Dependency Management Practices for the npm Software Ecosystem

Abbas Javan Jafari

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy (Software Engineering) at

Concordia University

Montréal, Québec, Canada

December 2023

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Abbas Javan Jafari**

Entitled: **Dependency Management Practices for the npm Software Ecosystem**

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

Doctor of Philosophy (Software Engineering)

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

Signed by the Final Examining Committee:

_____ Chair
Dr. Pantcho P. Stoyanov

_____ External Examiner
Dr. Raula Gaikovina Kula

_____ Examiner
Dr. Peter Rigby

_____ Examiner
Dr. Tse-Hsun (Peter) Chen

_____ Examiner
Dr. Jamal Bentahar

_____ Supervisor
Dr. Emad Shihab

Approved by

_____
Dr. Leila Kosseim, Graduate Program Director
Department of Computer Science and Software Engineering

December 12, 2023

_____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Dependency Management Practices for the npm Software Ecosystem

Abbas Javan Jafari, Ph.D.

Concordia University, 2024

Software ecosystems provide developers with the opportunity to accelerate development by relying on third-party dependencies. Developers use third-party packages to increase productivity and improve quality. However, the increased reliance on third-party dependencies has emphasized dependency-related challenges. Developers need to be aware of such challenges and be equipped with techniques to mitigate their impact. Poor management of third-party dependencies can subject the project to breaking changes, bugs and vulnerabilities, which negatively impact the quality of software. In this thesis, we use a mixture of quantitative and qualitative methods to understand dependency management challenges in the npm ecosystem and provide actionable mitigation techniques to help developers better manage their dependencies.

We first study, catalog and quantify recurring patterns of dependency mis-management in the npm ecosystem and provide evidence of their prevalence and accumulation. In the second part of the thesis, we analyze the relationship between the characteristics of npm packages and how they are used by the community. We propose to developers a technique to determine the update strategy of their direct dependencies based on the individual characteristics of each package. In the last part of the thesis, we focus on the impact of transitive dependencies and quantify the impact of dependency decisions on continued exposure to

security vulnerabilities. We propose a technique to select dependencies that mitigates the propagation of vulnerabilities to our project. Throughout our research, we identify implications that can serve both researchers and practitioners.

Acknowledgments

I would like to thank my amazing supervisor, Prof. Emad Shihab for his continued support throughout my PhD. It was a great pleasure having a supervisor who is both truly an expert in the field and who can always motivate his students to perform at their best. I am forever grateful to Emad for his trust, guidance and encouragement.

Throughout my PhD adventure, I had the privilege of working with Dr. Diego Elias Costa, Dr. Ahmad Ahmad Abdellatif and Dr. Rabe Abdalkareem at DAS Lab. Thank you so much for taking the time to share your invaluable experience and insights. I also had the wonderful experience of collaborating with Dr. Nikolaos Tsantalis at the Department of Computer Science and Software Engineering.

I would also like to thank my committee members. Dr. Raula Gaikovina Kula, Dr. Peter Rigby, Dr. Tse-Hsun (Peter) Chen and Dr. Jamal Bentahar for taking the time to read my thesis and providing me with insightful comments throughout my PhD.

I also wish to extend my appreciation to my fellow colleagues at the Data-driven Analysis of Software (DAS) Lab, specifically Mahmoud Alfadel, Jasmine Latendresse, Suhaib Mujahid and Khaled Badran. I wish you all the very best.

Above all, I thank god for blessing me with a wonderful family. I thank my mother, father and brother for always supporting me throughout the years. I thank my amazing wife who was always at my side and kept me motivated throughout the journey. I would not be here without you all and I hope to make you proud.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Research Statement

1.1 Introduction

Current software systems are large and complex and heavily rely on code reuse to accelerate their development and improve quality (Bombonatti, Goulão, & Moreira, 2017; Keswani, Joshi, & Jatain, 2014). More than 80% of the codebase in a modern application is comprised of third-party and open source components (GitHub, 2020). Software packages (i.e. reusable code libraries) are a common way to achieve code reuse and their popularity is driven by software ecosystems. Software packaging ecosystems serve as a platform for developers to share their own packages, and also use packages shared by others. The increasing number of packages can facilitate code reuse, but it can also lead to an increase in dependencies between packages (i.e. packages rely on other packages to function). Dependencies between packages are the major underlying principle for software ecosystems (Cogo, Oliva, & Hassan, 2019). One example of such ecosystems is the Node Package Manager (npm), which is currently the world's largest software ecosystem where a high reliance on third-party dependencies is commonplace. By late July 2021, the npm ecosystem had over 1.9 million packages (up 16% from the previous year) and 21 million package versions (Sonatype, 2021). According to Laurie Voss (COO and co-founder of

npm Inc.), 97% of the code in modern web applications come from npm packages, meaning the application developer is directly responsible for only 3% of the code-base (npm, 2018). In the case of the npm ecosystem, the average number of total dependencies has been growing at a super-linear rate (Zimmermann, Staicu, Tenny, & Pradel, 2019).

The sheer size and complexity of the dependency networks, along with their speedy growth and evolution, has created difficult challenges in software ecosystems, particularly in dependency management practices such as tracking and updating packages (Artho, Suzaki, Di Cosmo, Treinen, & Zacchiroli, 2012; Bogart, Kästner, Herbsleb, & Thung, 2016; Decan, Mens, & Grosjean, 2019). In fact, a 2022 survey of industry practitioners found that 85% of organizations are are not fully confident in their open source dependency management practices (Tidelift, 2022). The multitude of dependency management challenges in software ecosystems have also given birth to the colloquial term, "Dependency Hell" (Decan & Mens, 2019a; Fan et al., 2020; Matt Rickard, 2021). Examples of the technical issues which directly relate to dependency management are software bugs and vulnerabilities, technical lag, breaking changes and bloated installations (Bogart et al., 2016; Chinthanet, Kula, Ishio, Ihara, & Matsumoto, 2019; Cogo et al., 2019; Decan, Mens, & Constantinou, 2018a; Jafari, Costa, Abdalkareem, Shihab, & Tsantalis, 2021; Kula, German, Ouni, Ishio, & Inoue, 2018a; Soto-Valero, Harrand, Monperrus, & Baudry, 2021).

Many software ecosystems such as npm (Node Package Manager), PyPi (Python Package Index) and RubyGems (Ruby Package Manager) allow developers to determine their stance toward automatic updates using the dependency configuration file. This presents package developers with a dilemma. On one hand, we want to keep our dependencies up to date and take advantage of the latest features and bug/vulnerability fixes. On the other hand, updating dependencies often increases the chance of installing a backward-incompatible release of the dependency and break our code (Bavota, Canfora, Di Penta, Oliveto, & Panichella, 2013; Bogart et al., 2016; Cogo et al., 2019; Derr, Bugiel, Fahl,

Acar, & Backes, 2017). Proper dependency management practices can help in finding the right balance between staying up to date and ensuring code stability. Due to the complexity and variety in software packages and their inter-dependencies, ensuring (or even defining) a suitable dependency management strategy is not straightforward. Knowing when to update a dependency and knowing which versions to use are among the most important challenges faced by development teams (Tidelift, 2022). Motivated by the issues in the literature, we aim to tackle the following questions:

(1) **What is bad dependency management?** Developers have many alternative options in deciding how to manage their dependencies. However, these alternatives correspond to negative consequences on their project or on the ecosystem. Developers need to be aware problematic dependency configurations and their impact.

(2) **How should developers update dependencies?** Updating too eagerly can break compatibility and updating too conservatively can expose us to security risks. Every package has its unique set of characteristics and behaviors. Therefore, developers need to identify an update strategy suitable for each dependency.

(3) **How should developers select dependencies?** Regardless of how well developers manage their dependencies, they have no means to properly manage the dependencies of their dependencies. Managing these transitive dependencies is the responsibility of direct dependencies. Therefore, developers need to select suitable packages that have responsible dependency practices.

In order to better understand the challenges in open-source dependency management and help developers in mitigating these challenges, we conduct a series of empirical studies focused on the JavaScript ecosystem. JavaScript is the most widely used open source programming language (GitHub, 2021) and it also has the largest software ecosystem with over 1.9 Million packages (Sonatype, 2021). We first employ a mixture of quantitative

and qualitative techniques to study dependency management practices in packages that declare dependencies to examine dependency management issues, their prevalence and consequences. Next, in order to help developers better manage and update their dependencies, we study the upstream packages to model and understand the association between the characteristics of a package and the dependency practices selected by its dependent community. Finally, we aim to help developers in selecting suitable dependencies by examining the attributes of packages that quickly mitigate vulnerabilities. Our research aims to help developers in selecting and maintaining their dependencies to minimize their exposure and contribution to dependency-related issues in both their own project and the larger ecosystem.

1.2 Research Statement

Developers are faced with many challenges when managing their dependencies Section 1.1. Knowing what dependency packages to use and how to use them are key issues in the npm ecosystem. The goal of this Ph.D. thesis is to address these issues by studying dependency management practices and proposing suitable mitigation strategies. We state our research statement as follows:

Given the complexity of dependency management in software ecosystems, our goal is to employ information from the JavaScript ecosystem to identify challenges and propose mitigation techniques to help developers better manage their dependencies.

1.3 Thesis Overview

In this section, we provide a brief overview of the thesis. In the first part (Chapter 2), we provide the background required for our research and discuss how our work aligns with

the related works in literature. In the second part, we present our three empirical studies (Chapters 3, 4 and 5). We conclude our thesis in Chapter 6 and discuss the key avenues for future research in dependency management.

Chapter 3: Challenges in Dependency Management

Many software ecosystems (e.g. npm, PyPI) allow developers to choose from a variety of dependency management practices. For example, developers in npm can specify different constraints for each of their dependencies that determine how the dependency is fetched, installed and automatically updated. Choosing the wrong dependency configuration can have negative consequences on the project such as bugs, vulnerabilities and compatibility issues. We need to identify and quantify the advantages and drawbacks of different practices to understand the current landscape and formulate guidelines for good dependency management practices. Dependency practices and their consequences are referenced in online discussions and informal interviews, and sometimes mentioned as a byproduct of other research, but there is no comprehensive and focused study on cataloging and quantifying such practices. We use a combination of quantitative and qualitative techniques to catalog, understand and quantify the challenges posed by various dependency practices. We start by specifying bad practices and qualitatively analyzing their consequences by enlisting the help of practitioners. We then conduct an empirical study on the development history of 1,100+ npm projects on GitHub to quantitatively measure the prevalence of such bad practices. We also look at the evolution of such practices over time and investigate the reasons for their occurrence in the first place. This work has been published in the journal of IEEE Transactions on Software Engineering (Jafari et al., 2021).

Chapter 4: Practices for Updating Dependencies

The findings of Chapter 3 show that developers don't use the same blanket strategy to manage all of their dependencies and sometimes opt for problematic alternatives as a response to shortcomings of a particular dependency. Therefore, understanding downstream dependency practices requires an investigation into the characteristics and behaviors of different upstream packages. Previous research has hinted that the characteristics of a package may influence how it is perceived by its dependents (Decan & Mens, 2019a). We need to understand how upstream packages influence dependency management strategies in their downstream dependents. We also need to help developers assign a fitting dependency management strategy for each of their dependencies. Our objective is to better understand whether package characteristics can be used to model and predict the common dependency update strategy used by its dependents. We conduct an empirical study to explore the association between the 19 characteristics of more than 112,000 packages and the dependency update strategies used by their dependents in an effort to understand why certain dependency update strategies are favored by the community. We complement this work by a number of qualitative analyses on a sample of 160 packages to better understand the evolution of dependency update strategies over time. This work has been published in the journal of ACM Transactions on Software Engineering and Methodology (Jafari, Costa, Shihab, & Abdalkareem, 2023).

Chapter 5: Practices for Selecting Dependencies

While developers might maintain a certain amount of trust for their dependency packages, installing each dependency places an implicit trust on a much larger number of third-party packages that serve as the dependencies of their dependencies (i.e. transitive dependencies) (Zimmermann et al., 2019). The findings of Chapter 4 can be used to determine a suitable update strategy for dependencies. However, even if developers have flawless

dependency management practices for their dependencies, they have no control over the transitive dependencies of their project. Yet, transitive dependencies can expose the project to upstream issues such as security vulnerabilities. From a downstream client's perspective, there is no difference between being directly exposed to a vulnerability or being exposed to a vulnerability through a dependency (Decan, Mens, & Constantinou, 2018b). Therefore, developers need to select dependencies that are well-equipped for handling vulnerabilities. We propose a means to identify responsive packages (i.e. packages that quickly adopt a vulnerability fix) to help developers select dependencies that better mitigate the propagation of vulnerabilities to their project. We study 450 verified npm vulnerabilities and over 200,000 infected packages (through dependencies) to model the speed of vulnerability fix adoption. We observe 9 package attributes, verified by practitioners, that can be incorporated to dependency management practices to mitigate the exposure to vulnerabilities. We plan to submit this work to the journal of IEEE Transactions on Software Engineering (Jafari, Costa, Abdellatif, & Shihab, 2023).

1.4 Thesis Contributions

The key contributions of this thesis are as follows:

- Empirical study of dependency smells, their impact and their evolution in more than 1,100 JavaScript projects
- Catalog of 7 dependency smells crafted and validated by quantifying responses from JavaScript practitioners.
- Prototype tool named *Dependency Smells* that can analyze npm projects and identify dependency smells.
- Empirical study of over 112,000 npm packages to identify the update strategy used

by clients of each package.

- Identifying 19 package characteristics that can be used to determine a suitable update strategy.
- Qualitative analysis of 160 npm packages to investigate the evolution of update strategies.
- Empirical study on the speed of fix adoption for 450 vulnerabilities and more than 200,000 impacted packages.
- Identifying 9 attributes that can be used to select dependencies that mitigate vulnerability propagation.
- Validating the practical applicability of using dependency practices to mitigate vulnerabilities with a survey of 67 practitioners.

1.5 Related Publications

The following publications align with the content of this thesis:

- **Abbas Javan Jafari**, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. "Dependency smells in javascript projects." IEEE Transactions on Software Engineering 48, no. 10 (2021)
- **Abbas Javan Jafari**, Diego Elias Costa, Emad Shihab, and Rabe Abdalkareem. "Dependency Update Strategies and Package Characteristics." ACM Transactions on Software Engineering and Methodology 32, no. 6 (2023):
- **Abbas Javan Jafari**, Diego Elias Costa, Ahmad Abdellatif and Emad Shihab. "Dependency Practices for Vulnerability Mitigation." IEEE Transactions on Software Engineering [To be Submitted]

1.6 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 provides the background and related works for our research. Chapter 3, Chapter 4 and Chapter 5 present the details and results of our empirical studies that investigate the research questions of this thesis. Since the related works are mostly specific to each chapter, we do not attempt to amalgamate the related works into a standalone section of the thesis. We summarize our findings and suggest directions for future work in Chapter 6.

Chapter 2

Background and Related Work

2.1 Background

In this section, we explain the necessary background required to understand our work on dependency management. We describe the concept of software ecosystems and using npm as an example, we explain the dynamics of dependency management. We also explain the Semantic Versioning standard and how it impacts dependency management.

2.1.1 Software Ecosystems

We study software ecosystems in the context of package libraries for software programming languages. In this regard, a software ecosystems is a collection of software projects which are developed, distributed and evolve under the same environment (Lungu, Lanza, Gîrba, & Robbes, 2010). The environment in this context refers to package managers that are charged with maintaining (add, remove, update, configure) and distributing (install) software packages (Burrows, 2017; Decan et al., 2019). In the following, we present some examples of currently popular software ecosystems and their corresponding package managers:

- **npm (Node Package Manager)** is the main package manager for Node.js created in 2010. The package manager consists of an online database (the npm registry) which is accessed by the command line client.

- **PyPI (Python Package Index)** is an official package repository for Python released in 2003. The popular "pip" package manager in Python uses PyPI as the main source for packages.

- **RubyGems** is a package manager for the Ruby programming language launched in 2004. The package manager distributes ruby program and libraries in a standard "gem" format.

Different software ecosystems can also have different cultures and values, which are enforced through peer pressure or policies. For example, the community around the npm ecosystem favors ease of adoption for new technologies, even if it may lead to incompatible and breaking releases. On the other hand, the Eclipse community has long recognized stability as a key value, even if it may cause technical debt (Bogart et al., 2016).

2.1.2 Dependency Management

Dependencies between packages in a software ecosystem is a common technique to facilitate code reuse. The dependency relationship from one package to another can be *direct* (a package declares another as a dependency) or *transitive* (a package declares another as a dependency which itself declares a third package as a dependency). A package can reuse all of the functionalities provided by its dependencies or only make use of a subset of the functionalities. Based on the example relationships in Figure 2.1, our project has declared package A as a dependency. In this relationship, our project is referred to as the *dependent* or a *downstream package* and package A is referred to as the *dependency* or an

upstream package. Package A has declared package B as a dependency, making package B a transitive dependency for our project.

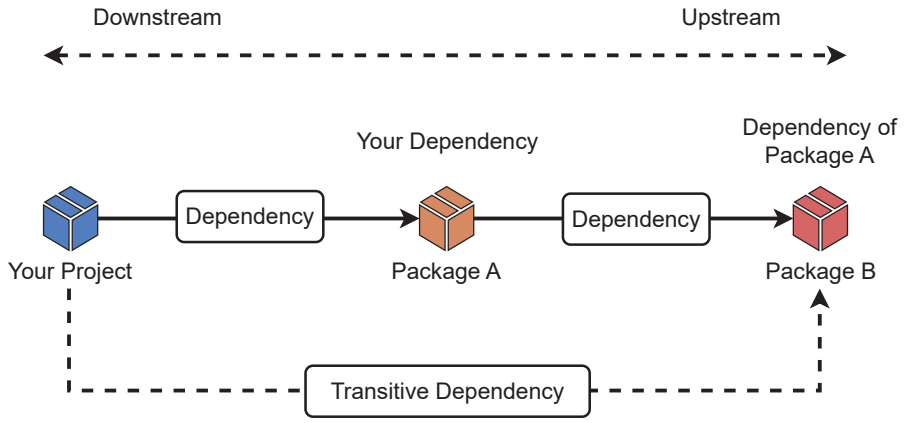


Figure 2.1: Dependency relationships

Different software ecosystems have their own standards in declaring and managing dependencies. In the JavaScript ecosystem of npm, packages use the package.json file (Figure 2.2) to specify dependencies, along with other metadata such as package name and version (npm Documentation, 2019a). This file consists of different sections for runtime, development and optional dependencies. When a package is installed, npm will fetch and install all upstream runtime dependencies, as they are necessary for the application to function. Failure to fetch or install runtime dependencies will raise errors by the package manager. Development dependencies are only required for development operations such as testing and linters and are not necessary for package users. As for optional dependencies, npm will try to fetch them, but failure to do so will not raise any errors since they are not necessary for the package to function correctly. The installation of packages is also performed for transitive dependencies (dependencies of dependencies) until the full dependency tree is installed. Upon using the *npm install* command, the package manager also creates a package-lock.json which includes the installed versions of all dependencies at the time. This allows future installations of a package to remain consistent.

```
# PACKAGE.JSON #
{
 "name": "npm-pkg",
 "version": "1.0.3",
 "description": "A sample npm package",
 "homepage": "http://github.com/npm-pkg",

 "dependencies": {
  "react": "^16.10.1",
  "jquery": "3.4.1",
  "lodash": "~4.17.15",
  "jest": "<=25.0.0",
  "moment": ">=2.24.0",
 },
 "devDependencies": {
  "webpack": "^4.41.2"
 },
 "optionalDependencies": {
  "eslint": "^6.7.2"
 }
}
```

Figure 2.2: Example of a dependency configuration file (package.json)

2.1.3 Semantic Versioning (SemVer)

Semantic Versioning (SemVer) is the recommended versioning standard for many software ecosystems including npm (npm, 2022a) and PyPI (Coghlan & Stufft, 2021). Tom Preston-Warner, the co-founder of the GitHub platform, first introduced this standard in 2011. SemVer 2.0 was released in 2013 and it is the version referenced throughout our research (Preston-Werner, 2019). SemVer addresses the dependency update issue by allowing package maintainers to communicate what type of changes are included in a new release. SemVer introduces a multi-part versioning scheme in the form of **Major.Minor.Patch[-Tag]**.

- **Major:** If a newly released version contains backward incompatible feature updates, the maintainer will increase the major version number.

- **Minor:** If a newly released version includes a backward compatible feature update, the maintainer will increase the minor version number.
- **Patch:** If the new release only contains bug or security fixes, the maintainer will increase the patch version number.
- **Tag:** The optional tag is used for specifying build metadata and pre-release or post-release numbers.

In a software ecosystem such as npm, developers can use SemVer, along with the dependency notations in npm, to specify the degree of freedom granted to the package manager in fetching new versions of a dependency. In order to be compliant with SemVer, developers should accept automatic updates for new minor and patch version for all post-1.0.0 releases. The common dependency notations in npm are as follows:

- The caret (^) notation is used to accept only minor and patch updates for post-1.0.0 versions. For example, ^2.3.4 is equivalent to [2.3.4-3.0.0).
- The tilde (∼) notation is used to accept only patch updates (when a minor version is specified). For example, (∼)2.3.4 is equivalent to [2.3.4-2.4.0).
- The star (*) wildcard will give npm complete freedom to install any new version of a dependency.
- Specifying a specific version will limit npm to only install that particular version.

SemVer Non-Compliance

While SemVer is a promising solution to many dependency update issues, and even though it is recommended by ecosystem maintainers (Coghlan & Stufft, 2021; npm, 2022a), it is not always followed in practice (Decan & Mens, 2019a). A survey of more than 2,000

developers from 18 different software ecosystems including the npm ecosystem showed that while 92% of the respondents for npm claim to only increment the left-most digit if they release an update that may break dependent code, 70% of the surveyed developers had a different experience when updating their dependencies (Bogart, Filippova, Kästner, & Herbsleb, 2017). In these cases, they experienced breaking changes even when updating in compliance with SemVer guidelines. This leads developers to exercise caution when following SemVer, especially for dependencies which have a history of non-compliance for their releases.

2.2 Related Work

In this section, we present the research works closely related to our thesis. We start by presenting the related studies on software ecosystems including npm. We then present the related works on the important dependency challenges of the npm ecosystem and how our thesis complements the literature.

2.2.1 Software Packaging Ecosystems

There is an extensive body of work that study the evolution and dependency dynamics of different software ecosystems. Decan et al. compared the evolution of dependency networks in seven package ecosystems and proposed metrics to capture the growth, reusability and fragility of the ecosystems (Decan et al., 2019). Kikas et al. studied the dependency networks of the JavaScript, Rust and Ruby ecosystems and found all ecosystems to be highly dependent on a subset of popular packages (Kikas, Gousios, Dumas, & Pfahl, 2017). Soto-Valero et al. empirically studied the diversity of releases in the Maven ecosystem and found that for popular packages, more than 50% of their versions are used by downstream

dependents (Soto-Valero, Benelallam, Harrand, Barais, & Baudry, 2019). Kula et al. proposed a model for visualizing package diffusion and adoption in the Maven and CRAN ecosystems and found the Maven ecosystem to have a more conservative approach to updating dependencies than CRAN (Kula, De Roover, German, Ishio, & Inoue, 2018). Vargas et al. conducted a qualitative study with 115 industry practitioners to understand the influence of technical, human and economical factors on how developers select third-party packages (Larios Vargas, Aniche, Treude, Bruntink, & Gousios, 2020). They observed many ad-hoc package selection procedures because developers do not have access to a unified infrastructure to support package selection. However, they found that developers generally look for stability and security when selecting dependencies. Dietrich et al. investigated developer choices on over 170 million dependency relationships across 17 software ecosystems (Dietrich, Pearce, Stringer, Tahir, & Blincoe, 2019). They found that developers struggle with finding the sweet spot between fixed and flexible dependency practices.

The npm ecosystem is not only the largest software packaging ecosystem in the world with over 2 million packages (npm, 2022b), it is also a highly interconnected ecosystem. Wittern et al. studied the evolution of the npm ecosystem and found an increasing amount of dependencies between packages, with more than 80% of the packages relying at least on one other package (Wittern, Suter, & Rajagopalan, 2016). Zimmermann et al. reported that the average number of packages in the ecosystem is growing every year, but also highlighted that the number of dependencies per package is growing at a super-linear rate (Zimmermann et al., 2019). In their study of the evolution of package dependency networks, Decan et al. confirmed the exponential growth of npm in terms of both size and complexity (Decan et al., 2019). Large packages are not the only popular packages in the ecosystem. Chowdhury et al. found that some small and trivial packages (Abdalkareem, Nourry, Wehaibi, Mujahid, & Shihab, 2017) (also known as micro-packages (Kula, Ouni, German, & Inoue, 2017)) impact close to 30% of downstream dependents, resembling points of failure

for the entire ecosystem (Chowdhury, Abdalkareem, Shihab, & Adams, 2021). In addition, developers in npm do not limit their dependencies to stable and mature packages. Decan and Mens studied the reliance on initial development releases (version number equivalent to 0.y.z) and found that even though these releases are deemed unstable, more than 43% of all downstream packages in npm are depending on such releases (Decan & Mens, 2021).

2.2.2 Breaking Changes and Technical Lag in the npm Ecosystem

Due to its rapid growth and highly connected nature, the npm ecosystem is plagued with dependency-related issues. Breaking changes are changes in new package releases that are backward incompatible with the older version, causing a breakage for downstream dependents that update their dependency. In their study of the cultural values of different software ecosystems, Bogart et al. (Bogart et al., 2016) observed that npm developers are less concerned with introducing breaking changes, as long as they are clearly communicated to downstream clients via version numbering (i.e. using SemVer). A follow-up study by Bogart and Thung confirmed theses findings (Bogart, Kästner, Herbsleb, & Thung, 2021). However, ensuring backward compatibility is difficult in practice. Venturini et al. empirically studied breaking changes and found that more than 11% of all downstream packages in npm experienced breaking changes while updating to a non-major release (Venturini, Cogo, Polato, Gerosa, & Wiese, 2023). The majority of these breaking changes were not a result of a change in a direct dependency, but trickled downstream from transitive dependencies. The authors also found that close to 22% of breaking changes are never documented. Moller et al. proposed a pattern matching technique based on static analysis for detecting locations in JavaScript programs that are affected by breaking changes (Møller, Nielsen, & Torp, 2020). Mujahid et al. proposed a crow-sourced technique that leverages the automated test cases of downstream packages that depend on the same upstream package to detect breaking changes and notify downstream dependents (Mujahid, Abdalkareem,

Shihab, & McIntosh, 2020).

Fear of breaking changes has given room for another dependency-related issue in the npm ecosystem- outdated dependencies. Zerouali et al. introduced the concept of technical lag for dependencies in a software ecosystem which measure how far behind a package is with respect to the latest version of their dependencies (Zerouali, Constantinou, Mens, Robles, & González-Barahona, 2018). They found an average technical lag of 3.5 months in npm, which indicates a reluctance to update dependencies. Decan et al. studied the evolution of technical lag in npm and found that one in four dependencies in the ecosystem suffer from technical lag (Decan et al., 2018a). They also found that most dependencies with technical lag, increase their technical lag over their lifespan. Outdated dependencies is not a problem specific to npm. Kula et al. studied the Maven ecosystem and found that 80% of the projects have outdated dependencies (Kula, German, et al., 2018a). Technical lag is often associated with a reluctance to update, but it may be created by deliberate decision-making. Cogo et al. studied dependency downgrades in the npm package ecosystem. They found that downgrades occur a reactive or preventive measure by the downstream package and often leads to a more conservative use of the upstream dependency (Cogo et al., 2019). In another study, Cogo et al. investigated the usage of deprecated packages in npm and found that 27% of downstream packages directly rely on deprecated releases (Cogo, Oliva, & Hassan, 2021). Additionally, they found that 54% of packages transitively rely on deprecated releases.

Software ecosystems suffer from a great deal of dependency related problems which arise from the decisions and habits of both downstream dependents and upstream maintainers. In order to understand the prevalence and impact of bad dependency management decisions in downstream dependents, **Chapter 3** presents an empirical study on JavaScript projects to identify, catalog and quantify recurring dependency management issues that can lead to breaking changes or technical lag in the npm ecosystem. In **Chapter 4**, we shift the

focus from the decisions of downstream dependents to understanding the characteristics of upstream packages that influence how downstream dependents update their packages. We propose a solution to updating dependencies by extracting the relationship between package characteristics and the wisdom of the crowds. Our findings help developers in managing their direct dependencies.

2.2.3 Security Vulnerabilities in npm Packages

The specific characteristics of software ecosystems create a breeding ground for security vulnerabilities. The npm ecosystem in particular, emphasizes heavy reuse (Kikas et al., 2017; Zimmermann et al., 2019) and tends to have a larger number of transitive dependencies (Decan, Mens, & Claes, 2017; Decan et al., 2019). Consequently, certain packages in npm have far-reaching influence (Decan et al., 2017) and can expose large portions of the ecosystem if they are infected with a vulnerability. Zimmermann et al. studied the security threats in the npm ecosystem. They found that installing an average npm package introduces an implicit trust on 79 third-party packages and highly popular packages can influence more than 100,000 other packages (Zimmermann et al., 2019). The authors also observed that up to 40% of the packages in npm rely on dependencies with publicly disclosed vulnerabilities. Zerouali et al. explored the transitive impact of vulnerabilities in the npm ecosystem and found that it takes up to 7 years to uncover and disclose half of the hidden vulnerabilities in the ecosystem (Zerouali, Mens, Decan, & De Roover, 2022). They also found that the number of dependents exposed to vulnerabilities through transitive dependencies are twice the number exposed through direct dependencies. Pashchenko et al. conducted interviews with industry practitioners and found that developers are concerned about using packages that result in too many transitive dependencies (Pashchenko, Vu, & Massacci, 2020).

The existence of long dependency chains in the npm ecosystem means that even when a

vulnerability is fixed, it is not immediately propagated through the ecosystem (Chinthanet et al., 2021) and portions of the ecosystem continue to suffer the effects of vulnerabilities. Decan et al. empirically studied the impact of vulnerabilities on the npm ecosystem and found that the number of vulnerabilities is on the rise (Decan et al., 2018b). The authors found that many a reluctance to update means packages remain vulnerable via their dependencies even after the fix has been released. Chinthanet et al. investigated the release and adoption of vulnerabilities in npm and found that relying on patch releases is not enough to receive vulnerability fixes as they are often bundled into minor and major releases (Chinthanet et al., 2021). The authors also observed that the severity of vulnerabilities influence the propagation speed of the fix throughout the ecosystem. Alfadel et al. analyzed Node.js applications and found that the main reason for vulnerable dependencies in a project was simply the refusal to update to a newer version of the dependency (Alfadel, Costa, Shihab, & Adams, 2023). They also observed that half of their applications were exposed to publicly disclosed vulnerabilities for more than 3 months.

Security vulnerabilities are a major problem in software ecosystems. Since package maintainers do not equally attend to their vulnerable dependencies, publicly disclosed vulnerabilities linger in the package dependency chains long after the fix has been released. Vulnerabilities from our direct dependencies are one of the negatives consequences of the bad dependency management decisions studied in **Chapter 3**. The impact of vulnerabilities can be alleviated by choosing a suitable strategy to update dependencies, as proposed in **Chapter 4**. However, update strategies for direct dependencies are less effective for controlling the transitive effect of upstream packages further along the dependency chain. In **Chapter 5**, we propose a solution to mitigate the impact of vulnerabilities from transitive dependencies. We conduct an empirical study on the responsiveness of npm packages to vulnerability fixes and identify the characteristics of fast-responder packages.

Chapter 3

Challenges in Dependency Management

Dependency management in modern software development poses many challenges for developers who wish to stay up to date with the latest features and fixes whilst ensuring backwards compatibility. Project maintainers have opted for varied, and sometimes conflicting, approaches for maintaining their dependencies. Opting for unsuitable approaches can introduce bugs and vulnerabilities into the project, introduce breaking changes, cause extraneous installations, and reduce dependency understandability, making it harder for others to contribute effectively.

In this chapter, we empirically examine evidence of recurring dependency management issues (dependency smells). We look at the commit data for a dataset of 1,146 active JavaScript repositories to catalog, quantify and understand dependency smells. Through a series of surveys with practitioners, we identify and quantify seven dependency smells with varying degrees of popularity and investigate why they are introduced throughout project history. Our findings indicate that dependency smells are prevalent in JavaScript projects with two or more distinct smells appearing in 80% of the projects, but they generally infect a minority of a project's dependencies. Our observations show that the number of dependency smells tend to increase over time. Practitioners agree that dependency smells bring

about many problems including security threats, bugs, dependency breakage, runtime errors, and other maintenance issues. These smells are generally introduced as developers react to dependency misbehaviour and the shortcomings of the *npm* ecosystem.

3.1 Introduction

Software ecosystems have completely changed the way we build software, by enabling code reuse in large scale through software packages. Developers now rely on an increasingly high number of packages to build their programs, reusing code to increase productivity, improve software quality and decrease time-to-market (Lim, 1994; Mohagheghi, Conradi, Killi, & Schwarz, 2004). However, this development paradigm creates a lot of dependencies, and managing these dependencies has become a key issue for developers (Artho et al., 2012; Bogart et al., 2016; Decan et al., 2019). An example incident in *npm* (Node Package Manager) is the release of a backward incompatible minor version 1.7.0 of the package "underscore" that caused many complaints among dependent packages about underscore not respecting Semantic Versioning (SemVer) (Chatfield, 2014). Another anecdote is the removal of the "left-pad" package that caused widespread breakage among big internet sites like Facebook, AirBnB, and Netflix (MacDonald, 2018).

Semantic Versioning (SemVer) has been presented as a solution to help effectively manage dependencies (Preston-Werner, 2019). It allows maintainers to automatically receive fixes and minor updates, while also limiting their exposure to breaking changes. However, previous research has shown that developers do not always conform to SemVer (Chinthanet et al., 2019; Dietrich et al., 2019; Kula, German, Ouni, Ishio, & Inoue, 2018b; Wittern et al., 2016). This has created major problems due to outdated dependencies and breaking changes. There is evidence that up to 40% of the packages on *npm* rely on at least one package with a publicly disclosed vulnerability (Zimmermann et al., 2019) and up to 53% of package releases on *npm* suffer from some sort of technical lag (Decan et al., 2018a).

Simply allowing dependencies to automatically update is also problematic. A survey of 2,000 developers across different ecosystems has reported that 70% of the Node.JS developers have experienced breaking changes caused from updates when building their package (Bogart et al., 2017). Although the reasons and impacts regarding the circumvention of guidelines such as SemVer and opting for alternative approaches have been referenced in the literature, they are usually studied as a side-topic to explore other issues such as technical lag (Decan et al., 2018a) or security vulnerabilities (Zimmermann et al., 2019).

We argue that in order to better improve the management of dependencies, there is a need for a study that specifically focuses on dependency issues. Hence, the objective of this chapter is to catalog, quantify, and understand these dependency issues, which we refer to as dependency smells. *Dependency smells are recurring violations of dependency management guidelines that have negative consequences on the project and the ecosystem.*

Through our empirical study, we curated and analyzed a dataset of 1,146 open source JavaScript projects. First, we provide the definition and description of seven identified dependency smells: pinned dependency, URL dependency, restrictive constraint, permissive constraint, no package-lock, unused dependency, and missing dependency. These definitions are validated with an initial survey of twelve practitioners. We then qualitatively investigate their advantages and disadvantages through a survey with 41 JavaScript practitioners. We then conduct an empirical study to examine the prevalence of these dependency smells, and investigate why they are introduced in the studied JavaScript projects. Our study is formalized through the following research questions:

RQ1: How prevalent are JavaScript dependency smells? We built a tool that detects the aforementioned defined dependency smells in the 1,146 JavaScript projects in our dataset. Our findings reveal that dependency smells are prevalent in JavaScript projects. The results reveal that 80% of the projects are infected with two or more distinct smells. Not all smells occur at a large degree. Four out of seven appear in less than 30% of projects and the

majority of smells infect a minority (17%) of a project's dependencies. However, the results hint at inadequate attention to dependency maintenance or a lack of awareness regarding best practices in dependency management.

RQ2: How do developers perceive dependency smells and their negative impact? We crafted a questionnaire and surveyed 41 practitioners to quantify their agreement/disagreement on the consequences and potential rationales of dependency smells. Developers confirmed the harmful nature of these smells and they were even more critical than we anticipated.

RQ3: Why are these smells introduced in JavaScript projects? We asked developers why they opted to introduce a smell rather than using the alternatives suggested by *npm* or SemVer. We aggregated the 28 responses into 14 reasons. Our findings show that experiencing breaking changes and needing a fix not yet published on *npm* were among the most cited reasons for introducing a dependency smell.

Looking through the evolution of the dependency smells over time, we observe that these dependency smells are addressed, but new smells are introduced more frequently than old smells are being fixed. This has caused an overall upward trend in their accumulation.

This study presents (i) A catalog of dependency smells crafted and validated, by quantifying responses from JavaScript practitioners, (ii) A large-scale empirical study of dependency smells in 1,146 popular JavaScript projects, and (iii) A prototype tool named DependencySniffer (Javan Jafari, 2020) that can analyze any JavaScript project that uses *npm* and detect the presence of dependency smells. The tool can potentially be incorporated into CI pipelines to prevent dependency smells from rippling through software projects.

The rest of the chapter is organized as follows. Section 3.2 introduces our catalog of dependency smells. Section 3.3 and 3.4 present our dataset and the smell detection technique for the empirical analysis. The results for the three RQs are presented in Section 3.5. We discuss smell evolution in Section 3.6 and we present our implications in Section

3.8. The related works are discussed in Section 3.9. Section 3.10 highlight the limitations of our study and Section 3.11 concludes the chapter.

3.2 Dependency Smells

In this section, we define each dependency smell, the rationale for why it might occur, and its negative consequences (Section 3.2.1). Similar to the code smell literature, whether or not something is a smell is based on the context (Fontana, Dietrich, Walter, Yamashita, & Zanoni, 2016). These smells have not been explicitly defined in the literature. They are initially observed by studying violations of *npm* recommendations, violations of SemVer guidelines and by studying the discussions surrounding the package.json file. We have followed up the initial observations with a survey from twelve practitioners in the field to better understand the negative consequences of each smell and why they might occur. We sent out an open-format questionnaire (where respondents are free to write anything) to seventeen JavaScript developers that we knew had sufficient experience with developing JavaScript projects. The developers were a convenient sample from Canada and Brazil that were known by the authors. They were contacted by email. We presented them with different approaches of depending on an *npm* package and asked them to write down the advantages and disadvantages associated with each approach. Since we did not want to bias the developers into associating negative consequences with the presented approaches, we refrained from calling an approach a "smell" and also included the SemVer-compliant alternative among our approaches. More importantly, we asked them to give us both advantages and disadvantages for each approach. The advantages not only help in reducing bias but also help in understanding why a particular dependency smell might occur. In order to ensure a similar understanding of dependency management and SemVer, we provided the participants with links to the *npm* dependency reference (npm Documentation, 2019a) and the SemVer standard guide (Preston-Werner, 2019) .

From the seventeen invites, twelve developers responded to our first survey (i.e,. 70% of response rate). From the 12 respondents, 10 identified themselves as industry practitioners and 2 participants were students. All participants have used *npm* and 11 out of 12 participants had at least one year of experience in Node.JS development. From the 12 respondents, 11 were familiar with SemVer. For each dependency smell, we categorized the 12 responses using an open-coding approach similar to the guidelines expressed by Philip Burnard (Burnard, 1991). Through an iterative process for each smell, we extracted and grouped relevant themes from developer responses into reasons for why a smell occurs and reasons for why it can cause problems.

The smells identified in this study focus on *how* dependencies are used and managed, rather than *what* dependencies are selected. Therefore, issues such as using too many dependencies or the wrong selection of dependencies are not covered, as they are specific to the requirements and functionalities of each project.

3.2.1 Dependency Smells Catalog

This smells catalog consists of a list of seven identified dependency smells. We do not claim this to be an exhaustive list, but given that *npm* limits the possible expressions of dependency constraints in the package.json file, the smells presented here cover the majority of different alternatives for depending on a package. Figure 3.1 presents examples of the restrictive constraint, pinned dependency, permissive constraint, and URL dependency smells. Other dependency smells are not identifiable solely by looking at the package.json file and they require closer inspection of the source code or project directory.

In identifying the negative consequences associated with each pattern, along with the reasons for why they might occur, we relied on developers' feedback as well as the inherent *npm* rules for evaluating dependency constraints. In each smell's definition, we give examples of developer responses (Tagged P1 through P12). Table 3.1 presents an overview

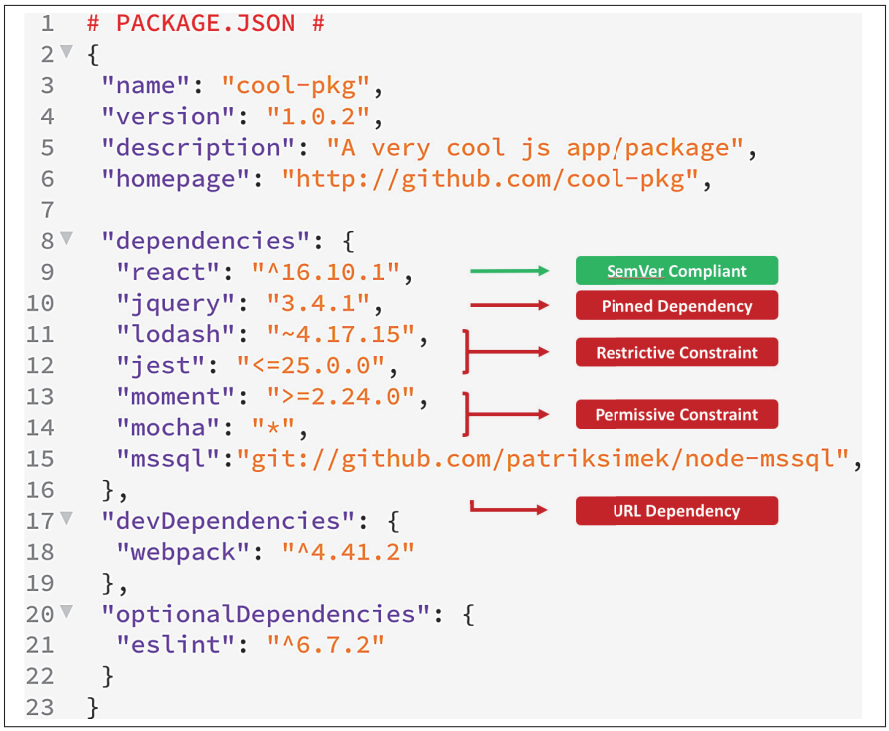


Figure 3.1: Example of a smelly package.json file

of the smells and their negative consequences.

Note that in this first survey we did not ask about the Unused Dependency and the Missing Dependency smell. These smells are included as they are both a clear case of dependency mismanagement. They create a mismatch between dependencies used in the source code and the ones defined in package.json which is a violation of npm guidelines (npm Documentation, 2019a). Hence, we do not expect any developers to consider them as a valid approach with positive outcomes. Missing dependencies can cause runtime errors or crashes when executing the code that relies on the dependency. Unused dependencies unnecessarily increase installation size and maintenance effort required for dependencies. Soto-Valero et al. (Soto-Valero et al., 2021) use the term bloated dependencies to study unused dependencies in the Maven ecosystem. Depcheck is an npm tool which reports unused and missing dependencies for npm projects (J. Li & Lukic, 2019). Additionally, we did not ask developer's opinion on not using package lock, which is the smell in our catalog (No

Table 3.1: Overview of dependency smells.

Smell	Description	Consequence
Pinned Dependency	Using a fixed version	Not receiving fixes and manual workload
URL Dependency	Fetching from a URL	Increased security risks and link breakage
Restrictive Constraint	Only updating to patch releases	Not receiving compatible features and fixes
Permissive Constraint	Allowing major version updates	Significantly higher risk of breaking changes
No Package-Lock	No package-lock in repository	No guarantee of consistent installation
Unused Dependency	Installing unused packages	Unnecessarily bloated dependency folder
Missing Dependency	Omitting a needed package	Code breakage or ambiguous dependencies

package-lock). Instead, we ask the advantages and disadvantages of *using* package.lock in the project, given this more intuitive and easier to assess than the consequences of not using an approach. We then use the pros of using a package.lock as the cons of not having package.lock in the project, and vice versa.

S1 - Pinned dependency:

Description: A pinned dependency is a type of restrictive approach in which only a single version of the dependency is accepted, for example pinning to version 1.2.3.

Why it occurs: Choosing to pin dependencies is a common approach to ensure fine-grained control over the software packages that a project depends on. It is sometimes even recommended for runtime dependencies of deployable projects to ensure consistent installations. This approach drastically reduces the risk of breaking changes and unknown bugs or vulnerabilities that may exist only in newer releases. For example, P12 said: *"Makes it extremely unlikely for breaking changes to happen ..."*.

When it is a dependency smell: By pinning a dependency, we will not receive important security and bug fixes, as P12 mentions: *"you are more exposed to security risks and bugs"*. It therefore leaves our project less and less secure over time, unless manual measures are taken to constantly update the constraint to a newer version, which will require extra effort. For example, P10 said: *"[It] creates technical debt in dependencies. Someone has to put in the time later to update them ..."*. Having pinned dependencies is even worse for packages

which will later be used as dependencies in other projects, since *npm* can no longer re-use similar dependencies for installed packages but must rather have separate installations for the different versions, which can cause a significant increase in the number of installed dependencies for a project. For example, P10 cited: *"...increases node modules size because different packages can't share the same installation"*.

S2 - URL dependency:

Description: URL dependencies are constraints that directly point to an online URL, often pointing to a repository link such as GitHub. The URL may point to a general repository or a specific release or branch in that repository. Figure 3.1 contains an example of a URL dependency.

Why it occurs: The most plausible reason for choosing a URL dependency is in cases where there is no equivalent on the *npm* package manager, as P2 states: *"Point to a package outside npm listings"*. Another potential reason is instances where the dependent strongly needs a new fix in the repository that has not yet made its way to the latest release on *npm*. For example, P7 said: *"... the fix is applied to the GH repo way before we see it on npm. ..."*.

When it is a dependency smell: URL dependency constraints are problematic for a variety of reasons. If the URL points to a master branch of a software repository it can lead to breaking changes since any new change to the master branch will be fetched. If the URL points to a specific release of a package on a repository or other type of webpage, it is akin to a pinned dependency. In any case, it is very difficult to adhere to SemVer guidelines whilst using URL constraints for dependencies. In addition, since developers are operating outside *npm*, they cannot take advantage of its project metrics and security advisories, and they are more vulnerable to typosquatting attacks (different and malicious packages with

very similar names) (npm, 2017). Apart from security issues, the unstable nature of unreleased packages can also introduce more bugs, as mentioned by P9: *"... vulnerable to typo-squatting attacks and even if the repository is legit, you are depending on something unstable ..."*. It can also be harder to identify if a new version is released if the URL does not point to a code repository. Additionally, *npm* prohibits the deletion of uploaded packages after a grace period, but a URL can change or be removed completely at the owner's discretion, leaving developers with missing dependencies. For example, P8 cites: *"If the linked library disappear then code will stop working ..."*. If the package exists on *npm*, one should use the published version. Otherwise, it may be possible to avoid this dependency in favor of its *npm*-established alternative.

S3 - Restrictive constraint:

Description: Restrictive constraints are any dependency constraints that are more restrictive than SemVer. For post-1.0.0 releases, this is equivalent to only accepting patch updates using the "~" notation (e.g. ~2.1.3) or by using the "x" notation in the patch component of a constraint (e.g. 2.1.x). While pinned dependencies are a special case of restrictive constraint, we opt for separating them in our study for clarity purposes.

Why it occurs: When a developer knowingly opts for a more restrictive approach in specifying dependency constraints, it is likely due to fear of breaking changes from automatic updates. For example, P12 said: *"... reduce[s] the chance of breakage"*. Indeed, adhering to SemVer only guarantees backward compatible updates if the maintainers of the dependency also adhere to SemVer. This verification is not straightforward and it can be difficult to ensure backward compatibility in practice (Decan & Mens, 2019b). Nonetheless, restrictive constraints still allow patch updates, as P8 states: *"This will make sure your libraries are up to date in regard to security patches"*.

When it is a dependency smell: Restrictive constraints prevent the project from receiving

backward compatible updates. For example, P10 mentions: *"I will not get "safe" feature updates"*. In practice however, the situation could be even worse. Since minor releases can also contain crucial fixes which are not necessarily back-ported to previous patch releases, using the restrictive constraint does not even guarantee we will receive the majority of important security or bug fixes. For example, P9 states: *"quite common to see lots of bugs and vulnerabilities addressed in a minor which never make it to any of the previous patch releases"*. Alternatively, restrictive dependencies have to be manually changed more frequently to keep up with newer versions. This creates extra work for developers, as mentioned by P2: *"Increased work required to update the system"*.

S4 - Permissive constraint:

Description: Permissive constraints are any dependency constraints that are more permissive than SemVer. For post-1.0.0 releases, this is equivalent to accepting major updates using the * or "" wildcards, or by specifying a minimum version range without a maximum limit such as $> 1.2.3$.

Why it occurs: In cases where there is an inter-dependency between dependencies maintained by the same developers, they might freely accept any updates since they are in charge of both projects and are therefore more confident in the type of changes they make. For some, removing restrictions is an easy way to make sure the latest updates are fetched. For example, P9 said: *"It caters to my inner laziness"*.

When it is a dependency smell: Permissive constraints expose projects to the constant risk of breaking changes, as stated by P10: *"Sooner or later, an update will break the API"*. If the maintainers pinpoint the breaking changes immediately, they have to manually change the constraint to an earlier version, which they have first to identify to work correctly. This is not always easy if semantically breaking changes have creeped up during several updates. In addition, it is irrational to claim a project will work with all future versions of a

dependency. In fact, the Cargo ecosystem has prohibited the use of wildcards for this very reason (Rust-lang, 2018).

S5 - No package-lock:

Description: The *npm* package-lock file includes the entire dependency tree with precise dependency versions at a specific time, which make sure the project will be built. The *npm* documentation recommends that the package-lock should be committed into source repositories (npm Documentation, 2019b). However, we observe that developers do not always include package-lock (or its similar counterparts: yarn.lock or npm-shrinkwrap.json) in their project repositories.

When it occurs: Since package-lock is auto-generated, this smell occurs because developers chose not to commit the file into their repository. Since the package-lock does not affect the package.json file and it can be rebuilt if the desired dependencies change, or it can be ignored altogether, we do not see any rationale for not including it, and developers seem to generally agree. For example, P12 said: *"I don't see any issues with having package-lock"*.

Why it is a dependency smell: In the absence of the package-lock file, there is no way to guarantee that installation at different times will yield the same result (even if the dependencies are pinned), as stated by P10: *"everyone can use it [package-lock] to install the same exact thing"*. In a sense, package-lock provides the best of both worlds by allowing developers to comfortably use SemVer in their package.json file while giving the option to install using a pinned dependency tree. Without a package-lock there is no way to track the history of dependencies without uploading the large node_modules folder. For example, P9 mentions: *"... if I don't use package-lock ... I need to push my node modules folder"*.

S6 - Unused dependency:

Description: The unused dependency smell represents runtime dependencies that exist in

the package.json file, but are not used in source code of the project.

Why it occurs: The unused dependency occurs when the code and import statements that needed the dependency were removed but the package.json file has not been cleaned up to properly reflect this. Alternatively, the dependency may only be used in development environments but it is incorrectly listed under runtime dependencies instead of the development dependencies in the package.json file.

When it is a dependency smell: If the dependency was actually removed from the project, then including it in the package.json file will cause an extraneous package installation and take up unnecessary space in the modules folder. Even if the dependency is used in development but incorrectly specified under runtime dependencies rather than development dependencies, it will result in the same extraneous installation when installing a project for production. Additionally, both cases will cause confusion for users and developers of the project. In any case, a mismatch between the dependencies imported in the source code and the ones defined in the package.json is a violation of *npm* guidelines (npm Documentation, 2019a).

S7 - Missing Dependency:

Description: The missing dependency smell represents packages that are used in the source code, but not specified in the runtime dependencies of the package.json file.

Why it occurs: The missing dependency smell can occur when a dependency is manually installed but not added to the package.json (by using the '–save' option with the *npm* install command). Additionally, it can appear because the dependency is removed from the project, and thus the package.json file, but the import statements are not fully cleaned up from the code. Alternatively, a missing dependency may still be used in the code but it has not yet raised errors because it exists in a seldom-reached execution path.

When it is a dependency smell: Missing dependencies can be very problematic if they

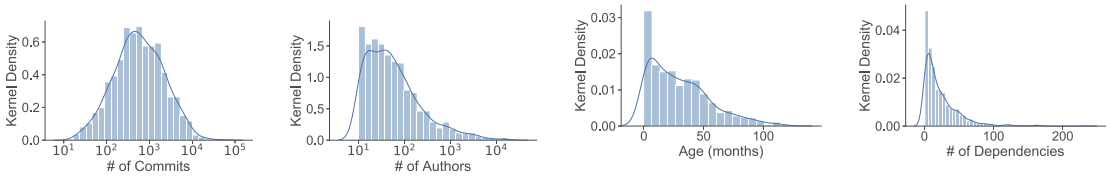
are in fact used in the source code but are not properly specified in the package.json file. This can result in bugs or crashes while executing the part of the source code that relies on this dependency. If the missing dependency occurs because the dependency is imported in source code but never actually used, it will still create ambiguity for users and developers of the source code. In any case, a mismatch between the dependencies imported in the code and the ones defined in the package.json is a violation of *npm* guidelines (npm Documentation, 2019a).

3.3 Dataset

We gather a large set of open-source JavaScript projects that are hosted on GitHub to create our dataset. We chose to study projects written in JavaScript since it is the most popular programming language on Github (GitHub, 2019). We also study *npm* since it is the official registry of JavaScript packages with more than 1.5 million packages (Libraries.io, 2020).

We start our data collection by retrieving the GitHub metadata from GHTorrent (Gousios, 2019) containing data until June 1st, 2019. We use this metadata to identify non-forked open-source Javascript projects that have a minimum of 10 contributing authors (as determined by GitHub) and at least 10 commits since January 2019, that is, six months before the data collection started. With this criteria, we aim at filtering projects that are actively maintained by at least a medium sized team of developers and reduce the chances of skewing our results towards abandoned and toy projects, prevalent on GitHub (Abdalkareem et al., 2017; Kalliamvakou et al., 2014).

Using our selection criteria, we identified a total of 2,216 candidate projects. We further prune this dataset by 1) removing 569 projects that do not contain package.json, making it impossible to reliably pinpoint their dependencies and 2) removing 413 projects that had no runtime dependencies in their package.json, as development and optional dependencies



(a) Number of commits (b) Number of authors (c) Age of the projects (months) (d) Number of dependencies

Figure 3.2: Distribution of projects in the dataset under four perspectives: age, authors, commits and dependencies

are not in the scope of this study. We were left with 1,146 projects as our dataset. This accounts for 26,924 dependency constraints (in the latest snapshot).

Figure 3.2 presents the dataset distribution for the number of commits per project, the number of authors per project, the projects' age in months and the number of dependencies per project. Our dataset is comprised of 1,541,504 commits, with a median of 555 commits per project. It also contains 29,400 total authors with a median of 21 authors per project. The median age for the projects is 24 months. The dataset contains a total of 26,924 dependencies with a median of 15 dependencies per project. These statistics are indicative of the diversity of our dataset, comprised of well-maintained and active open-source JavaScript projects. A replication package of our study is available on Zenodo (Javan Jafari, Elias Costa, Abdalkareem, Shihab, & Tsantalis, 2021).

3.4 Smell Detection

Since we are aiming at analyzing a large dataset of JavaScript projects, some of our analyses are automated using a tool we developed specifically for this purpose. Our tool can easily and effectively be used to analyze any JavaScript project and detect current dependency smells. The tool prototype (`DependencySniffer`) is open source and accessible to everyone (Javan Jafari, 2020).

For the first four smells (pinned dependency, URL dependency, restrictive constraint,

permissive constraint), we go through the projects in our dataset and parse the package.json and their related commits and commit diffs. The parser is a prominent part of our detection because the plain textual diffs are not useful for automated analysis and they can be error-prone. We first parse the package.json in the latest snapshot of the project and store all dependency profiles in a structured SQL database. These dependency profiles can be queried to obtain information such as the name, version constraint and existing smells. The first four smells are identified using regex rules on the version constraints. We manually checked the results of 50 smell extractions to fine-tune the rules and ensure the rules properly identify the first four smells. The fifth dependency smell (No-package-lock) can be identified by looking through a projects root directory. We used the depcheck external tool to find unused and missing dependencies. In order to investigate the evolution of dependency smells over project history, we need to gather the historical information. We use `git log` to identify the commits that modify the package.json file. We then run a large contextual `git diff` on those commits, i.e., diffs that also include unchanged lines, to properly parse the entire JSON dependency object into a SQL database. To do so, we perform an intersection of the dependencies in the *before* and *after* commits. The elements that are not in the intersection from the before commit are ”deleted” dependencies. The elements that are not in the intersection from the after commit are ”added” dependencies. The elements in both the ”added” and ”deleted” sets that refer to the same dependency but with a different version or different constraint are ”modified” dependencies. This database contains added, removed, and modified dependencies in each commit. Differentiating between these operations are important because we do not want to flag a smells as fixed, due to the removal of the dependency from the package.json file. Aside from the dependency profile, this database captures commit-relevant information (e.g., commit message, commit date) needed for answering RQ3.

3.5 Results

In this section, we motivate our research questions, describe the approaches to answer them, and present their results.

3.5.1 RQ1: How prevalent are JavaScript dependency smells?

Motivation: Thus far, we have catalogued seven dependency smells and described their negative consequences in software projects. In this question, we aim at finding empirical evidence of their existence to better understand their prevalence and importance. We look at the current snapshot of projects to see which smells are more common, which will give us a clear picture of the current landscape and guide our in-depth analysis.

Approach: We parse the package.json file of each project in our dataset and create a structured database for the dependencies (Section 3.4) to identify the occurrence of the dependency smells S1 to S4 (see Table 3.2). For the no package-lock smell (S5), we verify the existence of package-lock.json and other lock files including npm-shrinkwrap.json and yarn.lock in the root directory of the studied projects. To identify the Unused Dependency (S6) and Missing Dependency (S7) smells, we resort to the Depcheck tool (J. Li & Lukic, 2019). Depcheck is a tool for analyzing dependencies in projects, reporting instances of unused dependencies (i.e., mentioned in package.json but not used in the code) or missing dependencies (i.e., absent in package.json but used in the runtime source code). It works by analyzing the package.json and comparing the declared dependencies with the dependencies used in the code. We parse Depcheck's output to analyze the occurrence of the two dependency smells.

Results: Table 3.2 shows the dependency smells and the number and percentage of JavaScript projects in our study that have these dependency smells. Overall, our findings show that dependency smells are prevalent, with four out of seven appearing in more than 25% of the projects. As can be seen in Table 3.2, developers choose to pin some of their dependencies

Table 3.2: Dependency smells in extracted projects.

#	Dependency Smell	Projects (%)	
S1	Pinned dependency	598	(52.2%)
S2	URL dependency	117	(10.2%)
S3	Restrictive constraint	106	(9.2%)
S4	Permissive constraint	41	(3.6%)
S5	No package-lock	304	(26.5%)
S6	Unused dependency	915	(79.8%)
S7	Missing Dependency	729	(63.6%)

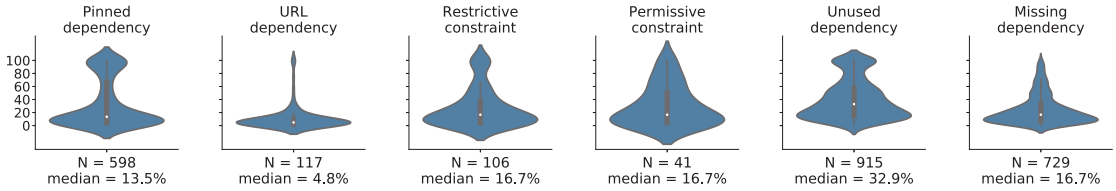


Figure 3.3: Distribution of dependency smells ratio in the latest snapshot of projects that contain at least one instance of the smell. We specify the number of projects plotted (N) and the median of the distribution (median).

in over 52% of the projects. This is because while pinning dependencies has many drawbacks, it is still common practice among developers who do not trust their depended-upon packages to properly abide by SemVer.

The package-lock could be used as a more appropriate alternative for pinning (although not exactly the same thing), since it pins the entire dependency tree while also allowing developers to abide by SemVer. Even if the project is not meant to be directly installed, it aides development cycles by preventing breakage in transitive dependencies (Section 3 presents other reasons for using package-lock). In any case, this lock file is effortless to generate and will be ignored when the package is fetched as a dependency from *npm*. However, package-lock is not used in over 26% of the projects.

In order to investigate this further, we cross-reference the data to identify whether these two dependency smells occur in the same projects, and present this analysis in Table 3.4. The results are quite interesting. More than 40% of the studied projects contain both

Table 3.3: Number of projects containing distinct smell types

# of Distinct Smells	# of Projects	% of Projects
Zero	41	3.6%
One	188	16.4%
Two	362	31.6%
Three	357	31.2%
Four	160	14%
Five	32	2.8%
Six	6	0.5%
Seven	0	0%

Table 3.4: Co-occurrence of pinned constraints with the existence of package-lock.

Package-lock	Dependency smell	Projects (%)
Exists	Contains pinned dep.	463 (40.4%)
Exists	Contains unpinned dep.	783 (68.3%)
Does not exist	Contains unpinned dep.	277 (24.2%)
Does not exist	Contains pinned dep.	135 (11.8%)

package-lock and pinned dependencies (row 1 in Table 3.4). We also observe that more than 68% of the projects that have a package-lock did not feel the need to pin all of their dependencies (row 2 of Table 3.4). Conversely, 11.8% of the projects had at last one instance of a pinned dependency but did not opt to include the package-lock. Note that the percentages in Table 3.4 should not add up to 100% because a single project can contain both pinned and unpinned dependencies.

As can be seen in Table 3.2, the URL dependency smell is less common but still appears in over 10% of the projects. This is followed by the restrictive constraint smell which appears in more than 9% of the projects. The Permissive constraint is perhaps the least common smell, appearing in less than 4% of the projects. This makes sense since this smell is almost never justified, unless we have complete control over the dependent package. Our results show that developers are much more likely to err on the side of caution when it comes to breaking changes, since restrictive constraints and pinned dependencies are

substantially more common than permissive constraints.

Unused and missing dependencies are surprisingly common, appearing in 79% and 63% of the projects, respectively. While the high incidence of unused dependencies indicates an overall lack of attention to dependency maintenance and can bloat the installation of some applications, our analysis show that the large majority of the projects require dependencies that are not declared in the package.json (missing dependencies), which can cause bugs and crashes. Overall, these results indicate JavaScript developers seem to struggle to maintain a healthy package.json file.

We studied the relationship between project characteristics such as age, number of dependencies, number of commits, and the number of maintainers on the occurrence of each dependency smell. Based on the mentioned characteristics, we found significant differences between smell-infected and smell-free projects using the Man-Whitney-U test for $p<0.05$. However, analyzing the effect size using Cliff's delta reveals the differences to be small or negligible for all characteristics except the number of dependencies (large effect size). This is not surprising as increasing the number of dependencies increases the likelihood of a project being infected by a smell.

So far, we have investigated whether projects contain dependency smells or whether smell occurrence is affected by project characteristics, but not the extent in which they occur within the projects. For instance, pinning a single dependency may have much less impact on a project maintenance than pinning half of its project dependencies. To put the extent of smell occurrences per project into perspective, we show in Figure 3.3 the distribution of *dependency smell percentage* on the projects with at least one instance of that smell. The dependency smell percentage normalizes the number of dependency smells with respect to the total number of the project dependencies, as projects with large number of dependencies will have a tendency to have more smells. In the case of Missing Dependency, we perform a different normalization, as these represent dependencies that are not specified

by the project. We calculate the dependency smell percentage by dividing the number of identified missing dependencies over the number of dependencies in package.json + the number of missing dependencies. Note that we did not include the no package-lock smell, as calculating the percentage for this particular smell makes no practical sense - a project either has the smell (100%) or not (0%).

All of the violin plots have a relatively non-uniform distribution, with the majority of smell distributions having a median of under 17%. In other words, smells usually infect a minority of a projects dependencies. In particular, projects tend to have a relatively small percentage of URL and Missing dependency smells, while Pinned and Unused dependencies occur in higher proportions in our dataset. Observing the pinned dependency plot near the 100% mark shows that some projects (7.5% of all projects) may decide to use pinning as a blanket dependency strategy regardless of the specifics of each dependency. On the other hand, the URL dependency smell is often used on specific cases. We further explore the reasons why these occur in Section 3.5.3.

The results reveal that 80% of the projects are infected with two or more distinct smells. However, not all smells occur at a large degree. Four out of seven appear in less than 30% of projects and the majority of smells infect a minority of a project's dependencies.

3.5.2 RQ2: How do developers perceive dependency smells and their negative impact?

Motivation: After realizing the prevalence of dependency smells in JavaScript projects, we want to understand how developers perceive these smells and their negative consequences. Do the developers agree that these smells are harmful? Do different smells have different levels of impact? This will help us understand if the prevalence of these smells are in fact a major concern.

Approach: In order to examine the negative consequences of dependency smells, we crafted a survey and asked respondents to rate their agreement/disagreement to a number of statements about each smell. We craft the statements by grouping the responses from the first survey from Section 3.2 into encompassing statements about each smell. We also included statements about using SemVer, to assess how developers see the use of this standard in practice.

To that aim, we create a questionnaire containing all statements in Table 3.6, and ask practitioners to rate their agreement on a likert-scale of 1 to 5 (strongly disagree, disagree, neither, agree, strongly agree). Note that some statements might seem positive/negative, but considering the level of disagreement from developers may reveal the opposite. For example, the positive statement that restrictive constraints "allow for all important security and bug fixes" has been met with disagreement, meaning the inverse is true. We were careful not to bias the participants in being negative towards the dependency smells. To do so, first, we refer to each of the smells as dependency approaches and include the SemVer compliant approach in the questionnaire. Second, to avoid confusing the participants, we did not use the labels used throughout this study (e.g., Restrictive constraint, URL Dependency) and instead focus in describing the smells as an "approach" for managing dependencies. For instance, instead of writing "Restrictive Constraint" we specified "Allowing *npm* to only install the latest patch updates". Instead of "URL Dependency" we use "Specifying a direct URL to fetch the dependency", and so on. The sole exception are the dependency smells Unused Dependency and Missing Dependency, which are clear dependency issues and we framed them as such in the questionnaire. The participants could also freely input comments on a text box for each dependency smell, to clarify and justify their agreement or further address aspects not mentioned in our questionnaire.

Similar to Section 3.2, we ask the advantages and disadvantages of *using* package.lock in the project, given this more intuitive and easier to assess than the consequences of not

Table 3.5: Background of participants in the survey.

Dimension	**Experience**	**%**
Background	Industry Practitioner	75.6%
	Academic Researcher	19.5%
	Student	4.9%
Node.JS	$>$ 5 years	24.4%
	4-5 years	24.4%
	1-3 years	41.5%
	$<$ 1 year	9.8%
npm use	Always	48.8%
	Most projects	43.9%
	Sometimes	4.9%
	Rarely	2.4%
SemVer	Completely familiar	80.5%
	Somewhat familiar	12.2%
	Not familiar	7.3%
Total participants		**41**

using an approach. We then use the pros of using a package.lock as the cons of not having package.lock in the project, and vice versa.

It is worth noting that we did not include statements that relates to how SemVer or *npm* operates, since our goal was to evaluate qualitative statements rather than quiz the respondents on how SemVer or *npm* works. For example, a common issue associated with pinned dependencies is that developers cannot automatically receive new updates. However, this is a fact and is not up to debate. Even though it exists in our smell definition, it does not appear in the survey. On the other hand, the security problems associated with pinning dependencies is a qualitative characteristic which depends on developers' experience.

To recruit our survey participants, we use a snowball sampling method (Goodman, 1961) where we posted this survey on Twitter and the Node.JS community on Spectrum. We received a total of 41 replies. As Table 3.5 shows, the set of respondents is composed primarily by industry practitioners (75.6%) and academic researchers 19.5%). The majority

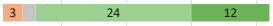
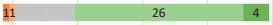
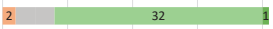
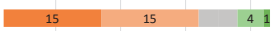
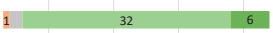
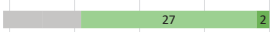
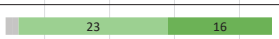
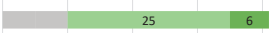
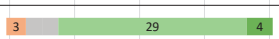
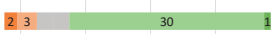
of the participants have more than a year of Node.JS development experience (83.3%), use *npm* on most or all of their projects (92.7%) and are completely familiar with *npm* (80.5%). We present the background and the experience of respondents with Node.JS, *npm* and SemVer in Table 3.5.

Results: We present the results of our survey in Table 3.6, showing per dependency smell and statement, the distribution of the agreement levels as a Divergent Stacked Bar chart. We center-align the bars on "Neither". Hence, a statement with mostly green bars tending to the right shows a more frequent agreement within the 41 participants. Conversely, statements with bars tending to the left (orange) convey a more frequent disagreement to the statement by the respondents. Note that each statement is either a rationale for using the dependency smell/approach (marked with '+') or a disadvantage of using the smell/approach as a valid approach (marked with '−').

Overall, the results shown on the third column of Table 3.6 indicates that developers substantially agree with 15 out of the 21 statements. Developers mostly agree with all statements related to the benefits of using SemVer, and the problems that may be caused by Unused and Missing Dependencies. Aside from this, we see a more common agreement on the downsides of using dependency smells. For instance, the three highly agreed upon statements related to Pinned Dependency are all related to the downsides of pinning dependency: increases bugs and vulnerabilities over time, increases installation size and creates extra overhead for maintainers. We observe a similar picture in the highly agreed upon statement for Permissive Constraint and URL Dependency, indicating that developers tend to agree with the downsides of using the dependency smells.

All the six statements where developers more frequently disagree are related to the rationale of using a dependency smell as a valid approach. For instance, most developers either disagree or strongly disagree that using a Permissive Constraint makes dependencies easier to manage. When justifying their position, developers pointed out that: *"It is*

Table 3.6: Survey results for quantifying smell characteristics. Each statement is either a rationale (+) for using the smell as a valid approach or a downside (−) of the dependency smell. The last column presents the aggregated total for each statement from 1 to 5 (strong disagreement to strong agreement)

Smell/Approach	Statement	Strongly Disagree / Disagree / Neither / Agree / Strongly Agree (-35 -25 -15 -5 5 15 25 35 45)	Overall
Pinned Dependency	− Increases bugs and vulnerabilities over time		4.1
	− Increases installation size		3.8
	− Creates extra overhead for manually updating		3.5
	+ Drastically reduces breaking changes		3.2
URL Dependency	− Can cause dependency breakage (link-removal)		4.6
	− Increases security risk and bugs		4.2
	+ Useful for non-npm packages and custom forks		3.8
	+ Useful for getting latest fixes not yet published		2.6
Restrictive Constraint	+ Significantly reduces breaking changes		3.3
	+ Allows for all important security and bug fixes		2.3
Permissive Constraint	− It is sure to lead to a breaking change		4.6
	+ Makes dependencies easier to manage		2.2
Package Lock	+ The best way to ensure consistent installations		4.9
	+ Provides a historical snapshots of dependencies		4.5
	− Only useful for applications not packages		2.1
Unused Dependency	− It will cause an extraneous installation of a dependency and waste storage memory		4.1
	− The mismatch between declared and used dependencies confuses users		3.8
Missing Dependency	− It will cause unexpected runtime errors		4.4
	− The mismatch between declared and used dependencies confuses users		3.9
SemVer Compliant	+ Ensures I receive necessary security and bug fixes		3.9
	+ Ensures I don't experience breaking changes		3.6

only easier to manage [the dependencies] if nothing breaks, which it will" and said that: *"Solves some vulnerabilities but causes others."* Another vastly disagreed upon statement, is related to Restrictive Constraint: Allows for all important security and bug fixes. Developers pointed out that: *"[...] maintainers bundle many fixes into non-patch releases"* and said that: *"(one) should never ONLY rely on patch releases for fixes."* Another highly contested view is related to the cons of using package.lock: Only useful for applications not packages. As we mentioned before, this statement actually points out a disadvantage of not using package.lock in projects. Most practitioners disagree, pointing out that: *"Even for packages, it decreases test flake in CI, causing less maintainer stress"*, showing that they believe there is not a good reason not to include a package.lock on JavaScript projects.

Overall, developers are even more conservative when it comes to rationalizing the usages of dependency smells than we anticipated. While all the downsides of using dependency smells as an approach were highly agreed upon by practitioners, they have disagreed with statements that attempt to rationalize dependency smells as valid approaches in particular cases.

Not only do practitioners confirm that dependency smells can be harmful to the maintenance and security of software projects, but practitioners are even critical towards rationales for turning to these smells in specific circumstances.

3.5.3 RQ3: Why are these smells introduced in JavaScript projects?

Motivation: Observing the prevalence and the impact of dependency smells leads us to ask the question: why are these smells introduced? We were keen to identify reasons on why developers have introduced a certain dependency smell and whether the introduction of a smell was a conscious effort by the developer or if it was merely a by-product of other development or maintenance tasks.

Approach: For this analysis, we reach out to 100 developers that have introduced a dependency smell in the studied projects in our dataset. We only consider the cases where a developer changed the dependency constraint from SemVer to a dependency smell. In such cases, we expect developers to be aware of semantic versioning and are more likely to remember what motivated them to use an alternative approach. We also prioritize more recent changes with the hope that the changes are still relatively fresh in their minds (although this resulted in an older change from 2015 for the permissive constraint smell to also be included, since this smell has less cases overall). We ignore ”no-reply” emails.

In total, we sent out 100 emails that reached a recipient, 25 for each of the 4 dependency smells: Pinned dependency, URL dependency, Restrictive constraint, and Permissive constraint. We omit Missing and Unused dependencies from this analysis, as these are clearly a symptom of dependency mismanagement and there are no valid rationales for introducing them. We also omit ”No Package-Lock” from this analysis as this is a project decision not easily attributed to a single developer we could contact to get information. It is also not a decision that regularly changes due to dependency modifications and is often related to the development culture of individual projects. Our email provided details of the change such as the dependency name, dependency constraint, and the commit message, along with GitHub links to the commit that caused the change. We asked developers ”why they switched to this approach instead of using SemVer like before”. Developers were suggested to justify their changes by simply replying to the email.

We received a total of 28 responses (i.e., response rate of 28%) where developers explained their reasoning for the change. The achieved response rate is significantly larger than the typical 5% rate found in questionnaire-based software engineering surveys (Singer, Sim, & Lethbridge, 2008). This is because developers were targeted based on their dependency actions and we also tried to provide all the relevant information in the email while keeping it concise and to the point. The responses were quite detailed in a free-text format,

providing not only the reasoning about the dependency change in question, but also rationales on why the developers might resort to dependency smells, along with cultural beliefs about semantic versioning and the centralized nature of *npm*.

To analyze the free-text responses provided by the developers, the first two authors have independently assigned a theme to each reason presented in the responses, using an open-coding approach (Fincher & Tenenberg, 2005). The two authors then discussed the themes encountered and merged them into 10 categories of reasons of why the dependency smells are introduced. In the two cases where the first two authors did not fully agree on the classification, the third author was consulted to provide a tie-breaker.

Results: The 10 reasons on why dependency smells are introduced are presented in Table 3.7. Note that the total number of responses in the table is more than 28, because many developers gave multiple reasons for introducing the dependency smells. We found at most four different reasons per dependency smell, while the most common reason on each smell was found in at least three responses. In the following, we explain each reason along with example responses from developers (Tagged P1 through P28):

R1-Breaking changes: The most cited reason for pinning dependencies and also for using restrictive constraints is breaking changes. Breaking changes are changes in a new release of a package that is incompatible with the previous API, causing a breakage in others that depend on the package. Developers tend to adopt a more cautious approach when a dependency breaks the API even though it was a minor or patch release. For example, P12 said: *"newer versions of "esm" broke GL JS tests in very obscure ways. I spent a day debugging and couldn't find a root cause for"*. It is also interesting to see that new features, while not causing API breakage, can still cause problems, as stated by P13: *"a new feature introduced in a minor release might be backwards compat in terms of the API, but not so much in terms of the user experience or expectations - e.g. a component might suddenly show some new option or expose some new behavior that we have yet to evaluate or have*

Table 3.7: Developer reasons on why dependency smells are introduced.

Smell	Reason	# Responses
Pinned Dependency	Breaking changes	5 (71%)
	Mistrust of SemVer compliance	3 (43%)
	Transitive dependency compatibility	1 (14%)
	Unaware of SemVer	1 (14%)
URL Dependency	Fix not on *npm*	7 (70%)
	Experimenting with features	4 (40%)
	Philosophical issue	3 (30%)
	Maintainer owns dependency	2 (20%)
Restrictive Constraint	Breaking changes	5 (83%)
	Mistrust of SemVer compliance	2 (33%)
	Transitive dependency compatibility	1 (17%)
Permissive Constraint	Breaking changes not likely	3 (60%)
	Unaware of SemVer	1 (20%)
	Reducing installation size	1 (20%)

approved by our clients / end users". Developers are in fact opting to pin dependencies as a direct response to breaking changes or unexpected updates.

R2-Mistrust of SemVer compliance: Some developers have not experienced breaking changes for the specific dependency, but past experiences with other dependencies have made them less trusting in SemVer and more conservative in their dependency approach. For example, P27 mentions: *"I think it is a result of several incidents with packages and my mistrust of [the] js community"*. In fact, not trusting dependency maintainers in *npm* to properly follow SemVer is the second highly cited reason for pinning a dependency or using a restrictive constraint, as cited by P16: *"The unfortunate reality is that not every maintainer correctly adheres to SerVer when publishing new versions of their own dependencies"*. Past experiences leave a bitter taste and it may take a while for some developers to opt for SemVer even after maintainers properly abide by it.

R3-Transitive dependency compatibility: Sometimes transitive dependencies can cause

incompatibility issues. Such cases arise when a transitive dependency requires only specific versions of a dependency to be installed, whereas *npm* will install multiple versions of a dependency if they are required by dependency packages. For example, P22 said: *"So to ensure that a transitive dependency issue do not occur I pinned the react-scripts version"*. Although developers are somewhat aware that this issue could be remedied by using peerDependencies (plugins that express compatibility with a specific host package but do not use it directly through the "require" statement (npm Documentation, 2019a)), as stated by P11: *"it's probably better to express the dependencies within the hierarchy as loose peer-dependencies"*.

R4-Unaware of SemVer: While not common, some developers mentioned their lack of familiarity with SemVer as the reason for pinning a dependency or using a permissive approach, as cited by P18: *"I really had no idea what the hell I was doing back in 2015, and had probably not yet even heard of semver"*. Some dependencies can still suffer from old practices in which SemVer was not a well-known standard.

R5-Fix not on *npm*: The most common reason for using a direct URL to fetch a dependency is the delay or reluctance of the maintainer in publishing a needed fix on *npm*. For example, P9 stated: *"There is a bug in the main npm module and there's a branch made to fix it but it hasn't been merged."*. In some cases, the developers even issued a fix to the repository but the maintainer did not respond, as mentioned by P2: *"I submitted a pull request to fix the vulnerability, but they never bothered merging it, and their system eventually closed the PR because it sat for too long"*. If the maintainers would work with users to publish a consistent and timely stream of patch releases with the newest fixes, there would be less incentive to circumvent *npm* to fetch dependencies.

R6-Experimenting with features: Another common reason for switching to a direct URL is to experiment with new features not yet ready for release, as mentioned by P9: *"I'm experimenting, and not sure if certain things are stable yet"*. This experimentation is also

useful if the developer wants to prepare for an upcoming transition to a new release. For example, P8 said: *"At the time I did that because I wanted to test against a specific branch"*. Some developers are fully aware of the unstable nature of using unpublished dependency versions and view it as a temporary experimentation.

R7-Philosophical issue: Interestingly, some developers have an inherent and somewhat philosophical issue with the idea of a centralized dependency registry. For example, P8 stated: *"I don't think it is beneficial to have a standard centralised registry for dependencies in any language. That's what npm has become to node and it is critical and prejudicial, as npm is a private company and can choose to do whatever it wants"*. In such cases, they believe using absolute URLs for JavaScript dependencies is a major step towards decentralization, as pointed out by P14: *"since JavaScript is an interpreted language I have a hunch that it's actually not a good thing to use a centralized registry"*. The philosophical standpoint against a centralized dependency model is an interesting and unexpected reason why some developers may never fully depend on *npm*, but such discussions often ignore the disadvantages of using URLs, such as link breakage and typo-squatting attacks.

R8-Maintainer owns dependency: We also observed that using a URL to use a development branch of a dependency managed by you can be useful for integration tests. For example, P26 cited: *"When we are developing something in '[anonymised]' or in any other dependency managed by us which are npm dependencies, we used to point to the Github development branch of the dependency in order to a more or less detailed validation of the changes"*. It is also useful if it is crucial to have a personal fork in order to have full control over the dependency. For example, P28 mentioned: *"We switched to our own forked version of the repo. This is to ensure that it works properly, that no vulnerabilities are introduced (since our app deals with people's money)"*. The main advantages and disadvantages of using URLs relates to "trust", but some developers trust a custom URL, or even trust their forks more than the official package due to increased control.

R9-Breaking changes not likely: Contrary to reasons 1 and 2, some developers have not experienced breaking changes. This is the most cited reason for using permissive constraints, as stated by P7: *"I might have done it differently if it was likely that we'd break people with this, but it just didn't seem likely."*. However, developers mention how this only works in specific circumstances when there is a high familiarity with the dependency, as mentioned by P25: *"while it works perfectly for our use case, this situation is not ideal from a semantic versioning perspective"*. Just as negative experience can shape one to rarely trust the compatibility promises of SemVer, those who have never experienced breaking changes may not be worried about their consequences.

R10-Reducing installation size: Another reason for using permissive constraints is so that users of your package can reuse dependency versions without *npm* installing multiple versions of the same package. For example, P7 mentioned: *"allowing more permissive use may allow users to reduce the total number of packages that they need to ship in their bundles"*. In these cases, if you already have a particular version of package A in your dependencies, and package B in your dependency tree relies on package A, it is more likely that package B does not need to install a separate version of package A. While overly permissive approach can create many issues, being overly restrictive could force clients into larger installations as *npm* cannot fully utilize reusability and needs to install multiple versions of the same package.

The introduction of dependency smells is generally the result of developers reacting to dependency misbehaviour and the shortcomings of the *npm* ecosystem. Nonetheless, some developers are unaware of SemVer and some are against a centralized dependency model altogether.

Table 3.8: Number of introduced and fixed instances per dependency smell.

#	Smell	Smell Instances	
		Introduced	Fixed
S1	Pinned dependency	5,814	1,640
S2	URL dependency	750	556
S3	Restrictive constraint	1,068	479
S4	Permissive constraint	99	96

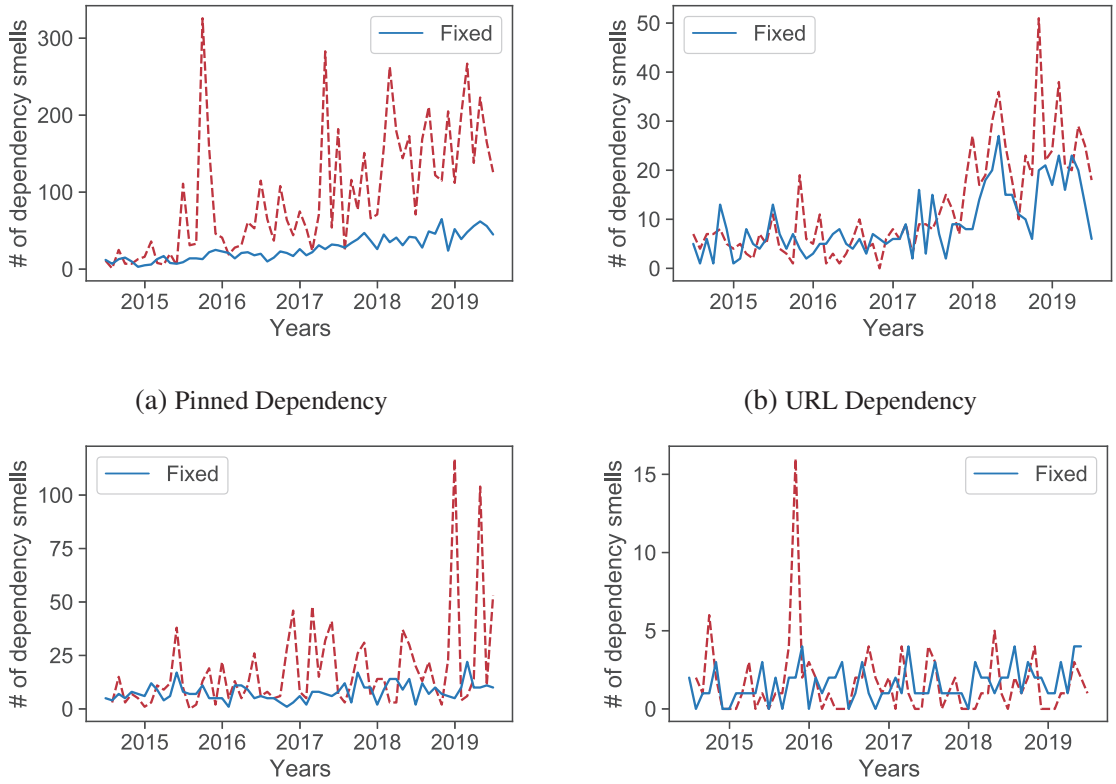
3.6 Dependency Smell Evolution

Now that we know dependency smells are prevalent, we want to investigate smell-inducing and smell-fixing commits throughout history to better understand their evolution. By analyzing the smells that were introduced or fixed in the past five years, we can better understand when these smells started to accumulate and where the current trend is moving towards.

Since we have the contextual diff for all commits parsed and stored into our database, we can identify when and how dependency constraints were changed throughout the project. We specifically focused only on dependencies that persist between two commits but their constraint was changed such that a dependency smell was introduced or fixed. This prevents noisy data due to fixes that are merely a result of dependency removal or migration. Since the analyses for this research question rely on the dependency constraints inside the package.json file, it can only be conducted on the first four dependency smells (pinned dependency, URL dependency, restrictive constraint, and permissive constraint). We have analyzed the changes over a 5 year period to provide a more comprehensive view of how the smells evolve through time.

Table 3.8 presents the total number of instances for each of the four smells that were introduced or fixed during a five-year period from mid 2014 to mid 2019. The line plots in Figure 3.4 show how the introduction and fix of dependency smells happen throughout project evolution. The fixed lines clearly show that developers are somewhat aware of these

smells and they regularly address them.



(a) Pinned Dependency

(b) URL Dependency

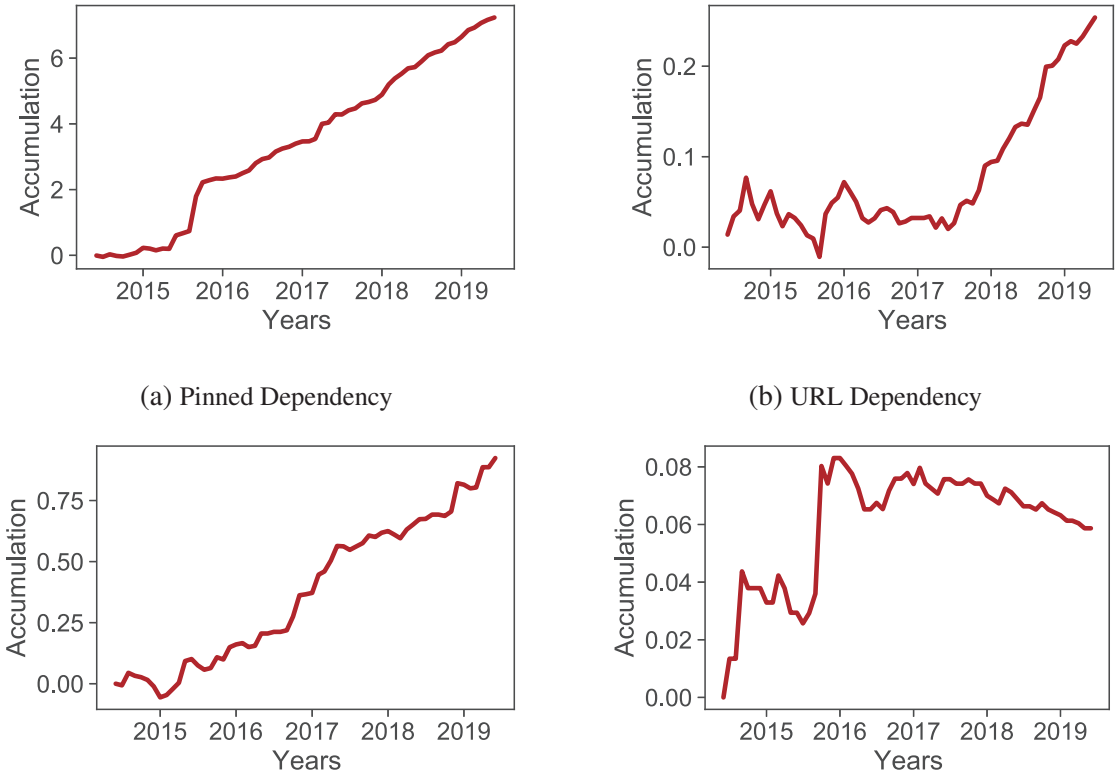
(c) Restrictive Constraint

(d) Permissive Constraint

Figure 3.4: Introduced and fixed dependency smells over time.

In order to investigate whether the regular fixes are enough to alleviate the regular introduction of dependency smells, we have plotted the accumulation of these dependency smells in Figure 3.5. This accumulation is normalized by the number of projects included in the dataset per month (since projects are added at different times). As can be seen, the pinned dependency, URL dependency, and restrictive constraint smells have an increasing trend. This shows that despite consistent dependency refactoring throughout the history of these projects, the number of newly added dependency smells are still higher.

The accumulation graph for the permissive constraint smell is noticeably different than the rest, both in its spiky form and its recently downwards trend. The non-gradual shape of the diagram might be due to the smaller number of instances for this smell (Table 3.8).



(a) Pinned Dependency

(b) URL Dependency

(c) Restrictive Constraint

(d) Permissive Constraint

Figure 3.5: Accumulation of dependency smells over time.

The permissive constraint smell subjects projects to a higher risk of breaking changes and the recently decreasing trend implies that developers generally agree that this smell is not justified, or the drawbacks far outweigh any benefits. It may be for this reason that another ecosystem like Cargo has banned the use of permissive wildcards in 2016 (Rust-lang, 2018). It may also be due to the severity of this smell. According to Table 3.6, Permissive Constraints have the highest likelihood of introducing breaking changes. Another reason for this downward trend could be the introduction of the semver-compliant caret (^) symbol in *npm* and its use as the default in February 2014 (Decan & Mens, 2019b; Oxley, 2014). This gives developers more freedom by allowing minor and patch updates (as opposed to only patch updates) and discourages the use of more permissive constraints..

Dependency smells are addressed and fixed through time, but most dependency smells

are introduced more frequently than they are fixed, causing an accumulation of smells over time. The permissive constraint smell is the only case where we observed a decrease over time.

3.7 Generalizability to other ecosystems

While our study focuses on *npm*, many of these smells can apply to other ecosystems with varying degrees. In fact, any package manager that allows developers to restrict dependencies to a particular version or version range is susceptible to pinned and restrictive dependency smells. It will also be susceptible to permissive and URL dependency smells if the package managers allow developers to always use the latest version of a package or fetch a dependency directly from a URL. Missing dependencies (and to a lesser extent, Unused dependencies) are mainly a problem for interpreted languages such as JavaScript and Python. However, compiled languages such as C++ (and even languages like Java) can catch these issues at compile time. For example, the PyPI ecosystem for Python allows developers to specify dependencies (requirements specifiers) and constraints (version specifiers) in a setup.py or requirements.txt file. The Python dependency specification (Coghlan & Stufft, 2021) allows for a pinned dependency using the version matching clause (==). PyPI also uses the compatible release clause (∼=) and the ordered comparison clause (>= and <=) which allows for the restrictive constraint and permissive constraint smells. Python projects can also suffer from unused and missing dependencies since one can specify dependencies regardless of their usage in the code, or forget to specify all dependencies in the setup.py or requirements.txt file. PyPI allows direct URL links using the "<Dep Name> @ <URL>" format. PyPI also uses the pip-freeze command to output installed package versions which can be used for locking dependencies, similar to package-lock in *npm*.

It is worthwhile mentioning that while other ecosystems can differ in the type of smells they can experience, the severity of the smells might also be different. A good example is

the difference in the way the JavaScript and Python package managers handle dependency conflicts. While *npm* handles conflicts by installing multiple versions of a transitive dependency, which are each nested inside their respective direct dependencies (npm, 2021), PyPI will either outright reject installing incompatible Python packages or break another dependency by installing an incompatible transitive dependency (PyPA, 2021). This means that while using pinned dependencies can cause security issues in both ecosystems (due to not receiving security updates), in the case of Python, it can also break dependencies, while for *npm*, it will only increase the installation size.

3.8 Implications

We present actionable implications for developers, package maintainers, researchers, and educators.

Implications for Developers and Maintainers: The prevalence of dependency smells (RQ1) and their increase (Section 3.7) implies that current dependency management practices are inadequate. We found that 80% of the projects are infected with two or more distinct smells. Many of the reasons for introducing smells in RQ3 (Section 3.5), such as R2, R3, R4, R6, R7, R8, R9 and R10 can be fully or partially addressed if developers regard them as a priority, perhaps by opting to use SemVer rather than the alternative. For example, some developers are against a centralized repository altogether (R7) so they opt to use direct URLs, But these external links can bring about many security and stability issues (Section 3.2). They also cite a mistrust that stems from historical practices but not necessarily from the package at hand (R2). It would be beneficial for developers to spend more effort in dependency maintenance, but such maintenance requires proper guidance to be efficient. Developers can use our open source tool (Javan Jafari, 2020) to quickly scan their projects and identify dependency smells as "hot-spots" requiring prioritized maintenance. Additionally, developers could add a "dependency maintenance" task to the code

review checklist, every time a dependency is added or modified in a pull request.

The number of dependency smells in a package (given by a tool such as DependencySniffer) can also be used as a metric to evaluate and compare package quality, especially in cases where the functionality is similar. When faced with a choice between two similar packages, selecting one with more dependency smells may propagate maintenance issues to our own project in the future. It also affects the ecosystem as more projects along a dependency chain could be infected by dependency smells. This can be accomplished by displaying "clean" or "infected" badges in the project's page on *npm*. Badges are a great way to provide status information for the package and they can be displayed on GitHub or the *npm* registry. In fact, the David DM tool attempts a similar approach by analyzing project dependencies and providing "up to date" or "out of date" badges for the project (Shaw, 2020). Additionally, IDEs can be a good platform for warning developers about smells. For example, the JetBrains IDE currently issues warnings for missing dependencies, but not the other smells mentioned in this study (JetBrains, 2021).

Developers agree that dependency smells are harmful (RQ2) but some key reasons for reluctantly resorting to dependency smells are rooted in the dependencies themselves (RQ3). Some of the reasons for introducing smells in RQ3 (Section 3.5), such as R1, R2 and R5 need to be addressed by the maintainers. As such, package maintainers play a very important role in reducing the spread of dependency smells in an ecosystem. For example, developers mentioned how unexpected breaking changes (R1) forced them to pin a dependency for which they previously adhered to SemVer, or how a maintainer's apathy towards releasing requested fixes (R5) forced them away from the official package towards custom URLs. They also state how they sometimes resort to alternatives because they do not trust that SemVer is correctly respected by the maintainers (R2). Thus, proper SemVer compliance by maintainers can alleviate some of the key cited reasons for turning to dependency smells. The Greenkeeper tool attempts to proactively warn developers when a dependency

update breaks their code by running CI tests each time a dependency is updated (Lehnardt & Haas, 2020). Package maintainers can also benefit from this by running a select number of tests from their dependents' projects before releasing a new version of the package. This can be further encouraged at the ecosystem level by labeling packages that frequently violate SemVer in their releases.

Implications for Researchers and Educators: Dependency smells occur in a very considerable portion of the projects in our dataset (RQ1). However, there are yet other research aspects which should be explored. One possibility is mapping a smell-inducing commit to its fixing commit to determine the lifetime of dependency smells. Also, observing how smell type or project characteristics influence the lifetime of smells in a project will help in better understanding the root cause of dependency smells. Additionally, it is worthwhile to study which smell instances have a considerable impact on project maintenance based on revision history, issue tracking records and other project data to better understand the impact of dependency smells on software evolution.

Our catalog of dependency smells contains full descriptions and consequences for seven *npm* dependency smells (Section 3.2). This catalog is backed up by empirical evidence regarding their prevalence (RQ1), and practitioner validations regarding their negative impact (RQ2). It is thus a great reference guide for educators looking to teach best practices in dependency management. While analyzing survey responses from 41 practitioners (RQ2), no single developer was aware of every smell, nor were they aware of every possible consequence for the smells. Teaching this collective knowledge to future developers is a good way to battle the increase in dependency smell accumulation (Section 3.7).

3.9 Related Work

While we could not find any research works that specifically target dependency smells, the issues surrounding dependency management in software ecosystems is a well known

and actively explored topic. There are also some studies which look at other types of configuration smells.

Dependency management issues: Being too restrictive in updating dependencies will prevent developers from receiving the latest security and bug fixes, and as observed by Cox et al., systems with outdated dependencies are four times more likely to have security vulnerabilities (Cox, Bouwers, Van Eekelen, & Visser, 2015). Derr et al. also found that almost 98% of the actively used android libraries that they investigated have a security vulnerability that can be fixed by simply updating the package versions (Derr et al., 2017). The result of this study hints that developers are sometimes aware of dependency issues, but they choose to ignore it because addressing them can be a lot of work. However, a survey by Kula et al. showed that up to 69% of developers are unaware of the fact that they have a vulnerable dependency in their project (Kula, German, et al., 2018b). The work of Decan et al. analyzes security reports for *npm* packages and found that more than 40% of releases depending on a vulnerable package could easily be protected if they were less restrictive in their updates (Decan et al., 2018b). A study by Zimmermann et al. looks at the potential of both packages and package maintainers in *npm* to affect larger parts of the ecosystem. They look at over five million package versions and found that up to 40% of all packages depend on a code that has at least one publicly disclosed security vulnerability (Zimmermann et al., 2019). They also observed how particular features of the *npm*, such as pinned dependencies and heavy reuse (especially in micro-packages (Abdalkareem et al., 2017)) exacerbate dependency issues.

Opting for a permissive approach will facilitate updates, but increases the exposure to breaking changes. Bogart et al. study breaking changes in Eclipse, CRAN, and *npm* and find that compared to other ecosystems, *npm* developers are more willing to perform breaking changes with the assumption that users that use SemVer will be protected (Bogart et al., 2016). Bogart et al. also conduct a survey of 2,000 developers across eighteen ecosystems

where 70% of the respondents for *npm* declare that they have experienced breaking changes when attempting to build their package (Bogart et al., 2017). A study on the Maven ecosystem finds that more than 35% of minor and more than 23% of patch releases contain at least one breaking change (Raemaekers, Van Deursen, & Visser, 2014). Semantic versioning is one of the proposed solutions to the dependency management issues. A recent work by Decan and Mens (Decan & Mens, 2019b) looks at the four ecosystems of Cargo, *npm*, Packagist, and RubyGems. They found that there is a promising trend towards SemVer compliance, but in the cases where SemVer is not followed, developers prefer to be restrictive rather than permissive. An empirical study by Decan et al. looks into dependency issues in *npm*, CRAN, and RubyGems, focusing on the extent of dependency relationships and the issues surrounding dependency constraints(Decan et al., 2017). They find that developers prefer to specify a maximum threshold on their constraints, rather than a minimum one. They also mention the co-installability issues with strict (aka pinned) dependencies. Another recent work looks at how dependencies adopt SemVer and how they change their approach over time. Dietrich et al. were unable to find a large-scale adoption of SemVer and there is evidence of both flexible and restrictive approaches to update(Dietrich et al., 2019). These studies are all solely based on *npm* packages and do not consider applications, which can have different approaches in managing dependencies. They also do not mention issues beyond permissive or restrictive constraints, or quantify how these issues evolve over time.

To the best of our knowledge, our study is the first work that specifically focuses on cataloging, quantifying, and understanding dependency smells.

Configuration smells: Infrastructure as Code (IaC) is another domain where system configuration is specified through code. While configuration code is different than source code, it is subject to similar issues of maintainability and complexity. Sharma et al. (Sharma, Fragkoulis, & Spinellis, 2016) analyzed more than 4,600 Puppet repositories and proposed

a catalog of configuration smells that violate best practices. Configuration scripts can also contain smells related to security vulnerabilities. These security smells are reoccurring patterns of configuration code snippets which can lead to security breaches. Rahman et al. conduct an empirical study on more than 1,700 IaC scripts (Rahman, Parnin, & Williams, 2019) to identify seven security smells. They then developed a detection tool and collected instances of those smells from 293 open-source repositories and submitted bug reports for a random subset of the instances. They found that security smells have a median lifetime of 20 months while some can persist for as long as 98 months. Since the package.json has a relatively well-defined structure and functionality, it is possible to automatically parse it to detect dependency smells. Semantic versioning guidelines can also be an effective solution in Infrastructure as Code. Opdebeeck et al. study 70,000 version increments in the Ansible infrastructure and discover that Ansible role developers generally abide by semantic versioning guidelines (Opdebeeck, Zerouali, Velázquez-Rodríguez, & De Roover, 2020).

3.10 Threats to Validity

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

Threats to construct validity: Threats to construct validity refers to the concern between the theory and the results of the study. The smells catalog in this study is neither exhaustive nor complete. There might be other dependency smells in the *npm* ecosystem that our study did not consider. In addition, many of our dependency smells are observed through the package.json file. However, it is possible for developers to manually install dependencies and not include them in package.json. While we have studied such cases under "Unused dependencies", they could also be subject to other smells such as "Pinned dependency", since a manually installed package may not be automatically updated. However, since package.json is the official dependency configuration file for *npm*, ad-hoc alternatives are rare and discouraged. Also, while investigating the reasons for the introduction of smells

(RQ3), we emailed the developers in charge of the smell-inducing commit. In reality, there may be other sources, such as other developers involved in group discussions, that may have also influenced the change. We believe the developer in charge of the commit would be the most-informed member of the team on this change and in fact, all respondents were able to elaborate on the commit.

Threats to internal validity: Threats to internal validity refers to the concerns that are internal to the study such as experimenter bias and errors. In our work, we used the depcheck (J. Li & Lukic, 2019) tool to extract the missing and unused dependency smells in the studied projects. Hence, we are limited by the accuracy of this tool. To mitigate the threats related to using the depcheck tool, we randomly selected five projects from our dataset and fed them to the depcheck tool. We then manually cross-checked the output of this tool. We found that in all cases, the tool produced the correct results for missing dependencies, as we were able to navigate to the file that uses a dependency not specified in the package.json. Validating unused dependencies can be much harder since there are different ways to use a dependency inside a code. While the results proved accurate in our manual checks, this is one area where depcheck users have experienced false positives.

Similar to the process used for empirically studying the smells, our proposed tool uses regular expressions to detect the first four dependency smells. To validate the accuracy of the regular expressions and the parser as a whole, we manually examined a sample of 10% of the cases and we observed no false positives or false negatives. However, our manual validation is still limited to 10% of the dataset. Additionally, we used the tool to identify smells and emailed developers for clarification (RQ3), and all respondents verified the existence of the smells. These results make us confident in our parsing process and our regular expressions.

Our initial survey gathered responses from 12 developers. These developers were a convenience sample which were known by the authors in Canada and Brazil which is not

representative of all JavaScript developers. However, this survey was conducted to augment the initial understanding of the smells. The initial definitions are observed by studying violations of *npm* recommendations, violations of SemVer guidelines and by studying the discussions surrounding the package.json file. In addition, to extract the reasons for introducing a dependency smell, we manually analyzed the free-text responses provided by the developers. Since this human activity and can be subject to human bias, we perform a thematic analysis with two independent coders where we have the first two authors independently analyze and categorize them.

Threats to external validity: Threats to external validity concerns the generalization of our findings. Our study is based solely on JavaScript projects, therefore our findings may not hold for projects written in other programming languages. However, our research methodology of defining and studying dependency smells can be easily replicated for other programming languages such as Python. Secondly, the datasets that we used in our work only represent open source projects hosted on GitHub that may not reflect proprietary projects. Furthermore, we examine the official dependency manager for JavaScript, *npm*. Hence, our results may not be fully generalized to other dependency managers such as yarn. However, yarn uses the same package.json file to evaluate dependencies and uses the same *npm* registry to download them (Yarn, 2020). The only difference related to our work is cases where projects have a yarn-lock file instead of a package-lock. That is why we considered projects with yarn.lock to be free of the no package-lock smell. To understand why dependency smells are introduced in projects, we surveyed 28 developers. Although we believe this to be a sufficient number of developers for our analysis, our results may not represent the opinion of all JavaScript developers. Also, asking a different sample of developers may result in a different set of reasons for introducing dependency smells. To mitigate the threat, we contacted practitioners from different studied projects and their backgrounds show that they are experienced JavaScript developers.

Threats to conclusion validity: Threats to conclusion validity concern the relation between the experiment and the conclusions. The empirical study of dependency smells is based on historical data in 1146 JavaScript projects, but investigating their impact and the reasons for their introduction is conducted using surveys. Therefore, the conclusions drawn are based on the type and number of respondents. The response rate for the emails we sent out to developers to understand the reasons for introducing the smells was 28%. While this is significantly larger than the typical rate in questionnaire-based software engineering surveys (Singer et al., 2008), having a larger portion of respondents may reveal new reasons or change the priority of current reasons. Additionally, we argue that the prevalence of dependency smells in RQ1 implies a lack of attention to dependency maintenance. The dependency smells in this study focus on how dependencies are used, but do not consider other aspects of dependency maintenance, such as what dependencies are selected. Therefore, dependency smells alone are not enough to compare dependency maintenance across projects. Hence, this study does not claim to measure dependency maintenance.

3.11 Chapter Conclusion and Future Work

Our objective was to catalog, quantify, and understand dependency smells in the *npm* ecosystem. We conducted an empirical study on a dataset of 1146 active JavaScript projects to identify which smells are more common and how they accumulate over time. We also consulted practitioners to quantify the consequences associated with each smell and understand why they are introduced.

We define seven dependency smells in *npm*. Our findings reveal that these smells are prevalent. While not all smells occur at a large degree, 80% of the projects are infected with two or more distinct smells. In our practitioner surveys, we found that practitioners recognized the multitude of security problems, bugs, dependency breakages, and other maintenance issues brought about by dependency smells. These smells are generally introduced

ass developers react to dependency misbehavior and the shortcomings of the *npm* ecosystem. We also observed that dependency smells are addressed/fixed, but most dependency smells are introduced more frequently than old smells are fixed, causing an accumulation of smells over time. Since we had to analyze a large number of projects, commits, and smells, we built a tool (DependencySniffer) for our analyses. Our prototype tool can be used to analyze any JavaScript project and detect dependency smells. The tool is open source and accessible to everyone (Javan Jafari, 2020).

As we observed in the results of this study, it is not rare to see developers introduce smells into some of their dependencies, while others remain clean. The communication with the developers further highlights that dependency smells in downstream packages are often introduced as a response to the shortcomings and misbehaviors of upstream packages. When it comes to managing dependencies, developers do not perceive (or trust) all packages equally. In the following chapter, we investigate how the characteristics of upstream packages influence the dependency management decisions of their downstream dependents and how developers can leverage such characteristics to better manage their dependencies.

Chapter 4

Practices for Updating Dependencies

Managing project dependencies is a key maintenance issue in software development. Developers need to choose an update strategy that allows them to receive important updates and fixes while protecting them from breaking changes. Semantic Versioning was proposed to address this dilemma but many have opted for more restrictive or permissive alternatives. This empirical study explores the association between package characteristics and the dependency update strategy selected by its dependents to understand how developers select and change their update strategies. We study over 112,000 npm packages and use 19 characteristics to build a prediction model that identifies the common dependency update strategy for each package. Our model achieves a minimum improvement of 72% over the baselines and is much better aligned with community decisions than the npm default strategy. We investigate how different package characteristics can influence the predicted update strategy and find that dependent count, age and release status to be the highest influencing features. We complement the work with qualitative analyses of 160 packages to investigate the evolution of update strategies. While the common update strategy remains consistent for many packages, certain events such as the release of the 1.0.0 version or breaking changes influence the selected update strategy over time.

4.1 Introduction

Software development is increasingly reliant on code reuse, which can be accomplished through the use of software packages. Utilizing packages to build software improves quality and productivity (Lim, 1994; Mohagheghi et al., 2004). These packages, along with the dependencies and maintainers have formed large software ecosystems (Sonatype, 2021). In the current landscape, managing dependencies among packages is an emerging challenge (Artho et al., 2012; Bogart et al., 2016; Decan et al., 2019). The popular Node Package Manager (npm) ecosystem has experienced several dependency-related incidents. One example is the removal of the backward-incompatible release of the "underscore" package that generated a lot of complaints among dependents that updated to the latest version (Chatfield, 2014). Another example is the removal of the "left-pad" package which, at the time, majorly impacted many web services (MacDonald, 2018). The ua-parser-js package is more a recent example of an npm package that had its maintainer account hijacked to release malicious versions of the library (Github, 2021) that would steal user information such as cookies and browser passwords. The package frequently experiences 6-7 million weekly downloads and was used by many large companies such as Facebook, Apple, Amazon, Microsoft, IBM, Oracle, Mozilla, Reddit and Slack (Cimpanu, 2021).

Knowing when and how to update dependencies are among the most important challenges faced by development teams (Tidelift, 2022). The npm package manager allows for various constraints for configuring when and how each dependency will automatically update (npm Documentation, 2019a). In order to study the dynamics of dependency updates, we draw inspiration from previous literature and group the various dependency constraints into 3 update strategies: the balanced update strategy, the restrictive update strategy and the permissive update strategy (Decan & Mens, 2019a). The specifics of each update strategy is further explained in Section 4.2. Different update strategies bring about different

consequences (Jafari et al., 2021). Opting for overly restrictive update strategies (e.g. preventing any automatic updates) will prevent timely security fixes for packages (Cox et al., 2015; Decan et al., 2018b; Prana et al., 2021). On the other hand, overly permissive update strategies (e.g. allowing any type of automatic updates) will increase the likelihood of breaking changes due to incompatible releases (Decan et al., 2018a; Jafari et al., 2021; Kula, German, et al., 2018a). Thus, a key issue in dependency management is choosing the right strategy for updating dependencies.

Semantic Versioning (SemVer) has been proposed as a solution to aid dependency management by allowing maintainers to communicate the type of changes included in their new package releases and allowing developers to determine backward-compatibility based on the semantic version number of the newly released version. This provides developers with a middle-ground between keeping dependencies up to date while ensuring a backward-compatible API (Preston-Werner, 2019). However, previous research has shown that SemVer is not always relied on in practice and it is not rare to see developers opting for alternative dependency update strategies (Chinthanet et al., 2019; Cogo et al., 2019; Dietrich et al., 2019; Kula, German, et al., 2018a; Wittern et al., 2016).

Developers may adopt or modify a dependency update strategy based on their perception of a package dependency. This is visible in the dependency configuration of npm packages (package.json) where different maintainers will opt for different strategies for managing their dependencies but more importantly, a maintainer will even opt for different strategies for different dependencies in the same project (Jafari et al., 2021). Certain events (e.g. breaking changes) may also shift a developer's perception in regards to the previously selected update strategy (Cogo et al., 2019). Different dependency update strategies may be selected based on the characteristics of the target packages. Additionally, the characteristics of a package dependency may serve as indicators of the community trust on the package

(e.g. age may signal maturity). Understanding how these characteristics relate to dependency decisions among the majority of developers can serve as a guide for how one should depend on each package, as well as a means to understand what package characteristics are associated with dependency update strategies.

In this study, we investigate the relationship between npm package characteristics and the dependency update strategy opted by its dependents. We focus on npm since it currently maintains the largest number of packages in any software ecosystem (Libraries.io, 2020) and consequently, a high number of dependency relationships between packages. Our dataset includes 112,452 npm packages and 19 characteristics derived from npm and the package repository. We use a machine learning module to investigate whether package characteristics can be used to predict the most popular dependency update strategy for each package. Specifically, we aim to tackle the following research questions:

RQ1: Can package characteristics be used as indicators of dependency update strategies?

We train several machine learning models using features collected and derived from package characteristics. Our experiments reveal Random Forest as the most suitable model for our purpose. As such, we select Random Forest as the model in this study. We evaluate our model and compare it against two baselines (stratified random prediction and npm-recommended balanced strategy). The results show a 72% improvement in the ROC-AUC score and 90% improvement in the F1-score compared to the stratified baseline. We observe that package characteristics can be used as indicators of the common update strategy and they can be leveraged for predicting dependency update strategies. Additionally, we found that our model results align considerably better with community decisions than always using the balanced update strategy.

RQ2: Which package characteristics are the most important indicators for dependency update strategies?

In order to help developers understand the key factors that impact dependency update strategies, we identify the most important features for the prediction model and analyze how a change in these features impacts the model's predictions. The *release status* of a package, the number of *dependents* and its *age* (in months) are the most important indicators for the common dependency update strategy. Dependents of younger, post-1.0.0 packages with more dependents are more likely to agree on the balanced update strategy. On the other hand, dependents of pre-1.0.0 packages are more likely to opt for more permissive update strategies.

RQ3: How do dependency update strategies evolve with package characteristics?

In an effort to understand the prominence of evolutionary features in predicting the common update strategy, we use a mixed-method technique on a convenience sample of 160 packages to analyze the evolution of update strategies over a period of 10 years. We found that for many packages in npm, the common update strategy remains consistent throughout a package's lifecycle, but the release of the 1.0.0 version causes a visible shift in the common update strategy. Restrictive update strategies proved to experience the weakest agreement (repeatedly challenged by other strategies), with more erratic evolutionary behavior that correlate with incidents such as breaking changes.

The rest of the chapter is organized as follows. Section 4.2 describes specialized packages and our data selection and feature extraction methodology. We present our results in Section 4.3 and highlight the study implications in Section 4.4. We review related work in Section 4.5 and discuss the threats to validity in Section 4.6. We conclude our work in Section 4.7.

4.2 Data and Methodology

We use the latest version of the libraries.io dataset available at the time of collection, containing package dependencies from January 2020[1] (Libraries.io, 2020) to collect all packages in the npm ecosystem. We filter and label the packages, extract characteristics and derive new features, and use them to train a Random Forest model.

A replication package of our study is available on Zenodo (Javan Jafari, Elias Costa, Shihab, & Abdalkareem, 2022).

4.2.1 Specialized packages

In order to identify the "common" dependency update strategy for a particular package, we rely on the "wisdom of the crowds" principle (Decan & Mens, 2019a). This means that a dependency update strategy is deemed the common strategy if the majority of its dependents are using the same strategy. A package is deemed specialized toward an update strategy if the majority of its dependents agree on that particular update strategy. In this study, we calculate the proportion of each of the 3 dependency update strategies and use 50% as the threshold to define specialized packages. If more than 50% of the dependents are not using a common update strategy, a package is deemed unspecialized and we can not use package characteristics to analyze dependency update strategies for that package. Section 3.1 explains the rationale for the selected threshold. By drawing inspiration from the work of Decan and Mens (Decan & Mens, 2019a), a package is considered specialized if more than 50% of its dependents agree on one of the following update strategies:

- **Balanced:** The update strategy is considered balanced if it allows for new updates but keeps us safe from breaking changes (with the assumption that SemVer is correctly followed by the target package). In specific terms, a post-1.0.0 constraint that allows

[1] At the time of this study, no other dataset has been published since 2020.

automatic updates to new minor and patch versions is considered balanced. This can be accomplished by using the caret notation in npm (e.g. "^1.2.3") but can also be expressed in other ways such as "1.x.x". A pre-1.0.0 constraint is considered balanced if it does not allow any updates (pinned). This is due to the fact that SemVer considers these versions to have an unstable API (Preston-Werner, 2019).

- **Restrictive:** The update strategy is considered restrictive if it is more restrictive than the balanced update strategy. In specific terms, a post-1.0.0 constraint that only allows automatic updates to new patch releases or no automatic updates at all is considered restrictive. This can be accomplished through the use of the tilde notation in npm (e.g. "∼1.2.3") but can also be expressed in other ways such as "1.2.x" or "1.2.3". Pre-1.0.0 constraints can not be restrictive since pre-1.0.0 releases have an unstable API and any freedom in updates is considered permissive.

- **Permissive:** The update strategy is considered permissive if it is more permissive than balanced update strategy. In specific terms, a post-1.0.0 constraint that allows automatic updates to all new versions (including major versions) is considered permissive. This can be accomplished through the use of wildcards (e.g. "*") but can also be expressed in other ways such as "latest" or ">=1.2.3". A pre-1.0.0 constraint that allows any automatic updates is considered permissive.

4.2.2 Data filtering and labeling

For this study, we only consider packages with two or more runtime dependents. We want to investigate the most common dependency update strategy for each package. Therefore, we should only consider packages that have downstream dependents. Additionally, looking for a majority agreement between dependents of a package is not a sound approach if the package has fewer than 2 dependents. The npm package manager allows developers

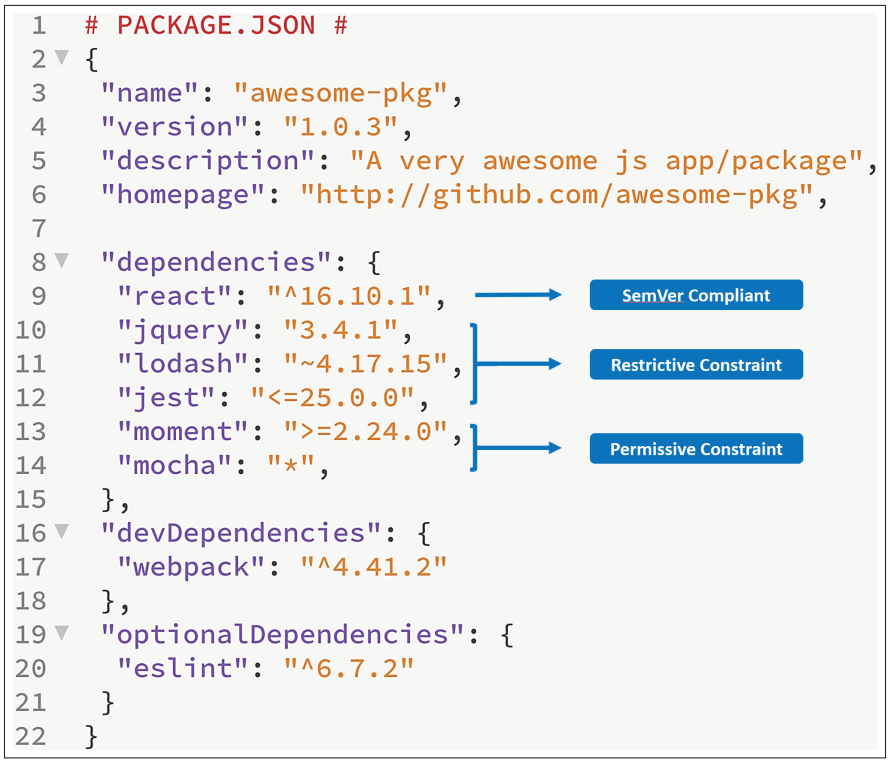


Figure 4.1: Example of a package.json file showing dependency update strategies

to specify development dependencies (will be used in development environment) and optional dependencies (npm will try to fetch them but will not raise errors if unsuccessful). We do not consider development and optional dependencies because they are not required for the dependent package to function and are sometimes incomplete. These thresholds help in removing unused and noisy packages from the dataset. However, we were still able to identify multiple spam packages which had the sole purpose of depending on all packages in npm. The ones we identified were all-packages-X, wowdude-X and neat-X, in all of which the X is replaced by various numbers.

In order to identify package specialization, we extracted the runtime dependency relationships from the latest published versions of all packages to other packages in our dataset (January 2020). We used the reverse relationship (from the target package to the source package) to determine the dependents of each package and their dependency constraints.

If more than 50% of a package's dependents agree on a dependency update strategy (Section 4.2), the package is labeled as specialized towards that strategy (i.e. balanced, restrictive, permissive). Otherwise, the package is labeled as unspecialized.

This groups all packages in the dataset into 4 categories (balanced, restrictive, permissive, unspecialized). We do not choose a threshold below 50% since a threshold of over 50% for one class is guaranteed to always represent the most accepted update strategy for that package. Increasing the threshold (higher majority agreement) bolsters the confidence in the "most common update strategy" when there is an agreement, but as the agreements become rare, the results become less meaningful in practice. As can be seen in Figure 4.2, our selected threshold also results in the lowest comparative percentage of "unspecialized" packages. Unspecialized packages are not helpful in studying the common update strategy, since by definition, they do not have a common agreed upon update strategy among their dependents.

The final dataset includes 112,452 total npm packages. From this total, 101,381 (90.2%) are specialized toward a particular update strategy and 11,071 (9.8%) are unspecialized. Looking at different update strategies we see that 54.2% of packages are specialized toward the balanced strategy, 6.7% are specialized toward the restrictive and 29.3% are specialized toward the permissive update strategy. The packages in our dataset have a median of 3 dependents and a median age of 39 months. The distribution of our dataset is shown in the first row (50% threshold) of Figure 4.2 and the distributions of agreement percentage (among dependents) for each class are presented in Figure 4.3.

4.2.3 Feature selection and extraction

In this section, we explain the rationale for selecting the package features. We then explain our feature extraction procedure and the necessary pre-processing of the features.

Feature selection rationale: In order to train a suitable model in predicting dependency

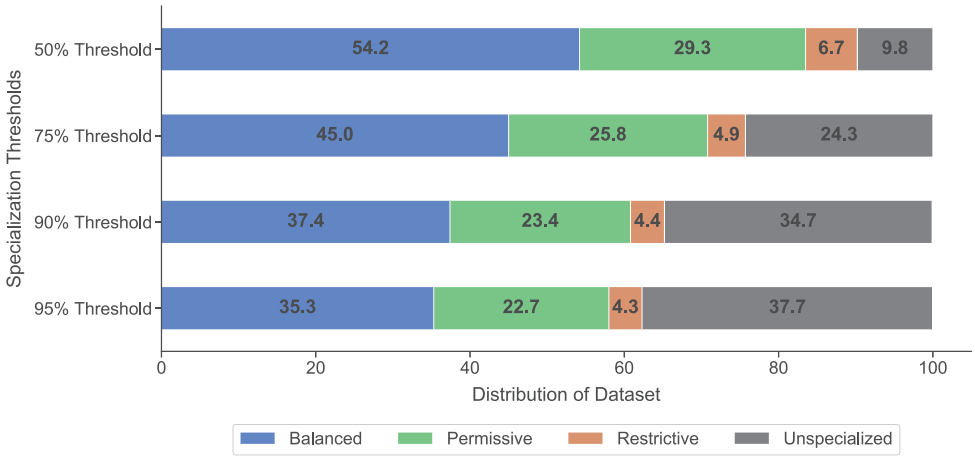


Figure 4.2: Impact of specialization threshold on class distribution

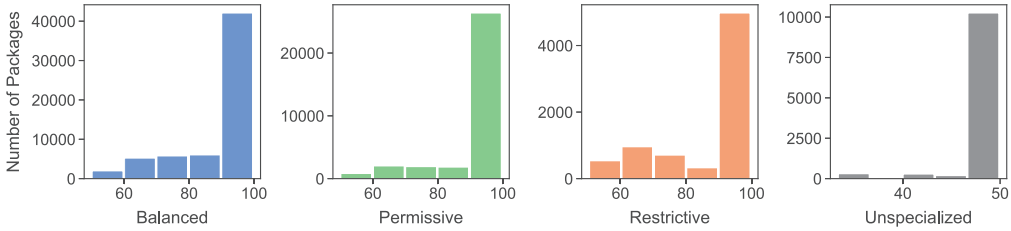


Figure 4.3: Distribution of dependent agreement percentage for packages in each class

update strategies, we first need to select appropriate features that can capture developer needs in choosing the correct strategy. The libraries.io dataset consists of over 50 characteristics for each package, although some are highly correlated. We use the term package features to refer to characteristics from both the package on npm and its project repository. In order to determine what features in our dataset are relevant and what other features might be needed, we studied the literature to identify which package characteristics are associated with the characteristics involved in choosing and managing dependencies.

Table 4.1 presents each of these features. All of the studies referenced in the table are comprised of developer surveys and interviews regarding practitioner needs and practices (see Section 4.5). The features listed here are deemed relevant in the literature in choosing and managing dependencies, but ours is the first study to investigate their influence on the dependency update strategy. According to the reviewed literature, developers use the

Table 4.1: Relevant features in selecting dependencies

Feature	Studies
Repository Stars Count	(Haenni, Lungu, Schwarz, & Nierstrasz, 2013; Larios Vargas et al., 2020; Pashchenko et al., 2020)
Repository Watchers Count	(Haenni et al., 2013; Larios Vargas et al., 2020)
Repository Forks Count	(Haenni et al., 2013; Larios Vargas et al., 2020)
Dependency Count	(Larios Vargas et al., 2020)
Dependent (Repository and Package) Count	(Bogart et al., 2016; Haenni et al., 2013; Larios Vargas et al., 2020; Pashchenko et al., 2020)
Repository Contributors Count	(Pashchenko et al., 2020)
Repository Open Issues Count	(Pashchenko et al., 2020)
Licenses	(Haenni et al., 2013; Pashchenko et al., 2020)
Days Since Last Release	(Bogart et al., 2016; Larios Vargas et al., 2020)
Age	(Larios Vargas et al., 2020)
Version Count, Version Frequency	(Bogart et al., 2016; Haenni et al., 2013; Larios Vargas et al., 2020)
Repository Readme, Description, Wiki, Pages	(Bogart et al., 2016; Haenni et al., 2013; Larios Vargas et al., 2020)
Repository Size	(Bogart et al., 2016; Larios Vargas et al., 2020)
Release Status	(Bogart et al., 2016)

following characteristic groups to select dependencies:

- **Package maturity and popularity** is a recurrent factor in the literature. Prominent projects that are established in the community are a priority in selecting dependencies (Bogart et al., 2016; Haenni et al., 2013; Larios Vargas et al., 2020; Pashchenko et al., 2020). Characteristics such as Age, Dependent Count, Repository Stars and Forks Count along with Repository size and Contributors count can be used as indicators for established package among the community. We hypothesize that packages with a more established history (whether positive or negative) provide more information for developers to decide on their preferred dependency update strategy. Popular packages are also encouraged to be more diligent in their updates as they are scrutinized by a larger user-base. Additionally, packages in initial stages of development are often deemed unstable by dependency guidelines such as SemVer, and thus warrant stricter update strategies.

- **Package activity and maintenance** is cited as one of the most important factors in selecting dependencies (Bogart et al., 2016; Larios Vargas et al., 2020; Pashchenko et al., 2020). Characteristics such as Version frequency, Open issues count and Days

since last release can be used as indictors for package activity. We hypothesize that highly active packages would be more problematic for dependents that opt for permissive dependency approaches as the likelihood of breaking changes may increase with more frequent releases. On the other hand, different dependency update strategies can be inconsequential for packages that have not released a new version for a long time as there is little meaningful difference between the latest version and an old version.

- **Documentation** is also among the highly stated factors for selecting dependencies (Bogart et al., 2016; Haenni et al., 2013; Larios Vargas et al., 2020; Pashchenko et al., 2020). License information is also important to prevent legal issues. Project readme and wiki files, along with license information can be used as suitable indicators for this category. We use the license code as a feature that represents the type of licenses for the package (e.g. MIT, BSD-2-Clause, ISC). We hypothesize that the resulting perception from better documentation can not only encourage developers to select a package, but also influence the perception of trust on the package. This in turn can sway them to opt for less restrictive update strategies. Adequate documentation may also bring comfort in knowing that the dependent's development team can rectify shortcomings in particular dependency versions.

Feature extraction: Some of the selected features are directly available in the libraries.io dataset and others are derived using the raw features in the dataset. In the following, we will explain the derived features:

- **Age** is derived using the package's "created timestamp" and comparing it against the date the dataset was released (Jan 2020).

- **Version Frequency** is derived by counting the number of releases and dividing it by the package age in months. In cases where the age was zero months, we used version

count instead of version frequency.

- **Dependent Count** for each package is the sum of reverse dependencies (dependents of a package) from the latest version of all packages in the dataset to that package. The dependent count available in libraries.io also includes dependents from old versions of all packages.

- **Transitive Dependent Count** is the total number of packages in the dependent tree of our package. It is calculated by converting the dependency relationships for each package into a graph and calculating the total ancestors from the selected package.

- **Dependency Count** is calculated by counting the number of dependencies for the package.

- **Transitive Dependency Count** is the total number of packages in the dependency tree of our package. It is calculated by converting the dependency relationships for each package into a graph and calculating the total descendants from the selected package.

- **Release Status** is extracted using the latest version of the package and determines if the package is in initial development (pre-1.0.0) or production stage (post-1.0.0).

- **Days Since Last Release** is derived by extracting the latest release and comparing its date against the date the dataset was released (Jan 2020).

We hypothesize that the **Domain** or type of the package may influence how developers depend on a package since certain dependencies may correspond to more critical aspects of a software project. This is further investigated in the manual analysis of Section 4.3. Seeing that we have access to package keywords, we can use them to assign domain/type to each package. Since there are many varied keywords in the dataset, we first need to prune the keyword set and map each package to a smaller set of keywords. To this aim,

we first address highly correlated keywords by finding the top 2000 trigrams and bigrams (n-grams are collections of n keywords that frequently appear together) with the highest Point-wise Mutual Information (PMI) scores. PMI is a metric provided by NLTK (NLTK, 2022) to quantify the likelihood of co-occurrence for two words, taking into account that this might be caused by the frequency of single words. We only consider trigrams and bigrams that appear at least 10 times in the dataset. In short, we group keywords into sets if they commonly co-appear. We then use one keyword to represent each set. This procedure reduces the average number of keywords per package. In the next step, we use the keywords to cluster the packages. To this aim, we use the top 15 keywords to build a term frequency vectorizer for package keywords. The vectorized keywords are fed into a K-means clustering algorithm with K=10 (derived using the elbow and silhouette methods (Géron, 2019)). The result is a numerical "Domain" feature which includes a value from 1 to 10 for each package.

Feature pre-processing: Many values in the dataset did not have a default of zero and instead, included missing values. Missing values were handled in such a way that would be meaningful for each feature. For example, if there were missing values for the number of dependencies or repository stars count, a value of zero was used as a replacement. However, this strategy would not be meaningful for all features. For example, missing values in repository size were replaced by the median repository size. Since we study packages with a dependent count greater or equal to 2, missing values in dependent count were automatically removed.

Highly correlated features negatively impact the model's performance and more importantly, its interpretability. We calculate the Pearson correlation and remove features with a correlation above 0.7.

When two features were highly correlated, we kept the feature with the more tangible description. For example "Repository Contributors Count" was removed as it was highly

correlated with ”Repository Size” and ”Repository Watchers Count” was removed due to its high correlation with ”Repository Stars Count”. In total, the following 12 features were removed due to correlation: Repository Host Type, Repository Wiki enabled?, Repository Pages enabled?, Repository Open Issues Count, Repository Issues enabled?, Repository Watchers Count, Repository Forks Count, Repository SourceRank, Versions Count, Repository Contributors Count, Repository URL, Transitive Dependent Count.

Table 5.2 presents the final set of features selected for this study along with a description for each feature. After dropping the aforementioned correlated features, the remaining feature set in Table 4.1 appears in our final set of features. We have also used the characteristic groups observed in the literature (maturity and popularity, activity and maintenance, and documentation) to utilize relevant features available in the dataset or synthesize relevant features. Transitive dependency count is an extension of dependency count which considers whether the dependencies of a package are ”dependency heavy” themselves. The existence of keywords and homepage URL is another means of evaluating package documentation. The domain is an attempt to identify package type by clustering the keywords (since the entire set of keywords are too numerous to use outright). The domain and keywords features have different objectives. Domain attempts to encapsulate package type while the existence of keywords is an indicator of package documentation. License code is also different from repository license in a similar manner. The former is a means of encapsulating package license type and permissions (to understand whether it affects how dependents use the package) while the latter is an indicator of documentation completeness. We also added SourceRank as a feature as it is the scoring algorithm used by Libraries.io to index the results (Libraries.io, 2020). SourceRank aggregates a number of metrics believed to represent high quality packages, some of which are also included in our features. For example: Is the package new? How many contributors does it have? and Does it follow SemVer?

Table 4.2: Selected features and their description

Feature	Description	Histogram
Dependency Count	The # of dependencies from the latest releases of npm packages.	
Transitive Dep. Count	The # of transitive dependencies from the latest package release.	
Dependent Count	The # of dependents from the latest releases of npm packages.	
Version Frequency	The # of released versions divided by the age.	
Age	The age of the project in months.	
Description	Whether or not the package provides a description.	
Keywords	Whether or not the package specifies keywords.	
Homepage URL	Whether or not the package specifies a homepage URL.	
License Code	The ID for the type of license(s) specified for the package.	
SourceRank	The SourceRank metric of a package provided by libraries.io.	
Release Status	Whether or not the package is at a pre-1.0.0 or post-1.0.0 state.	
Days Since Last Release	The # of months elapsed since the most recent release.	
Dependent Repositories	The # of dependent repositories on the package's repository.	
Repository Size	The size of the package repository in Kilobytes.	
Repository Open Issues	The # of open issues in the package repository.	
Repository Stars	The # of stars for the repository.	
Repository License	Whether or not the package repository specifies a license.	
Repository Readme	Whether or not the package repository provides a readme file.	
Domain	Package domain group extracted from the keywords.	

4.3 Results

We present the findings of our empirical study starting by our results for using package characteristics to predict the dependency update strategy. This is followed by a study on the impact of package characteristics on the popular dependency update strategy. In the last section of our results, we conduct a mix-method analysis with 160 packages to understand the contributing factors in the evolution of update strategies over a span of 10 years.

4.3.1 RQ1: Can package characteristics be used as indicators of dependency update strategies?

Motivation: Understanding the association between package characteristics and the commonly chosen dependency update strategy by its dependents can help the community to better grasp the dynamics of dependency update strategies. Knowing whether or not the

characteristics of a package are indicators of dependency update strategies will also help developers by providing them with meaningful and actionable information in the process of deciding the appropriate update strategy for their package dependencies. This can help prevent dependency issues that result from using unsuitable alternative strategies (Jafari et al., 2021).

Approach: In order to study the relevance of package characteristics to the commonly used dependency update strategy by the community, we use the features in Table 5.2 to train a Random Forest model. The multi-class model aims to use the characteristics to predict the commonly used update strategy for each package. The result of the prediction for each package can be one the four classes of Balanced, Restrictive, Permissive or Unspecialized. The unspecialized class does not represent an update strategy but rather, packages which do not have a common agreed-upon update strategy among their community of users. We use Random Forests since the objective of our study is to understand the association between package characteristics and dependency update strategies which necessitates descriptive models. In addition, we want good performance compared to the baseline in order to derive meaningful associations. We conducted preliminary experiments with Random Forest, Logistic Regression and SVM and compared their performance using ROC-AUC and F1-score metrics. The ROC (Receiver Operating Characteristics) is a probability curve where AUC (Area Under the Curve) is a value between 0 and 1 that represents the degree of which the model is capable of distinguishing between classes. The higher the AUC, the better the model is at correctly predicting classes. Since our problem is a multi-class model, we plot multiple ROC-AUC curves, one for each of the classes using the One-vs-Rest (OvR) methodology. The final ROC-AUC is the resulting average of the ROC-AUC scores. F1-score is a function between 0 and 1 that balances between precision (the fraction of true positive instances among the retrieved instances) and recall (the fraction of true positive instances that were retrieved). We did not modify the hyper-parameters of the three models

but we performed data normalization which is important for Logistic Regression and SVM when there is high cardinal variance between the features. All three models were trained on 80% of our dataset (training set) and evaluated on the held-out 20% (tests set). As can be seen in Figure 4.4, the Random Forest model yields considerably better performance, which is why it is selected as the Package Characteristics model in this study.

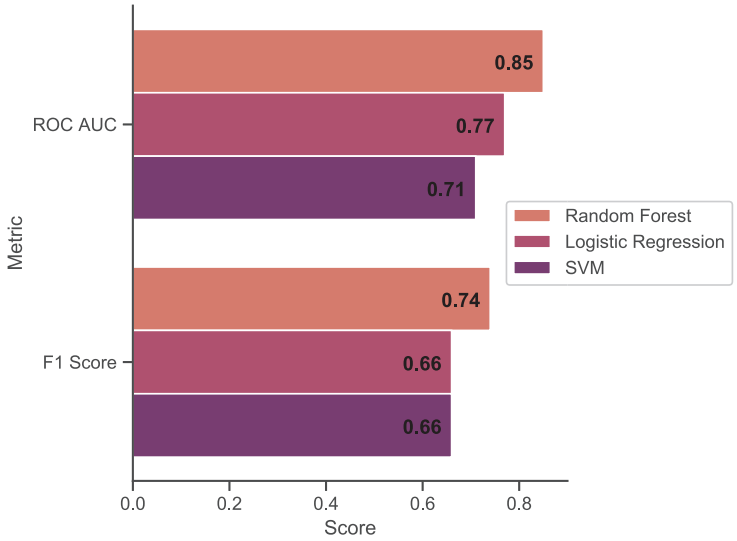


Figure 4.4: Comparison of performance for candidate models

Since there is no previous work on using package characteristics to predict dependency update strategies, the results are compared against two baselines; the stratified baseline model and the balanced model. The stratified baseline uses the class distribution in the training set for weighted random predictions about the suitable update strategy. The balanced baseline always predicts the balanced update strategy, as is recommended by npm (npm, 2022a). We evaluate the performance of the model using ROC-AUC and F1-score metrics (as explained previously in our preliminary experiments). We use 80% of the data as our training set and leave the remaining 20% for the final evaluation. We tuned the hyper-parameters of the Random Forest model using 10-fold validation on the training set which results in 500 estimators (trees) with 8 minimum samples required for a split. The 10-fold cross validation fits the model 10 times, with each fit being performed on a 90%

of the training data selected at random, with the remaining 10% used as a validation set. It is important to evaluate the model on the 20% of the data used as a held-out set since we want to assess the model's performance on unseen data.

Results: Figure 4.5 presents the evaluation results using the ROC-AUC, F1-score, Precision and Recall metrics. Compared to the baseline model, we can see a 72% improvement in the ROC-AUC for the Random Forest model, achieving an ROC-AUC of 0.86. The ROC-AUC for the Stratified baseline and the balanced-only approach round-up to 0.5, which is the expected behavior of ROC-AUC when the model makes random predictions or always predicts the same class. We also see a 90% improvement in the F1-score for the Random Forest model compared to the stratified baseline model, achieving a score of 0.74. Since the real world contains unspecialized cases where no agreement is observed, we have also included these unspecialized packages in the training and evaluation of our model.

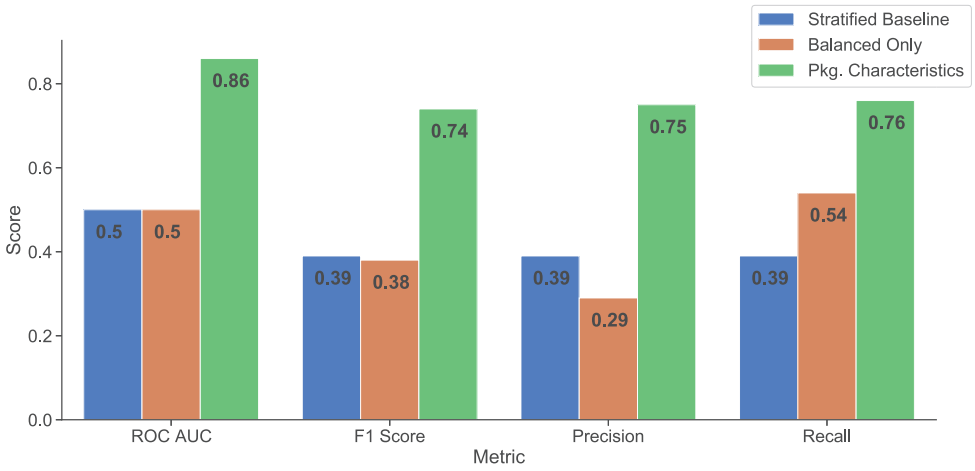


Figure 4.5: Performance evaluation results

The high ROC-AUC score of 0.86 shows that the package characteristics in Table 5.2 are not only relevant for selecting dependencies, but they can also be leveraged to predict the dependency update strategy opted by the majority of developers. In other words, they can be used as indicators of dependency update strategies. Another interesting observation are the results for the balanced baseline. While the balanced strategy is the recommended

Table 4.3: Comparing model performance across different specialization thresholds

Threshold	Model	ROC AUC	Min. Increase	F-1 Score	Min. Increase
50%	Stratified Baseline	0.50	-	0.39	-
	Balanced Only	0.50	-	0.38	-
	Package Characteristics	0.86	72%	0.74	90%
75%	Stratified Baseline	0.50	-	0.33	-
	Balanced Only	0.50	-	0.28	-
	Package Characteristics	0.85	70%	0.67	103%
90%	Stratified Baseline	0.50	-	0.32	-
	Balanced Only	0.50	-	0.20	-
	Package Characteristics	0.86	72%	0.68	113%
95%	Stratified Baseline	0.50	-	0.32	-
	Balanced Only	0.50	-	0.18	-
	Package Characteristics	0.88	76%	0.70	119%

default by the npm ecosystem (npm, 2022a), the results indicate that there is a considerable number of packages for which developers do not believe the balanced update strategy to be suitable.

In Section 4.2, we discussed the impact of alternative specialization thresholds on the class distribution. Additionally, we have analyzed the impact of alternative specialization thresholds on the performance of our model in Table 4.3. We look at the change in the ROC AUC and F1-score metrics and also calculate the minimum increase in model performance (i.e. the performance compared to the highest value among the stratified and the balanced only models). As can be seen in Table 4.3, increasing the specialization threshold to focus on higher majority agreements (i.e. 75%, 90%, 95%) actually results in a more performant model (when comparing each model to the corresponding baselines). However, as stated in Section 4.2, higher specialization thresholds result in a higher number of unspecialized packages for which there is no majority agreement on the update strategy. Our objective is to model the relationship between package characteristics and the common update strategy of its dependents in the npm ecosystem. A model that assumes a strictly high level of agreement among the dependents will be of limited use in practice as such agreement does not exist for many npm packages.

Finding #1: The quality of our classification model shows that package characteristics can be used as indicators of the common update strategy chosen by the package's dependent community.

Finding #2: While the balanced update strategy is recommended by npm, the recommended update strategy from the package characteristics model is better aligned with the update strategy selected by npm developers.

4.3.2 RQ2: Which package characteristics are the most important indicators for dependency update strategies?

Motivation: There is a large array of characteristics for packages in the npm ecosystem and some create extraneous noise in understanding and selecting the appropriate update strategy while others might even mislead the community. By identifying and studying the most important characteristics that are associated with update strategies, the community can better understand the type of packages that fall into each of the three specialization groups. As previously stated, opting for the suitable dependency update strategy for a package can prevent dependency issues that arise from alternative update strategies (Jafari et al., 2021). Therefore, developers also need to know which characteristics should be prioritized when deciding on an update strategy and how the increase or decrease of such characteristics would impact the commonly selected dependency update strategy.

Approach: Package characteristics which have a larger impact on the model's prediction of the commonly used dependency update strategy are better indicators of the update strategy. In order to calculate the feature importance in our model, we use the permutation feature importance instead of the default impurity-based feature importance of Random Forest. The impurity-based feature importance inflates the importance of high cardinality features and it is biased to the importance of features in training the model, rather than

their capacity to make good predictions (Scikit-learn, 2020). The 10-fold permutation importances in Figure 4.6 are calculated by randomly permuting each feature 10 times and observing its impact on the model's performance (ROC-AUC score). A feature is deemed more important if permuting its values has a larger impact on the model's performance.

In order to visualize how a change in a package characteristic (feature) impacts the model's decision making for each class, we present Partial Dependence Plots (PDP) for the top 3 important features in Figure 4.8 (since the top 3 are the most prominent). Partial dependence plots visualize the marginal effect of a feature on the prediction of the machine learning model (Molnar, 2020). PDPs can highlight linear, monotone or more complex relationships between the feature and the target. In the case of our model, the PDPs in Figure 4.8 can show how an increase or decrease in a feature (such as age) can increase or decrease the model's likelihood to predict the balanced class (or any other class). Since partial dependence is plotted across the distribution, we also plot the distribution plots of the top 3 features to emphasize where the PDPs have more weight. The Y-axis represents the predicted probability for an instance belonging to the mentioned class. The tick marks on the X-axis of the PDPs represent the deciles of the feature values, which are consistent with the distributions in Figure 4.7.

Results: The box-plots of Figure 4.6 present the top 10 most important features which are associated with the commonly used dependency update strategy. As can be seen, release status, dependent count and package age are the most important indicators for dependency update strategies. This hints that these features are highly relevant in influencing decisions about dependency update strategies. Release status is the most relevant feature for the model. Knowing if a package is in early development or post-production is one way to gauge the stability of new releases, which in turn is a way to gauge the degree of freedom dependents give to automatic updates for that package. Additionally, since SemVer considers pre-1.0.0 versions to be unstable, any update strategy that permits even the smallest

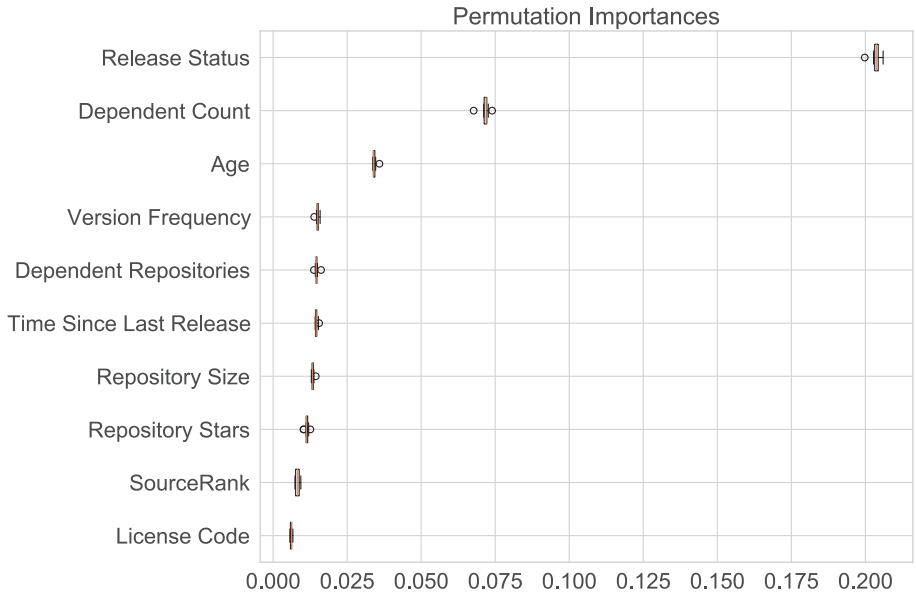


Figure 4.6: Importance of Features

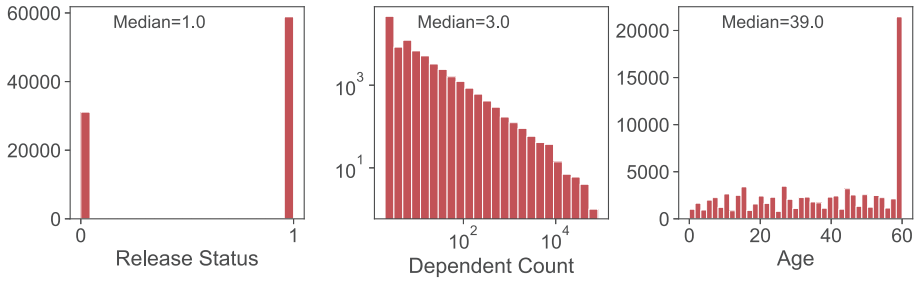


Figure 4.7: Distribution of the top 3 important features (Dependent Count is log10 distribution)

degree of freedom in receiving new versions (i.e. only accepting patch releases) is considered permissive. This allows the model to use release status to identify many instances of permissive-labeled packages. The high rankings of dependent count and age hints that both popularity and maturity are good indicators of the common dependency update strategy toward the package.

Finding #1: The most important indicators for the common dependency update strategy toward a package are its release status, number of dependents and age.

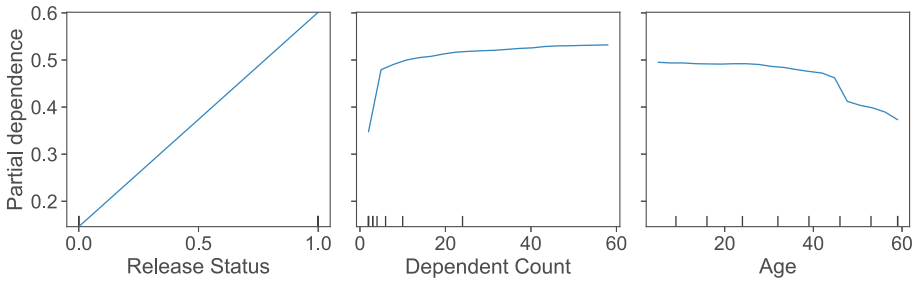
The distributions for the top 3 features can be seen in Figure 4.7. The majority of

packages (65.5%) are in a post-1.0.0 release state with a median of 3 dependent packages and 39 months (3+ years) of age. The distribution of values for most of the top features are highly skewed. Therefore, it is necessary to consider this skewed distribution when analyzing the impact of features.

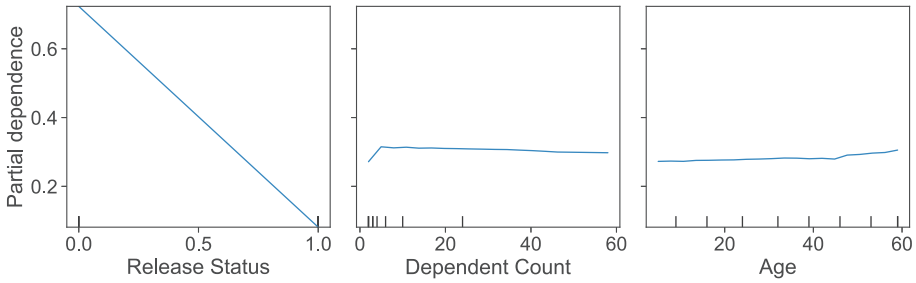
Figure 4.8 depicts the partial dependence plots for the top 5 features. The partial dependence plot for release status is unsurprisingly linear since release status is a binary feature. The steep slope of the release status dependence plot is also expected as we previously discovered this feature to be highly important for the model. The impact of release status on the common dependency update strategy is straightforward and intuitive. **Post-1.0.0 releases result in balanced dependency update strategies, and pre-1.0.0 releases result in more permissive update strategies**. In other words, knowing whether a package is in post-1.0.0 production or in pre-1.0.0 initial development is a good way to decide how permissive or restrictive one should be when depending on that package. As stated previously, this is partly due the treatment of pre-1.0.0 release by the SemVer standard. SemVer considers pre-1.0.0 versions to be unstable by nature and any update strategy that permits even the smallest degree of freedom in receiving new versions (i.e. only accepting patch releases) could introduce backward compatibility issues (Preston-Werner, 2019). This finding also aligns with the previous investigations of Decan et al. that found the majority of dependencies toward pre-1.0.0 releases to accept patch releases, which is more permissive than what SemVer recommends (Decan & Mens, 2021).

Looking at the partial dependence plots for dependent count, we see that **higher dependent count increases the likelihood of balanced update strategies** (i.e. dependents of a package tend to agree on the balanced strategy, when the package has more dependents). In a developer survey, Bogart et al. found that the value of avoiding breaking changes grows with the user base of a package (Bogart et al., 2016). Consequently, the user base of such packages may be more likely to perceive the balanced update strategy to be "good enough"

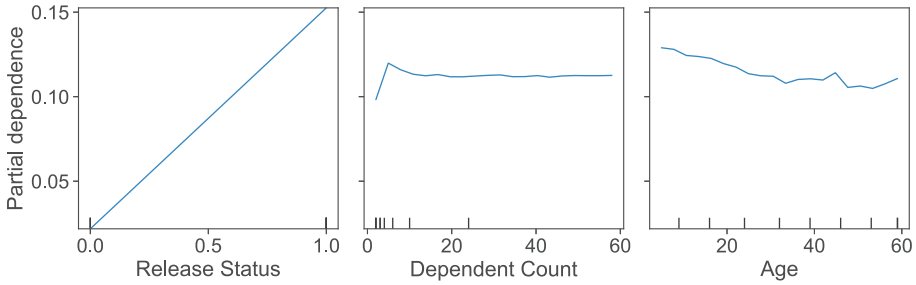
in preventing breaking changes for highly used and mature packages.



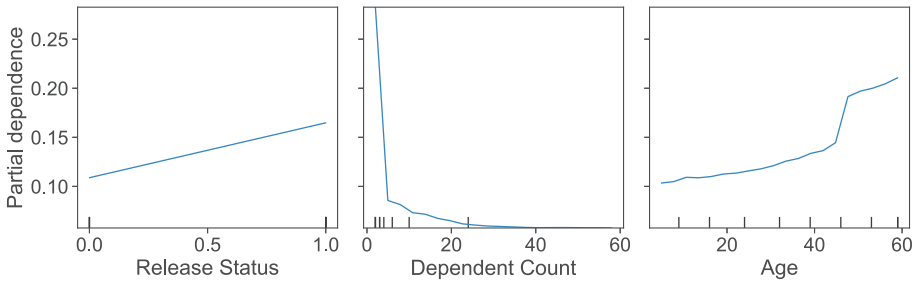
(a) Balanced Class



(b) Permissive Class



(c) Restrictive Class



(d) Unspecialized Class

Figure 4.8: Partial Dependence Plots (PDP) for each class

The distribution in Figure 4.7 should be taken into account when discussing the PDPs. Since the median dependent count is 3, the left portion of the plot has more weight. It is

also important to highlight that **packages with very few dependents (less than 5) have a considerably higher chance of not being specialized** (i.e. dependents of packages with few dependents are less likely to agree on a dependency update strategy). This is a natural consequence of lesser dependents as there is not yet enough dependents (and perhaps package history) to reach an agreement on how to treat that package as a dependency. Additionally, dependents may be more inclined to choose an update strategy based on personal preference if there is no established popular update strategy for the upstream package.

The partial dependence plots for age reveals that developers tend not to favor the balanced update strategy for old packages, specifically those older than 45 months. Cross referencing this information with the distribution gives further insight. Since the majority of the packages in the dataset are in fact more than 39 months old (right portion of plot has more weight), **we can conclude that in general, dependents of newer packages favor the balanced update strategies more than dependents of older packages.** The SemVer caret notation was established as the npm default in 2014 (Decan & Mens, 2019a; Oxley, 2014). This alone could gradually shape the update strategy the majority of developers choose for newer packages. On the other hand, some might deem an old project as stagnant and will not worry about a new release that breaks the API, which can justify permissive update strategies.

Finding #2: Package characteristics are highly skewed and packages with less than 5 dependents are less likely to be specialized toward a particular dependency update strategy.

Finding #3: Dependents of younger, post-1.0.0 release packages with more dependents are more likely to use the balanced update strategy while dependents of pre-1.0.0 release packages are more likely to use the permissive update strategy.

4.3.3 RQ3: How do dependency update strategies evolve with package characteristics?

Motivation: According to our model, characteristics such as release status, dependent count and age have the largest impact on the dependency update strategy. Interestingly, all of these top characteristics are indicative of how a package evolves over time (since dependent count generally increases over time and release status is changed once in a package's lifetime). Consequently, there can be multiple explanations for how the evolution of a package impacts the update strategy chosen by its dependents. For example:

- The common update strategy was different early on but dependents gradually shifted to a new update strategy.
- The common update strategy changed because new dependents are adopting a different strategy than old dependents.
- The common update was initially the same and dependents (new and old) simply followed the previous choice.
- The common update strategy experienced a shift due to the shift from a pre-1.0.0 version to a post-1.0.0 version.
- The common update strategy experienced a sudden shift due to an anomalous event in the package's lifecycle.

While we know that release status, dependent count and age are related to the currently popular dependency update strategy, we need to see if such a relationship was preserved through the package's evolution or if perhaps, it is a result of an external event. Understanding the evolution of dependency update strategies toward a package will provide much needed insight into why the characteristics that are most relevant to the dependents' update strategy are all related to a package's evolutionary behavior.

Approach: Evaluating the evolution of dependency update strategies is carried out through a mix of quantitative and qualitative techniques. We take a random sample of 160 packages from the dataset (40 packages from each of the three update strategies + 40 unspecialized packages) for a historical analysis of each package's dependents over the last 10 years up to the latest snapshot of the dataset. We want to look at packages with over 100 dependents in the hopes of disregarding packages with very limited historical dependent data. Therefore, half of this sample dataset consist of packages with 100 to 1000 dependents (in the latest snapshot) and the other half have more than 1000 dependents (in the latest snapshot). This sample of 160 packages is not meant to be a representative sample of the main dataset. Rather, it is "convenience sample" (Baltes & Ralph, 2022) consisting of reasonably used packages selected for an in-depth mix-method study that is otherwise not feasible on a large dataset.

For each package, we utilize a monthly snapshot of the ecosystem to identify dependents at each month. We then analyze the dependency requirement constraints to identify the number of dependents using a particular update strategy per month. Since the age of a package increases with time, visualizing the dependency update strategies over time is akin to plotting the evolution of update strategies over the package's lifecycle. It is important to note that even though we take 40 samples from each group (balanced, restrictive, permissive, unspecialized), we still plot all update strategies for each package, since a package currently specialized toward a restrictive update strategy for example, may have other strategies used by its dependents throughout time. To eliminate the bias toward dependents that release more frequently, we only consider the latest version of each dependent at each month (i.e. each dependent package is counted only once per month, regardless of how many versions it maintains).

Results: We present the commonly observed evolution patterns for dependency update strategies along with real examples that embody the findings. While age and dependent

count do not increase at the same rate, their relationship with the evolution of update strategies proved to be similar. Thus, we focus our analyses on the evolution of dependency update strategies across package age. The complete set of visualizations for each package can be accessed through our replication package (Javan Jafari et al., 2022).

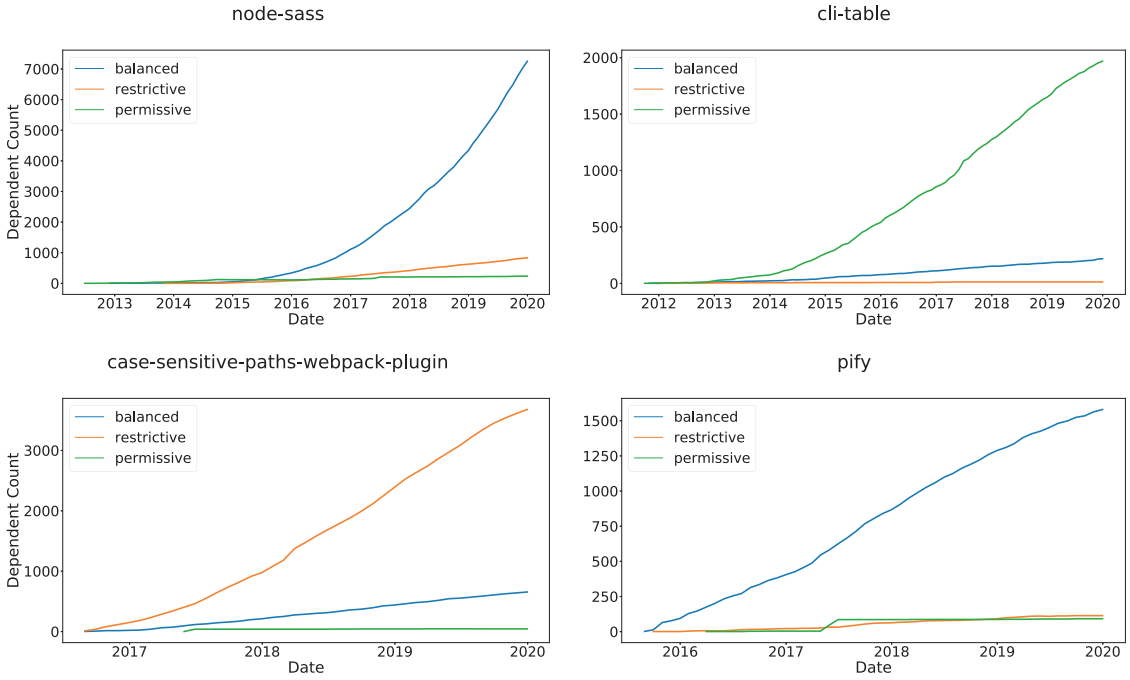


Figure 4.9: Example packages for which dependents follow the previously popular update strategy

One common evolution pattern is the tendency of dependents to follow the previously popular update strategy (i.e. agreement on the common update strategy does not change throughout the package's lifecycle). This evolution pattern was observed across the dependents of all package groups as shown in Figure 4.9. We observed this pattern for 18 instances of balanced packages, 28 instances of permissive packages and 6 instances of restrictive packages. This finding aligns with the observation of Dietrich et al., which state that packages tend to stick to their dependency habits for a particular dependency (Dietrich et al., 2019). It is also worth noting that this behavior was observed in example packages specialized to all of the three update strategies, meaning it is not a result of dependents

merely using the default npm update strategy (which leans toward the balanced update strategy).

Finding #1: For many npm packages, the common update strategy of its dependents remains consistent.

The pre-1.0.0 release versions of an npm package is considered to be unstable due to its initial development stage. However, Decan et al. studied package usage for pre-1.0.0 releases and found that there is no considerable difference between the number of dependents for pre-1.0.0 and post-1.0.0 releases (Decan & Mens, 2021). In our sample dataset, we observed an interesting phenomenon when a package releases its 1.0.0 version. When a highly used pre-1.0.0 package releases switches to a post-1.0.0 status, there is a very observable shift from permissive to balanced update strategies among its dependents. The examples in Figure 4.10 clearly show the impact of the 1.0.0 release (red line) on the update strategy evolution. While there are still dependents that use the permissive update strategies after the 1.0.0 release, the majority of new dependent relationships shift to the balanced strategy. The pattern generally appears when the pre-1.0.0 releases were already used by many dependents (which is why it can not be observed in the examples of Figure 4.9). This pattern may have occurred because the npm community is less accepting of the SemVer standard as it pertains to pre-1.0.0 releases and does not believe pre-1.0.0 dependencies should necessarily be pinned to a particular version (Decan & Mens, 2021). This particular pattern is observed for 12 instances of balanced packages, 6 instances of permissive packages and 15 instances of unspecialized packages.

Finding #2: For highly used pre-1.0.0 packages, the release of the 1.0.0 version can change the common update strategy from permissive to balanced.

The evolution of update strategies for dependents of packages specialized toward the restrictive update strategy exhibits unusual and anomalous behavior that is not observed in

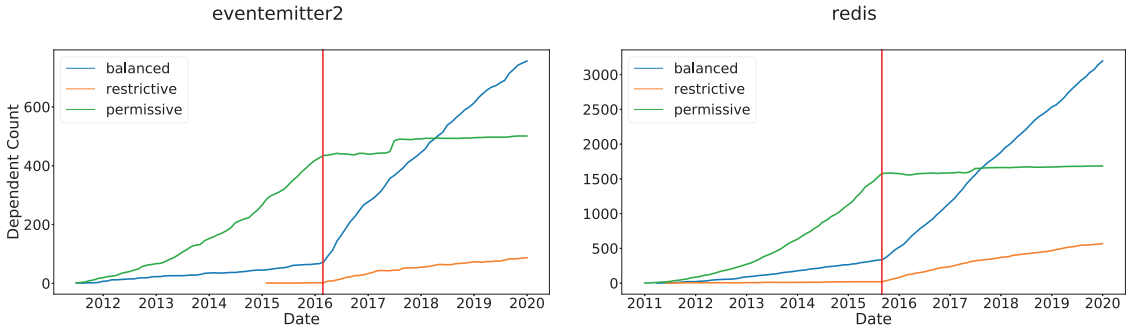


Figure 4.10: Example packages for which the dependent strategy shifts at the 1.0.0 release mark (red vertical line)

the other two package groups (balanced and permissive). First of all, it is more common to see packages that have a borderline agreement in the restrictive cases. The examples in Figure 4.11 show that while the evolution of update strategies for these packages ultimately leads the restrictive update strategy as the dominant one, a very considerable number of dependents still use the balanced update strategy when depending on these packages. Restrictive update strategies are a reluctant response to breaking changes or other problems with automatically updating to new minor versions of the dependency (Jafari et al., 2021). Therefore, the observed disagreement on the restrictive update strategy can happen because either a portion of the community is not aware of an existing issue with the package or because the issues do not equally affect all dependents. We observed this pattern in 10 instances of restrictive packages and 6 instances of unspecialized packages.

Finding #3: Even when restrictive update strategies are the majority, they experience weaker agreements due to many dependents opting for balanced update strategies.

The other unusual observation for restrictive dependency update strategies is their anomalous evolutionary behavior. For example, in the evolution of update strategies for packages in Figure 4.12, we see a sudden spike in the number of restrictive update strategies starting at a specific point in time that is very dissimilar to the gradual increase of the other

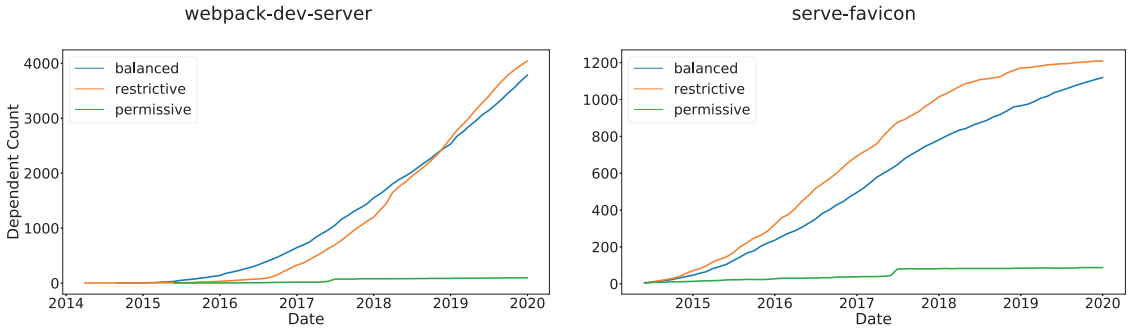


Figure 4.11: Example packages for which there is a weak agreement on the restrictive update strategy

two update strategies. This can happen if a particular event in time (perhaps a breaking change) causes a shift in community perception toward that package. The observation may also be due to a new set of dependents with more conservative strategies that started using the package for the first time. The latter is more likely in cases such as *detect-port* and *identity-obj-proxy*. Alternatively, in cases such as *promise* and *raf* where the community moves back to the balanced strategy after a certain amount of time, the former explanation is more likely. We found such anomalous behavior in 4 instances of balanced packages, 8 instance of restrictive packages and 3 instances of unspecialized packages.

The findings for the evolution analysis of the restrictive update strategy warrants a closer look into the capability to identify them using package characteristics. While RQ1 presents the overall performance of our model, the per-class evaluation results can provide further insight. Table 4.4 presents the precision, recall and F1-score for each of the 3 main classes of the model, along with the unspecialized label (since some npm packages are not specialized toward any update strategy and they must also be included in the evaluation). We have also included the per-class F1-scores for the two baseline models for comparison. F1-Stratified denotes the F1-score for the stratified baseline and F1-Balanced denotes the F1-score for the Balanced only model. While our model outperforms the baseline for all 3 main classes, the restrictive class seems to be more difficult to predict across all models.

Table 4.4: Per-Class Evaluation

Class Label	Precision	Recall	F1-score	F1-Stratified	F1-Balanced
Balanced	80%	84%	82%	54%	70%
Permissive	74%	85%	79%	29%	0%
Restrictive	77%	32%	45%	6%	0%
Unspecialized	47%	33%	39%	9%	0%

Specifically, our model achieves high precision but low recall for the restrictive cases, indicating the model is mostly correct when classifying a restrictive package, but it also misses many of the other restrictive cases. The challenges in predicting the restrictive update strategy can be due to the limited number of packages specialized toward the restrictive strategy in the ecosystem (7% of our main dataset) or due to the incidental nature of such strategies that are caused due to target package misbehavior (e.g. breaking changes) rather than its characteristics.

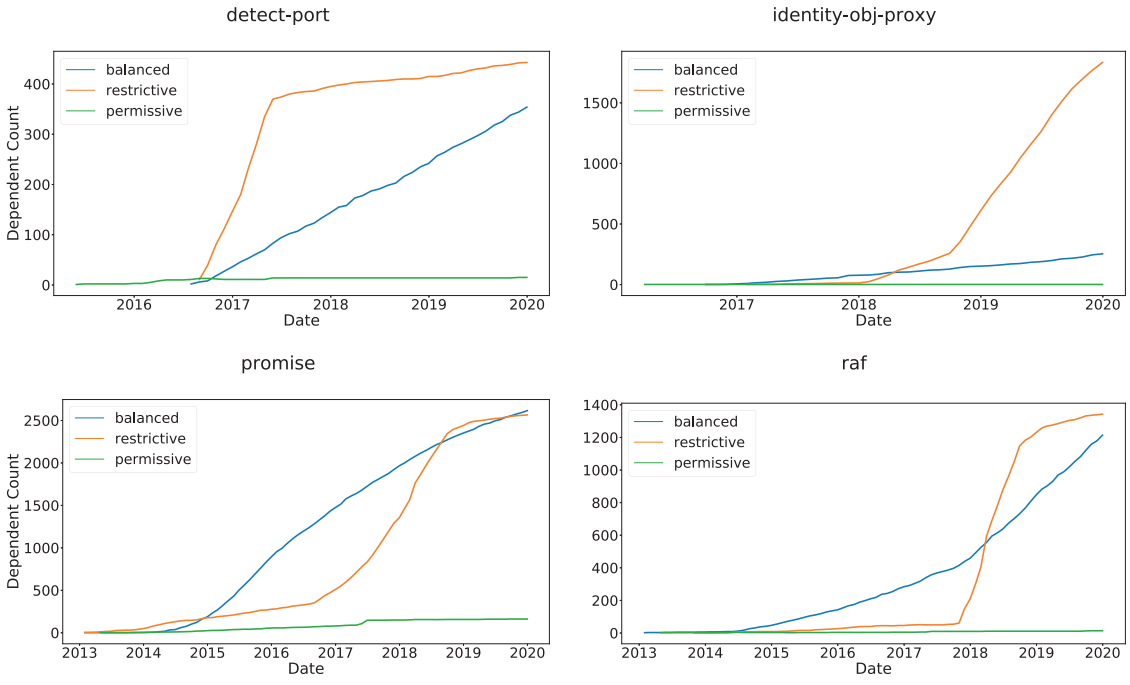


Figure 4.12: Example packages for which the restrictive update strategy exhibits anomalous behavior

Further examination of the anomalous behavior in the evolution of restrictive update

strategies necessitates a qualitative approach. Thus, we manually analyze 1) The npm registry (npm, 2022b), 2) The snyk open source advisory (snyk, 2022) and 3) The GitHub repositories of the 40 sampled packages in the restrictive group. The npm registry provides information regarding installation notes, current weekly downloads of each version and build status badges. The snyk advisory provides information about known security vulnerabilities along with a package health score that considers security in addition to package popularity and maintenance. The GitHub repository provides the development history of the package. Using the repository information, we can filter created and resolved issues during a specific historical window to identify breaking changes that may correspond to the rise of a restrictive update strategy for that package.

We started with the npm registry page of each package to search for mentions of SemVer non-compliance from maintainers of the package. We hypothesized that one reason for the popularity of restrictive update strategies for this group of packages would be the official statements by package maintainers that indicate their misalignment with SemVer compliance. None of the 40 packages had stated anything about the recommended update strategy. Thus, we can speculate that the choice of a restrictive update strategy is solely on the dependents' side. One interesting observation was the maintainer's recommendation to install their packages as a development dependency, as opposed to a runtime dependency, in 65% of these packages. Since our dataset is filtered to only include runtime dependency relations, many dependents have obviously not followed this recommendation.

The snyk advisory provides a package health score that combines security, popularity, maintenance and community factors into a single metric (snyk, 2022). More importantly, snyk is a vulnerability dataset that catalogs low, medium, high and critical severity vulnerabilities recorded for each version of a package. We hypothesized that vulnerable releases will encourage package dependents to restrict their update strategies while they wait for a fix to be released. With the exception of the "webpack-dev-server" package in which 144

Table 4.5: Examples of created issues that correspond with a rise of restrictive update strategies

Package Name	Issue Date	Issue Title
postcss-loader	Jan 2017	"v1.2.1 runs fine, but v1.2.2 throws error"
eslint-plugin-jsx-a11y	Jun 2016	"Exception after update to 1.4.0"
jest-resolve	Dec 2018	"medium severity vulnerability [...] introduced via jest @23.6.0"
eslint-loader	Apr 2015	"npm error after update to version 0.11.0"
fsevents	Feb 2017	"breaking change in 1.1.0"

versions were infected by a high severity vulnerability, the rest of the packages in our sample had no recorded vulnerabilities. In simpler terms, we could not find sufficient evidence that indicate restrictive update strategies are mainly the result of vulnerable releases.

The GitHub repository of the packages allows open access to the development history of the package, along with recorded issues and feature requests.

We hypothesized that breaking changes from new releases may be a reason why dependents opt for a more restrictive update strategy. To this aim, we searched through repository issues created for each package during the one year window in which we observed a rise in restrictive update strategies from the dependents of that package. We found concrete evidence of breaking updates in 18 of the 40 packages in the restrictive group. Not all breaking updates lead to newly created issues about the problem, so our findings are actually a lower bound on the number of packages that experience breaking changes. In fact, out of the 22 packages with no evidence of breaking changes, 11 packages had low activity (less than 50 open and closed issues combined) or no activity in their repository issue tracker throughout the project's history. Table 4.4 presents example issues from package repositories where users voice their concerns about breaking changes (or other problems) caused by updating to a new version. These findings align with prior research that identifies breaking changes and dependency misbehavior as highly influential factors in restrictive dependency update policies by the dependents (Jafari et al., 2021).

Finding #4: Restrictive update strategies exhibit a more erratic evolutionary behavior that corresponds to breaking changes, making them harder to predict

4.4 Implications

We present actionable implications for both practitioners (developers and package maintainers) and researchers in the field.

Implications for Practitioners:

The package characteristics model presented in this study has been shown to outperform the default balanced update strategy in npm (RQ1). **The predictions of the model can be used as a recommendation for developers to help them in deciding on a suitable dependency update strategy for a package. Alternatively, practitioners can rely on the most important features such as release status, dependent count and age (RQ2) to aid their dependency update strategy selection.** For example, using packages with a smaller number of dependents poses an inherent risk of not yet having an agreed upon update strategy in the community. In addition to the number of dependents, the prominence of those dependents should also be taken into account.

The release status of a package (pre-1.0.0 vs. post-1.0.0) has shown to be a relevant feature in identifying the common update strategy (RQ2) and there is an observable shift from the permissive update strategy to the balanced strategy when the 1.0.0 version is released (RQ3). The use of permissive constraints for pre-1.0.0 packages shows that developers in the npm community do not fully align with the SemVer standard for pre-1.0.0 releases. It is also a testament to the relatively high popularity of some pre-1.0.0 packages. We looked at the number of dependents for both the pre-1.0.0 and post-1.0.0 packages and found that while post-1.0.0 packages have a median of 4 dependents, pre-1.0.0 have a median of 3 dependents. This is surprising as SemVer considers pre-1.0.0 initial development releases

to be unstable by nature and depending on them poses an inherent risk. Yet, a considerable portion of developers are already using such packages as dependencies. This confirms the findings of Decan et al. (Decan & Mens, 2021) and highlights the importance of initial development releases for package maintainers. **Package maintainers should assume that initial development releases may already be used by dependents which could be stakeholders in future changes.**

While studying the evolution of dependency update strategies, we observed many instance where the initially established update strategy was also selected by new dependents, creating a compounding effect that ultimately leads to a clearly dominant dependency update strategy for dependents of that package. We did not find significant evidence of target packages recommending a particular update strategy to their users and this continuous trend was observed for all 3 types of update strategies (i.e. it can not simply be attributed to the use of the default balanced update strategy). Therefore, this behavior likely stems from independent decisions from package dependents, some of which may consider the previously common update strategy to be the best one. **Ecosystem maintainers should be attentive to the early adopter community of their packages as the first impressions set by the initial community can have long-lasting influence on how new dependents use their package.**

Implications for Researchers:

While the package characteristics model in this study can be leveraged to predict the suitable dependency update strategy (RQ1), there are other characteristics to explore. Further research is needed to extract and **look into other features such as the package downloads count, code complexity, the experience level of package maintainers and the quality of the documentation to see if and how these features can improve the model**. Additionally, since we know that restrictive update strategies may be influenced by specific events rather than package characteristics (RQ3), future work is needed to cross-reference

the time of the change with relevant events in the repository such as a bug/vulnerability fix or a newly opened issue to understand how such events can influence a change in the dependency update strategy. **We should also look at the frequency of change and the duration between changes in the dependency update strategy to better understand whether some events such as breaking changes have long-term impact on the trust of a particular package**.

The current model proposes a predicted update strategy based on the characteristics of a target package. However, it is beneficial to know the confidence in the recommended update strategy and the rankings of the non-recommended alternatives. While developers can use the important features discovered in this study as the basis for their own judgment, **a probabilistic model that complements the predictions by presenting a ranking of recommended update strategies can prove useful.**

Not knowing why different dependency update strategies occur in a package creates data noise when analyzing the strategies. We previously discussed how npm default constraints for newly added dependencies (RQ2) create a challenge when analyzing the wisdom of the crowds since we do not fully know whether the developer chose the constraint or simply trusted the default update strategy. Using the balanced strategy can be traced back to meticulous planning by the dependent or a simple disregard toward dependency maintenance. **A valuable avenue for research is to study how much the ecosystem is impacted by developer decisions versus ecosystem policies, such as default dependency constraints.**

Restrictive update strategies are a response to issues such as breaking changes when updating dependencies. However, the entire dependent community of a package may not be equally aware or equally affected by such issues, which leads to weaker agreements on the restrictive update strategy (RQ3). In the wisdom of the crowds model, a high level of restrictive strategies (and their underlying cause) may be disregarded simply because

they do not represent the majority. **An improved version of the model presented in this study can allow the specialization threshold to differ per each class to allow a strategy-sensitive model that is tuned to better predict the probability of a particular update strategy.**

4.5 Related Work

To the best of our knowledge, there is no other work that utilizes package characteristics to predict the most suitable dependency update strategy and studies the impact of those characteristics on the selected strategy. The related work for our study is comprised of research that focuses on dependency update strategies, studies that focus on relevant characteristics in selecting dependencies and research in the npm ecosystem supply chain.

Dependency update strategies:

Decan and Mens conducted an empirical study to compare SemVer compliance across four software ecosystems including npm (Decan & Mens, 2019a). They proposed an update strategy based on "the wisdom of the crowds" to help developers choose the best dependency update strategy. They accomplished this by analyzing the dependency constraints of all dependents of a package and recommending the most common update strategy. This study is the most relevant to our work as it uses past dependency decisions to predict the most common update strategy in the future. However, the work of Decan et al. does not use package characteristics for prediction and requires a complete and updated dependency graph of the npm ecosystem, making it unscalable in practice. Our method is scalable as it only looks at the current characteristics of the package and does not need dependency information from the dependents. More importantly, our work is the first to study the relationship between package characteristics and the predicted dependency update strategy. In another study, Decan et al. empirically investigated the pre-1.0.0 versions and their usage

in 4 software ecosystems. They found that there is no practical difference between the usage of pre-1.0.0 and post-1.0.0 versions but ecosystems are more permissive than SemVer guidelines when it comes to using pre-1.0.0 versions (Decan & Mens, 2021).

Dietrich et al. studied dependency versioning practices across 17 software ecosystems including npm (Dietrich et al., 2019). Their study is complemented by a survey of 170 developers. They found that most ecosystems support flexible versioning practices but developers still struggle to manage the trade-offs between the predictability of more restrictive update strategies and the agility of more flexible ones. Feedback from more experienced developers suggest they favor the stability that accompanies restrictive update strategies. Dietrich et al. did not look at how package characteristics can impact the selected dependency update strategy and how such package characteristics can be used to guide developers towards the suitable strategy.

Jafari et al. empirically studied problematic dependency update strategies in JavaScript projects (Jafari et al., 2021). They cataloged and analyzed 7 dependency smells including restrictive constraints and permissive constraints. Their findings indicate that while smells are prevalent, they are localized to a minority of each project's dependencies. Through a developer survey, they highlighted the negative impacts of such update strategies and they also quantified the reasons for their existence. They found that such alternative update strategies are often the result of dependency misbehaviour or issues in the npm ecosystem. While Jafari et al. did not look at the impact of package characteristics on dependency update strategies, their work highlights the importance of studying such characteristics to understand why some npm packages implicitly push their dependents to use non-balanced dependency update strategies.

Package characteristics for selecting dependencies:

Bogart et al. performed an empirical study on three software ecosystem including npm

to study how developers make decisions in regard to change and change-related practices (Bogart et al., 2016). In their interview with 28 developers, they found that various signals are used to select dependencies. These include the level of trust on the developers of the package, activity level, user base, project history and artifacts such as documentation. The respondents believed such characteristics to be important in deciding what package to depend on, but the study did not look at how package characteristics can influence the chosen dependency update strategy.

Vargas et al. surveyed 115 developers to study the factors that impact the selection of dependency libraries (Larios Vargas et al., 2020). They observed several technical factors such active maintenance, code stability, release frequency, usability and performance to be relevant factors. The authors also observed human factors such as community perception and popularity along with economic factors such as license and cost of ownership to be contributing factors in selecting a dependency.

Pashchenko et al. interviewed 25 industry practitioners to investigate the influence of functional and security concerns on decision making with regards to software dependencies (Pashchenko et al., 2020). The authors found that developers rely on high-level information that demonstrates the community support of a library such as popularity, commit frequency and project contributors. Developers prefer libraries that are safe to use and do not add too many transitive dependencies. The authors observed that dependency selection is often assigned to more skilled members of the team.

Haenni et al. conducted a survey and asked developers about their information needs with respect to their upstream and downstream packages (Haenni et al., 2013). Developers stated that they consider factors such as popularity, documentation, license type, update frequency and compatibility when looking for a new dependency. The authors also found that in practice, developers monitor news feeds, search through package websites and blogs and run their unit tests to achieve these goals.

The four aforementioned studies all focus on relevant characteristics in selecting a package as a dependency. They do not study the impact of these characteristics on the update strategy used for each dependency.

The npm ecosystem supply chain:

Zimmerman et al. studied how the packages and package maintainers in npm have the potential to impact large chunks of the ecosystem (Zimmermann et al., 2019). They looked at a collection of more than five million package versions in npm and observed that installing an average npm package is the equivalent of implicitly trusting 79 packages and 39 maintainers. Additionally, they realized that up to 40% of npm packages depend on a vulnerable package with a publicly disclosed vulnerability. The authors found that, among other things, locking dependencies exacerbates the security issues in the ecosystem since it hinders the automatic adoption of a vulnerability fix.

Zerouali et al. empirically analyzed the technical lag in the npm ecosystem and its relationship to dependency update strategies (Zerouali et al., 2018). The authors used a subset of the libraries.io dataset comprised of 610K packages and over 4.2 million package versions. They found that while npm packages are frequently updated, dependencies are rarely added or removed. They also discovered that restrictive dependency update strategies are the main culprit for technical lag in the ecosystem.

Cogo et al. conducted an empirical study on same-day releases in the npm ecosystem (Cogo, Oliva, Bezemer, & Hassan, 2021). They found same day releases to be common in popular packages, interrupting a median of 22% of regular release schedules. More importantly, they observed that 32% of such releases encompass even larger changes than their prior (planned) release. In general, downstream dependents of popular packages tend to automatically adopt same-day releases due to their dependency update strategies. The authors believe same-day release to be a significant occurrence in the npm ecosystem and

dependency management tools should consider flagging such releases for downstream dependents.

Chowdhury et al. studied trivial packages in the npm ecosystem (micro-packages with only a few lines of code) (Chowdhury et al., 2021). They found that close to 17% of the packages in the ecosystem can be considered trivial, but removing one of these packages can impact up to 29% of the entire ecosystem. While such small packages are small in size and complexity, they are responsible for a high percentage of API calls. Trivial packages play an important and significant role in the npm ecosystem.

4.6 Threats to Validity

This section discusses the threats to the validity of our study.

Threats to construct validity consider the relationship between theory and observation, in case the measured variables do not measure the actual factors. Our specification of dependency update strategies considers version constraints and assumes developers use the official npm registry to fetch their dependencies. In reality, developers can look outward and use external sources to fetch dependencies (e.g. direct link to Github repository). One issue with such cases is that the update strategy could change depending on the contents of the external source. For example, linking to the master branch is equivalent to a permissive update strategy and linking to a specific release is equivalent to a restrictive update strategy. Another issue is that there is no way to identify all package dependents if the package is hosted on an external link. In order to study both the dependencies and the dependents of the packages, our study only considers packages hosted on the official npm registry and dependencies pointing to other packages in the npm ecosystem. Additionally, we assume the information provided by the libraries.io dataset (Libraries.io, 2020) is accurate, and this assumption has been verified by other researchers (Decan et al., 2019).

Threats to internal validity refer to internal concerns such as experimenter bias and error. The npm ecosystem is very large and susceptible to noisy/toy packages. We disregard packages with less than 2 dependents which removes unused packages from our dataset. We also manually remove multiple spam packages (and their dependencies) which had the sole purpose of depending on every other package in the ecosystem (Section 4.2). In order to train our model, we use 19 features that we believe to influence dependency decisions based on the literature. In reality, there may be other relevant information for deciding on the dependency update strategy that were not captured (or not feasible) using our feature set. For example, developers can change dependency update strategies following a recommendation from a senior member of the team or because the specific section of the code relying on the dependency is critically important. We believe our features to be suitable since we cross-referenced the relevant characteristics for dependency selection and management that we found in the literature, with the package characteristics available in the npm registry and the code repository. We discovered features with missing data in the repository fields of the libraries.io dataset, warranting a look into the accuracy of the dataset. For many features (e.g. Dependency Count) the null value was used to denote zero as the minimum value starts at one. However, in 3 out of the 19 features selected for our model (Repository Stars Count, Repository Size, and Repository Open Issues Count), we found missing values where a value of zero was also present. We took a sample of 1000 packages that had missing data corresponding to the three features and realized 96.1% of these packages do not have a working repository link (repository no longer exists). Section 4.2 explains how we handled missing values in our dataset. Our findings regarding the accuracy of the libraries.io dataset corroborates the previous analysis of Decan et al. in which they manually cross-checked the libraries.io dataset against their own collected metadata from the npm registry and verified its accuracy (Decan et al., 2019).

Threats to external validity concern the generalization of our findings. The observed

findings are specific to the npm ecosystem since previous research has shown that different ecosystems have different practices and cultural values (Bogart et al., 2017, 2016). However, the package characteristics, the methodology to extract the features and the update strategy to train the model can be replicated on other ecosystems that provide similar dependency information. In fact, since the libraries.io dataset (Libraries.io, 2020) used in this study utilizes the same schema to store metadata for other ecosystems such as PyPI and Maven, our replication package (Javan Jafari et al., 2022) can easily be used to replicate the study on other ecosystems. Additionally, the libraries.io dataset used in this study does not contain npm package data after January 2020. However, re-collecting the dataset for an entire ecosystem such as npm does not only require a lot of effort, but it is error-prone. The accuracy of the libraries.io dataset has previously been verified in the literature (Decan et al., 2019). More importantly, our study is more focused on the dynamics of dependency management in the npm ecosystem, rather than predicting the update strategy for the latest available version. Therefore, we believe the dataset to be suitable for our study. The findings of RQ3 are derived from a sample of 160 packages. While these packages are selected at random, we want to focus on packages with adequate historical dependent data. Therefore, our selection criteria requires packages to have more than 100 dependents, which threatens the generalizability of the results of this particular RQ to packages with a small number of dependents. As previously mentioned, the sample of 160 packages is not meant as a representative sample of the entire ecosystem. It is a convenience sample of highly used packages for an in-depth mixed-method study that is otherwise infeasible for such a large ecosystem.

4.7 Chapter Conclusion

In our study, we use a curated dataset of over 112,000 npm packages to collect and derive 19 package characteristics from the their npm registry and code repository. We use

these characteristics to train a model to predict the most commonly used dependency update strategy for each package. Based on the wisdom of the crowds principle, we believe the update strategy used by the majority to be favorable to the alternatives. We show that these characteristics can in fact be used to predict dependency update strategies. We analyze the most important features that influence the predicted update strategy and show how a change in these features influences the predictions. Developers should take note of the highly important characteristics and their impact when making dependency decisions about a package. The results show that our model outperforms the alternative of merely using the balanced update strategy in all instances. We complement the work with a manual analysis of 160 packages to investigate the evolutionary behavior of dependency update strategies and understand how they are impacted by events such as the 1.0.0 release or breaking changes.

The findings of this study can be used to better manage direct dependencies. However, managing the dependencies of our project is not enough to mitigate dependency issues, especially those arising from transitive dependencies. Just like direct dependencies, transitive dependencies are installed along the code-base and can influence the project. Yet, developers have no control over what transitive dependencies are installed and how they are managed. Developers must rely on their direct dependencies to properly manage their own dependencies in order to prevent the propagation of problems (e.g. vulnerabilities) downstream. In the following chapter, we investigate the attributes of packages that quickly respond to vulnerabilities and how developers can leverage these attributes in their dependency selection criteria to mitigate the risk of vulnerabilities from transitive dependencies.

Chapter 5

Practices for Selecting Dependencies

Relying on dependency packages accelerates software development, but it also increases the exposure to security vulnerabilities that may be present in dependencies. While developers have full control over which dependency packages (and which version) they use, they have no control over the dependencies of their dependencies. Such transitive dependencies, which often amount to a greater number than direct dependencies, can become infected with vulnerabilities and put software projects at risk. To mitigate this risk, Practitioners need to select dependencies that respond quickly to vulnerabilities to prevent the propagation of vulnerable code to their project. To identify such dependencies, we analyze more than 450 vulnerabilities in the npm ecosystem to understand why dependent packages remain vulnerable. We identify over 200,000 npm packages that are infected through their dependencies and use 9 features to build a prediction model that identifies packages that quickly adopt the vulnerability fix and prevent further propagation of vulnerabilities. We also study the relationship between these features and the response speed of vulnerable packages. We complement our work with a practitioner survey to understand the applicability of our findings. Developers can incorporate our findings into their dependency management practices to mitigate the impact of vulnerabilities from their dependency supply chain.

5.1 Introduction

Software ecosystems have facilitated large scale code reuse by providing access to a wide range of software packages. In turn, developers are increasingly reliant on third-party packages to accelerate development (Bombonatti et al., 2017). However, developers also need to expend effort and expertise to manage their dependencies (Kula, German, et al., 2018a). In a recent survey, developers reported that they are not confident in their current dependency management practices (Tidelift, 2022). The Node Package Manager (npm) is the world's largest software package ecosystem with more than 2 million packages (npm, 2022b). The average number of packages in the npm ecosystem is growing year after year (Sonatype, 2021), but more importantly, the number of dependencies per package is growing at a super-linear rate (Zimmermann et al., 2019). The scale and complexity of the npm dependency network has created many dependency management challenges (Artho et al., 2012; Decan et al., 2017, 2019; Matt Rickard, 2021).

Security vulnerabilities are a prevalent issue in software ecosystems. The increase of deep dependency supply chains provides an exploitable opportunity for attackers. In the year 2021, there was a 650% increase in attacks aimed at exploiting vulnerabilities in upstream software packages (Sonatype, 2021). Up to 40% of all npm packages depend on code that is infected with a publicly disclosed vulnerability (Zimmermann et al., 2019) and the number of reported vulnerabilities in npm is increasing exponentially (Zerouali et al., 2022). The interconnected nature of software ecosystems increases the threat surface for vulnerabilities (Decan et al., 2018b; Liu et al., 2022). For example, when the popular "lodash" package was infected with a high severity vulnerability, more than 4 million open-source projects were exposed to a potential attack (Tal, 2019). When a client installs an npm package, they are implicitly trusting up to 80 other packages (on average), many of which are due to transitive dependencies (dependencies of dependencies) (Zimmermann et al., 2019). More than one third of the latest package releases in npm are exposed to

vulnerabilities through their transitive dependencies (Zerouali et al., 2022). Vulnerabilities in transitive dependencies can propagate to our project and yet, we have no direct control over what transitive dependencies (or what version) are installed alongside our project. Long dependency chains created through layers of transitive dependencies also impede the propagation of vulnerability fixes throughout the affected packages in the ecosystem (Alfadel et al., 2023; Chinthanet et al., 2021).

Since transitive dependencies are installed based on the requirements of our direct dependencies, the only means to address our exposure to vulnerabilities in transitive dependencies is relying on the responsiveness of our direct dependencies. *Responsive packages are packages that install the published fix for their vulnerable dependencies in a timely manner*. Developers need to align their dependency practices to select responsive dependency packages. We aim to identify and understand the attributes of npm packages that indicate better responsiveness to vulnerable dependencies. By selecting packages that quickly adopt vulnerability fixes in their dependencies, developers can reduce the risk of publicly disclosed vulnerabilities from transitive dependencies. Our research is formulated through the following questions:

We extract vulnerability data for 458 vulnerabilities from the npm advisory database. We then cross-reference the vulnerabilities with more than 154 million dependency relationships in the npm ecosystem, extracted from libraries.io to identify 201,027 unique packages that install (and later fix) vulnerable dependencies. We measure how long it takes for the dependent packages to adopt the vulnerability fix. We train a random forest model that uses 9 package attributes (e.g. release frequency) to classify the *fast-reponder* and *slow-responder* packages.

RQ1: How does dependency management impact the risk of vulnerabilities for downstream packages? While it is up to the maintainer of a vulnerable package to release a fix, dependent packages must decide on when and how they install the fix in their dependency.

The average public vulnerability is fixed in a little over 1, but it takes on average more than 6 months for dependents to install the fixed version, which highlights the importance of *package responsiveness* on mitigating vulnerabilities. Dependency management decisions of packages with a vulnerable dependency also influence what portion of fixes can be installed. By relying only on patch updates, dependents miss out on 30% (on average) of vulnerability fixes.

RQ1: How does dependency management impact the risk of vulnerabilities for downstream packages? Developers need a means to identify and select responsive packages to mitigate the risk of vulnerabilities from transitive dependencies. The high ROC-AUC score of 0.85 for our classification model shows that package attributes are useful indicators for differentiating between fast-responder and slow-responder packages. We found that non-restrictive dependency update strategy have a shorter exposure to vulnerabilities. Package age, and release frequency are also good predictors of how quickly packages adopt a vulnerability fix.

RQ3: How do developers perceive dependency practices for vulnerability mitigation? In order to gauge the applicability of our results in practice and understand the perception of practitioners on our findings, we design a survey and disseminate it among 67 practitioners from industry and academia. Many practitioners are unaware that the lack of updates from downstream dependents is the key culprit for prolonged exposure to public vulnerabilities. In regards to the package attributes that indicate a faster adoption of vulnerability fixes, the experience of practitioners is generally aligned with our results. Survey participants are in favor of incorporating our findings into their dependency management practices.

This chapter is structured as follows. Section 5.2 explains the methodology and dataset used to conduct the study. We present the findings of our research in Section 5.3 and discuss the practical and research implications of our results in Section 5.4. In Section 5.5, we summarize the key related works. Section 5.6 presents the threats to validity. We conclude

our study in Section 5.7.

5.2 Data and Methodology

Our objective is to identify and understand the characteristics of responsive packages. Consequently, we require data regarding vulnerable packages in npm. We also need information about the attributes of downstream packages that depend on these vulnerable packages. We combine the data provided by the npm security advisories and package and dependency metadata for the npm ecosystem to curate a dataset of vulnerable dependency relationships in npm. We then extract the relevant features from the downstream dependents that install (and later fix) a vulnerable version of the upstream dependency to prepare our dataset for training a machine learning model. A complete replication package for this study is available on Zenodo (Javan Jafari, Elias Costa, Abdellatif, & Shihab, 2023).

5.2.1 Vulnerable dependency dataset

In order to analyze the package attributes that indicate a faster adoption of vulnerability fixes, we need first to identify packages that install a vulnerable dependency. We gather the dataset of reviewed npm security advisories from GitHub (GitHub, 2023b). The vulnerability advisories include the vulnerability name, published date, severity, CWE classification (Mitre, 2023b), CVE identifier (Mitre, 2023a), affected versions, fixed versions, along with a description of the vulnerability. We also use the latest available version of the libraries.io dataset (Libraries.io, 2023) to identify dependency relationships in the npm ecosystem so we can extract packages that install a vulnerable dependency. This dataset has been used and validated in previous research (Decan et al., 2019; Jafari, Costa, Shihab, & Abdalkareem, 2023). The dataset contains metadata for 1,275,082 unique packages, 11,400,714 package versions and 154,914,774 dependency relationships for the npm ecosystem. We

then cross-reference both datasets and select the packages that have a vulnerable version in our npm dependency dataset, resulting in 118, 253, 160 and 44 critical, high, medium and low severity vulnerabilities, respectively. We maintain 4 separate datasets (one for each vulnerability severity type) to facilitate detailed analyses.

For each vulnerability, we first collect the entire list of its downstream dependents from the dataset to identify the packages that are 'potentially vulnerable' to the upstream vulnerability. We only consider runtime dependency relationships from the downstream dependents to upstream packages because they are required for the package to properly function and should be complete (missing runtime dependencies are considered a bad practice (Jafari et al., 2021)). This first step amounts to more than 1,940,000, 4,545,000, 4,770,000 and 2,320,000 potentially vulnerable dependencies for critical, high, medium and low severity vulnerabilities, respectively. We then evaluate each of the mentioned dependency relationships to determine whether the downstream dependency actually installs a vulnerable version of the upstream package (i.e. downstream dependent installs a vulnerable version). Similarly, we determine if the downstream package installs a version including the fix or any version higher than that (i.e. downstream dependent installs a non-vulnerable version). Since disclosed vulnerabilities often affect all previous releases of a package (Zerouali et al., 2022), the most reliable way to mitigate vulnerabilities is to update to a fixed version.

Many packages specify flexible version ranges for their dependencies which may install a different upstream version depending on the evaluation time frame (i.e. constraint of $>$1.2 may install version 1.5 or version 2.0 depending on when the constraint is evaluated). In order to verify if a package installs a vulnerable version, we evaluate the current downstream constraint the day before the fix could be adopted. If no downstream dependent in our dataset installs the vulnerable upstream version, the upstream package are removed from our analysis. In order to verify if the downstream package installs a fix, we evaluate the current downstream constraint at the time the fix was released by the upstream package

Table 5.1: The dataset for our study

Dataset Attribute	Critical Severity	High Severity	Medium Severity	Low Severity	Total
Vulnerabilities	59	213	146	40	**458**
Vulnerable Dependencies	113,068	165,311	151,471	79,746	**509,596**
Vulnerable Dependents	104,149	134,329	126,087	75,432	**201,027**

and every subsequent date the downstream releases a new version and measure the first time the fixed version (or any higher version) of the upstream package is installed. Table 5.1 presents the detailed statistics of the final dataset used in our study. Note that total vulnerable dependents are less than the sum of vulnerable dependents for each severity type since a dependent package can appear as a vulnerable dependent in multiple severity type groups.

5.2.2 Package feature extraction

We aim to identify and collect features in downstream packages that are relevant for responding to vulnerable dependencies. The features will be used to train a model to predict the speed of response to a vulnerable dependency fix. We aim to consider dimensions regarding package popularity, maturity & stability, activity and dependency management.

Popularity: We use *Dependent Count* as our popularity metric as it is the de facto measure of the number of unique downstream clients for a package. We hypothesize that highly used packages may take greater care in quickly addressing vulnerable dependencies as their vulnerable dependencies can transitively impact a larger number of downstream packages and consequently, invite greater repercussions. Multiple studies highlight the number of downstream dependents as a metric used by developers to aid in their dependency selection (Bogart et al., 2016; Haenni et al., 2013; Larios Vargas et al., 2020; Pashchenko et al., 2020).

Maturity & Stability: We use *Age* and *Release Status* as our indicators for package

maturity & stability. Packages with a longer history provide practitioners with more reliable information regarding the responsiveness of the package to security vulnerabilities. On the other hand, older projects are less likely to still be well maintained (software also rots after all). Developers have cited package maturity as a criterion for selecting dependencies (Larios Vargas et al., 2020). The release status determines whether a package has released their first 1.0.0 version or if they are still in their initial development releases (e.g. v0.2.3). SemVer considers pre-1.0.0 releases unstable by nature and we believe their use is a reliable means to gauge the maintainers' perception regarding the maturity of their own package. Release status has previously been referenced by developers as a metric in selecting packages (Bogart et al., 2016).

Activity: We use the *Release Frequency* as our metric for measuring activity. We hypothesize that packages with a more active development and maintenance schedule are more proactive in responding to vulnerable dependencies. If a package rarely releases a new version, they need to rely on automatic dependency updates as their means of installing new fixes for vulnerable dependencies. However, a package that limits automatic dependency updates can still install the fix manually if they release a new version of their package with a modified configuration file. Release frequency normalizes the number of total releases by the age of a package and it is an indicator used by developers when selecting dependencies (Bogart et al., 2016; Haenni et al., 2013; Larios Vargas et al., 2020).

Dependency Management: Since we want to identify features that best predict the responsiveness to vulnerable dependencies, we need to consider a feature group that focuses on the dependency management of packages. We hypothesize the dependency-related decisions made by a potential direct dependency has a considerable influence on our exposure to transitive dependencies. We use *Dependency Count*, *Dependency Modifications* and *Dependency Update Strategy* as our dependency management features. *Dependency Count* is a measure of the number of dependencies, and it is considered by developers to aid

in dependency selection (Larios Vargas et al., 2020; Mujahid, Abdalkareem, & Shihab, 2023). We hypothesize that a larger number of dependencies may make it difficult to properly manage and keep track of vulnerable dependencies. *Dependency Modifications* is a measure of dependency change and measures how many releases modified the dependency configuration. Changing a dependency configuration file (package.json) more frequently can be a sign of continuous upkeep to maintain dependency health or it could be a sign of frequent dependency problems. When viewed alongside release frequency, the number of dependency modifications highlights whether an active package with frequent releases also frequently modifies their dependencies.

The *Dependency Update Strategy* encapsulates the dependency constraints used by the npm package manager to determine acceptable versions of each package dependency (Decan & Mens, 2019a; Jafari, Costa, Shihab, & Abdalkareem, 2023). These constraints specify the degree of freedom given to the package manager to automatically install new versions of a dependency and it is an important part of the dependency management practices in each project. We hypothesize that limiting automatic updates for dependencies increases the likelihood of depending on a vulnerable version of a dependency that has already released a fix. In the following, we elaborate on the specific definition of our three dependency update strategies. Figure 5.1 presents the distribution of dependency update strategies in our dataset.

- **Balanced Update Strategy:** This update strategy encompasses dependency constraints that allow the package manager to automatically install new minor and patch releases for post-1.0.0 package dependencies and prevent automatic updates for pre-1.0.0 package dependencies (because SemVer considers pre-1.0.0 releases to have an unstable API (Preston-Werner, 2019). A common practice to use a balanced update strategy in npm is to use the caret symbol as a dependency constraint (e.g. ^1.2.3).

- **Restrictive Update Strategy:** This update strategy encompasses dependency constraints that either prevent automatic updates entirely or restrict the package manager to only install new patch updates for post-1.0.0 package dependencies. The tilde notation is commonly used in npm to specify restrictive update strategies (e.g. $\sim$1.2.3). We do not define restrictive strategies for pre-1.0.0 package dependencies as any automatic updates for such packages is considered permissive.

- **Permissive Update Strategy:** This update strategy encompasses dependency constraints that allow the package manager to automatically install all new releases (including major releases) for post-1.0.0 package dependencies and any dependency constraint that allows updates of any kind for pre-1.0.0 package dependencies. The npm ecosystem allows the use of wildcards (e.g. *) as a dependency constraint but one can also use >= (e.g. >=1.2.3).

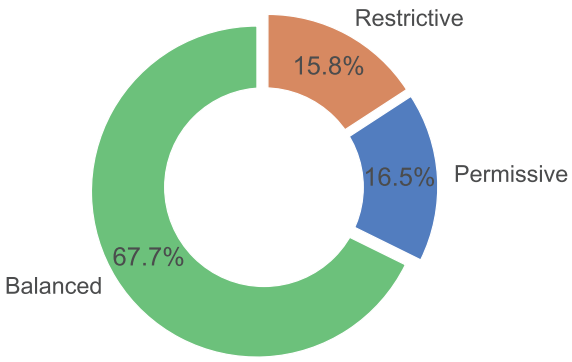


Figure 5.1: Distribution of the update strategy feature

In order to curate an accurate depiction of the relationship between features and vulnerability fixes, we need to calculate downstream package features at the time a vulnerability was fixed in an upstream package (e.g. we are interested in the release frequency of the downstream package at the time the upstream package released the fix, not many years after). Since the same npm package can depend on multiple vulnerable dependencies at

Table 5.2: Selected features for downstream packages

Feature	Description	Histogram
Package Age	The number of days since the dependent package was published.	
Balanced Update Strategy	Whether or not the dependent used the balanced update strategy.	
Restrictive Update Strategy	Whether or not the dependent used the restrictive update strategy.	
Permissive Update Strategy	Whether or not the dependent used the permissive update strategy.	
Release Frequency	The average number of dependent version releases per month.	
Dependency Count	The total number of dependencies for the dependent package.	
Dependent Count	The total number of dependents for the dependent package.	
Release Status	Whether the package was in a pre-1.0.0 or post-1.0.0 release state.	
Dependency Modifications	The number of releases in which the dependencies were modified.	

different times, the features have to be calculated separately every time.

The dependency update strategy is a categorical feature with no ordered relationship and we must use one-hot encoding to encode each class into a binary (0,1) feature. Otherwise, the model may assume a natural order between the update strategies, which might result in poor performance or irrational results. We used the pandas library to retrieve one-hot encoded values for the update strategies (Pandas, 2023).

Introducing highly correlated features while training a machine learning model can impact both its performance and its interpretability. Due to the skewed distribution of our features, we use Spearman's correlation (Hollander, Wolfe, & Chicken, 2013) to identify and remove features with a correlation score of above 0.7. In such cases, we kept the feature which we believed to have a more tangible definition. We dropped *Package Version Count* in favor of *Dependency Modifications* and *Days Since Last Release* in favor of *Package Age*. The final set of features for our study is presented in Table 5.2. It is worth noting that some of the collected features represent the characteristics of a selected dependency (e.g. Age and Dependent Count) while others reflect their behavior (e.g. Update Strategy and Release Frequency).

5.3 Results

In this section, we motivate our 3 research questions, describe our approach, and present our findings.

5.3.1 RQ1: How does dependency management impact the risk of vulnerabilities for downstream packages?

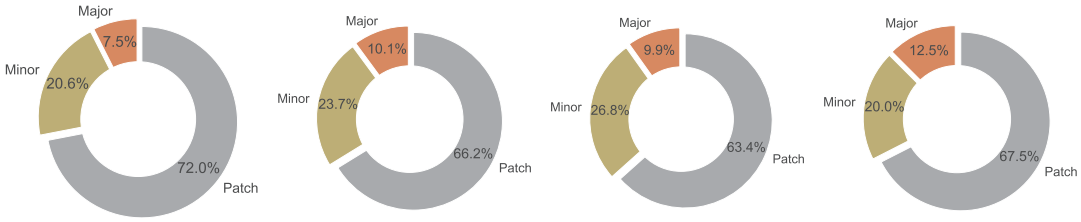
Motivation: Security vulnerabilities rooted in dependency packages are a major risk to software (Sonatype, 2021; Zimmermann et al., 2019). Packages play a key role in mitigating vulnerabilities in their dependencies. We want to understand the role of dependency management decisions of packages in their exposure to vulnerabilities from upstream packages. While previous research has discussed how long it takes for a npm dependency to remain vulnerable (Alfadel et al., 2023; Chinthanet et al., 2021; Decan et al., 2018b), they have not studied the relationship between vulnerabilities and the dependency management decisions of downstream dependents.

Approach: A vulnerability fix can be released as a major, minor or patch version (Section 2). Since we have the information for package versions and vulnerability metadata, we can compare the version number of the fixing release (r) with the version number of the release right before the fix (r-1) to evaluate the type of fixing release. (Figure 5.2). We use the npm SemVer package (npmjs, 2023) to evaluate the difference between these two packages. For this analysis, we exclude uncommon release types such as pre-major and pre-minor and focus on the main types of releases according to semantic versioning guidelines (Preston-Werner, 2019) (i.e. major, minor and patch). Since the adoption of different release types is determined by the dependency decisions of downstream dependents, comparing the proportion of release types for vulnerability fixes shows how downstream dependency management decisions can influence the proportion of vulnerability mitigation.

To evaluate the speed of vulnerability mitigation, we measure how long it takes for the upstream packages to fix the vulnerability after public disclosure **(i.e. fix delay)** and how long it takes for the downstream dependents to adopt the fix after the fix is released **(i.e. adoption delay)**. Since the adoption of a fix is the responsibility of downstream dependents (through dependency management decisions), comparing the fix delay and adoption delay shows how much downstream dependency management decisions influence the total mitigation time.

The fix delay is measured by comparing the public exposure date and the fix release date which are both provided by the vulnerability advisory. In order to measure the adoption delay, we determine which dependents actually install a vulnerable version of the upstream vulnerable package and later adopt the fix (Section 5.2). We then look at the release date of the fix and compare it with the first release of the dependent that accepts the upstream fix. In addition, 74,909 dependent packages in our dataset never adopt a fix, so we measure the adoption delay from when the fix was published to the most recent date in the dataset (12 Jan 2020). This is a lower bound for the adoption delay as the dependents can opt to receive a fix at a later date.

Figure 5.3 presents the distributions for the fix and adoption delays for different vulnerability severities. We use the Mann-Whitney U test to measure the statistical significance of the difference between the distributions (Mann & Whitney, 1947). The distributions should not be biased towards popular vulnerabilities that impact many dependents or the decisions of packages that have many vulnerable dependencies. In order to prevent a bias towards vulnerabilities that affect many dependents, the distributions of fix delays are calculated for **unique vulnerabilities**. Similarly, to prevent a bias towards downstream packages that are affected by many vulnerabilities, the distributions of adoption delays are calculated for **unique dependent packages** (i.e. when a package has multiple vulnerable dependencies, we consider the most recent vulnerable dependency).



(a) Critical (b) High (c) Medium (d) Low

Figure 5.2: Release type for the vulnerability fix (separated by vulnerability severity)

Findings: The release types of the vulnerability fixes for each severity class are depicted in Figure 5.2. The majority of vulnerability fixes are released as a patch update, which is expected as these types of releases are meant for fixing bugs and vulnerabilities. Yet, a sizeable portion (more than 30% on average) of vulnerability fixes are released in minor and major release types, which aligns with the findings of previous research (Chinthanet et al., 2021; Zerouali et al., 2022). This creates a problem for downstream dependents because previous research has shown that even though dependents frequently use the non-latest version groups, back-porting fixes to previous version groups is an infrequent practice (Decan, Mens, Zerouali, & De Roover, 2021). Consequently, downstream dependents that refuse to accept all updates will not receive all vulnerability fixes. In other words, downstream dependency practices impact what proportion of vulnerability fixes can be adopted. The solution is not as straightforward as "accepting all updates", as new *major* versions will (by definition) contain backward incompatible changes that can break the downstream package. Even new minor or patch releases may introduce such breaking changes (Cogo et al., 2019; Mezzetti, Møller, & Torp, 2018).

Finding 1: Since vulnerability fixes are packaged in different release types, downstream dependency management decisions that limit updates to specific release types (e.g. patches) also limit their capacity of receiving the fixes.

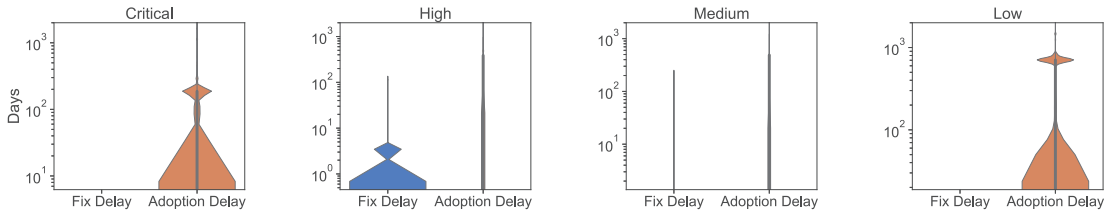


Figure 5.3: Comparing the distributions of upstream fix delay with downstream adoption delay for different vulnerability severities (statistically significant difference with $p<0.05$).

The more important finding is the relationship between downstream dependency management decisions and the adoption delay of vulnerability fixes. The violin plots in Figure 5.3 provide insights into how long npm packages remain vulnerable. For each severity type, we have plotted the distribution for the fix delay and the adoption delay. While the median for all delays is zero (meaning many publicly disclosed vulnerabilities are fixed within one day and adopted by the dependents on the same day), the distribution of the delays clearly show that adoption delay is usually much longer than the fix delay. The difference between two distributions are statistically significant (for all severities) using the Mann-Whitney U test with $p<0.05$. The mean time to release a fix is 1.17 days, while the mean time to adopt a fix is 192.5 days. While only 4 vulnerabilities take more than a day to release a fix (after disclosure), 73,155 dependent packages need more than a day to adopt an upstream fix.

The minuscule delay to release a fix is due to the vulnerability disclosure policies that recommend the initial vulnerability report to be made privately to package maintainers so they can release a fix before publicly disclosing the vulnerability (GitHub, 2023a). In fact, apart from a few outliers, almost all publicly disclosed vulnerabilities have a vulnerability-fixing release on the same day. This is not the case for the adoption of vulnerability fixes. It is the delay in downstream fix adoption, not the delay in the upstream fix release, that is keeping dependent packages vulnerable. What makes matters worse, is that the post-disclosure vulnerability risk period is more dangerous than the pre-disclosure period because the details of the vulnerability and the means to exploit it are now made public. A

crucial factor in determining how fast publicly disclosed vulnerabilities are mitigated across the ecosystem is for downstream dependents to adopt appropriate dependency management practices.

One interesting observation in Figure 5.3 is that the delay in fix adoption corresponds intuitively and consistently to the severity of the vulnerability. The lower the severity of a vulnerability, the higher the delay in adopting the fix. This hints that developers are aware of vulnerability severities and use the information to prioritize the adoption of fixes. As previous research has shown, developers often do not update their dependencies due to the necessary extra effort (Kula, German, et al., 2018a). The built-in *npm audit* command scans package dependencies for known vulnerabilities and reports the list and severity of public vulnerabilities (npm Docs, 2023). A similar functionality is provided by the popular Dependabot dependency management tool (GitHub Docs, 2023).

Finding 2: Since the majority of packages with a vulnerability release a fix within a day of public disclosure, downstream dependency management decisions that delay fix adoption are the key culprit for prolonged exposure to such vulnerabilities.

5.3.2 RQ2: How can we identify packages that quickly mitigate vulnerabilities?

Motivation: In our first research question, we find that dependency decisions impact the responsiveness of packages to adopting a vulnerability fix. We need to identify the attributes of responsive dependency packages. By identifying responsive packages, we can help developers mitigate the risk of vulnerable dependencies. We wish to use our selected package attributes to model the adoption delay of vulnerability fixes. Our results will help developers better understand how opting for different dependency practices (e.g. choosing a package with a higher release frequency) can increase or decrease the adoption speed of

vulnerability fix propagation from upstream packages.

Approach: We use our set of 9 package attributes (Section 5.2) to train a random forest model to predict the adoption delay of the fix. We use random forest because it is known to offer a good balance between performance and interpretability and is commonly used in software engineering research (Dey & Mockus, 2020; Ohm, Boes, Bungartz, & Meier, 2022). When feeding the vulnerability instances to the model, we use the latest vulnerable dependency relationship for each downstream dependent. We do this as we do not want non-unique downstream packages across our training and test data and we do not want to bias the data towards downstream packages with a high number of dependencies. For example, package X may depend on a vulnerable package in 2016 and another vulnerable package in 2019. Since each dependency relationship is an instance used to train the model, relationships involving package X may appear in both training and test sets. While the features for package X are calculated separately per dependency relationship, a hidden feature (e.g. cultural habits) may remain consistent for both relationships, causing a data leak from the training set to the test set. Additionally, package X may have many more vulnerable dependencies than another downstream package, which would bias the model results to the features of package X due to increased presence in the training set. This amounts to 201,027 vulnerable dependency relationships in our data. Since no previous work has used dependency practices to model adoption delay, we use a stratified predictor (random prediction based on class weight) as our baseline (Scikit, 2023). Both our random forest and our baseline model are trained and tuned on 80% of our dataset (training and validation set) and evaluated on the held-out 20% (test set) (Géron, 2019). We use ROC-AUC and F1-score to evaluate our models (Tharwat, 2020). The ROC metric (Receiver Operating Characteristics) depicts a probability curve and the AUC (Area Under the Curve) is a value in the range of 0 and 1 that shows the capability of the model in distinguishing between

classes. Higher ROC-AUC means the model is better at correctly predicting classes. F1-score is a function in the range of 0 and 1 that measures the balance between precision (the portion of true positive cases among all the retrieved cases) and recall (the portion of true positive cases that were retrieved).

We used a grid search with a 10-fold cross validation on the training set to tune the hyper-parameters of our model. This results in 1000 estimators (trees) with a minimum sample split of 8. The 10-fold cross validation fits the model 10 times, where each fit is performed on 90% of the training set (randomly selected) and the remaining 10% is used as a validation set.

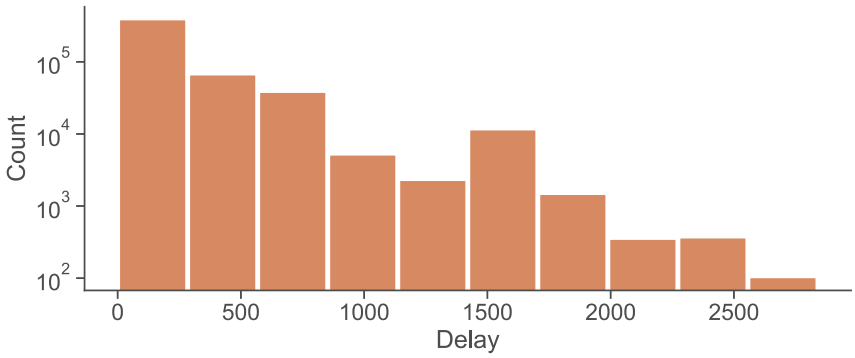


Figure 5.4: Distribution of adoption delay (days)

We are interested in predicting a range of classes of adoption delay (e.g. fast, slow), instead of the actual number of days because the adoption delay ranges from 0 to 2841 days and the delay is not equally distributed along the range. Figure 5.4 presents the distribution of delay in 10 bins. As can be seen, the majority of vulnerability fixes are adopted in a short time (note the log scale on the Y-axis of Figure 5.4).

We train a binary classifier and use a threshold of 48 hours for identifying the fast-responder class. Since it is not intuitive to classify anything below 48 hours as fast and anything slightly above it (e.g. 50 hours) as slow, we use a disjoint threshold of below 48 hours for the fast class and above 14 days for the slow class to better separate the classes. This removes 546 dependents between the two thresholds (that adopted the fix later than

48 hours and sooner than 14 days) from our dataset. We conducted a sensitivity analysis to analyze our binary class threshold and ensure minor changes do not significantly impact the target class distribution (see Section 5.6).

Since not all dependency practices have the same impact in predicting the adoption delay, we need to identify, rank and analyze the important features of the model (Figure 5.7). We calculate the permutation feature importance on the test set in which each feature is randomly shuffled (repeated 10 times) to observe its impact on the model's performance (ROC-AUC). Important features have a larger influence on the model's performance when their values are permuted (Scikit-learn, 2020).

We use Partial Dependence Plots (PDP) to visualize the impact of the important dependency practices on the predicted adoption delay (Figure 5.8). PDPs depict the marginal effect of a feature on the model's predictions (Molnar, 2020) and they can highlight linear, monotone or more complex relationships between the dependency practices and the adoption delay. In other words, we can visualize how a change in a feature can change the adoption delay predicted by the model. The Y-axis on the PDPs in Figure 5.8 is the predicted probability for an instance being predicted as a fast response. The tick marks on the X-axis are the deciles for the feature distribution which indicate which part of the plots represent the majority of our dataset. Individual Conditional Expectation (ICE) plots also show the effect of a feature on the target variable. However, unlike PDP which average the effect over all instances, ICE plots visualize the relationship for a single instance (Goldstein, Kapelner, Bleich, & Pitkin, 2015). We have shown a random sample of 20 ICE plots in Figure 5.8 for each of the top 5 features (depicted using the light-colored lines).

Findings: The evaluation results in Figure 5.6 present the ROC-AUC, F1 score, Precision and Recall for our dependency practices model compared with the stratified baseline. Our dependency practices model achieves an ROC-AUC of 0.85, which is a 70% improvement

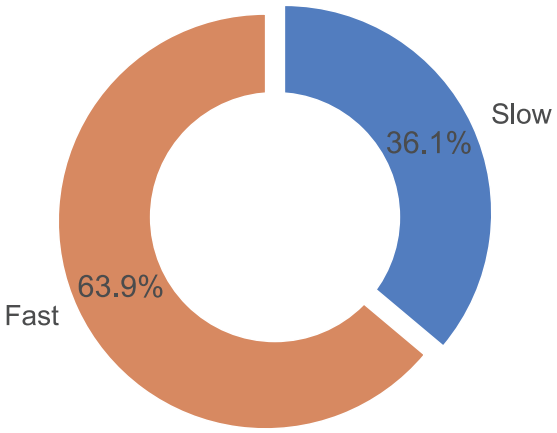


Figure 5.5: Distribution of model classes

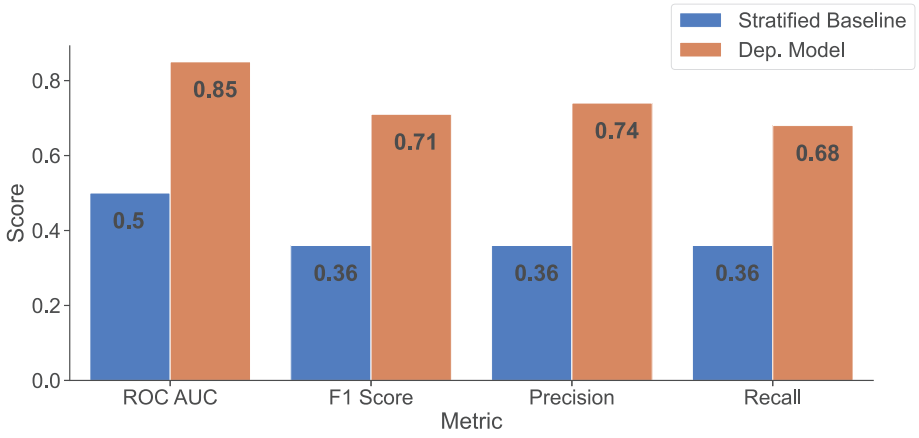


Figure 5.6: Performance evaluation results for the dependency practices model

over the baseline. Our model also achieves an F1-score of 0.71, which is a 97% improvement compared to the baseline. These results indicate that package features can be used to model and predict the adoption delay of vulnerability fixes in the npm ecosystem.

We also train and evaluate the model on four subsets of the data in Table 5.1, which separates the dataset based on vulnerability severity to see if there is a considerable difference between our results when we focus on critical, high, medium and low severity vulnerabilities. As can be seen in the evaluation breakdown of Table 5.3, our data subset models perform at least on par with the main model. In some cases, we observe even stronger

Table 5.3: Performance evaluation of alternative models on severity-specific subsets of the data

Subset model	Model ROC-AUC	Baseline ROC-AUC	Increase	Model F1-score	Baseline F1-score	Increase
Critical	0.92	0.5	84%	0.82	0.35	134%
High	0.85	0.5	70%	0.73	0.38	92%
Medium	0.91	0.5	82%	0.80	0.34	135%
Low	0.92	0.5	84%	0.83	0.41	102%

performance results (e.g. ROC-AUC of 0.92 for critical vulnerabilities).

Finding 1: Practitioners can use the attributes of their dependency packages to influence the adoption of upstream vulnerability fixes.

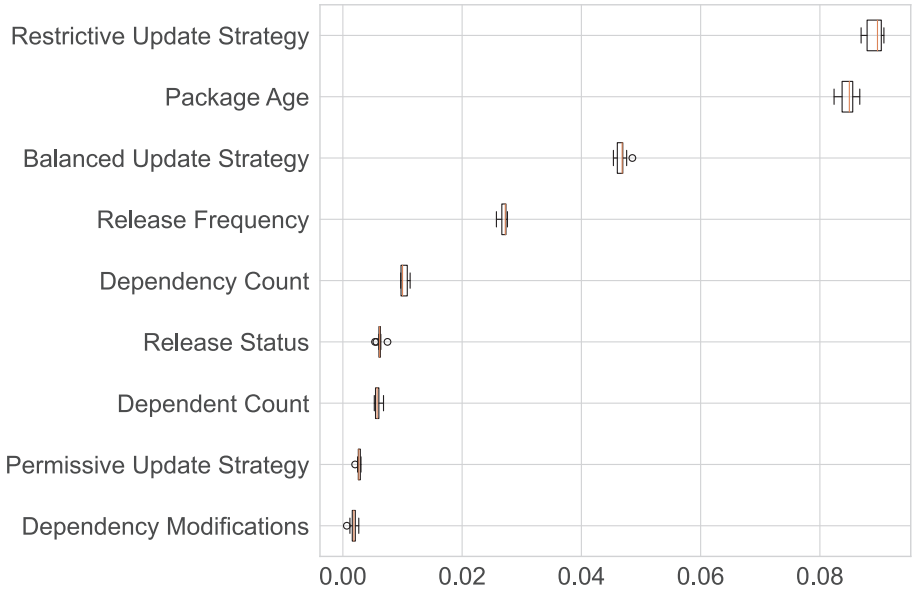


Figure 5.7: Ranking the features of the model based on permutation importance

We have ranked the dependency practices based on their permutation importance in the box plots of Figure 5.7. The use of a restrictive update strategy, the age of the package, the use of a balanced update strategy, the release frequency of the package and the number of dependencies for a package are the most important indicators for determining the responsiveness of an npm package to a vulnerability fix. Developers should incorporate these characteristics in their dependency management and selection practices.

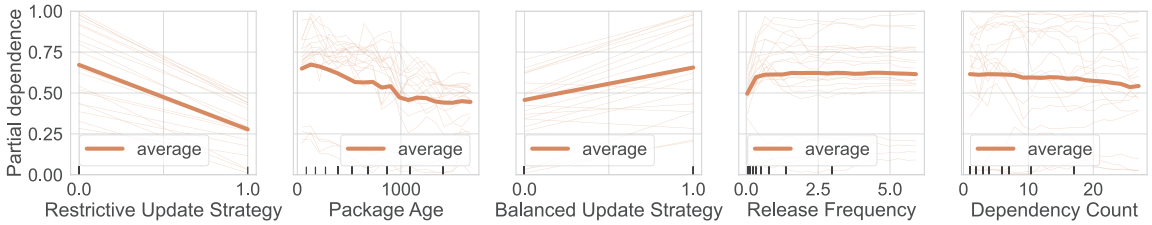


Figure 5.8: Partial Dependence Plots and Individual Conditional Expectations for the top 5 dependency practices

While feature importance tells us which dependency practices are instrumental for the model, the PDPs in Figure 5.8 tell us how the dependency practices impact the prediction results. Specifically, we want to analyze how a change in the top 5 dependency practices increase or decrease the likelihood of a package being classified as a fast responder to vulnerabilities. We can see that packages with restrictive update strategies take longer to adopt vulnerability fixes. Consequently, their increased exposure to vulnerabilities further exposes their downstream dependents (transitive exposure). While this observation aligns with our intuitive understanding of restrictive updates, it is not always the expected outcome. Developers that use restrictive update strategies can always decide to manually maintain their dependencies and keep track of important updates, but in a generalized sense, they fall short of the latest vulnerability fixes; especially compared to developers who opt for a more balanced update strategy.

Finding 2: Downstream dependents of packages with a non-restrictive dependency update strategy tend to have shorter exposure to security vulnerabilities.

A less intuitive finding is the impact of package age on the responsiveness to vulnerabilities. Older packages in our dataset are predicted less likely to quickly adopt vulnerability fixes. We believe this is due to the impact of old and unmaintained packages, since a well-maintained mature package (while seemingly rare) can actually be a more reliable dependency choice. This hypothesis is reinforced by the PDPs for release frequency. Packages that release more often are more likely to be predicted as a fast responder. It is also

worth noting the initial feature of "Days Since Last Release" was removed due to a high correlation with "Package Age", indicating that older packages are more likely to not have a recent release.

Finding 3: Downstream dependents of younger packages tend to have shorter exposure to security vulnerabilities.

A higher release frequency of a package initially increases the likelihood of a fast response to vulnerabilities. However, once the package has a new release every 4 months, a further increase in release frequency does not seem to make a difference. There are two potential explanations for this observation. First, a non-zero release frequency of around 0.25 *per month* is already enough to differentiate between regularly maintained and abandoned packages and further increase in the release frequency may not provide further insight into adequate maintenance. The second explanation for this observation is the tick marks on the X-axis, which represent the distribution deciles for the feature. As can be seen, the majority of the distribution is collected in the left side of the PDP, indicating that the initial positive slope is in fact a better representation of the overall effect of release frequency on the likelihood of a fast response.

Finding 4: Downstream dependents of packages with a faster release cycle tend to have shorter exposure to security vulnerabilities.

Packages with a higher number of dependencies are less likely to be predicted as a fast responder. Having more dependencies can make it more difficult to keep track of (and respond to) vulnerable dependencies in a timely manner, especially if relying on manual updates. The PDP slope in Figure 5.8 only indicates a modest effect by dependency count that does not influence the model's predictions as strongly as the other top features. It is also worth noting that our findings are focused on the delay of adopting vulnerability fixes, but having more dependencies can also increase the threat surface for vulnerable dependencies.

Finding 5: Downstream dependents of packages with less dependencies tend to have shorter exposure to security vulnerabilities.

5.3.3 RQ3: How do developers perceive dependency practices for vulnerability mitigation?

Motivation: The key objective of this study is to assist developers in identifying responsive packages to reduce the risk of vulnerability exposure from dependencies. Therefore, we need to understand the perception of developers regarding our findings. Are developers aware of the impact of downstream dependency practices on vulnerability mitigation? Do our findings reinforce or contradict their experiences? Will they use our findings to complement their dependency selection process? Answering these questions will help us understand the applicability of our findings in practice.

Approach: To understand the perception of developers on the use of dependency management practices for mitigating vulnerabilities, we crafted a survey that aims to compare the findings of RQ1 and RQ2 with the real-world dependency management experience of practitioners.

The survey is composed of 4 parts. In the first part, we ask respondents about their background and experience. In part 2, we ask the participants about their opinion (based on their experience) on the relative effect of publishing the fix compared to adopting the fix as the main reason for exposure to publicly disclosed vulnerabilities (RQ1). In part 3, we ask the participants about the impact of the features used for our model (RQ2) on the responsiveness to vulnerability fixes. In the final part, we provide respondents with the initial findings of our study and ask if they would incorporate our findings in their dependency selection and management practices in the future.

All of the questions included an ”other” option which allowed developers to provide an answer not already included in the choices or to expand on their answer. We also asked

Table 5.4: Background of participants in the survey.

Dimension	Responses	%
Background	Industry Practitioner	59.7%
	Academic Researcher	29.8%
	Both	7.5%
	Student	3%
Development Experience	>= 7 years	38.8%
	4-6 years	28.4%
	1-3 years	31.3%
	< 1 year	1.5%
Total participants		**67**

respondents if they wish to suggest additional package attributes that affect the response speed to vulnerable dependencies. Some of these free-form responses are exclusively referenced in Section 5.4. The complete set of questions and response choices are included in our replication package (Javan Jafari et al., 2023).

In order to recruit the participants for the survey, the authors distributed it among their existing network of industry developers and academic researchers (mostly in Canada) which are actively working in the industry or academia. We contacted 114 practitioners and received a total of 67 responses. As Table 5.4 shows, the respondents are primarily composed of industry practitioners (59.7%), followed by academic researchers (29.8%). The majority of our participants (67.2%) have 4 years or more experience in software development.

Findings: In the following, we will present the findings of our practitioner survey. We have also included sample responses from the participants. We empirically discovered (RQ1) that it is the adoption delay, not the fix delay, that is the main contributing factor for the survival of publicly disclosed vulnerabilities in the npm dependency ecosystem. This aligns with the experience of a considerable portion (46.3%) of our respondents. However, 40.3%

of the respondents believe that both the fix delay and adoption delay are equally responsible for exposure to publicly disclosed vulnerabilities.Surprisingly, 10.4% of respondents believe the fix delay to be the main contributing factor. Respondents also highlighted the criticality of the downstream dependent in the outcome.

> *"The projects deployed in industry are usually relatively large and extensive. [...] nobody dares to manipulate dependencies as any change may stop the system's regular functionality" - R12*

Indeed, updating dependencies is a more sensitive decision when backward compatibility is crucial.

> *"It depends on the project. For some projects, the security level is critical, so if there is any known vulnerability, it can be more option one [fix delay], and for less critical project, more option 2 [adoption delay]." - R58*

In other words, a security critical downstream client is less likely to be exposed to a vulnerability through their own fault, but rather because the fix is not yet released.

Finding 1: The majority of the practitioners do not believe that the delay in downstream fix adoption is more responsible than the delay in the upstream fix release in keeping dependent packages vulnerable.

The next section of the survey focuses on the important package features for predicting the adoption delay (RQ2). The responses align with our findings for RQ2, both for the importance of the features and for the positive/negative relationship between the features and the response delay. When asked about the relationship between package age and the speed of handling vulnerable dependencies, 46.3% of practitioners believed older packages to be slower in addressing vulnerable dependencies, which aligns with our finding in RQ2. Only 10.4% believed older packages to be faster in addressing vulnerable dependencies. 37.3% of the practitioners selected the "Neither" option. Respondents also highlighted the

distinction between old active packages and old unmaintained packages.

> *"[...] I believe packages that see active development are more likely to address vulnerable dependencies" - R20*

> *"If its really old then maybe its kinda abandoned but if its too young then it might be immature" - R39*

When asked about the relationship between release frequency and how fast vulnerable dependencies are addressed, 85.1% of practitioners believed packages that release more often are faster in handling vulnerable dependencies. Respondents also highlighted the importance of the reason behind a frequent release cycle.

> *"packages in early development release more often. Not to fix vulnerabilities but to change features. After that initial period release frequency might be more related to vulnerabilities" - R66*

> *"[...] a healthy release cycle is good but too much may just be a sign of bad versioning practice by the devs" - R39*

Our findings in RQ2 show that release frequency does have a positive relationship with a fast response to vulnerabilities, but only to an extent. We observed no further improvement after a release frequency of around 0.25 per month. 11.9% of practitioners selected the "Neither" option.

We previously found in RQ2 that higher dependency count in our model decreases the likelihood of a fast response to vulnerabilities. In the survey, 64.2% of the practitioners also believe packages with a high number of dependencies are slower in addressing vulnerabilities. 19.4% believe the opposite is true and 11.9% believe there is no difference either way. Practitioners highlighted that the number of dependencies should be viewed in conjunction with project size and policies.

> *"If the dev team is small then more dependencies will make it harder to keep track so it might increase risk but a large dev team should maintain good dependency health anyway" - R66*

> *"Depends on project/org policies" - R16*

This may explain the reason we observed a *weak* relationship between dependency count and the response to vulnerabilities in RQ2.

We asked practitioners to rank the various dependency update strategies based on which update strategies in a package lead to faster handling of vulnerable dependencies with a publicly disclosed vulnerability. Among the respondents, 58.2% agree that the balanced update strategy leads to a faster response to vulnerable dependencies whereas a restrictive update strategy leads to a slower handling of such vulnerable dependencies. Going further than a balanced update strategy and adopting a more permissive strategy allows for even more updates but increases the risk of breaking changes.

> *"The best practice is to allow automatic updates for new patch and minor versions [i.e. balanced strategy]. For major changes, it is not technically possible to allow automatic updates since the major updates [i.e. permissive strategy] include breaking changes for other dependencies [...]" - R38*

Indeed, developers are aware that fixes are not always released in (or backported to) patch releases, but they are wary of the trade-offs (between receiving all fixes and breaking changes) for allowing too much freedom in automatic updates.

> *"believe it or not, some fixes are in major updates! but its usually not a good idea to update all the time like that" - R39*

Even though we consider both patch only updates and no updates as restrictive update strategies, we gave respondents the option to rank these two approaches separately. 86.6% of respondents ranked patch only updates as a better approach than no automatic updates.

Finding 2: The majority of practitioners agree that younger packages with frequent releases that adopt a balanced update strategy and have fewer dependencies are faster in addressing vulnerable dependencies.

Even though the Update Strategy, Package Age, Release Frequency and Dependency Count are the important features for our model (Figure 5.7), we asked practitioners about all the features in our study. When asked about the number of modifications to dependency configuration, 50.7% of practitioners believe more frequent modifications of the dependency configuration file is associated with a faster response to vulnerable dependencies, while 32.8% do not believe this feature to be relevant to assess a packages response to vulnerable dependencies.

> *"Depends. touching things too much could be a sign of diligence or an inexperienced dev" - R66*

For dependent count, 44.8% of respondents believe packages with higher dependent counts to be faster in addressing vulnerable dependencies whereas 26.9% believe they would be slower to react. Having a large community can motivate the package maintainers to be more diligent since vulnerabilities from their dependencies can propagate to their large client base.

> *"Usually yes because they get bombarded from their community if they don't [...]" - R66*

In regards to release status, 37.3% believe post-1.0.0 packages to be faster in addressing vulnerable dependencies while 38.8% do not believe this feature to be relevant to how fast packages respond to vulnerable dependencies.

> *"Some packages just enjoy staying in pre-1.0.0 (god knows why!) but they have a large following so they are good at handling vulnerable dependencies on time. Generally though, post-1.0.0 is better" - R66*

In the final section of the survey (after asking the respondents about their perceptions on our proposed attributes) we presented our own findings on the most important package attributes that lead to better responsiveness to vulnerable dependencies. We asked participants how likely they are to use our proposed attributes in their dependency practices in the future. Figure 5.9 presents the results for each of the attributes, ranging from Never using the attribute to Definitely using the attribute in the future. As can be seen, participants have a generally positive outlook on the applicability of our findings in practice. However, the ranking of the features proposed by developer is not the same as the rankling of our model. Specifically, practitioners have a strong tendency to use release frequency as a criterion for selecting packages in order to mitigate vulnerabilities from transitive dependencies.

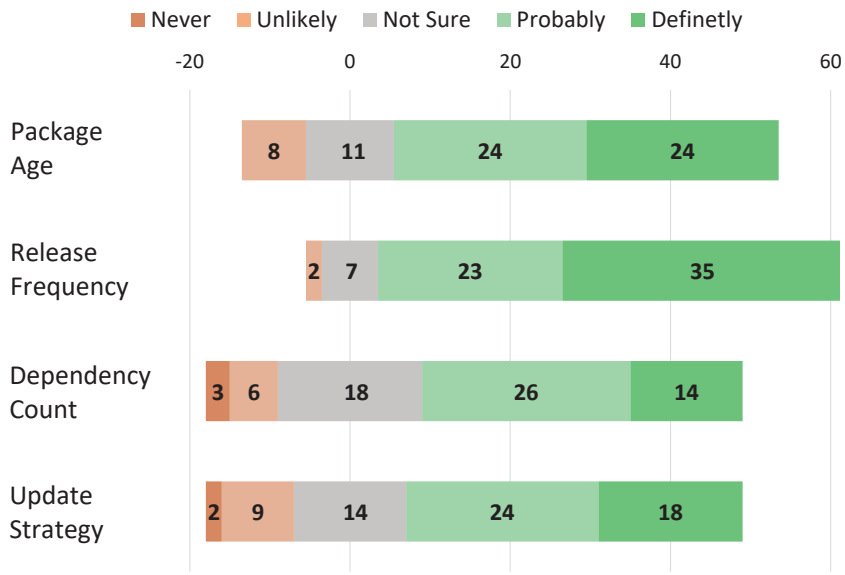


Figure 5.9: Likelihood of our top features being used in practice.

While the top attributes in our model are a considerable indicator of how fast packages address vulnerable dependencies, they are not equally accessible to all developers.

> *"The age and release frequency are attributes that are often easy to check on a package management website or GitHub. These attributes are often good indicators of the health of the package, thus, the likelihood that maintainers will address vulnerable dependencies in a fast manner[...]" - R22*

> *"[...] Regarding dependency counts, I know having more dependencies to manage carries more risks, but sometimes there are specific packages you need that don't have other equivalents in npm. [...] I'm not sure if we can see somewhere in the npm registry or github if a package has automatic updates, but I'd definitely take a look at their github repo and see if the package has recent updates and contributors." - R26*

Finding 3: Practitioners are likely to incorporate our findings into their dependency management practices, but not all attributes are readily accessible to downstream dependents.

5.4 Implications

We present actionable implications for practitioners and researchers.

Implications for Practitioners & Maintainers:

Delay in the adoption of fixes is the main reason packages remain vulnerable (RQ1). Almost all vulnerabilities in our dataset are fixed within a day after public disclosure, but it takes an average of 6 months for the fix to be adopted by downstream dependents (RQ1). There are many tools available for notifying developers of vulnerabilities in direct dependencies. For example, Dependabot is an open source tool can identify vulnerable dependencies and raise a pull request to adopt the fix (GitHub Docs, 2023). SonarQube (Sonar, 2023) is an enterprise-ready alternative that can identify vulnerable dependencies using its dependency check plugin (Dallig, 2023). However, developers must also be aware of the *transitive* influence of upstream packages on their project. Specifically, **developers**

should maintain an inventory of their entire dependency tree (direct and transitive) and periodically analyze their transitive dependencies for vulnerabilities. As one of the respondents in our survey (RQ3) highlighted: "Maintain some sort of SBOM [Software Bill of Materials] and monitor for transitive dependencies with known CVEs". In addition, **developers can use our proposed set of attributes (RQ2) as dependency selection criteria, because the consequences of bad dependency management trickles downstream**.

Multiple respondents in our practitioner survey (RQ3) cited the lack of accessibility as a reason for not using some of the suggested attributes. The npm registry uses badges to display certain metadata about a package such as the number of downloads and test coverage. **Package maintainers should expand the use of badges to include information sought after by potential downstream dependents to aid their dependency selection process**. A good example would be to display information regarding dependency update strategies (RQ2) such as the number of pinned dependencies. This functionality is currently available with third-party tools such as Dependency Sniffer (Javan Jafari, 2020). Additionally, **Vulnerability databases should add information regarding vulnerable dependencies for a package to provide a more holistic depiction of risk for potential dependents**.

Implications for Researchers:

We use a set of package attributes to model how fast the package will respond to a vulnerability fix in their dependencies (RQ2). However, as highlighted by our survey respondents (RQ3), socio-technical factors play a significant role in the responsiveness of a package to vulnerable dependencies. When asked about identifying packages with a speedy response to vulnerability fixes, one respondent stated: "For me, looking at who is behind the package also helps. Is it a college kid pushing his first npm package or is it a new project from Microsoft?". Another respondent said: "[...] packages owned by large corps (e.g. Microsoft, Meta...) will be faster in checking for these issues even before they have many users". **Future research should investigate the organizational practices that**

influence the responsiveness to software vulnerabilities, especially in the case of open source projects tied to large organizations.

We found that the dependency update strategy is one of the important indicators for predicting the response speed to vulnerability fixes (RQ2). An important reason why developers are hesitant to freely their dependencies is the fear of breaking changes. For example, one of the respondents in our survey (RQ3) said: "[...] when packages update, and break code (which can happen even for patch releases, even though it shouldn't)[...]". Another respondent stated: "For major changes it is not technically possible to allow automatic updates since the major updates includes breaking changes for other dependencies". Although Semantic Versioning was proposed to alleviate such issues, it is still not adopted by all packages in the npm ecosystem (Decan & Mens, 2019a). In fact, it is not uncommon for developers to downgrade to a previous version, following a seemingly compatible update (Cogo et al., 2019). Different ecosystems favor different practices and policies. Different packaging ecosystems can also have a differing levels of Semantic Versioning adoption (Decan et al., 2019; Dietrich et al., 2019; W. Li, Wu, Fu, & Zhou, 2023) and different cultural habits (Bogart et al., 2017, 2016). **Future research should study the ecosystem-specific attributes and policies that indicate responsive packages**, especially across packaging ecosystems with diverse policies.

5.5 Related Work

In addition to the studies cited throughout the chapter, this section describes the key research works that study vulnerable dependencies in the npm ecosystem and approaches for selecting dependencies.

Vulnerable dependencies:

Decan et al. (Decan et al., 2018b) explored the impact of security vulnerabilities in the npm ecosystem. They found that the number of vulnerabilities in the ecosystem are

on the rise but most vulnerabilities are fixed before they are publicly disclosed. They observed that a large fraction of packages do not immediately adopt the fix released by the upstream package, leaving them vulnerable despite the availability of the fix. The authors also highlighted that from a package user's perspective, there is no difference between being directly exposed to a vulnerability or being exposed to a vulnerability through a dependency.

Chinthanet et al. (Chinthanet et al., 2021) studied the release and adoption of vulnerability fixes in the npm ecosystem. They found that vulnerability fixes are not always released as a patch, but often bundled into other release types. Additionally, the majority of the commits in the majority of the fixing releases are not related to the security vulnerability. The authors found that even when the fix is released as a patch, the direct dependent package often releases the fixed package as a minor or major version, meaning as we travel downstream in the dependency chain, relying on patch-only fixes is increasingly ineffective. They also observed that the type of the release and the severity of the vulnerability influence the propagation of the fix across the ecosystem.

Alfadel et al. (Alfadel et al., 2023) conducted an empirical study on Node.js applications to analyze the discoverability of npm vulnerabilities. They found that 67% of applications have at least one vulnerable dependency. The main reason for the existence of publicly disclosed vulnerable dependencies in projects was the refusal to update the dependency to a newer version. In half of the applications studied by the authors, exposure to publicly disclosed vulnerabilities persists for more than 3 months. Additionally, the authors found that the majority of the projects (77%) are infected by a small subset of 5 vulnerability types.

Zimmermann et al. (Zimmermann et al., 2019) studied the potential of individual packages and package maintainers to threaten the security of the npm ecosystem. They found that installing an average npm package creates an implicit trust on 79 unique packages and

39 unique maintainers. Additionally, they found that the top 5 packages in npm are used by more than 100,000 downstream dependents, which makes such packages a primary target for attackers. The authors cited characteristics of the npm ecosystem such as heavy reuse, micropackages (small packages with few lines of code) and an open publishing model as potential security threats.

Zerouali et al. (Zerouali et al., 2022) empirically analyze the impact of vulnerabilities on transitive dependents in the npm and RubyGems ecosystems. They observe that it takes up to 7 years to disclose half of the lingering vulnerabilities in the npm ecosystem. They also found that more than 15% of the latest dependent releases in npm are exposed to vulnerabilities from direct dependencies and 36.5% of the latest releases are exposed to vulnerabilities from transitive dependencies. The authors found that the number vulnerabilities from transitive dependencies decrease as you go deeper along the dependency chain but vulnerabilities are still existent at the deepest levels.

Dependency selection:

Suhaib et al. (Mujahid et al., 2023) examined the characteristics of highly selected packages in the npm ecosystem. Through their qualitative analysis, they found that developers gravitate towards popular packages that have adequate documentation and are generally free from vulnerabilities. Their quantitative analysis of more than 2,500 packages confirmed their observations. The authors highlighted that developers should carefully consider the attributes of a package before adding it as a dependency of their project. They further mentioned that package maintainers should strive to make such attributes more accessible to their downstream dependents.

Vargas et al. (Larios Vargas et al., 2020) studied the technical, human and economic factors considered by practitioners when selecting dependencies. The authors present the importance of release characteristics such as active maintenance and stability of packages in the dependency selection process, but underline the lack of a standard means of measuring

such factors. The authors also observe disagreements between developers on the correct approach to selecting dependencies. They highlight the need to move away from ad-hoc decision-making towards a more systematic means of identifying suitable packages.

Pashchenko et al. (Pashchenko et al., 2020) conducted 25 interviews to understand how developers select packages. The found that developers consider the community support of a package as an important factor when selecting dependencies. They also observed that developers have different dependency management practices but generally regard vulnerabilities as an important factor in dependency management decisions. Developers expressed frustration with packages with packages with a high number of dependencies due to the lack of control over transitive dependencies.

Our study builds on the previous works regarding vulnerabilities and dependency selection in the npm ecosystem by proposing empirically extracted dependency management practices that are associated with a faster response to vulnerability fixes. In addition to providing an in-depth analysis of how certain package attributes and behaviors influence the adoption of vulnerability fixes, our practitioner-aligned solution to selecting dependencies provides a means to mitigate the impact of vulnerabilities from transitive dependencies which are commonplace in the npm ecosystem.

5.6 Threats to Validity

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

Threats to construct validity: Threats to construct validity refer to the concern between the theory and the results of the study. In order to measure *responsiveness*, we categorize the adoption delay into fast and slow classes based on a threshold of less than 2 days and more than 14 days. However, there is no consensus on what is defined as fast or slow. We initially experimented with a multi-class model by distributing the delay into 4 classes. A response of 2 days or less was classified as fast; a response of more than 2 days but less than 2 weeks

was classified as acceptable; a response between 2 weeks and 3 months was classified as mediocre and a response later than 3 months was classified as slow. However, the fast and slow classes combined made up over 96% of our class distribution, indicating that we are in fact dealing with a binary classification problem. We then conducted a sensitivity analysis to ensure minor changes in our threshold does not translate into a considerable change in our class distribution. Reducing the threshold from 48 hours to 0 hours decreases the distribution of the Fast class from 63.7% to 63.6% (less than 1% change). Increasing the threshold from 48 hours to 30 days increases the distribution of the Fast class from 63.7% to 64.2% (less than 1% change). When identifying *exposure to vulnerabilities* from dependencies, we assume all dependencies in a package are fetched from the npm package manager. In reality, developers can install a dependency from any source (e.g. GitHub). The problem with considering sources outside of the official package registry (npm) is that there is no way to extract the list of dependents for an ad-hoc package hosted on the web. There is also no way to guarantee what is installed by the package manager as the contents of the hosted package can change at any time.

A vulnerability fix can be released as a major, minor or patch version (Section 2). Since we have the information for package versions and vulnerability metadata, we can compare the version number of the fixing release (r) with the version number of the release right before the fix (r-1) to evaluate the type of fixing release.

Threats to internal validity: Threats to internal validity refer to the concerns that are internal to the study such as experimenter bias and errors. We use 9 features that we believe can serve as indicators of the response to vulnerabilities. There could be additional indicators that are not captured (or not feasible) using our feature set. One example is the experience of the development team for a package, which can influence how they respond to vulnerabilities regardless of the package attributes (as hinted in the responses of RQ3).

We do not claim our collection of features to be an exhaustive list of all of the relevant characteristics and behaviors for predicting the adoption of vulnerability fixes. However, as can be seen in Section 5.3, our model model has a high capability of predicting the response to vulnerabilities. In order to extract the release type of the vulnerability fix in RQ1, we compare the version of the fixing release against the version of the release right before the fix. However, some packages may have multiple simultaneous release streams which means the chronological order of release may not align with the numerical order. The libraries.io dataset used in this study dates to January 2020. Collecting the metadata for an entire ecosystem from scratch requires great effort but is also prone to errors. The libraries.io dataset has been used in multiple prior studies (Decan & Mens, 2019a, 2021; Zerouali et al., 2022; Zerouali, Mens, Robles, & Gonzalez-Barahona, 2019) and its accuracy has been verified by other researchers (Decan et al., 2019). Additionally, we study vulnerabilities that have been discovered, disclosed, fixed and propagated across the ecosystem. We are more interested in the dynamics of dependency management practices for vulnerability mitigation, rather than the response to the latest vulnerabilities.

Threats to external validity: Threats to external validity concern the generalization of our findings. The methodology for identifying package attributes that indicate a fast response to vulnerabilities is applicable to other ecosystems. However, the scope of our findings is focused on the npm ecosystem. Therefore, our results may not be applicable to other software packaging ecosystems, especially if they follow dependency guidelines that are considerably different than npm. We conduct a survey to understand the developer's perception on our findings. While our response rate of 59% is considerably higher than the the usual rate in software engineering surveys based on questionnaires (Singer et al., 2008), having more respondents may influence our understanding on practitioner perspectives.

5.7 Chapter Conclusion

The objective of our study was to propose a means to mitigate vulnerabilities in transitive dependencies. We curated a dataset of 450 vulnerabilities and over 200,000 unique dependents that are exposed to these vulnerabilities through their dependencies. We use 9 features to train a model that predicts the adoption speed of vulnerability fixes. We found that packages that younger packages that release more often, favor non-restrictive update strategies and have less dependencies are faster in adopting vulnerability fixes. We also conducted a survey of 67 industry practitioners to obtain their perception on our findings. We found that the experience of practitioners generally align with our proposed set of features. However, many were not aware of the importance of downstream dependency decisions in mitigating vulnerabilities. Previous research has frequently suggested that practitioners need to be wary of the risk of vulnerabilities from transitive dependencies. Developers can use our findings to incorporate the mitigation of vulnerabilities from transitive dependencies in their dependency selection criteria.

Chapter 6

Conclusion and Future Work

In this chapter, we summarize the conclusions and findings from each chapter and discuss future directions for research.

6.1 Conclusion

Software ecosystems accelerate software development through third-party dependencies. However, increased reliance on third-party packages exacerbates dependency related challenges such as breaking changes and vulnerabilities. In this thesis, we empirically study the npm ecosystem to understand such challenges and propose mitigation techniques to help developers in their dependency management. We first start by cataloging dependency management problems and quantifying their impact. Next, we extract how upstream packages are used by downstream dependents and propose a technique to help developers decide on a suitable update strategy for their direct dependents. Finally, we help developers control the impact of vulnerabilities from transitive dependencies by proposing a solution to identify and select responsive packages. In the following, we discuss the main findings and contributions of each chapter.

6.1.1 Challenges in Dependency Management

We use a mixture of quantitative and qualitative methods to catalog dependency management issues (i.e. *Dependency Smells*) in the npm ecosystem. We conduct an empirical study on the history of over 1,100 npm projects and measure the prevalence of 7 dependency smells. We found that 80% of the projects in our dataset were infected with at least two distinct smells. The evolution of these smells in the dataset suggests that there is an upward accumulation of dependency smells in the ecosystem. We then surveyed 41 practitioners to quantify the impact of such smells and found that developers agree on the harmful nature of dependency smells. We identified the smell-introducing commits and contacted the associated developers to investigate the reasons behind such smells and aggregated the responses into 14 reasons. Dependency smells were largely a reactionary decision to shortcomings in the upstream package or the npm ecosystem. We also developed a tool named *Dependency Sniffer* that analyzes npm projects and detects the presence of dependency smells.

6.1.2 Practices for Updating Dependencies

Developers have a different perception and trust towards their various dependencies and do not utilize the same constraints for all of their dependencies. We conduct an empirical study on more than 112,000 npm packages to identify characteristics in packages that indicate the favored update strategy by their downstream dependents. We use 19 characteristics to build a model that predicts the update strategy of dependents with high accuracy. We further investigate the characteristics to show how a change in a characteristic influences the prediction of the model. We found that the release status, number of dependents and the age of the packages are prime indicators of the update strategy favored by the community and developers should consider these indicators when determining the update strategy of each dependency. We complement the study with a deeper analysis of 160 packages in

the dataset to analyze the evolution of their dependent update strategy over 10 years. We recommend that developers take note of the 1.0.0 release milestone, as it creates a shift in the update strategy of the corresponding dependent community.

6.1.3 Practices for Selecting Dependencies

The number of transitive dependencies for the average package in the npm ecosystem is much higher than direct dependencies. Transitive dependencies can expose software projects to security risks but developers cannot directly influence which transitive dependencies are installed with their project. We extract 450 vulnerability reports for the npm ecosystem and study over 200,000 packages infected with these vulnerabilities. We found that the main contributing factor for continued exposure to such vulnerabilities is the reluctance of packages to update their dependencies. We use 9 attributes from npm packages to build a model that predicts the adoption speed of vulnerability fixes. We recommend that developers consider the release frequency and update strategy of a package in the dependency selection criteria to shorten their exposure to vulnerabilities. In our survey of 67 practitioners, we found that developers were generally unaware of the reluctance to update as the main culprit of vulnerability exposure. However, developers are in favor of using our proposed attributes in their dependency selection practices.

6.2 Future Work

This thesis has made many contributions towards understanding dependency management challenges in software ecosystems and providing actionable mitigation techniques. In the following, we summarize the key directions for future research that were identified throughout our research.

6.2.1 Impact of the ecosystem on external projects

With the exception of Chapter 3, our research is focused on the dependency relationships of packages inside the npm packaging ecosystem. There are many packages outside the npm ecosystem that rely on (and are influenced by) packages inside the ecosystem. There is currently no means to accurately account for the number of external projects that rely on npm. The favored update strategy in Chapter 4 and the responsiveness to vulnerabilities in Chapter 5 may significantly differ if we were to include external projects. When discussing both the impact of dependent decisions and the influence of upstream packages on downstream dependents, having a comprehensive inventory of all dependent projects, internal and external to the ecosystem, results in more accurate observations and suggestions. Future work should at the very least include the entirety of GitHub projects alongside npm packages and consider the number of downloads from npm (in addition to dependent count) to better account for external projects.

6.2.2 Generalizability to other ecosystems

Chapter 3 investigate the applicability of our dependency smell catalog on other ecosystems such as PyPI (the Python package ecosystem). However, developers in packaging ecosystems for programming language with features distinct from JavaScript (e.g. Java) may have a different experience. The differences between various ecosystems is not limited to the technicalities of how packages are fetched, installed and used. Previous research suggests that there are also cultural and behavioral differences between ecosystems that can influence the dynamics of dependency managements. We have included replication packages for all of our studies (Javan Jafari et al., 2021, 2023, 2022) to facilitate future research efforts that aim to replicate our findings on other ecosystems.

6.2.3 Industry vs. Open-source priorities

The findings from our practitioner surveys in Chapter 3 and Chapter 5 show that developers are sometimes aware of the advantages and disadvantages of various dependency management decisions, but are forced to heavily prioritize certain dependency management objectives. In particular, projects in the industry may tend to favor system stability above new features available in new package versions, which could highly influence their update behavior. On the other hand, there is a greater responsibility of handling security vulnerabilities for critical systems compared to open source projects. Future research should investigate the dependency management practices of industry projects to evaluate how the difference in priorities between the industry and open source initiatives influences dependency management.

6.2.4 Extent of dependency utilization

This thesis studies the dynamics of dependency management rather than the extent of dependency utilization. While a package with many declared dependencies is heavily influenced by third-party code, it does not necessarily heavily utilize each dependency. When it comes to dependencies, a project may use as little as a function or as much as entire classes from the declared dependency. In fact, Chapter 3 reveals that some declared dependencies may not be used at all. Other research has also found that, despite what developers declare, not all packages are equally used in production (Latendresse, Mujahid, Costa, & Shihab, 2022). Major dependency issues such as breaking changes and security vulnerabilities can often be tracked down to specific portions of a package and may not equally impact downstream dependents that use different portions of the upstream packaged code. Knowing the details of utilization can provide further insight into why packages use particular update strategies (Chapter 4) or react differently to vulnerable dependencies (Chapter 5). Future

research should consider the degree of dependency utilization in addition to the raw dependency relationship between packages.

6.2.5 Impact of package functionality

The Practitioners in Chapter 5 remind us that the treatment of a dependency is directly tied to the criticality of the software. Consequently, the functionality of a package is tied to the criticality of its utilization. A packages that handles database queries can be considered more critical than a package that facilitates unit testing. Chapter 4 attempts to consider package domain and functionality as an indicator of the update strategy, but the automated clustering approach did not yield actionable package groups. The research field would greatly benefit from a large-scale analysis and catalog of packages based on their functionality to observe how both the behavior of package maintainers and its downstream dependents are impacted by the criticality of package functionality.

References

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

Alfadel, M., Costa, D. E., Shihab, E., & Adams, B. (2023). On the discoverability of npm vulnerabilities in node. js projects. *ACM Transactions on Software Engineering and Methodology*, *32*(4), 1–27.

Artho, C., Suzaki, K., Di Cosmo, R., Treinen, R., & Zacchiroli, S. (2012). Why do software packages conflict? In *Proceedings of the 9th ieee working conference on mining software repositories* (pp. 141–150).

Baltes, S., & Ralph, P. (2022). Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering*, *27*(4), 1–31.

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

Bogart, C., Filippova, A., Kästner, C., & Herbsleb, J. (2017, October). *How ecosystem cultures differ: Results from a survey on values and practices across 18 software ecosystems.* `http://breakingapis.org/survey/`. ((accessed on 10/16/2020))

Bogart, C., Kästner, C., Herbsleb, J., & Thung, F. (2016). How to break an api: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software*

engineering (pp. 109–120).

Bogart, C., Kästner, C., Herbsleb, J., & Thung, F. (2021). When and how to make breaking changes: Policies and practices in 18 open source software ecosystems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *30*(4), 1–56.

Bombonatti, D., Goulão, M., & Moreira, A. (2017). Synergies and tradeoffs in software reuse–a systematic mapping study. *Software: practice and experience*, *47*(7), 943–957.

Burnard, P. (1991). A method of analysing interview transcripts in qualitative research. *Nurse education today*, *11*(6), 461–466.

Burrows, D. (2017, October). *What is a package manager?* https://www.debian.org/doc/manuals/aptitude/pr01s02.en.html.

Chatfield, D. (2014, September). *Fix the versioning · issue #1805 · jashkenas/underscore.* https://github.com/jashkenas/underscore/issues/1805. ((Accessed on 10/16/2020))

Chinthanet, B., Kula, R. G., Ishio, T., Ihara, A., & Matsumoto, K. (2019). On the lag of library vulnerability updates: An investigation into the repackage and delivery of security fixes within the npm JavaScript ecosystem. *arXiv preprint arXiv:1907.03407*.

Chinthanet, B., Kula, R. G., McIntosh, S., Ishio, T., Ihara, A., & Matsumoto, K. (2021). Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering*, *26*, 1–28.

Chowdhury, M. A. R., Abdalkareem, R., Shihab, E., & Adams, B. (2021). On the untriviality of trivial packages: An empirical study of npm javascript packages. *IEEE Transactions on Software Engineering*, *48*(8), 2695–2708.

Cimpanu, C. (2021, October). *Malware found in npm package with millions of weekly downloads.* https://therecord.media/malware-found-in-npm-package-with-millions-of-weekly-downloads.

Coghlan, N., & Stufft, D. (2021, February). *Version identification and dependency specification.* https://www.python.org/dev/peps/pep-0440. ((accessed on 3/20/2021))

Cogo, F. R., Oliva, G. A., Bezemer, C.-P., & Hassan, A. E. (2021). An empirical study of same-day releases of popular packages in the npm ecosystem. *Empirical Software Engineering*, *26*(5), 89.

Cogo, F. R., Oliva, G. A., & Hassan, A. E. (2019). An empirical study of dependency downgrades in the npm ecosystem. *IEEE Transactions on Software Engineering*.

Cogo, F. R., Oliva, G. A., & Hassan, A. E. (2021). Deprecation of packages and releases in software ecosystems: A case study on npm. *IEEE Transactions on Software Engineering*, *48*(7), 2208–2223.

Cox, J., Bouwers, E., Van Eekelen, M., & Visser, J. (2015). Measuring dependency freshness in software systems. In *2015 ieee/acm 37th ieee international conference on software engineering* (Vol. 2, pp. 109–118).

Dallig, P. (2023). *Dependency check plugin for sonar.* https://github.com/dependency-check/dependency-check-sonar-plugin.

Decan, A., & Mens, T. (2019a). What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*.

Decan, A., & Mens, T. (2019b). What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*.

Decan, A., & Mens, T. (2021). Lost in zero space - an empirical comparison of 0.y.z releases in software package distributions. *Science of Computer Programming*, *208*, 102656.

Decan, A., Mens, T., & Claes, M. (2017). An empirical comparison of dependency issues in oss packaging ecosystems. In *2017 ieee 24th international conference on software analysis, evolution and reengineering (saner)* (pp. 2–12).

Decan, A., Mens, T., & Constantinou, E. (2018a). On the evolution of technical lag in the npm package dependency network. In *2018 ieee international conference on software maintenance and evolution (icsme)* (pp. 404–414).

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

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

Decan, A., Mens, T., Zerouali, A., & De Roover, C. (2021). Back to the past–analysing backporting practices in package dependency networks. *IEEE Transactions on Software Engineering*, *48*(10), 4087–4099.

Derr, E., Bugiel, S., Fahl, S., Acar, Y., & Backes, M. (2017). Keep me updated: An empirical study of third-party library updatability on Android. In *Proceedings of the 2017 acm sigsac conference on computer and communications security* (pp. 2187–2200).

Dey, T., & Mockus, A. (2020). Deriving a usage-independent software quality metric. *Empirical Software Engineering*, *25*, 1596–1641.

Dietrich, J., Pearce, D. J., Stringer, J., Tahir, A., & Blincoe, K. (2019). Dependency versioning in the wild. In *Proceedings of the 16th international conference on mining software repositories* (pp. 349–359).

Fan, G., Wang, C., Wu, R., Xiao, X., Shi, Q., & Zhang, C. (2020). Escaping dependency hell: finding build dependency errors with the unified dependency graph. In *Proceedings of the 29th acm sigsoft international symposium on software testing and analysis* (pp. 463–474).

Fincher, S., & Tenenberg, J. (2005). Making sense of card sorting data. *Expert Systems*,

22(3), 89-93. doi: 10.1111/j.1468-0394.2005.00299.x

Fontana, F. A., Dietrich, J., Walter, B., Yamashita, A., & Zanoni, M. (2016). Antipattern and code smell false positives: Preliminary conceptualization and classification. In *2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner)* (Vol. 1, pp. 609–613).

Géron, A. (2019). *Hands-on machine learning with scikit-learn, keras, and tensorflow: Concepts, tools, and techniques to build intelligent systems*. " O'Reilly Media, Inc.".

GitHub. (2019). *The state of the octoverse — the state of the octoverse celebrates a year of building across teams, time zones, and millions of merged pull requests.* https://octoverse.github.com/.

GitHub. (2020). *The 2020 state of the octoverse security report.* https://octoverse.github.com/static/github-octoverse-2020-security-report.pdf.

GitHub. (2021). *The 2021 state of the octoverse security report.* https://octoverse.github.com/static/octoverse-report-2021.pdf.

Github. (2021, October). *Security issue: compromised npm packages of ua-parser-js.* https://github.com/faisalman/ua-parser-js/issues/536.

GitHub. (2023a). *About coordinated disclosure of security vulnerabilities.* https://docs.github.com/en/code-security/security-advisories/guidance-on-reporting-and-writing/about-coordinated-disclosure-of-security-vulnerabilities.

GitHub. (2023b). *Github advisory database.* https://github.com/advisories.

GitHub Docs. (2023). *Dependabot.* https://docs.github.com/en/code-security/dependabot.

Goldstein, A., Kapelner, A., Bleich, J., & Pitkin, E. (2015). Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation.

journal of Computational and Graphical Statistics, *24*(1), 44–65.

Goodman, L. A. (1961). Snowball sampling. *The annals of mathematical statistics*, 148–170.

Gousios, G. (2019, Mar). *The GHTorrent project.* https://ghtorrent.org/. ((accessed on 10/16/2020))

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

Hollander, M., Wolfe, D. A., & Chicken, E. (2013). *Nonparametric statistical methods*. John Wiley & Sons.

Jafari, A. J., Costa, D. E., Abdalkareem, R., Shihab, E., & Tsantalis, N. (2021). Dependency smells in javascript projects. *IEEE Transactions on Software Engineering*, *48*(10), 3790–3807.

Jafari, A. J., Costa, D. E., Abdellatif, A., & Shihab, E. (2023). Dependency practices for vulnerability mitigation. *arXiv*.

Jafari, A. J., Costa, D. E., Shihab, E., & Abdalkareem, R. (2023). Dependency update strategies and package characteristics. *ACM Transactions on Software Engineering and Methodology*.

Javan Jafari, A. (2020, September). *Dependency Sniffer.* https://github.com/abbasjavan/DependencySniffer.

Javan Jafari, A., Elias Costa, D., Abdalkareem, R., Shihab, E., & Tsantalis, N. (2021, March). *Replication package for dependency smells in JavaScript projects.* https://doi.org/10.5281/zenodo.4701497.

Javan Jafari, A., Elias Costa, D., Abdellatif, A., & Shihab, E. (2023, October). *Replication package for dependency practices for vulnerability mitigation.* https://doi.org/10.5281/zenodo.8432714.

Javan Jafari, A., Elias Costa, D., Shihab, E., & Abdalkareem, R. (2022, August). *Replication package for dependency update strategies and package characteristics.* https://doi.org/10.5281/zenodo.5643627.

JetBrains. (2021, March). *Code inspections in JavaScript and TypeScript.* https://www.jetbrains.com/help/phpstorm/code-inspections-in-javascript-and-typescript.html#Imports_and_dependencies. ((accessed on 26/04/2021))

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., & Damian, D. (2014). The promises and perils of mining GitHub. In *Proceedings of the 11th working conference on mining software repositories* (p. 92–101). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2597073.2597074 doi: 10.1145/2597073.2597074

Keswani, R., Joshi, S., & Jatain, A. (2014). Software reuse in practice. In *2014 fourth international conference on advanced computing & communication technologies* (pp. 159–162).

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

Kula, R. G., De Roover, C., German, D. M., Ishio, T., & Inoue, K. (2018). A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem. In *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)* (pp. 288–299).

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

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

Kula, R. G., Ouni, A., German, D. M., & Inoue, K. (2017). On the impact of micro-packages: An empirical study of the npm javascript ecosystem. *arXiv preprint arXiv:1709.04638*.

Larios Vargas, E., Aniche, M., Treude, C., Bruntink, M., & Gousios, G. (2020). Selecting third-party libraries: The practitioners' perspective. In *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 245–256).

Latendresse, J., Mujahid, S., Costa, D. E., & Shihab, E. (2022). Not all dependencies are equal: An empirical study on production dependencies in npm. In *Proceedings of the 37th ieee/acm international conference on automated software engineering* (pp. 1–12).

Lehnardt, J., & Haas, S. (2020, September). *Greenkeeper.* https://greenkeeper.io/docs.html. ((accessed on 10/14/2020))

Li, J., & Lukic, D. (2019, October). *Depcheck: Check your npm module for unused dependencies.* https://github.com/depcheck/depcheck. ((accessed on 10/16/2020))

Li, W., Wu, F., Fu, C., & Zhou, F. (2023). A large-scale empirical study on semantic versioning in golang ecosystem. *arXiv preprint arXiv:2309.02894*.

Libraries.io. (2020, September). *The npm ecosystem.* https://libraries.io/npm. ((accessed on August, 2021))

Libraries.io. (2020, September). *Overview and documentation.* https://docs.libraries.io/overview.html. ((accessed on August, 2021))

Libraries.io. (2023). *npm.* https://libraries.io/npm.

Lim, W. C. (1994). Effects of reuse on quality, productivity, and economics. *IEEE software*, *11*(5), 23–30.

Liu, C., Chen, S., Fan, L., Chen, B., Liu, Y., & Peng, X. (2022). Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th international conference on software engineering* (pp. 672–684).

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

MacDonald, F. (2018, September). *How a programmer nearly broke the internet by deleting just 11 lines of code.* https://www.sciencealert.com/how-a-programmer-almost-broke-the\protect\discretionary{\char\hyphenchar\font}{}{}internet-by-deleting-11-lines-of-code.

Mann, H. B., & Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, 50–60.

Matt Rickard. (2021). *The nine circles of dependency hell (and a roadmap out).* https://about.sourcegraph.com/blog/nine-circles-of-dependency-hell.

Mezzetti, G., Møller, A., & Torp, M. T. (2018). Type regression testing to detect breaking changes in node. js libraries. In *32nd european conference on object-oriented programming (ecoop 2018)*.

Mitre. (2023a). *Common Vulnerabilities and Exposures (CVE).* https://www.cve.org/About/Overview.

Mitre. (2023b). *Common Weakness Enumeration (CWE).* https://cwe.mitre.org/about/index.html.

Mohagheghi, P., Conradi, R., Killi, O. M., & Schwarz, H. (2004). An empirical study of

software reuse vs. defect-density and stability. In *Proceedings of the 26th international conference on software engineering* (pp. 282–292).

Møller, A., Nielsen, B. B., & Torp, M. T. (2020). Detecting locations in javascript programs affected by breaking library changes. *Proceedings of the ACM on Programming Languages*, *4*(OOPSLA), 1–25.

Molnar, C. (2020). *Interpretable machine learning*. Lulu.com.

Mujahid, S., Abdalkareem, R., & Shihab, E. (2023). What are the characteristics of highly-selected packages? a case study on the npm ecosystem. *Journal of Systems and Software*, *198*, 111588.

Mujahid, S., Abdalkareem, R., Shihab, E., & McIntosh, S. (2020). Using others' tests to identify breaking updates. In *Proceedings of the 17th international conference on mining software repositories* (pp. 466–476).

NLTK. (2022, April). *Collocations documentation.* http://www.nltk.org/howto/collocations.html. ((accessed on August, 2021))

npm. (2017, December). *The npm blog — new package moniker rules.* https://blog.npmjs.org/post/168978377570/new-package-moniker-rules. ((accessed on 10/16/2020))

npm. (2018). *This year in javascript: 2018 in review.* https://blog.npmjs.org/post/180868064080/this-year-in-javascript-2018-in-review-and-npms.html.

npm. (2021, February). *npm install documentation.* https://docs.npmjs.com/cli/v6/commands/npm-install.

npm. (2022a, February). *About semantic versioning.* https://docs.npmjs.com/about-semantic-versioning.

npm. (2022b, April). *The npm registry.* https://www.npmjs.com/.

npm Docs. (2023). *npm-audit.* https://docs.npmjs.com/cli/v9/commands/

npm-audit.

npm Documentation. (2019a, October). *npm-package.json — npm documentation.* https://docs.npmjs.com/files/package.json.

npm Documentation. (2019b, October). *npm-package-lock.json — npm documentation.* https://docs.npmjs.com/files/package-lock.json. ((accessed on 10/16/2020))

npmjs. (2023). *The semver package.* https://www.npmjs.com/package/semver.

Ohm, M., Boes, F., Bungartz, C., & Meier, M. (2022). On the feasibility of supervised machine learning for the detection of malicious software packages. In *Proceedings of the 17th international conference on availability, reliability and security* (pp. 1–10).

Opdebeeck, R., Zerouali, A., Velázquez-Rodríguez, C., & De Roover, C. (2020). Does infrastructure as code adhere to semantic versioning? an analysis of ansible role evolution. In *2020 ieee 20th international working conference on source code analysis and manipulation (scam)* (pp. 238–248).

Oxley, T. (2014, September). *Semver: Tilde and caret.* https://nodesource.com/blog/semver-tilde-and-caret. ((accessed on 3/22/2021))

Pandas. (2023). *Pandas api reference.* https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html.

Pashchenko, I., Vu, D.-L., & Massacci, F. (2020). A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 acm sigsac conference on computer and communications security* (pp. 1513–1531).

Prana, G. A. A., Sharma, A., Shar, L. K., Foo, D., Santosa, A. E., Sharma, A., & Lo, D. (2021). Out of sight, out of mind? how vulnerable dependencies affect open-source projects. *Empirical Software Engineering*, *26*(4), 1–34.

Preston-Werner, T. (2019). *Semantic versioning 2.0.* Retrieved from https://semver

.org/

PyPA. (2021, February). *pip documentation.* https://pip.pypa.io/en/latest/user_guide/. ((accessed on August, 2021))

Raemaekers, S., Van Deursen, A., & Visser, J. (2014). Semantic versioning versus breaking changes: A study of the maven repository. In *2014 ieee 14th international working conference on source code analysis and manipulation* (pp. 215–224).

Rahman, A., Parnin, C., & Williams, L. (2019). The seven sins: security smells in infrastructure as code scripts. In *Proceedings of the 41st international conference on software engineering* (pp. 164–175).

Rust-lang. (2018, December). *Rust edition guide.* https://doc.rust-lang.org/edition-guide/rust-2018/cargo-and-crates-io/crates-io-disallows-wildcard-dependencies.html. ((accessed on September, 2020))

Scikit. (2023). *Scikit api reference.* https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html.

Scikit-learn. (2020). *Permutation importance vs random forest feature importance.* Retrieved from https://scikit-learn.org/stable/auto_examples/inspection/plot_permutation_importance.html

Sharma, T., Fragkoulis, M., & Spinellis, D. (2016). Does your configuration code smell? In *2016 ieee/acm 13th working conference on mining software repositories (msr)* (pp. 189–200).

Shaw, A. (2020, September). *David-dm.* https://david-dm.org.

Singer, J., Sim, S. E., & Lethbridge, T. C. (2008). Software engineering data collection for field studies. In *Guide to advanced empirical software engineering* (pp. 9–34). Springer.

snyk. (2022, April). *snykadvisor.* https://snyk.io/advisor/.

Sonar. (2023). *Sonarqube.* https://www.sonarsource.com/products/sonarqube/.

Sonatype. (2021). *2021 state of the software supply chain.* https://www.sonatype.com/hubfs/SSSC-Report-2021_0913_PM_2.pdf?hsLang=en-us.

Soto-Valero, C., Benelallam, A., Harrand, N., Barais, O., & Baudry, B. (2019). The emergence of software diversity in maven central. In *2019 ieee/acm 16th international conference on mining software repositories (msr)* (pp. 333–343).

Soto-Valero, C., Harrand, N., Monperrus, M., & Baudry, B. (2021). A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering*, *26*(3), 1–44.

Tal, L. (2019). *Snyk research team discovers severe prototype pollution security vulnerabilities affecting all versions of lodash.* https://snyk.io/blog/snyk-research-team-discovers-severe-prototype-pollution/security-vulnerabilities-affecting-all-versions-of-lodash.

Tharwat, A. (2020). Classification assessment methods. *Applied computing and informatics*, *17*(1), 168–192.

Tidelift. (2022). *The 2022 open source software supply chain survey report.* https://tidelift.com/2022-open-source-software-supply-chain-survey.

Venturini, D., Cogo, F. R., Polato, I., Gerosa, M. A., & Wiese, I. S. (2023). I depended on you and you broke me: An empirical study of manifesting breaking changes in client packages. *ACM Transactions on Software Engineering and Methodology*, *32*(4), 1–26.

Wittern, E., Suter, P., & Rajagopalan, S. (2016). A look at the dynamics of the JavaScript package ecosystem. In *2016 ieee/acm 13th working conference on mining software*

repositories (msr) (pp. 351–361).

Yarn. (2020, 2020). *Documentation — yarn.* https://classic.yarnpkg.com/en/docs/. ((accessed on 10/16/2020))

Zerouali, A., Constantinou, E., Mens, T., Robles, G., & González-Barahona, J. (2018). An empirical analysis of technical lag in npm package dependencies. In *New opportunities for software reuse: 17th international conference, icsr 2018, madrid, spain, may 21-23, 2018, proceedings 17* (pp. 95–110).

Zerouali, A., Mens, T., Decan, A., & De Roover, C. (2022). On the impact of security vulnerabilities in the npm and rubygems dependency networks. *Empirical Software Engineering*, *27*(5), 107.

Zerouali, A., Mens, T., Robles, G., & Gonzalez-Barahona, J. M. (2019). On the diversity of software package popularity metrics: An empirical study of npm. In *2019 ieee 26th international conference on software analysis, evolution and reengineering (saner)* (pp. 589–593).

Zimmermann, M., Staicu, C.-A., Tenny, C., & Pradel, M. (2019). Small world with high risks: A study of security threats in the npm ecosystem. *arXiv preprint arXiv:1902.09217 (To appear in the 28th USENIX Security Symposium)*.