted copyright notices; and (3) F1-score, which is the harmonic mean of the precision and the recall. The results are shown in Table 3.4.

The results suggest that Decision Tree and Random Forest perform best on automatic copyright notice identification of source files. We will use Random Forest to conduct the evaluation experiment in Section 3.4.

3.4 Comparison to FOSSology

In this section, we first describe how we construct the dataset. We then aim to evaluate the proposed method by comparing the performance of the proposed method and FOSSology, an existing method based on regular expression.

3.4.1 Dataset Construction

We still use the source files of the Linux kernel as the target to construct the dataset. To achieve this goal, we randomly select 500 source files from the source files in the same Linux kernel. Note that all these 500 source files are unseen in training classifier. For each source file, we manually check the comments to extract the copyright notices. We extract 537 copyright

Table 3.5: Evaluation of the proposed method.

Method	Precision	Recall	F1-score
Proposed method (RF)	1.0	1.0	1.0
Fossology	0.78	1.0	0.88

notices from these randomly selected 500 source files. Among them 351 copyright notices are not duplicated. The task of the evaluation experiment is to identify all 537 copyright notices from the source files.

3.4.2 Experiment and Results

In our evaluation experiment, we first use the proposed method and FOSSology to identify the copyright notices in the selected 500 source files respectively, and then compare their performances. To evaluate our classifier, we use precision, recall, and F1-score as metrics as well.

Table 3.5 shows the results. It is easy to know that the proposed method outperforms FOSSology. Especially, both the proposed method and FOSSology achieve 100% recall, which suggests that both two methods do not miss any copyright notice. This is important because identifying all copyright notices is an important metric to evaluate the automatic copyright notice identification. However, the proposed method outperforms FOSSology by achieving 100% precision. Specifically, 152 sentences that are not copyright notices are wrongly identified as copyright notices by FOSSology. The results suggest the effectiveness of the proposed method in filtering out sentences that are not copyright notices.

3.5 Conclusion of This Chapter

In this chapter, we proposed a machine learning method for automatic copyright notice identification of source files. The results of experiments suggest that Decision Tree and Random Forest perform best on automatic copyright notice identification of source files, and the proposed machine learning method outperforms the existing method. Our work highlights the possibility of applying machine learning method to solve software copyright-related issues, and also creates a possibility of studying the copyright notices in FOSS projects, which is overlooked by the software engineering researchers. In our future work, we will implement a tool to provide the copyright notice identification service to end-users, and also test the proposed method on a larger scale. We also plan to study the copyright notice issues in FOSS projects, such as the reliability of the copyright notices in source files.

Chapter 4

Study on Dependency-related License Violation in the JavaScript Package Ecosystem

4.1 Introduction

Software reuse has long been proved to be a good method to increase software productivity [11, 56, 64]. As a practice, reusing open source software (OSS) has become more and more popular.

Open source license describes the terms and conditions when OSS is used, modified, and shared. OSS should be distributed under one or multiple open source licenses so that it can be reused by others. These licenses are usually included in the header comments of source files. To standardize the use of open source licenses, the Open Source Initiative (OSI) determines the definition of open source licenses and publishes the list of all approved licenses[1]. When developers reusing OSS, they should pay special attention to open source licenses to prevent potential legal risks [80].

As the definition of open source license describes, OSS can only be reused as long as the particular terms and conditions are satisfied. Therefore, the developed software should satisfy all the terms and conditions in licenses of all reused OSS. In other words, the developed software should select a license that does not violate the licenses of all reused OSS. In this chapter, we define *license A violates license B* as the situation that the software licensed under license B can not be combined into the software licensed under license A according to their terms and conditions. If the

[1] `https://opensource.org/licenses/alphabetical`

selected license violates any license of the reused OSS, potential legal risks may occur.

OSS ecosystem consists of software projects that are developed and evolve together in a shared environment [54]. The user-contributed OSS ecosystem is an ecosystem where software projects are contributed by its users. When a developer develops a new software project, other software projects in the user-contributed OSS ecosystem can be reused easily by being declared as dependencies. In this chapter, a *dependency* of a package refers to the other packages in the user-contributed OSS ecosystem used by this package. Meanwhile, this package is defined as a *dependent*. Generally, the dependency can also refer to the dependency relation between the packages [18], but note that in this chapter we define dependency as the "package" instead of the "relation". Introducing dependencies makes a project dependent on them.

The license of the new software project under development should not violate the licenses of any dependencies as well. Dependency-related license violation occurs when the selected license violates any license of the dependencies. In a user-contributed OSS ecosystem, it is more difficult to judge whether the selected license violates the licenses of its dependencies or not.

To address the dependency-related license violation issue, in this chapter we propose an approach to detect dependency-related license violations of software projects in such OSS ecosystems. We are interested in studying the prevalence of dependency-related license violation in user-contributed OSS ecosystems and the developers' attitudes.

We select `npm`[2] as the target in our research. `npm` serves as a large repository of JavaScript-based software packages. It hosts over 1.3 million JavaScript packages and becomes the largest JavaScript ecosystem, with millions of packages being installed from the `npm` repository on an everyday basis. We select `npm` for 3 reasons: (1) `npm` is one of the most popular and successful OSS ecosystems and hosts a large number of packages. (2) `npm` has a perfect mechanism of including other packages as dependencies, which make the usage of dependencies prevalent in `npm`. (3) `npm` has a strict requirement of adding meta-data, for which we can utilize the meta-file of packages conveniently. These features make `npm` a suitable target to study dependency-related license violations. Studying dependency-related license violations in such typical OSS ecosystems will benefit the practitioners. The findings will also be a good baseline to study dependency-related license violations in other OSS ecosystems.

Our research questions are set as follows:

RQ1: How prevalent are dependency-related license violations in `npm`?

RQ2: What are the developers' attitude towards dependency-related license violation?

[2] `https://www.npmjs.com`

Many studies in software engineering have been done on the software license. Some effective approaches and tools are proposed to identify the license of source code files automatically [27, 32]. Some works aim to detect and understand the license violation in code siblings or similar files [26, 78], but no research has been done to understand the license violation occurring in the dependencies of packages in the OSS ecosystem.

The contributions of this study are as follows.

1. An empirical study on `npm` to understand the prevalence of dependency-related license violation with the proposed method of detecting dependency-related license violation.

2. A preliminary questionnaire on the authors of packages detected as having dependency-related license violation to reveal the developers' attitudes.

This chapter is organized as follows. Section 4.2 first provides a brief background on dependency-related license violation. Our empirical study on `npm` is described in Section 4.3, followed by Section 4.4 with the preliminary questionnaire. Section 4.5 describes threats to validity. After a description of related work in Section 4.6, Section 4.7 concludes this chapter.

4.2 Background

4.2.1 License Violation

A software license permits software to be reused under certain terms and conditions. An open source license is a software license that follows Open Source Definition[3] and is approved by Open Source Initiative. Here is an example of a license statement abstracted from `grunt` package in `npm`, which states that the file is licensed under MIT license:

```
...
Permission is hereby granted, free of charge, to any
person obtaining a copy of this software and associated
documentation files ( the "Software" ) , to deal in the
Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software,
and to permit persons to whom the Software is furnished to
do so, subject to the following conditions:

The above copyright notice and this permission notice shall
be included in all copies or substantial portions of the
```

[3] https://opensource.org/definition

```
Software.
...
```

Licenses can be basically grouped into two types - the permissive license and copyleft (protective) license. Some examples of the permissive license are MIT License, BSD licenses, and Apache license. A typical example of the copyleft ones is the GNU General Public License. When OSS reuses another OSS, if the reused OSS is licensed under a permissive license, the developed OSS does not need to open its source code. But if the reused OSS is licensed under a copyleft one, the developed OSS is enforced to open its source code. Usually, a permissive license violates a copyleft one.

For example, MIT license[4], which is a permissive license, violates GPL-2.0+ license[5], which is a copyleft one. GPL-2.0+ license declares:

```
...
This program is free software; you can redistribute it
and/or modify it under the terms of the GNU General Public
License as published by the Free Software Foundation;
either version 2, or (at your option) any later version.
...
```

According to the terms of GPL-2.0+ license, if OSS licensed under MIT license reuse another OSS licensed under GPL-2.0+ license, illegal reuse occurs since OSS licensed under GPL-2.0+ license can only be redistributed and/or modified under the terms of the GNU General Public License either version 2, or (at your option) any later version as published by the Free Software Foundation.

Furthermore, some licenses share the same name but are with different versions. An example is the GNU General Public License. GNU General Public License has versions 1, 2, and 3. Each version has different terms and conditions. According to these different terms and conditions, three versions violate each other. Usually, licenses share the same name are called a license family. For example, the GPL family includes GNU General Public License versions 1, 2, 3, and some other versions and MPL family includes Mozilla Public License versions 1, 1.1, 2.0, and some other versions.

4.2.2 Package Dependency

Introducing other packages in the user-contributed OSS ecosystem makes the package under development dependent on them. At the same time, the introduced packages may have their dependencies, seen as the indirect dependencies of the package under development. In this chapter, a *direct dependency* of package A refers to package B whose name is explicitly recorded in package A's meta-data as its dependency. The *indirect*

[4] https://opensource.org/licenses/MIT
[5] https://opensource.org/licenses/GPL-2.0+

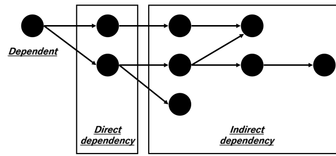


Figure 4.1: The examples of direct dependency and indirect dependency.

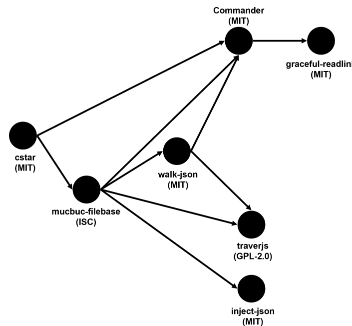


Figure 4.2: A part of the dependency network of `cstar` package in `npm`.

dependencies of package A refer to the transitive collection of package A's dependencies excluding package A's direct dependencies. Figure 4.1 shows examples of direct dependency and indirect dependency.

Finally, direct dependencies and indirect dependencies could form a dependency network. The dependency network is prevalent in the user-contributed OSS ecosystem [18]. For example, the number of dependencies of packages in `npm` has grown 60% over 2016 and the mean number of the dependencies per package has grown to 60.1 [42]. Figure 4.2 shows an example of the dependency network of `cstar` package.

Developers are not likely to update the dependencies of their packages [8, 49]. Developers might also overlook the indirect dependencies since they do not include those dependencies by themselves [42]. These problems harm the health of the OSS ecosystem, causing problems such as security vulnerability, API breaking conflicts, and illegal reuse.

4.2.3 Dependency-related License Violation

License violation related to dependency occurs when a package's license violates any license of its dependencies. To achieve our research goal, we will define dependency-related license violation.

We first construct a license compatibility network to specify the compatibility between two licenses. Our license compatibility network is constructed based on the one created by David A. Wheeler [75]. Our license

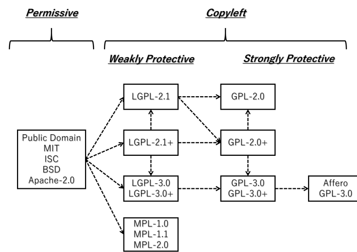


Figure 4.3: License compatibility network.

compatibility network is shown in Figure 4.3. Each arrow denotes one-directional compatibility. An arrow pointing from license A to license B means that license B does not violate license A. In other words, software licensed under license A is allowed to be combined into the software licensed under license B according to the proposed definition of violation in Section 4.1.

In this chapter, we define *dependency-related license violation* of a package X licensed under license A for another package Y licensed under license B, as the situation that Y is a dependency of X but no path from B to A exists in the license compatibility network (i.e. A violates B).

For example, `walk-json` package licensed under MIT license has dependency-related license violation for `traverjs` package licensed under GPL-2.0 license since `traverjs` package is a dependency of `walk-json` package, as shown in Figure 4.2, and there is no path from GPL-2.0 license to MIT license in the license compatibility network (i.e. MIT license violates GPL-2.0 license), as shown in Figure 4.3. This is an example of dependency-related license violation caused by direct dependency.

An example of dependency-related license violation caused by indirect dependency is `cstar` package. `cstar` package has two direct dependencies. The MIT license does not violate the licenses of these two direct dependencies. However, `traverjs` package licensed under GPL-2.0 license is also reused by `cstar` package as an indirect dependency. Because there is no path from GPL-2.0 license to MIT license in the license compatibility network, dependency-related license violation occurs in `cstar` package as well.

Note that our detection method proposed in Section 4.3.2 only detects dependency-related license violation using the dependency network and the license compatibility network constructed in this chapter. Although we can define license violation in different ways, we use the above definition for simplicity and clarity of the implementation.

4.3 The Prevalence of Dependency-related License Violation

In this section, we aim to answer RQ1.

We first introduce the proposed method and then report the results of our empirical study.

4.3.1 Data collection

We collect the target packages in `npm`. The observation period is from October 1st, 2010 to April 7th, 2017. We end up with 419,708 packages in total. Note that for each package, we only count it once no matter how many versions it has. We use the public API[6] of `npm` to get the historical meta-data of all versions of the target packages.

For each version, the license and direct dependencies are recorded. For example, a part of the historical meta-data of `cstar` package is shown as follows:

```
...
        "version": "0.0.5",
        "dependencies":
        "commander": "*","mucbuc-filebase": "*",
        "license": "MIT",
...
        "version": "0.0.2",
        "dependencies":"mucbuc-filebase": "*",
        "license": "MIT",
...
```

4.3.2 Method

It is not easy to detect dependency-related license violations of a package in the user-contributed OSS ecosystem. There are three main challenges:

1. A dependency is not always with the latest version, since packages in the user-contributed OSS ecosystem usually evolve frequently.

2. Using dependencies is very common in the user-contributed OSS ecosystem. As a result, a package has a high probability of having a deep and complex dependency network.

3. The same license is not always written in the same ways. For example, both GPL-2.0 and GPL version 2 refer to the same license.

[6] `https://registry.npmjs.org/-/all`. Note that for some reason, `npm` has stopped providing this public API. Therefore, we can not get the historical meta-data by using this API now.

To address these issues, we proposed a method that consists of five steps.

1) Constructing the license dictionary:

As mentioned above, the same license is usually written in different ways by different developers. We first construct a license dictionary, with which we can transform different forms of a license into a normalized one. The license dictionary includes the popular licenses published by the Open Source Initiative.

We collect all licenses written in the historical meta-data of the collected packages in `npm` and remove the duplicate one. We then use regular expression matching to do a preliminary classification. Regular expression matching is able to classify most licenses. We then manually check the results and move the license wrongly classified into the correct one. For the licenses which can not be matched by the regular expression, we manually classify them into the correct category. Note that since the license is written by the developer manually, some developers may not write the version number of the license. For example, some developers declare their packages are licensed under "GPL", but no version is declared. For these cases, we record them as "license no version" such as "GPL no version" and "MPL no version". "License no version" is seen as not violating any license in its family, but if a license violates any license in the family of "license no version", it will be detected as violating "license no version". For licenses that are not published by the Open Source Initiative, we classify them into a special category named "unknown" license.

All the normalized forms of our selected licenses are listed in Table 4.1. By constructing the license dictionary, we succeed in solving the third challenge - the various ways of writing the same license.

Table 4.1: The list of the selected licenses.

License	**Normalized form**	**License family**
Public Domain License	Public Domain	None
MIT License	MIT	None
ISC License	ISC	None
Apache License 2.0	Apache-2.0	None
3-clause BSD License/"New" or "Revised" license	BSD-3-Clause	BSD family
2-clause BSD License/"Simplified" or "FreeBSD" license	BSD-2-Clause	BSD family
Mozilla Public License version 1.0	MPL-1.0	MPL family
Mozilla Public License version 1.1	MPL-1.1	MPL family
Mozilla Public License version 2.0	MPL-2.0	MPL family
GNU General Public License version 2	GPL-2.0	GPL family
GNU General Public License version 2 or any later version	GPL-2.0+	GPL family
GNU General Public License version 3	GPL-3.0	GPL family
GNU General Public License version 3 or any later version	GPL-3.0+	GPL family
GNU Lesser General Public License version 2.1	LGPL-2.1	GPL family
GNU Lesser General Public License version 2.1 or any later version	LGPL-2.1+	GPL family
GNU Lesser General Public License version 3.0	LGPL-3.0	GPL family
GNU Lesser General Public License version 3.0 or any later version	LGPL-3.0+	GPL family
GNU Affero General Public License version 3	AGPL-3.0	GPL family

2) Constructing the license compatibility network:

We implement the license compatibility network proposed in Section 4.2.3. By searching this license compatibility network, we can check whether or not a license violates another one.

3) Constructing the historical meta-data dataset:

OSS in the user-contributed OSS ecosystem usually evolves frequently. With the evolution of software, source code, license, and dependencies are also changing. However, OSS does not always reuse the latest version of its dependencies, thus deciding the proper version of dependencies is important. To accelerate the detection, we construct the historical meta-data dataset, in which the license and the direct dependencies with the proper versions of all historical versions of each package are recorded.

We have collected the historical meta-data of all versions of the target packages. However, it is necessary to process historical meta-data. Firstly, for licenses, we normalize them according to the license dictionary constructed in step 1. Secondly, for direct dependencies, we list all direct dependencies and choose the proper version for each direct dependency. Note that `npm` uses semantic versioning standard to manage versions. The first release should start from 1.0.0. After this, changes should be handled according to Table 4.2. Developers can specify acceptable version ranges of the dependencies of their packages based on the semantic versioning standard used by `npm`. The specification is flexible by using different symbols to achieve different goals. Table 4.3 shows some examples of how to specify the ranges. Note that we identify the version of a dependency as the latest version in the acceptable version ranges recorded in the meta-data of a package, which is consistent with the mechanism of `npm`. The details of the semantic versioning standard used by `npm` can be found in the documents of `npm`[7].

[7] `https://docs.npmjs.com/getting-started/semantic-versioning`

Table 4.2: The rules of how changes should be handled in `npm`.

CODE STATUS	STAGE	RULE	EXAMPLE
First Release	New Product	Start with 1.0.0	1.0.0
Bug fixes, other minor changes	Patch Release	Increment the third digit	1.0.1
New feature that don't break existing features	Minor release	Increment the middle digit	1.1.0
Changes that break backward compatibility	Major release	Increment the first digit	2.0.0

Table 4.3: The examples of how to specify the ranges.

TYPE	EXAMPLE
All patch releases of major release 1.0	1.0 or 1.0.x or ~1.0.0
All minor releases of major release 1	1 or 1.x or ^1.0.0
All releases	* or x

Table 4.4: The historical meta-data dataset constructed for `cstar` package in `npm`.

Version	License	Direct dependency (Version)
0.0.5	MIT	commander (2.9.0), mucbuc-filebase (0.0.4)
0.0.4	MIT	commander (2.9.0), mucbuc-filebase (0.0.4)
0.0.3	MIT	mucbuc-filebase (0.0.4)
0.0.2	MIT	mucbuc-filebase (0.0.4)
0.0.1	MIT	mucbuc-filebase (0.0.4)
0.0.0	GPL-2.0	mucbuc-filebase (0.0.4)

After we process the historical meta-data, we can build the historical meta-data dataset for this package. The historical meta-data dataset includes information on license and direct dependencies of all versions of a package. Table 4.4 shows the historical meta-data dataset constructed for `cstar` package in `npm`.

4) Constructing the dependency network: In this step, we construct the dependency network. The direct and indirect dependencies in the dependency network are attached to its version and license. Figure 4.2 shows a part of the dependency network constructed for `cstar` package in `npm`. For each dependency in the dependency network, the attached information of version and license is shown in Table 4.5. Note that the version of the `cstar` package is the latest version. The version of the direct dependency consists with the dependency's version recorded in the meta-data of the `cstar` package. The version of indirect dependency is selected in a similar way.

By constructing the dependency networks for the packages in `npm`, we succeed in solving the first and the second challenges mentioned above - the variability of the dependencies' versions and the complexity of the dependency networks.

5) Dependency-related license violation detection:

Since we have constructed the dependency network and the license compatibility network, it becomes possible to detect dependency-related license violations. We detect whether or not the license of this package violate the licenses of direct and indirect dependencies in the dependency network ac-

Table 4.5: The attached information of version and license for the dependencies in the dependency network constructed for `cstar` package.

Dependency	**Version**	**License**
cstar	0.0.5	MIT
mucbuc-filebase	0.0.4	ISC
walk-json	0.0.2	MIT
commander	2.9.0	MIT
graceful-readlink	1.0.1	MIT
traverjs	0.0.7	GPL-2.0
inject-json	0.0.9	MIT

cording to the license compatibility network. Dependency-related license violation is detected when the violation is found.

For example, by observing the dependency network constructed for `cstar` package (Figure 4.2 and Table 4.5), we can find that MIT license - the license of `cstar` package - violates GPL-2.0 license - the license of `traverjs` package which is an indirect dependency. Therefore, `cstar` package is detected as having a dependency-related license violation.

4.3.3 Results and discussion

We detect dependency-related license violations in the collected 419,708 packages in `npm`. As a result, only 2,704 packages are detected as having dependency-related license violations, accounting for 0.644% of all packages. The result suggests that only a few packages (0.644%) in `npm` have dependency-related license violations. We also observe that among these 2,704 packages, 3,624 dependency-related license violations are detected. Table 4.6 shows the top 10 list of those violations classified by licenses. The result shows that most dependency-related license violations are caused by the violation between the permissive licenses and copyleft licenses, while usually copyleft licenses do not violate each other.

A potential reason is the developers' manner of choosing licenses for their packages. To ascertain this reason, we count the proportion of the different licenses in `npm`. Table 4.7 shows the result. We observe that the permissive licenses take a large part of all licenses while the copyleft licenses are not widely used in `npm`. The preference of the permissive licenses may be the reason for the low proportion of the packages detected as having dependency-related license violations in `npm`.

Furthermore, because of the high proportion of the permissive licenses and low proportion of the copyleft licenses used in `npm`, we could assume that including the packages licensed under copyleft licenses in the dependency network highly potentially causes dependency-related license viola-

Table 4.6: The top 10 dependency-related license violations.

Package	Dependence	Number	Proportion
MIT	GPL-3.0	582	16.1%
MIT	LGPL-3.0	349	9.6%
Public Domain	GPL-3.0	300	8.3%
MIT	LGPL no version	281	7.8%
ISC	GPL-3.0	231	6.4%
MIT	GPL no version	224	6.2%
MIT	LGPL-2.1	159	4.4%
MIT	GPL-2.0	134	3.7%
Public Domain	GPL no version	114	3.1%
Public Domain	LGPL-3.0	104	2.9%

Table 4.7: The proportion of the selected licenses in `npm`.

License	Number	Proportion
MIT	254,972	60.75%
None	80,331	19.14%
ISC	33,827	8.06%
Apache-2.0	13,598	3.24%
BSD family	17,794	4.24%
GPL family	9,158	2.18%
Unlicense/Public Domain	2,098	0.50%
MPL family	1,132	0.27%
Unknown	6,798	1.62%
All	419,708	100%

tion. To ascertain it, we selected GPL family licenses as the target to study. We collected the packages having direct or indirect dependencies with GPL family licenses. As a result, we collected 4,067 packages. Among them, 2,704 packages are detected as having dependency-related license violations, accounting for 66.84%. Note that all packages detected as having dependency-related license violations in the detection of 419,708 packages are included in these 4,067 packages. The result proves our assumption. A possible explanation is that the developers do not understand the copyright notice well and overlook the license violation when they use a package as a dependency. Another possible explanation is that the developers overlook the indirect dependencies since they do not include those packages by themselves. To ascertain it, we calculate the proportion of the dependency-related license violation caused by direct and indirect dependency respectively. As a result, among 2,704 packages detected as having

dependency-related license violations, 2,115 packages are detected as having violations caused by direct dependencies, accounting for 78.2%. Meanwhile, 1,507 packages are detected as having violations caused by indirect dependencies, accounting for 55.7%. The result suggests that both direct dependency and indirect dependency play an important role in the occurrence of dependency-related license violations.

We will study more on developers' attitudes in Section 4.4.

Hence, we answer RQ1:

> Only a few packages (0.644%) in `npm` have dependency-related license violations. However, including the packages licensed under copyleft licenses as dependency is highly related with the occurrence of dependency-related license violations.

4.4 Preliminary Questionnaire

In this section, we aim to answer RQ2.

We first introduce the preliminary questionnaire we conducted and then report the results.

4.4.1 Questionnaire Design

Our preliminary questionnaire targets the authors of packages detected as having dependency-related license violations. Therefore, our questionnaire first describes how the license of the target package violates the licenses of its dependencies by explaining the terms and conditions of the licenses. The second part of the questionnaire then asked developer opinions on the following two questions: *1) Do you think this is a kind of risk? If so, were you aware of this kind of risk when you are developing your packages? 2) In this question, you can share anything you want to say with this kind of risk with us.*

For the analysis, we first record the responses of the first question according to whether or not the developer thinks the dependency-related license violations in their packages as issues. We then analyze the responses of the second question through a systematic method: (i) reading of each response, (ii) checking, summarizing and categorizing text, and (iii) looking for similarities or differences in other responses. We finally end up with a list of important observations in the responses.

4.4.2 Data collection

In Section 4.3, 2,704 packages are detected as having dependency-related license violations. We randomly select 100 packages and send their authors

email invitations for our preliminary questionnaire. In the end, we received 20 responses which equals a response rate of 20%.

4.4.3 Results and discussion

Question 1

For the first question in our questionnaire, 11 participants out of 20 participants regard the dependency-related license violations as risks. On the contrary, 6 participants do not regard the dependency-related license violations as risks and 3 participants are not sure whether or not the dependency-related license violations are risks respectively. The results suggest the divergence of developers towards dependency-related license violations. To study the reasons why developers regard or do not regard the dependency-related license violations as risks, we also analyze the developers' explanations for their choices.

Among 11 participants regarding the dependency-related license violations as risks out of all 20 participants, 5 participants were aware of the dependency-related license violation issue when they were developing their packages while 6 participants were not. Only one participant out of 6 participants who was not aware of this issue explains the reason: as an individual developer, this participant will not take time to properly study the licenses in dependencies, especially in the case that the package is not developed for commercial use. Meanwhile, 4 participants out of 5 participants who were aware of this issue explain the reasons: 1) Developers will not check the licenses of the dependencies for personal projects. 2) The project is weakly maintained so the license violation issue is not addressed carefully. 3) The developer is the authors of both the detected package and its dependency. 4) The developer knows the license violation issue but is not well informed. The results suggest that the developers potentially overlook the dependency-related license violations even they regard them as risks.

All 6 participants who do not regard the dependency-related license violations as risks out of all 20 participants explain their reasons. There are two main reasons: 1) There is no need to care about the license violation issue if the package is not developed for commercial use. 2) Reusing other packages as dependencies is not "redistribution".

The first reason is obviously wrong since the terms and conditions in licenses are effective no matter the package is not developed for commercial use or not. For the second reason, judging whether or not reusing other packages as dependencies is "redistribute" is a complicated legal problem and may differ in different countries. However, even reusing other packages as dependencies is not regarded as "redistribution", it could potentially

become a risk for the end-users who will reuse these packages for various purposes.

The results reveal the developers' overlooking and misunderstanding of the dependency-related license violations.

Question 2

We analyze the responses of the second question through a systematic method and observe the following important observations:

1) It is not easy to understand the terms and conditions of the licenses. Most participants describe their understanding of the terms and conditions of the licenses. For the same license, participants may have different understanding. The most important problem is understanding the meaning of particular words in the licenses such as "redistribute", "reuse", and "modify". It is also difficult for developers to identify the difference between two licenses in the same license family, such as "GPL-2.1" and "GPL-3.0". The difficulties in understand the terms and conditions also hinder developers in choosing the proper licenses.

2) Managing dependency-related license violations is difficult in practice. Some participants describe the difficulty in managing dependency-related license violations in practice. One participant says that a lot of open source components are reused in his project, and the collection of dependencies grows and changes during the life cycle of the project. He usually generates reports on the licensing requirements for dependencies manually and thus it takes a lot of time. Another participant says that he always has to re-write his packages because of the license of the dependencies. Note that the licenses of the dependencies may change during the life cycle and it is difficult to trace [49]. The dependency-related license violations may also occur when a developer is required to change the license of his package to meet the requirement of the end-users.

3) Help in managing the dependency-related license violations is demanded. Some participants agree that tools that help to manage the dependency-related license violations will help a lot. The following functions are wanted: 1) detecting the dependency-related license violations; 2) tracing the change of the licenses of the dependencies; 3) choosing the proper license.

The results highlight the difficulties in managing dependency-related license violations and the developers' demands for help.

Hence, we answer RQ2:

> The attitudes of developers towards dependency-related license violations vary. The dependency-related license violations are overlooked and misunderstood by the developers for various reasons. Managing

dependency-related license violations is difficult and the developers are demanding help.

4.5 Threats to Validity

This section discusses the threats to the validity of our research. Threats to construct validity concern the relationship between theory and outcome, and relate to possible measurement imprecision when extracting data we used in this study. `npm` hosts over 1.3 million JavaScript packages. This number is still increasing rapidly every day. It is impossible to conduct the empirical study on packages of this large number, especially considering that the packages depend on each other. Otherwise, because of the rapid increment of the packages in `npm`, the public API we use to get the historical meta-data is not accessible. The collected data is the only one we can use to study dependency-related license violations in `npm`. We set the observation period from October 1st, 2010 to April 7th, 2017, and collect the meta-data of 419,708 packages finally. 419,708 packages are enough to represent all packages in `npm` statistically. Furthermore, these 419,708 packages only depend on each other and do not depend on other packages in `npm`, which makes the empirical study executable.

Threats to internal validity concern factors internal to the study that could impact our results. Such threat does not affect exploratory study like the one in this chapter. The only case worthwhile being discussed is our answering to RQ2, where we observe and understand the responses manually based on our knowledge.

Threats to external validity are related to the ability to generalize the findings in our study. Our empirical study is only conducted on `npm`. `npm` is one of the most popular and successful OSS ecosystem, which makes itself a suitable target to study copyright inconsistencies. However, we find that the developers in `npm` do not prefer to choose copyleft licenses for their packages, which is a possible reason for the low proportion of packages having dependency-related license violations. But according to the different situations, other OSS ecosystems may have different results. As the first work focusing on dependency-related license violations in OSS ecosystems, our findings could be a good baseline to study dependency-related license violations in them. We agree that it is necessary to replicate our empirical study on different OSS ecosystems.

Another threat worthwhile being discussed is the scale of our preliminary questionnaire. Our preliminary questionnaire only selects 100 packages from 2,704 packages detected as having dependency-related license violations. Compared with the large numbers of packages and developers

in `npm`, the scales of the target packages and developers are not enough to achieve a result with statistical sense. However, as a preliminary qualitative analysis of dependency-related license violations, it can still reveal developers' attitudes, some of which are important. But we also agree that a large-scale developer survey is necessary in our future works.

4.6 Related Work

4.6.1 Software License

Many studies in software engineering investigated software license. There are some studies that devote to identify licenses [27, 32, 67]. Based on these studies, some researchers analyzed software licenses in open source projects and revealed some license issues. Di Penta et al. [20] provided an automatic method to track changes occurring in the licensing terms of a system and did an exploratory study on license evolution in six open source systems and explained the impact of such evolution on the projects. German et al. [28] proposed a method to understand licensing compatibility issues in software packages. They mainly focused on the compatibility between licenses declared in packages and those in source files. Different from their work, we mainly focused on the compatibility between licenses declared in packages and its dependencies in this chapter. In another research by Di Penta et al. [26], they analyzed license inconsistencies of code siblings (a code clone that evolves in a different system than the code from which it originates) between Linux, FreeBSD, and OpenBSD, but they did not explain the reasons underlying these inconsistencies. Wu et al. proposed an approach to find license inconsistencies in similar files [78]. By investigating the revision history of these files, they summarized the factors that caused these license inconsistencies and tried to decide whether they are legally safe or not. [26] and [78] focused on the license violations on the source file level and source code level, while we focused on the package level. [26] analyzed license inconsistencies of code siblings between Linux, FreeBSD, and OpenBSD. [78] analyzed license inconsistencies in Debian and concluded the reasons why license inconsistencies occurred. Alspaugh et al. [3] proposed an approach for calculating conflicts between licenses in terms of their conditions. Vendome et al. [71] performed a large empirical study of Java applications and found that changing license is a common event and a lack of traceability between when and why the license of a system changes. Vendome et al. performed a study on GitHub and found that developers adopt a license may depend on various factors and they discovered the lack of traceability of when and why licensing changes are made and highlighted the need for better tool to support in guiding developers in choosing and changing licenses and in keeping track of the rationale of license changes [72].

4.6.2 License Compliance

License compliance is an important area of research that draws attention from many researchers. Zhang et al. have developed a tool named LCheck that utilizes Google Code Search service to check whether a local file exists in an OSS project and whether the licenses are compatible [80]. German et al. proposed a tool named Kenen that checks license compliance for Java components that uses component identification, provenance discovery, license identification, and licensing requirements analysis [25]. Van et al. proposed an approach that can uncover license compliance inconsistencies by analyzing the Concrete Build Dependency Graph of a software system [69]. They proposed an approach to construct and analyze the Concrete Build Dependency Graph of a software system by tracing system calls that occur at build-time. Kapitsaki et al. proposed an approach of automating license compliance with a process that examines the structure of Software Packages Data Exchange [41]. Different from the above works, we mainly focused on dependency-related license violations in OSS ecosystems. We utilize the meta-data of the packages to detect license violations, which is an important characteristic of OSS ecosystems. Vendome et al. studied the rationale of developers in choosing and changing licenses and investigated the problem of traceability of license changes [70]. They provided a vision of ensuring license compliance of a system.

4.7 Conclusion of This Chapter

In this chapter, we propose a method to detect dependency-related license violations in OSS ecosystems, with which we conduct an empirical study on `npm` to study the prevalence of dependency-related license violations. The result suggests that only a few packages (0.644%) in `npm` have dependency-related license violations, but including the packages licensed under copyleft licenses in the dependency network still highly potentially causes dependency-related license violation. We also conduct a preliminary questionnaire on the authors of packages detected as having dependency-related license violations, revealing the developers' overlooking and misunderstanding of the dependency-related license violations, the difficulties in managing dependency-related license violations, and the developers' demands for help. Our work highlights the importance of the dependency-related license violation issue, and also creates a possibility of studying dependency-related license violations in OSS ecosystems, which is overlooked by the software engineering researchers.

In our future work, we plan to extend our study on dependency-related license violations to other OSS ecosystems. Furthermore, based on the results of the preliminary questionnaire, we also consider a new large-scale

developer survey as our future works. We also plan to implement some tools to help developers in maintaining licenses and managing license violations.

Chapter 5

Study on Popularity Growth of Packages in the JavaScript Package Ecosystem

5.1 Introduction

A software ecosystem consists of software projects that are developed and evolve together in a shared environment [54]. Open-source software (OSS) ecosystem is software ecosystem consisting of open-source software (OSS) software projects. Usually, these OSS software projects are contributed by the users of this OSS ecosystem. The development of software ecosystems has resulted in an abundance of free software packages that are easily reused by both new and existing projects. Also, this kind of software reuse has long been proved to be a good method to increase software productivity [11, 56, 64]. One example is `npm`, which serves as a large repository of JavaScript-based software packages. It hosts over 650,000 JavaScript packages to become the largest software ecosystem, with millions of packages being installed from the `npm` repository on an everyday basis. In such a large number of packages, some packages are significantly more attractive than others, being downloaded more times by end users. Software popularity is used to measure this difference. It is a useful indicator of whether a package is successful and is attracting and gaining acceptance in the software ecosystem [12]. Understanding the popularity growth of packages in OSS ecosystem is very important. Firstly, developers are continually wanting to know whether or not their software is attracting and gaining acceptance. Especially, the competition of packages in OSS ecosystem is becoming more and more fiercely nowadays. For example, it's reported by the homepage of `npm`[1] that about 200,000 new packages were uploaded onto

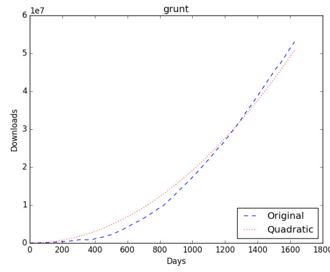
[1] https://www.npmjs.com/

npm during the last year, which account for nearly 30% of all the packages uploaded in the past 8 years. It's no doubt that the large and rapidly growing number of packages makes OSS ecosystem more competitive. Therefore, understanding the popularity growth will also help developers on improving their packages to survive in the competitive OSS ecosystem. Secondly, the software with the rapid growth of popularity is a double-edged sword for the OSS ecosystem and its reusers. On one hand, the vulnerabilities or defects involved in this software will propagate quickly in the OSS ecosystem with the rapid growth of its popularity. On the other hand, if the software with the rapid growth of popularity is removed from the OSS ecosystem, much other software that depends on this software will be impacted as well.

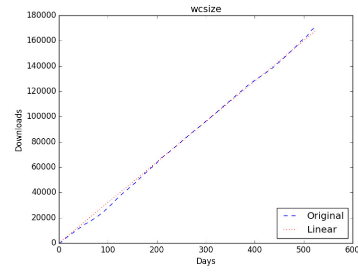
Studies of popularity are firstly conducted on the social platforms such as YouTube [1] and Twitter [53] aimed for recommendations on how to produce successful content. Unfortunately, to date, there have been few studies on software popularity growth. One exception is an effort on understanding the evolution of popularity by case study [77], but they only examine the top 5 popular packages in npm. Borges et al. aim to investigate common patterns of popularity growth of the projects on GitHub [12]. They use KSC algorithm [50] to achieve this, but due to the restriction of the algorithm, this approach can hardly be applied to a large and irregular dataset.

In this chapter, we would like to understand how fast packages become popular (defined as **popularity growth**). We propose that popularity growth over time can be modeled as a mathematical equation, and is plotted visually as a growth curve. Actually, this kind of application of the mathematical models is very common in many fields of biology, medicine, economics and the social sciences [4]. Especially, growth curve models have been used in various disciplines as well, for example in biological sciences to study crop growth, population processes, and bacterial growth. They are often estimated to understand defining characteristics, including initial levels, rates of change, periods of acceleration and deceleration, and final or asymptotic levels [34]. In software engineering, different models have been applied in the context of software reliability (e.g., [2], [79]) and modeling the evolution of library usage [47].

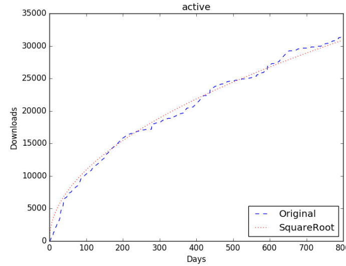
For an empirical evaluation of our popularity growth models, we conducted an exploratory study of packages in npm to understand: (1) the characteristics of popularity growth, and (2) the factors that could affect popularity growth. This chapter aims at answering two research questions: **(RQ1) Do packages in npm share common characteristics of popularity growth? If so, what are these characteristics?** The goal is to find different kinds of characteristics of popularity growth and the proportion of them. The answer to this question will provide a general view of how fast the popularity grows for packages in npm.



(a) accelerated growth model



(b) steady growth model



(c) decelerated growth model

Figure 5.1: The popularity growth for three packages representing the three proposed models. In each plot, the blue curve is the one created with the original data while the red curve is the model that fits best. Note that Figure 5.1(a) represents the accelerated growth model (`grunt`), Figure 5.1(b) represents the steady growth model (`wcsize`) and Figure 5.1(c) illustrates the best-fits for decelerated growth model (`active`).

(RQ2) Are there some factors which could affect the popularity growth of packages in `npm`? If so, what are the effects of these factors? This investigation can reveal the factors that could be utilized to accelerate popularity growth or to prevent the deceleration of it.

To answer RQ1, we first proposed a method of modeling popularity growth as a curve. To answer RQ2, we select some main factors including total downloads count, age, the number of contributors, dependencies, dependents, versions, and functionalities. We reveal the impact of these selected factors by examining their significant difference across proposed models. We also give some suggestions based on the results we observed.

5.2 Modeling Popularity Growth as a Curve

In this chapter, we are interested in the popularity growth of the packages in `npm`. Kula et al. have succeeded in modeling the evolution of library

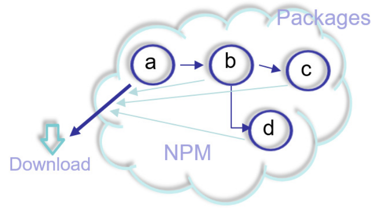


Figure 5.2: The effect of dependencies on downloads in `npm`. When package a is installed, package b, c and d is downloaded at the same time.

Table 5.1: Mathematical models

Model	**Equation**
accelerated growth model	$Popularity = at^2 (a > 0)$
steady growth model	$Popularity = at (a > 0)$
decelerated growth model	$Popularity = a\sqrt{t} (a > 0)$

usage as a curve to study the library aging [47]. Inspired by this work, we proposed a method of modeling popularity growth as a curve. We assume that popularity growth has the following three types: (1) grow slowly at first but accelerate over time, (2) grow steadily, with no sign of accelerating or decelerating, (3) grow rapidly at first but gradually decelerate.

We define these three kinds of growth as the accelerated growth model, steady growth model and decelerated growth model. When popularity growth is modeled as a curve, the accelerated growth can be represented by a convex while the decelerated growth can be represented by a concave. The convex on curve indicates the speed of growth is accelerating over time while the concave indicates the speed is decelerating. Similarly, the steady growth can be represented by a straight line, indicating that the speed is steady over time. Furthermore, we use three mathematical equations to represent the characteristics of the proposed three growth models. The linear growth model is the most commonly fit growth curve to describe a steady growth. For nonlinear change, many researchers turn to the quadratic growth model when a linear change model does not fit well or when a nonlinear trend is seen in the longitudinal plot [34]. When we limit the coefficient to be positive, the quadratic growth model becomes a good choice to describe an accelerated growth. Another reason we select quadratic equation is that its rate of growth increases slowly and gently. So if popularity growth has a tendency to accelerate, even not dramatically, it will still be well fitted by this equation. For the same reason, we select square root equation as the opposite equation to describe the decelerated growth.

To summarize, we use three models as shown in Table 5.1 and Figure 5.1 for our curve fitting. Thus, for the relationship between the popularity and time t, key characteristics of each model are described below:

- **Accelerated growth model**. The quadratic equation is depicted in Figure 5.1(a) as having a convex toward the lower right corner. The curve of this model indicates that popularity grows slowly at first but accelerates over time.
- **Steady growth model**. The linear equation is depicted in Figure 5.1(b) as having the single linear line and no convex and concave. It indicates that the popularity grows steadily, with no sign of accelerating or decelerating.
- **Decelerated growth model**. The square root equation is depicted in Figure 5.1(c) as having a concave toward the lower right corner. The growth of popularity fitting this model grows rapidly at first but gradually decelerates.

For the metric to measure popularity, we choose downloads count. Downloads count is incremented every time a package is installed from `npm` in any cases — redistribution, test or development. We use downloads count because it is a value to show how many times the package is downloaded, indicating how popular the package is used by end users intuitively. Note that, as shown in Figure 5.2, when a package is installed from `npm`, those packages which are in the dependency chain of this package are also installed. In this case, downloads count is incremented for all these packages. We obtained the historical data of downloads counts for each package on a daily basis through the web API of `npm`[2]. Finally, we accumulate the downloads counts to represent popularity growth.

5.3 Empirical Evaluation

5.3.1 Research Question 1

– **Research Method:**

Our research method comprises of two steps. In the first step, we need to collect empirical data that represents popularity and other main software ecosystem factors for `npm`. Table 5.2 shows a summary of collected `npm` packages. The observation period is from October 1st, 2010 to April 7th, 2017, and all data we collected only cover this range as well. At last, we collect 152,812 packages. Then we filter out packages whose age is

[2] source: https://api.npmjs.org/downloads/range/2010-10-1:2017-04-07/packageName

Table 5.2: Summary Statistics of the collected dataset

	Dataset statistics
observation period	2010-Oct to 2017-Apr
# packages	152,812
total size of projects	365 GB

Table 5.3: Best Fitting Results for the 102,341 target packages.

Model	**# Fitted**	**% of Studied Packages**
accelerated growth model	40,953	40.02%
steady growth model	52,769	51.56%
decelerated growth model	7,366	7.20%
Not Fit	1,253	1.22%
Total	102,341	100%

younger than 1 year or total downloads count is less than 1,000. This step ensures the reliability of the following curve fitting because the popularity growth of packages with few downloads or short lifetime is meaningless. For example, a package was published two days ago and downloaded once for each day. In this extreme case, this package will still be fitted by the steady growth model, which is not expected. We end up with 102,341 packages in total. Then for the second step, we run the experiment by which popularity growth is fitting against the three proposed growth models. Firstly, for the curve fitting, we rely on a Python-based package called `scipy`[3] to fit popularity growth against the proposed three growth models. Secondly, We use the widely-used the coefficient of determination [55], denoted by R^2, to evaluate the goodness of fit for each growth model. The one with the largest R^2 value among three proposed models is determined as the best-fitted model. Additionally, no matter which model is determined as the best-fitted model, the R^2 value of it must be larger than 0.7. The value is decided as 0.7 because if the R^2 value is larger than 0.7, this value is generally considered strong effect size [17]. For the case that all R^2 values of three proposed models are less than 0.7, the package is not well fitted by any models. By this, we ensure that the best-fitted model fits the popularity growth curve enough well. Meanwhile, we can also examine whether our three proposed models are effective or not. If a large number of packages could not be well fitted by any proposed models, our three proposed models are obviously not good enough to model popularity growth.

[3]https://www.scipy.org/

– **Findings:**

Table 5.3 lists the percentage of packages that are best-fit by each model. Notice that the percentage of not fit is only 1.22%. It suggests that only a small part of all packages could not be fitted by any proposed growth model. The percentage proves that our method is effective in modeling the popularity growth of packages. The result shows that 51.56% of the studied packages depict steady growth model, followed by accelerated growth model and decelerated growth model, 40.02% and 7.20% respectively. The most important finding is that only 7.20% of the studied packages are best-fitted by decelerated growth model, which indicates that `npm` is still very active and the number of packages installed from `npm` is still growing with no sign of deceleration. Specifically, 40.02% of the studied packages are best-fitted by accelerated growth model, which interprets that a large number of packages in `npm` are gaining popularity in accelerating speed. The result suggests that the reuse of packages in `npm` is still active, with more and more packages being installed from the `npm`.

Hence, we answer RQ1:

Packages in `npm` do share common characteristics of popularity growth. Specifically, 51.56% of the studied packages depict steady growth model, followed by accelerated growth model (40.02%) and decelerated growth model (7.20%). The distribution suggests that the reuse of packages in `npm` is still active.

5.3.2 Research Question 2

Table 5.4: Summary Statistics of the 101,088 fitted packages.

	Fitting Model	Min.	1st Qu.	Median	3rd Qu.	Max.	Mean	p-value
	accelerated growth model	1001.0	3091.0	8312.0	48718.0	800785605.0	2505140.14	
# Total downloads	steady growth model	1001.0	1511.0	2430.0	4876.0	92901071.0	49305.13	1.22e-09
	decelerated growth model	1001.0	1619.0	2820.0	6039.75	15443288.0	24208.82	
	accelerated growth model	366.0	759.0	1179.0	1507.0	17263.0	1169.85	
Age (# days)	steady growth model	366.0	585.0	781.0	1017.0	2300.0	817.89	1.12e-54
	decelerated growth model	366.0	423.0	533.0	796.0	2291.0	640.79	
	accelerated growth model	0.0	1.0	1.0	1.0	134.0	1.53	
# Contributors	steady growth model	0.0	1.0	1.0	1.0	64.0	1.27	0.63
	decelerated growth model	0.0	1.0	1.0	1.0	55.0	1.29	
	accelerated growth model	0.0	1.0	2.0	4.0	106.0	3.79	
# dependencies	steady growth model	0.0	1.0	3.0	5.0	114.0	4.24	0.24
	decelerated growth model	0.0	2.0	3.0	6.0	122.0	4.81	
	accelerated growth model	0.0	1.0	3.0	10.0	32130.0	38.13	
# dependents	steady growth model	0.0	1.0	1.0	2.0	3456.0	3.31	9.38e-49
	decelerated growth model	0.0	1.0	1.0	2.0	61.0	2.29	
	accelerated growth model	0.0	3.0	6.0	13.0	4204.0	12.95	
# versions	steady growth model	0.0	2.0	3.0	7.0	742.0	6.55	0.005
	decelerated growth model	0.0	4.0	7.0	14.0	1813.0	12.58	

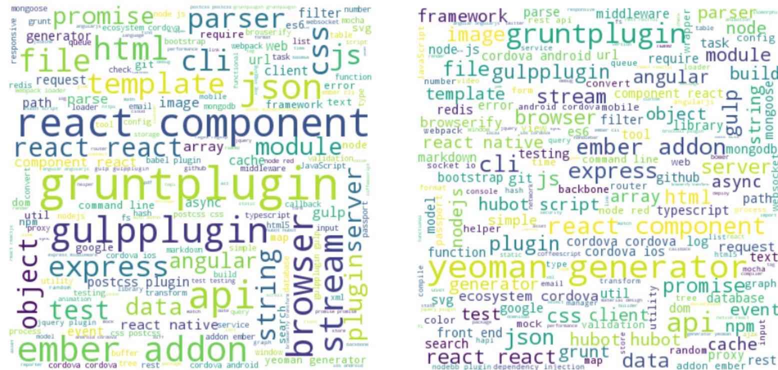
The result from RQ1 indicates that most studied packages depict steady growth model and accelerated growth model while only a few packages depict decelerated growth model. Hence our motivation for RQ2 is to make use of a quantitative approach to examine the effect of some main software ecosystem factors on popularity growth.

– **Research Method:**

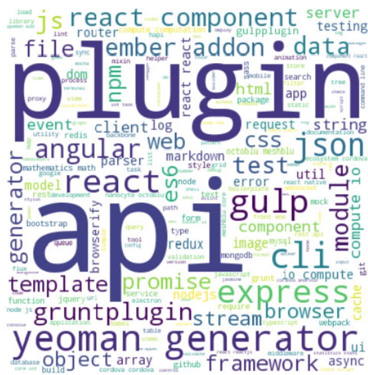
To solve this research question, we select some main software ecosystem factors and examine whether or not these factors are significantly different among the packages fitted by three proposed growth models. By this way, we aim to find the impact of these selected factors on popularity growth. The main software ecosystem factors we selected include *total downloads count, age* and *the number of contributors, dependencies, dependents, versions.*

The study on total downloads count could interpret if the packages with accelerated popularity growth are also the popular ones. Note that the downloads counts are accumulated on a daily basis. So the downloads count of the last day is the total downloads count. The investigation on age answers the question that whether or not the early packages are more likely to gain popularity in accelerated speed than the new ones. The age is represented by the number of passing through days after the first publication of a package. The number of contributors is the indicator of the scale of the development team. The contributors refer to any developer who ever contributed to any version of a package. A package in `npm` is usually maintained and contributed by an individual developer or some developers working as a team. While the numbers of dependencies and dependents reveal that if reusing or being reused by other packages have effects on popularity growth. The study on the numbers of dependencies and dependents will suggest the impact of relationships among packages on popularity growth. At last, the study on the number of versions interprets the impact of new features on popularity growth. Additionally, the larger number of versions not only means frequent update and more new feature but also suggests whether or not the package is continuously maintained.

For these selected factors, we make use of statistic analysis to examine whether or not there is a strong relationship between them and the proposed growth models. Specifically, we randomly pick 1,000 packages from each growth models and run Kruskal-Wallis H test [46] on them to re-check whether or not the selected factors are significantly different across three proposed growth models. Kruskal-Wallis H test is an effective and widely used method to test whether two or more samples of equal or different sample sizes originate from the same distribution [16]. Additionally, We investigate if the functionalities of packages play roles in popularity growth as well. For this purpose, we collect the keywords of every package fitted by



(a) accelerated growth model (b) steady growth model



(c) decelerated growth model

Figure 5.3: The word-cloud graph using the keywords of every package fitted by each growth models.

each growth models and draw the word-cloud graph using these keywords. The keywords are extracted from the meta-files of packages and we believe that they interpret the functionalities of packages to some extent.

– **Findings:**

Table 5.4 shows the summary of the statistic analysis result on *total downloads count, age* and *the number of contributors, dependencies, dependents, versions* for each growth model.

Total download count: The result shows that the packages with accelerated popularity growth are also the popular ones, which also conforms to our general impression.

Age: The packages fitted by accelerated growth model are definitely older than other two models, and the ones fitted by steady growth model are also a bit older than decelerated model. It suggests that the early

packages in `npm` are much more easily to get popularity in accelerated speed with the time passing by while the new packages are not.

The number of contributors: The number of contributors interprets the scale of development team. The result suggests that there is no significant difference across packages fitted by the proposed three growth models. It interprets that the scale of development team have no definite impact on popularity growth.

The number of the dependencies and dependents: In this investigation we want to answer the question whether reusing or being reused by other packages have effects on popularity growth or not. The result suggests that the numbers of the dependencies are similar across the three proposed growth models while the numbers of the dependents are significantly different. The packages fitted by accelerated growth model attract a lot of dependents while steady growth model and decelerated growth model attract few. It illustrates that being reused by other packages plays a significant role in accelerating popularity growth while the number of dependencies does not. This can be explained by the download mechanism of `npm` — when a package is installed from `npm`, those packages which are in the dependency chain of this package are also installed.

Versions: The p-value tells that the numbers of versions are significantly different across the three proposed growth models. The result suggests that packages fitted by accelerated growth model and decelerated growth model have a tendency to maintain more releases and add new features. On the contrary, the releases of packages fitted by the decelerated growth model are less. This illustrates that adding new features is a double-edged sword to popularity growth: both of acceleration and deceleration are possible due to adding new features. On the contrary, less changes make popularity grow steadily.

Functionalities: Figure 5.3 shows the the word-cloud graphs created with the keywords of every package fitted by three proposed growth models. The result shows that the functionalities have a large variation across the three proposed growth models. The top 3 keywords of packages in accelerated growth model are *gruntplugin, gulpplugin* and *react-component*, which are all related with one particular popular package. Among them, *grunt*[4] package is a JavaScript task runner, which can free developer from repetitive tasks like minification, compilation, unit testing, linting, etc. *gulp*[5] package is a toolkit that helps developer automate painful or time-consuming tasks in the development workflow. *react*[6] package is a JavaScript package for building user interfaces, a traditional client-side application on web development. It illustrates that *grunt*, *gulp*, and *react* are probably

[4]https://gruntjs.com/
[5]http://gulpjs.com/
[6]https://github.com/facebook/react

most widely used packages in `npm`, so the packages related to them can easily gain popularity in accelerated speed. Additionally, the packages fitted by decelerated model also get two significant hot keywords — *plugin* and *api*. This might suggest that some packages of this kind suffer from higher risk of losing popularity as time passes by.

Hence, we answer RQ2:

Some main factors including age, dependents, new features, and functionalities have effects on popularity growth. Among them age and dependents have a positive impact on popularity growth while the new features have possibility of both acceleration and deceleration. We also reveal some possible effects of functionalities on popularity growth.

5.4 Discussion

In this section, we aim to give some suggestions to practitioners in `npm` community to help them on developing and evolving their packages.

- `npm` community is health and well maintained. Therefore, practitioners in such community could possibly get chances to publish successful packages gaining a lot of acceptance.

- The popularity also complies with the Matthew Effect - the popular and rapid growing packages gain more and more acceptance easily. It's not wise for developers to have their packages compete with these packages.

- The early packages have strong advantage. The packages tested by time are proved to be of good quality and much more easily to be accepted. To compete with these early packages, developers should aim to introduce their packages with the distinctive features and good reliability to persuade other packages to make a change.

- Being widely reused by other packages will make a package popular. Easy-to-use features and well-maintained documents will help on that. Especially, being merged into the dependency chain of a popular and rapid growing package is a convenient way to accelerate the popularity growth of a package. So developers can find some popular and rapid growing packages which could possibly reuse their packages and provide more reliable and effective functionalities to them.

- Adding new features is a double-edged sword. On one hand, frequent changes of versions usually refer to the package is well maintained. On the other hand, the new features will also possibly result in dependency chain breakages and confuse the re-users. Developers should pay attention to the balance between them.

- Packages related to some popular packages also easily become popular in `npm`. Developing packages related to popular packages may be a good choice for practitioners in `npm` community.

5.5 Conclusion of This Chapter

In this chapter, we model the popularity growth of packages as curves and fit it against the three proposed growth models to understand the common characteristics of the popularity growth of packages in `npm`. We found that 51.56% of the studied packages depict the steady growth model, followed by accelerated growth model (40.02%) and decelerated growth model (7.20%). The result suggests that the reuse of packages in `npm` is still active, with more and more packages being installed from the `npm`. Also, we select and examine some main software ecosystem factors to understand their impacts on popularity growth. Our study shows that age, dependents, new features, and functionalities play significant roles in popularity growth. Based on these findings, we give some suggestions to practitioners in `npm` community. We hope our results provide valuable insights to help practitioners in developing and evolving packages in a competitive OSS ecosystem.

For the future work, although our approach of modeling the popularity growth of packages is proved to be effective because of the low percentage of packages not fitted by any growth models, we still don't know why these packages could not be fitted. By observing several packages, we find that the popularity of some packages not fitted is changing dramatically in a short time. It will be very interesting to find out why it happens. Secondly, although we have found the common characteristics of popularity growth and some software ecosystem factors having an impact on the growth, the detailed mechanism of how these factors impact popularity growth is still not very clear. Additionally, we presume that there are other factors that can have an impact on the popularity growth of packages since our selected factors are only a portion of all the factors of a software ecosystem. We consider these as future work. Our future work also includes extending the research to different ecosystems such as `RubyGems` or `CRAN` R ecosystem. Also, it would be interesting to do predictions on popularity growth based on the factors we observed.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This dissertation presented our studies on the four reuse issues in OSS: copyright inconsistency, copyright notice identification, dependency-related license violation, and popularity growth.

Firstly, we conducted an empirical study on the Linux kernel to analyze the issue of copyright inconsistency. From the results, we found that copyright inconsistency is prevalent in the Linux kernel. Meanwhile, we investigated the causes of these license inconsistencies and generalized them into 4 reasons: *code reuse*, *affiliation change*, *refactoring*, *support function*, and *others' contributions*. Our findings suggesting that the copyright notices recorded in the source files do not always reflect the actual contributors. It also provides a new perspective to study and improve the management of software copyright in OSS projects.

Secondly, we proposed a machine learning method for automatic copyright notice identification of source files and evaluated it by comparing it with FOSSology - a widely used regular expression matching-based method. The results suggest that the proposed machine learning method outperforms the existing method. This work will help practitioners and researchers in studying the copyright notices in OSS projects.

Thirdly, we conducted an empirical study on `npm` to analyze the issue of dependency-related license violations. The result suggests that only a few packages have dependency-related license violations. However, there is a high possibility of the occurrence of dependency-related license violation if the packages licensed under copyleft licenses is included in the dependency network. Furthermore, we also designed a preliminary questionnaire, which aims to find out the developers' attitude towards dependency-related license violations. We found that developers overlook and misunderstand dependency-related license violations. The results also suggest the difficulties in managing dependency-related license violations and the developers'

demands for help. Our work highlights the importance of dealing with the dependency-related license violation in OSS.

Lastly, we conducted an empirical study on `npm` to understand the popularity growth of packages in it. The result suggests that the reuse of packages in `npm` is still active. Meanwhile, we found that age, dependents, new features, and functionalities have an impact on the popularity growth of packages in `npm`. This study will help practitioners in developing and evolving their OSS in a competitive OSS ecosystem. It will also help researchers to study OSS reuse in OSS ecosystems.

We believe that the findings in this dissertation will help practitioners who reuse OSS in practice and researchers who are to create a better platform for OSS reuse.

6.2 Future Directions

In our first study of software inconsistency and the second study of copyright notice identification, we discovered the difficulty of managing software copyright of source code files in OSS. For example, how to trace the provenance of source code to determine its original copyright notice. Future research could be conducted to find a solution to manage the copyright notices efficiently and effectively in OSS.

Our third study of the dependency-related license violation creates a possibility of studying dependency-related license violations in OSS ecosystems. For example, studies on the propagation of dependency-related license violations would reveal how dependency-related license violations impact OSS ecosystems. We believe these researches will be helpful in maintaining licenses and managing license violations.

As the future research of our fourth study of popularity growth, studies could be done on predicting the popularity growth of OSS projects based on the observed factors. It will help in understanding how an OSS project becomes successful.

Furthermore, our studies are only conducted on some particular OSS projects, studies on more OSS projects are worthwhile of being conducted in further research.

Bibliography

[1] Ahmed, M., Spagna, S., Huici, F., and Niccolini, S. (2013). A peek into the future: Predicting the evolution of popularity in user generated content. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 607–616, New York, NY, USA. ACM.

[2] Almering, V., van Genuchten, M., Cloudt, G., and Sonnemans, P. (2007). Using software reliability growth models in practice. *Software, IEEE*, **24**(6), 82–88.

[3] Alspaugh, T., Asuncion, H., and Scacchi, W. (2009). Intellectual property rights requirements for heterogeneously-licensed systems. In *17th International Requirements Engineering Conference (RE2009)*, pages 24–33.

[4] Annadurai, G., Rajesh Babu, S., and Srinivasamoorthy, V. R. (2000). Development of mathematical models (logistic, gompertz and richards models) describing the growth pattern of pseudomonas putida (nicm 2174). *Bioprocess Engineering*, **23**(6), 607–612.

[5] Apte, U., Sankar, C. S., Thakur, M., and Turner, J. E. (1990). Reusability-based strategy for development of information systems: implementation experience of a bank. *MIS Quarterly*, pages 421–433.

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

[7] Batista, G. E., Prati, R. C., and Monard, M. C. (2004). A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD explorations newsletter*, **6**(1), 20–29.

[8] Bavota, G., Canfora, G., Di Penta, M., Oliveto, R., and Panichella, S. (2015). How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering*, **20**(5), 1275–1317.

[9] Berger, T., Pfeiffer, R.-H., Tartler, R., Dienst, S., Czarnecki, K., Wąsowski, A., and She, S. (2014). Variability mechanisms in software ecosystems. *Information and Software Technology*, **56**(11), 1520 – 1535. Special issue on Software Ecosystems.

[10] Bird, C., Nagappan, N., Murphy, B., Gall, H., and Devanbu, P. (2011). Don't touch my code! examining the effects of ownership on software quality. In *19th ACM SIGSOFT symposium and 13th European conference on Foundations of software engineering (ESEC/FSE 2011)*, pages 4–14.

[11] Boehm, B. W. (1987). Improving software productivity. *Computer*, **20**(9), 43–57.

[12] Borges, H., Hora, A. C., and Valente, M. T. (2016). Understanding the factors that impact the popularity of github repositories. *CoRR*, **abs/1606.04984**.

[13] Christley, S. and Madey, G. (2007). Analysis of activity in the open source software development community. In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, pages 166b–166b. IEEE.

[14] Corbet, J. and Kroah-Hartman, G. (2017). 2017 linux kernel development report. *A Publication of The Linux Foundation*.

[15] Da Veiga, A. (2016). A cybersecurity culture research philosophy and approach to develop a valid and reliable measuring instrument. In *2016 SAI Computing Conference (SAI)*, pages 1006–1015. IEEE.

[16] Daniel, W. (1978). *Applied nonparametric statistics*. Houghton Mifflin.

[17] David, S. M., William, I. N., and Michael, A. F. (2013). *The Basic Practice of Statistics*, page 138. W. H. Freeman.

[18] Decan, A., Mens, T., and Grosjean, P. (2019). An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, **24**(1), 381–416.

[19] Di Penta, M. and German, D. (2009). Who are source code contributors and how do they change? In *16th Working Conference on Reverse Engineering (WCRE2009)*, pages 11–20.

[20] Di Penta, M., German, D. M., Guéhéneuc, Y.-G., and Antoniol, G. (2010). An exploratory study of the evolution of software licensing. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE2010)*, pages 145–154.

[21] Forman, G. (2003). An extensive empirical study of feature selection metrics for text classification. *Journal of machine learning research*, **3**(Mar), 1289–1305.

[22] Foucault, M., Falleri, J.-R., and Blanc, X. (2014). Code ownership in open-source software. In *18th International Conference on Evaluation and Assessment in Software Engineering (EASE 2014)*, pages 1–9.

[23] Gao, Z., Bird, C., and Barr, E. T. (2017). To type or not to type: quantifying detectable bugs in javascript. In *IEEE/ACM 39th International Conference on Software Engineering (ICSE 2017)*, pages 758–769. IEEE.

[24] German, D. (2006). A study of the contributors of postgresql. In *2006 International workshop on Mining software repositories (MSR 2006)*, pages 163–164.

[25] German, D. and Di Penta, M. (2012). A method for open source license compliance of java applications. *IEEE software*, **29**(3), 58–63.

[26] German, D., Di Penta, M., Gueheneuc, Y.-G., and Antoniol, G. (2009). Code siblings: Technical and legal implications of copying code between applications. In *Proceedings of the 6th Working Conference on Mining Software Repositories (MSR2009)*, pages 81–90.

[27] German, D., Manabe, Y., and Inoue, K. (2010a). A sentence-matching method for automatic license identification of source code files. In *25th International Conference on Automated Software Engineering (ASE2010)*, pages 437–446.

[28] German, D., Di Penta, M., and Davies, J. (2010b). Understanding and auditing the licensing of open source software distributions. In *18th International Conference on Program Comprehension (ICPC2010)*, pages 84–93.

[29] German, D., Adams, B., and Stewart, K. (2019). cregit: Token-level blame information in git version control repositories. *Empirical Software Engineering*, **24, issue 4**, 2725–2763.

[30] Gharehyazie, M., Ray, B., and Filkov, V. (2017). Some from here, some from there: Cross-project code reuse in github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 291–301. IEEE.

[31] Girba, T., Kuhn, A., Seeberger, M., and Ducasse, S. (2005). How developers drive software evolution. In *8th international workshop on principles of software evolution (IWPSE 2005)*, pages 113–122.

[32] Gobeille, R. (2008). The fossology project. In *2008 International working conference on Mining software repositories (MSR2008)*, pages 47–50.

[33] Golder, T. and Mayer, A. (2009). Whose ip is it anyway? *Journal of Intellectual Property Law & Practice*, **4**(3), 165–175.

[34] Grimm, K. J., Nilam, R., and Fumiaki, H. (2011). Nonlinear growth curves in developmental research. *Child Dev.*, **82**, 1357–1371.

[35] Hammad, M., Hammad, M., Bani-Salameh, H., and Fayyoumi, E. (2014). Measuring developers' design contributions in evolved software projects. *Journal of SOftware*, **9**(12), 3005–3011.

[36] Hata, H., Treude, C., Kula, R. G., and Ishio, T. (2019). 9.6 million links in source code comments: purpose, evolution, and decay. In *IEEE/ACM 41st International Conference on Software Engineering (ICSE 2019)*, pages 1211–1221. IEEE.

[37] Hauge, Ø., Sørensen, C.-F., and Conradi, R. (2008). Adoption of open source in the software industry. In *IFIP International Conference on Open Source Systems*, pages 211–221. Springer.

[38] Hindle, A., German, D., and Holt, R. (2008). What do large commits tell us? a taxonomical study of large commits. In *2008 International working conference on Mining software repositories (MSR 2008)*, pages 99–108.

[39] Japkowicz, N. and Stephen, S. (2002). The class imbalance problem: A systematic study. *Intelligent data analysis*, **6**(5), 429–449.

[40] Jones, T. C. (1984). Reusability in programming: A survey of the state of the art. *IEEE Transactions on Software Engineering*, (5), 488–494.

[41] Kapitsaki, G. M., Kramer, F., and Tselikas, N. D. (2017). Automating the license compatibility process in open source software with spdx. *Journal of Systems and Software*, **131**, 386–401.

[42] Kikas, R., Gousios, G., Dumas, M., and Pfahl, D. (2017). Structure and evolution of package dependency networks. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories (MSR2017)*, pages 102–112.

[43] Koch, S. (2005). *Free/open source software development*. Igi Global.

[44] Krejcie, R. V. and Morgan, D. W. (1970). Determining sample size for research activities. *Educational and psychological measurement*, **30**(3), 607–610.

[45] Krueger, C. (1992). Software reuse. *ACM Comput. Surv.*, **24**, 131–183.

[46] Kruskal, W. H. and Wallis, W. A. (1952). Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, **47**(260), 583–621.

[47] Kula, R. G., German, D. M., Ishio, T., Ouni, A., and Inoue, K. (2017). An exploratory study on library aging by monitoring client usage in a software ecosystem. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 407–411.

[48] Kula, R. G., German, D. M., Ouni, A., Ishio, T., and Inoue, K. (2018a). Do developers update their library dependencies? *Empirical Software Engineering*, **23**(1), 384–417.

[49] Kula, R. G., German, D. M., Ouni, A., Ishio, T., and Inoue, K. (2018b). Do developers update their library dependencies? *Empirical Software Engineering*, **23**(1), 384–417.

[50] Langone, R., Mall, R., Alzate, C., and Suykens, J. A. K. (2016). *Kernel Spectral Clustering and Applications*, pages 135–161. Springer International Publishing, Cham.

[51] Larabel, M. (2020). The linux kernel enters 2020 at 27.8 million lines in git but with less developers for 2019. Web page at linux.com, https://www.phoronix.com/scan.php?page=news_item&px= Linux-Git-Stats-EOY2019, Jan.

[52] Laurent, A. M. S. (2004). *Understanding open source and free software licensing: guide to navigating licensing issues in existing & new software*. O'Reilly.

[53] Lehmann, J., Gonçalves, B., Ramasco, J. J., and Cattuto, C. (2012). Dynamical classes of collective attention in twitter. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 251–260, New York, NY, USA. ACM.

[54] Lungu, M., Lanza, M., Gîrba, T., and Robbes, R. (2010). The small project observatory: Visualizing software ecosystems. *Science of Computer Programming*, **75**(4), 264–275.

[55] Magee, L. (1990). R 2 measures based on wald and likelihood ratio joint significance tests. *The American Statistician*, **44**(3), 250–253.

[56] McIlroy, M. D., Buxton, J., Naur, P., and Randell, B. (1968). Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering (ICSE1968)*, pages 88–98.

[57] Meeker, H. (2017). Patrick mchardy and copyright profiteering. Web page at opensource.com, https://opensource.com/article/17/8/patrick-mchardy-and-copyright-profiteering, Aug.

[58] Oracle (2019). Oracle contributor agreement - version 1.7.1. Web page at oracle.com, https://www.oracle.com/technetwork/community/oca-486395.html#list.

[59] Ozdemir, S. (2016). *Principles of Data Science*. Packt Publishing.

[60] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., *et al.* (2011). Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, **12**, 2825–2830.

[61] Scacchi, W. (2007). Free/open source software development: Recent research results and methods. *Advances in Computers*, **69**, 243–295.

[62] Sebastiani, F. (2002). Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, **34**(1), 1–47.

[63] Sojer, M. and Henkel, J. (2010). Code reuse in open source software development: Quantitative evidence, drivers, and impediments. *Journal of the Association for Information Systems*, **11**(12), 868–901.

[64] Standish, T. A. (1984). An essay on software reuse. *IEEE Transactions on Software Engineering*, **SE-10**(5), 494–497.

[65] Stanik, C., Montgomery, L., Martens, D., Fucci, D., and Maalej, W. (Sept. 2018). A simple nlp-based approach to support onboarding and retention in open source communities. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain*, pages 172–182.

[66] Tsikerdekis, M. (2018). Persistent code contribution: a ranking algorithm for code contribution in crowdsourced software. *Empirical Software Engineering*, **23**(4), 1871–1894.

[67] Tuunanen, T., Koskinen, J., and Kärkkäinen, T. (2009). Automated software license analysis. *Automated Software Engineering*, **16**(3-4), 455–490.

[68] Uden, L., Damiani, E., Gianini, G., and Ceravolo, P. (2007). Activity theory for oss ecosystems. In *2007 Inaugural IEEE-IES Digital EcoSystems and Technologies Conference*, pages 223–228. IEEE.

[69] Van Der Burg, S., Dolstra, E., McIntosh, S., Davies, J., German, D. M., and Hemel, A. (2014). Tracing software build processes to uncover license compliance inconsistencies. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 731–742. ACM.

[70] Vendome, C. and Poshyvanyk, D. (2016). Assisting developers with license compliance. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 811–814. ACM.

[71] Vendome, C., Linares-Vásquez, M., Bavota, G., Di Penta, M., Germán, D. M., and Poshyvanyk, D. (2015). License usage and changes: A large-scale study of java projects on github. In *The 23rd IEEE International Conference on Program Comprehension, ICPC 2015*.

[72] Vendome, C., Bavota, G., Di Penta, M., Linares-Vásquez, M., German, D., and Poshyvanyk, D. (2017a). License usage and changes: a large-scale study on github. *Empirical Software Engineering*, **22**(3), 1537–1577.

[73] Vendome, C., Linares-Vásquez, M., Bavota, G., Di Penta, M., German, D., and Poshyvanyk, D. (Aug. 2017b). Machine learning-based detection of open source license exceptions. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina*, pages 118–129.

[74] Welte, H. (2018). Report from the geniatech vs. mchardy gpl violation court hearing. Web page at gnumonks.org, https://laforge.gnumonks.org/ blog/20180307-mchardy-gpl/, Mar.

[75] Wheeler, D. A. (2017). The free-libre / open source software (floss) license slide. Web page at https://www.dwheeler.com/essays/floss-license-slide.html.

[76] Wikipedia (2020). Google llc v. oracle america, inc. Web page at wikipedia.org, https://en.wikipedia.org/w/ index.php?title=Google$_{L}LC_{v}._{O}racle_{A}merica,_{I}nc.oldformat = true, Dec.$

[77] Wittern, E., Suter, P., and Rajagopalan, S. (2016). A look at the dynamics of the javascript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 351–361, New York, NY, USA. ACM.

[78] Wu, Y., Manabe, Y., Kanda, T., German, D. M., and Inoue, K. (2015). A method to detect license inconsistencies in large-scale open source projects. In *2015 International working conference on Mining Software Repositories (MSR 2015)*, pages 324–333.

[79] Yamada, S., Ohba, M., and Osaki, S. (1983). S-shaped reliability growth modeling for software error detection. *Reliability, IEEE Trans.*, **R-32**(5), 475–484.

[80] Zhang, H., Shi, B., and Zhang, L. (2010). Automatic checking of license compliance. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–3. IEEE.