UNTANGLING THE WEB OF DEPENDENCIES

A COMPLEX NETWORK ANALYSIS OF THE NPM ECOSYSTEM

University of Cape Town

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

UNTANGLING THE WEB OF DEPENDENCIES

A COMPLEX NETWORK ANALYSIS OF THE NPM ECOSYSTEM

Thesis

This thesis is presented in the partial fulfilment
of the requirements for the degree of
Masters of Commerce: Economic Development
at the University of Cape Town

by

EMILIE-ROSE OLDNALL

Supervisor: Associate Prof.Co-Pierre Georg
Student number: OLDEMI001

To my network of support, who enabled me to succeed.

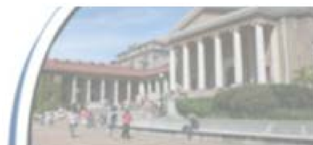
ABSTRACT

Open-source software development is a collaborative effort resulting in complex dependencies between software packages. Unlike proprietary software, the open-source model offers a unique opportunity to analyse and trace these dependencies due to its public availability. This thesis maps out the complex dependency network within the npm ecosystem, the package manager for JavaScript. JavaScript is the world's most widely used programming language, and its package manager is a tool responsible for storing and distributing thousands of third-party software packages to the developer community. Yet, with greater interconnectivity comes greater vulnerability, a reality sharply highlighted in 2016 when removing the small utility `left-pad` package from the npm registry. This event precipitated widespread software breakage as many web applications transitively and unknowingly depended on it for functionality.

This thesis uses complex network science to demonstrate how network measures can be used to determine the structure and level of complexity of the npm network and, more interestingly, how these parameters evolve over time. I analyse the npm network over five years, from 2012 to 2016. To the author's knowledge, no study at the time of writing has analysed the npm package ecosystem at a version level from the perspective of complex network science.

This thesis finds that the npm network exhibits small-world behaviour and a scale-free architecture, concurring with existing studies on open-source software systems. It underscores the pivotal role of hierarchical software design in moulding npm's network topology and identifies versioned packages that disproportionately influence the network's functionality. Notably, it reveals that central nodes can have up to 200,000 reverse transitive dependencies, highlighting the ecosystem's vulnerability to cascading failures. By providing a detailed exploration of npm's complex dependency network, this research deepens our understanding of npm's infrastructure and highlights the critical network dynamics at play in open-source software development. These insights pave the way for further research on mitigating potential vulnerabilities and improving the resilience of software dependency networks.

UNIVERSITY OF CAPE TOWN
FACULTY OF COMMERCE
Igniting Knowledge and Opportunity



Plagiarism Declaration

COMPULSORY DECLARATION:

1. This dissertation has been submitted to Turnitin (or equivalent similarity and originality checking software) and I confirm that my supervisor has seen my report and any concerns revealed by such have been resolved with my supervisor.
2. I certify that I have received Ethics approval (if applicable) from the Commerce Ethics Committee.
3. This work has not been previously submitted in whole, or in part, for the award of any degree in this or any other university. It is my own work. Each significant contribution to, and quotation in, this dissertation from the work, or works of other people has been attributed, and has been cited and referenced.

Student number	**EMILIE-ROSE OLDNALL**
Student name	**OLDEMI001**
Signature of Student	Signed by candidate
Date:	**2023/02/12**

TOTAL WORD COUNT: 28 303

NOMENCLATURE

ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
npm	node package manager
OSS	Open-Source Software
OOP	Object-Oriented Programming
RAM	Random Access Memory
SCC	Strongly Connected Component
SD	Standard deviation
TC	Transitive Closure
UI	User Interface
WCC	Weakly Connected Component

VARIABLES

$\mathscr{G}$	Dependency network
$\mathscr{N}$	Number of nodes in a network
$\mathscr{E}$	Number of links in a network
k_{in}	In-degree of a node/the number of links coming into a target node.
k_{out}	Out-degree of a node/the number of links going out of a target node.
j_{in}	In-degree of a node/ the number of links coming into a source node.
j_{out}	Out-degree of a node/the number of links going out of a source node.
p_k	Degree distribution of a network
r_{in}	In-Assortativity of a directed network
r_{out}	Out-Assortativity of a directed network
γ	Scale-free exponent of a network

1

INTRODUCTION

"Scientists and nonscientists alike...can easily mislead themselves about complexity if they are not properly attuned to it." - James Gleick. Chaos Making a New Science

1.1. Background

Package managers have become essential tools for modern software development, facilitating the management of complex dependencies and ensuring the stability and reliability of software projects. A *dependency* between two software packages arises when a project requires certain functionality from another package. Instead of a coder developing this functionality herself, she reuses the existing code from one package within the project she is developing.

Software projects may rely on multiple dependencies for functionality, creating complex interconnections between software components. Dependency resolution is the process of finding and installing the correct versions of a project's dependencies (libraries, modules, packages, etc.) in order to run the project successfully. It involves checking the version requirements of the project, comparing them with the available versions of the dependencies, and selecting the most suitable ones for installation. Package managers simplify dependency resolution by providing a central repository of third-party packages that can be automatically installed, upgraded or removed. Package managers select the most suitable versions of the dependencies to install and resolve any conflicts that may arise, thus reducing the risk of broken builds or missing dependencies. Furthermore, package managers rely on security advisories and databases to detect and prevent the installation of packages with known security vulnerabilities. In this way, package managers ensure the stability and reliability of project code.

Understanding and managing the complex ecosystems that are created by large package managers can be challenging. For instance, in March 2016, a coder intentionally removed a small utility package, called the `left-pad` package, from JavaScript's central online repository. The unexpected removal of the popular open-source package caused widespread software breakage throughout the JavaScript ecosystem, breaking over five thousand transitive dependents (Decan et al., 2019).

A *transitive dependency* is defined as an indirect dependency that a package has on another package through one or more intermediary packages. Due to their indirect nature, they are often not visible or directly managed by coders. The "left-pad incident" revealed the intricate network organisation of JavaScript's ecosystem and, more importantly, its vulnerability due to its substantial interconnectivity through transitive dependencies.

1.2. COMPLEX NETWORK SCIENCE

To address these challenges, complex network science provides a valuable approach to understanding the structure and behaviour of package manager ecosystems. This recent branch of science aims to uncover the principles behind networked systems' formation, evolution, robustness, and adaptability in response to changing environments. It is an interdisciplinary field with applications in physics, mathematics, biology, economics and sociology (Durlauf and Blume, 2008; Newman et al., 2008).

Surprisingly, network science has revealed that a universal set of fundamental laws and principles may govern the structure and evolution of different networks. For example, the underlying structure of the World Wide Web and the interactions between proteins in yeast are governed by the same network concepts, such as scale-free distributions and small-world properties (Dorogovtsev et al., 2003). These features make complex networks different from random or simple networks, which have a more uniform distribution of connections.

Formally, a network is a system of interconnected nodes, where each *node* represents an entity, and each *edge* represents a relationship between two entities. In a directed network, each edge has a direction of effect, represented as an arrow from the source node to the target node. Conversely, the connections between nodes are symmetrical in undirected networks. Directionality is a critical aspect of many real-world networks, including package managers, as it captures the direction of relationships and interactions between entities. The directionality of a network can have important implications for the structure and function of the network.

The way in which the components of a network are interconnected is a network's topology. For example, tree topology is when nodes are organised in a hierarchical structure with a central node and multiple levels of child nodes branching out from it. Star topology is when all nodes are connected to a central node, which acts as a relay for communication between nodes, and scale-free topology is a type of network structure characterised by highly connected nodes, called hubs, and many poorly connected nodes. The topology of a network is a critical aspect to consider when understanding its function, as it influences the efficiency of the network's performance. As a result, topological analysis has become a well-studied area in the last decade. Researchers have leveraged new data from various fields to gain a deeper understanding of network topology (Barabási et al., 2000; Dorogovtsev et al., 2003). New techniques in graph theory have been developed to analyse the large and complex networks generated by this data.

By analysing the relationships between packages in package managers, complex network science can reveal important insights into the functioning of package manager ecosystems and highlight ways to improve their stability, reliability, and maintainability.

1.3. THE NPM ECOSYSTEM

The npm (Node Package Manager) ecosystem is one of the world's largest and most widely used open-source software ecosystems. It serves as a repository of software packages for the Node.js platform, providing JavaScript developers with a vast array of tools and libraries for their software development projects. JavaScript is one of the foundational pillars of web development.[1] It is a programming language widely used for creating dynamic and interactive web pages, and thus, one of the key reasons for the npm ecosystem's popularity. Reusing open-source npm packages allows for significant time and cost savings, improved code quality, increased functionality, and access to a supportive developer community (Zimmermann et al., 2019). The npm ecosystem has proliferated in recent years, with millions of packages available on its registry with billions of total downloads, making it an important area of research.

The npm ecosystem has been the subject of limited research, and previous studies have focused on analysing package popularity, distribution of packages, and identification of popular packages(Decan et al., 2019; Kikas et al., 2017; Wittern et al., 2016; Zimmermann et al., 2019). Decan et al., 2019 compared several package ecosystems, including npm. Their findings showed an unequal distribution of connectivity across each ecosystem. The authors conclude that open-

[1] It's important to note that there are other technologies and programming languages used in web development as well, such as server-side languages like PHP, Ruby, and Python, and front-end frameworks like React, Angular, and Vue.js. The choice of technologies depends on the specific requirements of a project.

source package dependency networks may exhibit complex network behaviour, and they identify the analysis of these networks using complex network science as a potential area for future research. This thesis extends this body of work by conducting a comprehensive analysis of the npm ecosystem as a complex network, taking into account the dependencies between packages at a version level and the underlying structure of the network.

1.4. OBJECTIVES

This thesis focuses on the topological properties of npm's package manager ecosystem. I analyse how npm's topology and complexity evolved from the beginning of 2012 to the end of 2016 using data from the publicly accessible repository librairies.io (Nesbitt & Nickolls, 2017). I analyse the npm package manager ecosystem along three dimensions to better abstract and characterise JavaScript's software development process at a version level. First, I analyse the structure and complexity of the npm package dependency network by investigating the presence of scale-free and small-world qualities. Second, I reveal assortative mixing patterns based on node degree and explore the implications of large-scale network parameters of software design. Finally, I describe how complex network analysis can be used to identify packages vulnerable to targeted attacks which could lead to large-scale perturbation.

1.5. APPROACH & PRINCIPAL CONTRIBUTIONS

In this thesis, I map out the npm dependency network using static dependency graphs. Static dependency graphs represent all possible package interactions, whereas dynamic dependency graphs capture interactions that occur during program execution.

Previous software system studies generally adopt an aggregated approach when constructing dependency graphs. However, this thesis proposes that each version of a package should be considered a unique entity. One of the reasons is that versioned packages have different dependencies and functionalities compared to other versions of the same package. To account for all versioned dependency relations, I developed a set of rules and functions in line with npm's semantic guidelines such that the SemVer expression associated with each versioned package is interpretative. Semantic Versioning, or SemVer, is a convention widely embraced by developers in both open-source and proprietary software communities. It serves as a versioning schema that clearly delineates the evolutionary states of a software package. By applying this framework, this thesis successfully generates a refined edge list that captures the version-specific dependencies, facilitating the construction of an accurate directed network graph. This graph not only reflects the complex web of package interdependencies but also aligns with the dynamic nature of soft-

ware development, where each version release can significantly alter the dependency landscape.

I adopt a versioned approach for three reasons. First, the aggregated approach overstates the number of dependencies. Second, npm has a unique way of handling dependency resolution, such that a JavaScript project's directory structure mirrors the dependency tree. Thus, the version approach accounts for the nuance in how npm conducts dependency management. Lastly, this thesis supports the notion that package versions are a central consideration regarding software system evolvability. The need to accommodate change is a major driving force in open-source software development, and versions encapsulate the software demands of enhanced functionality and evolvability. Therefore, by studying the npm ecosystem at a version level, a more granular and accurate representation of the dependencies between packages can be established.

The following complex network analysis methods and techniques are employed: graph theory, assortative mixing, centrality measures, and network visualisation. The findings show that the npm network has scale-free and small-world characteristics. However, the strength of these properties varies over time for both in-degree and out-degree distributions. The scale-free implications for npm show that a small number of hubs have a disproportionately important role in providing functionality to the rest network.

Furthermore, the results indicate that the value of the scale-free exponent, and the deviations from the power law, provide valuable information on the mechanisms driving the underlying formation of the npm network. The complex network analysis demonstrates how the hierarchical nature of software design plays a fundamental role in shaping the overall network topology.

1.6. THESIS STRUCTURE

The rest of the thesis is organised as follows:

Chapter 2 provides a comprehensive review of the literature on complex network science and its application to open-source software ecosystems.

Chapter 3 describes the methodology used to collect and analyse data on the npm ecosystem, with a specific focus on network construction.

Chapter 4 presents the results and provides a discussion of the complex network analysis of the npm ecosystem.

Chapter 5 concludes this thesis with a summary of the main findings and their implications and suggestions for future research.

1.7. CONCLUSION

In conclusion, this thesis aims to advance the understanding of the npm ecosystem as a complex network by comprehensively analysing its structure and dynamics. These findings have the potential to contribute to the development of effective strategies for managing and improving the npm ecosystem and to provide insights into the structure and evolution of other open-source software ecosystems.

2

BACKGROUND

2.1. Introduction

This chapter explains several key concepts and examines previous research that has applied dependency modelling and complex network analysis techniques to open-source software systems. The concepts introduced are necessary for understanding; object-oriented programming, package management, cascading failures through dependency relations, and complex network science—the basic subject matter of this thesis.

2.2. Object-Oriented Paradigm

According to Luna & Stefansson (2012), the process of constructing a mechanical model of a material system using concrete components is analogous to creating an algorithmic model of a physical system using software objects. Under the programming paradigm, *object* refers to a component of a software model. An object is a data structure containing variables and functions. Variables determine the object's state, while functions define the object's behaviour (Luna & Stefansson, 2012). Functions, also known as *methods*, provide instructions regarding how an object should respond to input, changes in its state, or interact with other objects.

Definition: *A program is a set of object definitions and rules determining how objects interact.* [1]

Developers store object definitions in text files written in a readable programming language. Text files are a universal format for storing source code and can be easily edited, version-controlled, and shared between developers. [2] The state of the object, represented by variables, and its func-

[1] Program definition program (Luna & Stefansson, 2012).

[2] Text files make it convenient for multiple people to work on the same codebase, collaborate on a project, or use code from one project in another project.

tions (*methods*) are defined in a code template known as a *class*, which determines the object's behaviour. These text files are then translated by a compiler into machine instructions, which the computer's microprocessor can execute. During the execution of a program, the computer designates a portion of the Random Access Memory (RAM) for storing the variables and methods associated with a particular object. This process creates an instance, an object bearing a particular value that only exists during a program's execution. In other words, an object is an instance of a class that exists in the memory of a computer system. In practicality, instances may possess the same data structure. However, they do not occupy the same memory allocation.

Object-oriented programming (OOP) dominates the modern software development paradigm because its methodology is exploitable in a wide range of applications (Abadi & Cardelli, 2012). These include the construction of user interfaces, operating systems, and databases.[3] The design resilience of OOP software models is one property that confers this broad range of applicability. Objects create data abstraction boundaries, allowing coders to focus on the logical structure of the design rather than algorithm development. An algorithm can be implemented as a *method* in a class, allowing it to be encapsulated within an object, thereby allowing the objects to be behaviourally autonomous (modular) (Abadi & Cardelli, 2012). This is a key benefit of object-oriented programming, as the complexity of writing code is greatly reduced, and designs can develop without pervasive reorganisation.

Another beneficial property refers to the reuse mechanisms unique to object-oriented programming. When a software component can be used in multiple contexts, it is considered reusable. Reusability can be improved provided the time saved in functional specification outweighs the time lost in learning, memorising, researching and developing (Brooks, 1978). This ratio of gain to cost has reduced significantly as more complex functions have been implemented into the system of modern programming.

Inheritance, a fundamental concept in OOP, facilitates reuse and prevents code redundancy. It is a mechanism which allows a class to inherit all the methods and attributes of another class, thus saving coders from replicating instructions and attributes. Notably, inheritance allows developers to build functionality onto existing classes. When a new object is created through inheritance, generally referred to as a *child-object*, the coder can add new functional layers with minimally restraining specialisations while retaining the same features and behaviours of the *parent-object*. The hierarchical relationships generated through inheritance give rise to a directed acyclic graph.

[3]User Interface (UI) refers to the interaction between the user and a computer system in a device,i.e. the appearance of a desktop. An operating system is a software system which manages the computer's memory, processes and hardware.

Although JavaScript is not a fully object-oriented language, it provides many key features and design patterns needed to write code in an object-oriented style. JavaScript supports OOP features such as encapsulation and inheritance. Polymorphism, another key concept in OOP, allows npm packages to provide a common interface for different implementations, allowing for a high degree of flexibility and adaptability in how packages are used and combined. Moreover, the properties and methods can be added to objects dynamically, and objects can inherit properties and methods from other objects. The principles of OOP provide a useful framework for understanding the npm ecosystem and its various components, as well as its strengths and challenges.

2.3. PACKAGE MANAGEMENT

Today, one of the most popular and significant forms of software reuse is open-source software (OSS) libraries. A *library* is an object used to group related objects. For example, the modules "add", "subtract", "multiply", and "divide" would be included in the "calculator library".[4] It is a directory used to call a group of objects. Similarly, a *package* is a compiled form of the library which consists of meta-data such as timestamps and version number, source code, distribution code, executables, documentation, examples and test suite. A package manager stores all the versions of the package in a compressed form on its online registry. Note that the terms library and package are interchangeable; however, I use the term package for this thesis.

A package manager is included with every widely-used programming language: npm for JavaScript, PyPI for Python, and CRAN for R. Package managers offer similar functions with the purpose of assisting the respective developer communities of a specific programming language (Zimmermann et al., 2019). By hosting thousands of third-party packages on central repositories, developers can freely access a rich pool of building blocks for software projects. Moreover, package managers automate and streamline the installation, upgrading, configuring and removing packages.[5] Therefore, as a distributor, a package manager's main purpose is to facilitate software reuse.

2.4. THE TRANSITIVE TRAIL OF DEPENDENCIES

This extensive reuse of existing software packages results in complex dependencies between software packages, best described as a dependency network. The npm network consists of a

[4] A module refers to an isolated piece of code that performs a distinct task.

[5] Ian Murdock stated that package management technology was *"the single biggest advancement Linux brought into the industry."*

set of versioned packages (*nodes*) and a set of dependency relationships (*edges*) between the versioned packages. Even though many direct dependencies may exist between packages, it is the dependency relations which stem from transitive dependencies that significantly expose the network to vulnerability.

Zimmerman et al.(2019) discovered that the average npm package installation involves an implicit trust in approximately 80 other packages by a coder. Since npm automates the dependency resolution process, coders are not always aware of any given package's transitive dependencies. Thus, transitive dependencies can be considered hidden risks that may cause software projects to break due to increased complexity, exposure to security vulnerabilities, compatibility issues, and maintenance issues (Decan et al., 2019; Zimmermann et al., 2019). Therefore, dependencies — both indirect and direct — are possible contagion mechanisms. This is because any error, bug, or removal of a package may impact the entire dependency chain: the initiation of a cascading failure, which leads to widespread software breakage in the package manager ecosystem.

2.5. Cascading Failure

Cascading failure is a common phenomenon in complex networks, previously observed in power grids, transportation networks, and financial networks (Georg, 2012; Hines et al., 2009; Yang et al., 2017). A frequently cited example of cascading failure is the 2008 Global Financial Crisis, initiated by a credit crisis in the US. A few missed mortgage payments and property foreclosures snowballed into an event which paralysed the global economy, leaving a trail of failing banks, businesses and bankrupt states in its wake (Elliott et al., 2014). Similarly, in 2003, a software bug at a power station in Ohio led to overloaded power lines drooping into the foliage. This resulted in the collapse of the entire regional electricity distribution network of the US Northeast, causing approximately 50 million people in the United States and Canada to be enveloped in darkness (Barabási, n.d.).

These events illustrate how small failures within a network can unexpectedly lead to catastrophes. Importantly, complex network science shows that the scale of a cascading event is determined by the initial failed nodes' position and capacity in the network. To mitigate cascading failure, one must understand the underlying structure of the network and uncover the interaction patterns between nodes.

Small-world and scale-free qualities fundamentally change a system's behaviour. Small-world properties are when a network exhibits a short average path length and a high clustering coefficient. They make a network more efficient as information can flow quickly throughout it, and

nodes tend to form highly connected groups (Goyal, 2009). From a systemic risk viewpoint, however, small-world properties can render networks more susceptible to cascading failures than random or simple networks(Georg, 2012).

Scale-free networks have power-law degree distributions and are robust to random node failure, possibly explaining why scale-free properties are frequently found in numerous evolved networks in biological systems (Dorogovtsev et al., 2003). Some nodes are more central and interconnected in scale-free networks than others. These nodes are referred to as hubs and play a crucial role in network stability. However, evidence shows the network will likely experience cascading failure when hubs fail (Barabási, 2009).

Typically, greater interconnectivity leads to significant non-local effects, whereby failure or contagion can spread throughout the network regardless of the distance from the origin (Barabási, n.d.). Therefore, by studying the structure, patterns of interactions and relationships in the package manager ecosystem, researchers can gain greater insight into the mechanisms behind cascading software failures and develop strategies for mitigating them.

2.6. RELATED RESEARCH

Myers(2004) examines open-source software collaboration networks using call graphs. Using network science, the author observed scale-free and small-world phenomena. Myers(2004) argues that software systems' structural complexity is due to the need to support evolvability. Šubelj and Bajec (2012) argue that networks relate to the information flow between the open-source software components and also coincide with the comprehension of object-oriented systems. They use class dependency graphs to address the inter-class structure of Java software projects. Their findings show that the class networks exhibit small-world and scale-free properties. The authors conclude that "seed" nodes—nodes with high incoming links—can determine the vulnerability of an ecosystem to bug propagation.

The software systems methodology has been expanded to encompass the examination of software package networks. Zheng et al.(Qin et al., 2008) developed growth models for the Gentoo Linux distribution system using packages as nodes and edges as dependencies. Zheng et al.(Qin et al., 2008) observed that the primary challenges in applying complex network analysis to software systems are the collection of data and the absence of suitable models and tools. Mora-Cantallops et al. (2020) empirically examined CRAN (the R package manager ecosystem) employing complex network analysis. Using dependency graphs, the results indicate that the CRAN network exhibits small-world and scale-free qualities. Mora-Cantallops et al.(2020) iden-

tify that an important dimension of vulnerability is missing from the study as their analysis does not account for package versions.

Mora-Cantallops et al.(2020), Myers(2004), and Zheng et al.(Qin et al., 2008) find a negative correlation between nodes with incoming links (reverse dependencies) and nodes with outgoing links (dependencies). However, Myers(2004) extends his correlation analysis to include assortativity. In network science, assortativity describes the extent to which nodes tend to form links with other nodes with similar properties, such as degree (number of connections), node type, or other attributes. Understanding assortativity aids the classification of networks, provides insights into the design and evolution of networks, and provides useful information for developing defence or attack mechanisms (Dorogovtsev et al., 2003). For instance, in epidemiological networks, the progression of infection can be contained by removing crucial nodes based on their assortative nature (Barabási, n.d.). Myers (2004) argues assortativity is a useful analysis for understanding the structure and dynamics of open software collaboration networks.

Kikas et al.(2017) employed a versioned approach for a dependency network analysis of JavaScript, Rust and Ruby. Their findings suggest that the average size of dependencies in JavaScript increases by 60% annually. Moreover, the authors argue that in order to study transitive dependencies and the role they have in the network structure, project versions should be accounted for. Zimmerman et al.(2019) performed a comprehensive investigation of security risks arising from the densely interconnected structure of npm packages and the implicit trust placed in maintainers. Due to significant dependencies, the authors characterise the npm as a 'small world' ecosystem. However, this conclusion is not based on complex network metrics. The authors conclude that the npm ecosystem is associated with high-security risks as malicious/vulnerable code in a single package may greatly impact others. Decan et al. (2019) analyse the differences and similarities between seven packaging ecosystems, including npm for JavaScript, over a period of five years. Their findings indicate that the npm network at an ecosystem level has a high number of transitive dependencies relative to other ecosystems, implying that package failures can have a substantial impact on the ecosystem. As stated before, they proposed that applying complex network analysis to open-source software ecosystems would be a worthwhile task.

2.7. CONCLUSION

Overall, there is a rich research domain on open-source software ecosystems with an emphasis on dependency network modelling. However, to the author's knowledge, the application of complex network science techniques to the npm ecosystem is limited. Inspired by Kikas et al.(2017) and Decan et al.(2019), this thesis addresses the gap in the literature by mapping the

npm ecosystem using a versioned approach and applying complex network analysis. This thesis focuses on three research questions:

RQ1: Evolution How did the npm ecosystem grow between 2012-2016?

RQ2: Structure & Complexity Does the npm ecosystem exhibit scale-free and small-world behaviour? What are the underlying mixing patterns between nodes in the npm network, and how have they been influenced by software design?

RQ3:Vulnerability What are the most critical versioned packages in the network?

The next chapter provides a framework for understanding how the npm ecosystem can be analysed at a version level. The remainder of Chapter 3 describes the methodology employed for data collection and network construction.

3

METHOD & DATA

3.1. Introduction

Dependency graphs play a crucial role in software development as they help to understand the relationships between different components of a system. In particular, the npm dependency resolution process is an essential aspect of managing dependencies in a software project. In this chapter, I provide a detailed description of the methodology used to construct a network representation of npm packages and their dependencies at a version level. This chapter provides a comprehensive overview of the framework for versioned dependency graphs, network theory, network construction, data, and complex network techniques to address the research questions.

3.2. Dependency Graphs

A reliable and healthy software ecosystem requires packages to be frequently updated. Semantic Versioning notation (SemVer) is a widespread practice adopted by coders in closed and open-source software environments. It is a versioning system which differentiates between unique states of the package. SemVer uses a sequence of three digits in the format `x.y.z`, where the digits stand for `MAJOR.MINOR.PATCH` respectively. Here, an incremental change in the `PATCH` number indicates a bug fix, and an increase in the `MINOR` number denotes an update that introduces new features. `PATCH` and `MINOR` modifications are backwards compatible, while an increase in the `MAJOR` indicates incompatible API changes. A `MAJOR` update represents a change in the whole mechanism of the package and introduces changes that may cause the previous version code to break.

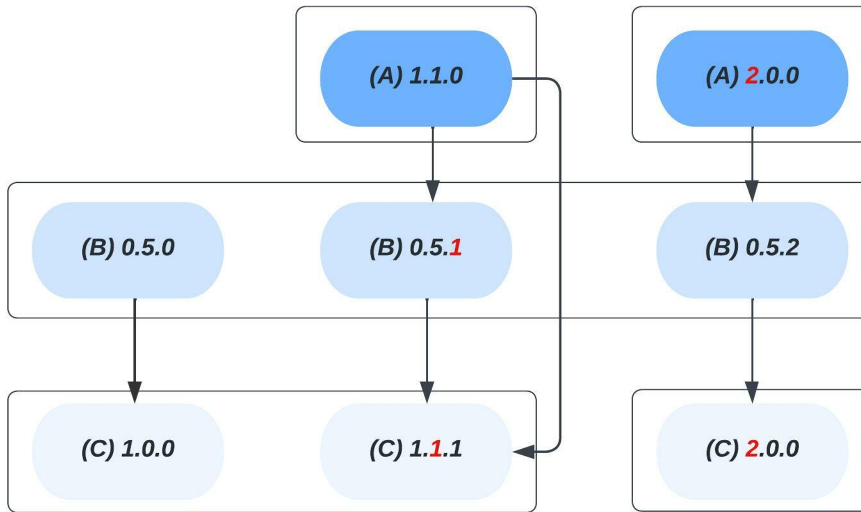
Like the interpretation of decimal numbers, semantic version numbers of packages are first compared by the magnitude of their MAJOR type, then by their MINOR, and finally by PATCH types. For instance, version 2.2.1 is lower than versions 3.0.0 (by a MAJOR), 2.3.1 (by a MINOR) and 2.2.2 (by a PATCH) but greater than versions 1.2.1(by a MAJOR), 2.1.1(by a MINOR) and 2.2.0 (by a PATCH). Additionally, optional tags are available for coders to specify pre-releases and unstable packages that are under development and still within the testing phase. Therefore, semantic versioning allows the package developer to signal to coders what modifications have been made to the underlying code of an existing package.

The relationships between software components are complex and varied, and any depiction of a software system is merely a simplified view of those intricate interactions (Myers, 2004). Dependency graphs describe the calling relationships between packages. They serve as a blueprint for understanding the structure of a software system. Static graphs describe a set of *possible* interactions, whereas dynamic graphs represent interactions generated during program execution and outline interactions that occur under certain runtime conditions. This thesis focuses on static dependency graphs arising in the npm ecosystem. **Figure 3.1** below illustrates an example of a static dependency graph for three packages, "A", "B", and "C", with different versions following the semantic version notation. Package (B) 0.5.0 directly depends on (C)1.0.0. Package (A)1.1.0 directly depends on (B) 0.5.1 and indirectly on (C) 1.1.1, since (B) 0.5.1 directly depends on (C)1.1.1. Then there is a MAJOR refactoring where (A) 2.0.0 directly depends on the PATCH-updated package (B) 0.5.2 and indirectly on (C) 2.0.0, which has independently undergone a MAJOR refactoring.

Another important function of semantic versioning is that it allows developers to specify the dependency requirements of their packages to their third-party users. This enables packages to benefit from compatible updates while preventing backwards incompatible ones. Packages may be compatible with a range of dependency versions. Hence, for dependency requirement specification, a notation is employed for describing a valid version. To obtain any version or the latest version of a package, the requirement is specified with the **wildcard** (*) or with an explicit condition (≦ /0). While the **caret** (^) matches with the latest MINOR version available with its highest PATCH version. The **tilde**(~) will select the most recent PATCH version.

Notably, the package resolution of versions is time-dependent. When dependency resolution is initiated, under the conditions of the specified constraint, the package manager automatically resolves the latest version of the package available in the central repository at the given time. For retrospective analysis, this suggests that results derived from the static dependency graph may be fragile. For instance, consider that the project "Hello World" depends on package A (from **Figure 3.1**) with a wildcard (*) version constraint. Recall a wildcard constraint means that

Figure 3.1: A dependency graph of three packages ("A", "B", "C") with different versions following semantic versioning notation



Notes: Packages with the same MAJOR number have the same API, indicated by a box. Arrows denote directed dependencies between different versions of packages.

the package is compatible with all possible versions of that dependency package. However, only one version can be called, so the package manager resolves the dependency to its latest version for optimal functionality. If package A (`1.1.0`) is created at the time $(t-1)$ and package A (`2.0.0`) is created at the time $(t+1)$, then the dynamic dependency graph for the execution of the project "`Hello World`" at time t will only demonstrate the dependency relations for package A (`1.1.0`). Dynamic analysis is beyond the scope of this thesis. Therefore, all possible interactions between dependencies are considered.

To prevent packages from breaking from a dependency update, the npm framework allows a coder to declare dependencies in a configuration file that states the name of the dependency package and a version constraint. Coders can choose to lock their direct dependencies to a fixed version or range of compatible versions using a '`package-lock.json`' file or allow dependencies to resolve to the latest release available. In other words, a coder may choose between the uniform installation of the packages associated with her code or up-to-date dependencies. Developers gain greater runtime predictability by locking in fixed versions and forgoing automated critical fixes in later versions. However, there is the caveat that a dependency may declare further dependencies —transitive dependencies— using ranges. Thus, projects are not necessarily guaranteed to be deterministic even when the coder declares fixed version dependencies.

In contrast, locking version ranges has the downside that projects can fail as changes between versions may introduce backward incompatibility. On the upside, locking a version range allows for package resolution flexibility. Thus, declaring version ranges enables bug fixes, facilitates security vulnerability mends and enhances performance. This is automatically deployed whenever dependency resolution takes place for a project. The combination of dependency constraints and semantic versioning eases the management of dependency updates for package maintainers. Unfortunately, package maintainers may voluntarily or not break the associated syntax (Raemaekers et al., 2014).

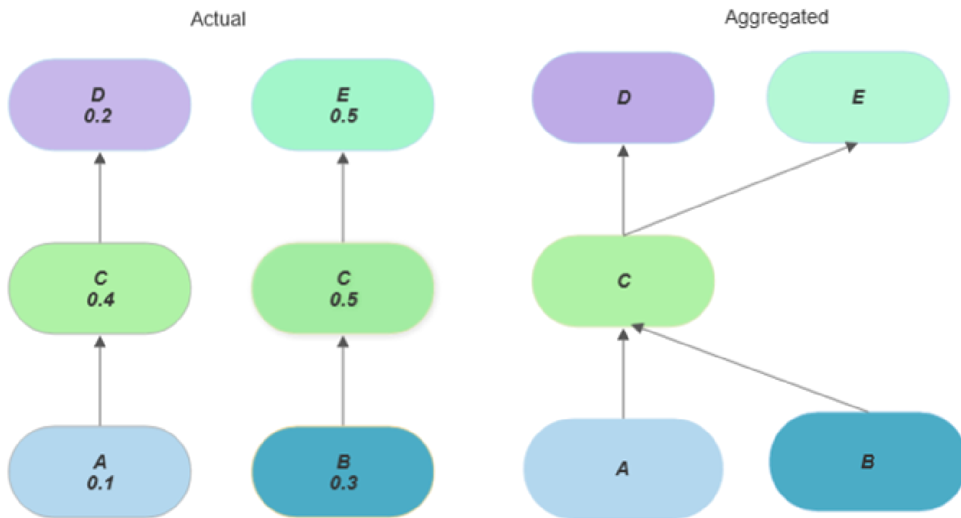
3.3. NETWORK CONSTRUCTION

3.3.1. THE VERSIONED-DEPENDENCY MODEL

This thesis adopts a versioned-level approach for defining the nodes and edges for network construction. This method is motivated by three factors. First, Kikas et al. (2017) suggest using the actual dependencies in software network construction where each node is a specific version of the same package, and links are actual dependencies between these versioned programs. The other possibility would be to treat each package as a node which subsumes all versions of the same package, thereby aggregating all dependencies between different versions of the same program. However, this method overstates the number of dependencies (Kikas et al., 2017). **Figure 3.2** below illustrates the aggregated approach versus the versioned approach for network modelling dependencies (Kikas et al., 2017). Package C is a dependency for packages A and B. Still, only package C version `0.4` depends on package D. Thus, the aggregated network model indicates that package D is a dependency for A, C and B, which is not valid.

The second factor relates to npm's novel approach to dependency management. Most package managers, such as RubyGems or pip, only allow one package version to be installed simultaneously. The project manager will raise a runtime error when a project includes packages with similar dependencies yet conflicting versions. In this case, the package manager's primary responsibility is to determine a set of package versions that will simultaneously satisfy every dependency constraint for a given project (King, 2016). This is commonly referred to as *dependency hell*, as solving one dependency for one software component may subsequently break other dependencies. Unlike these package managers, npm offers a simpler solution for the issue of dependency conflict package by installing a tree of dependencies. Notably, every versioned package installed is assigned its own set of dependencies rather than forcing them to share the same canonical set of packages. In turn, the directory structure of a JavaScript project closely mirrors the actual dependency tree, where each transitive dependency would have its module directory. Using a version-level approach, this thesis accounts for the nuance in how npm conducts dependency

Figure 3.2: Dependency Network Construction Approaches



Notes: There are six programs ("A", "B", "C", "D", "E") where the actual approach differentiates programs by versions, and the aggregated diagram subsumes both versions of "C", thereby overstating the dependencies associated with program "B". (*Image source from Kikas et al., 2017*)

management.

Lastly, dependency graphs that account for the relationships between versioned packages may effectively encapsulate variability and change. By disregarding versions through aggregation, one forgoes the opportunity to analyse the unique configurations that occur within software systems. This thesis argues that package versions are a central consideration regarding support in software system evolvability. Therefore, a crucial question is how fine-grained analyses on dependency relations between versioned packages contribute to forming large-scale software networks.

3.3.2. NETWORK THEORY AND NOTATION

Consider a software system consisting of $\mathscr{N} = (1, ..., n)$ versioned packages with $N = |\mathscr{N}| \geq 3$, where each package is developed and maintained by a different coder. A coder decides to implement all code for a package herself or to reuse existing code from the repository to build upon her package. This reuse of code gives rise to linkages between packages. This results in a network of packages and dependencies. Formally, a network graph of the npm ecosystem is defined as $\mathscr{G} = (\mathscr{N}, \mathscr{E})$, which consists of a nonempty set of versioned packages $\mathscr{N}$ and a set of ordered

pairs of dependency relations between the versioned-packages $\mathscr{E}$ called edges. $\mathscr{G}$ is analysed as a binary, directed graph where $g_{ij} = 1$ if $(i, j) \subseteq E$ is an existing link from versioned-package i to versioned-package j, indicating that i depends on j.

The interconnectedness and significance of a node can be defined by the in- and out-degree of the node. Denote $g = g_{ij}$ the resulting $\mathscr{N} x \mathscr{N}$ adjacency matrix $\mathscr{A}(\mathscr{G})$ are 1 if there is a dependency between i and j, and zero otherwise. The in-degree $k_{in}(i)$ and out-degree $k_{out}(i)$ of a node i are defined as:

$$k_{in}(i) = \sum_{j=1}^{n} a_{ji} \tag{3.1}$$

$$k_{out}(i) = \sum_{j=1}^{n} a_{ij} \tag{3.2}$$

In the npm network, the in-degree k_{in} of node i indicates the number of packages which call it (reverse dependencies). While the out-degree k_{out} represents the number of packages required for node i to function (dependencies). The size of node i in terms of the value of in- and out-degree defines its interconnectedness within a directed graph as $\mathscr{G} = (\mathscr{N}, \mathscr{E})$. The average in- and out-degree are equal for directed graphs.

TRANSITIVE CLOSURE

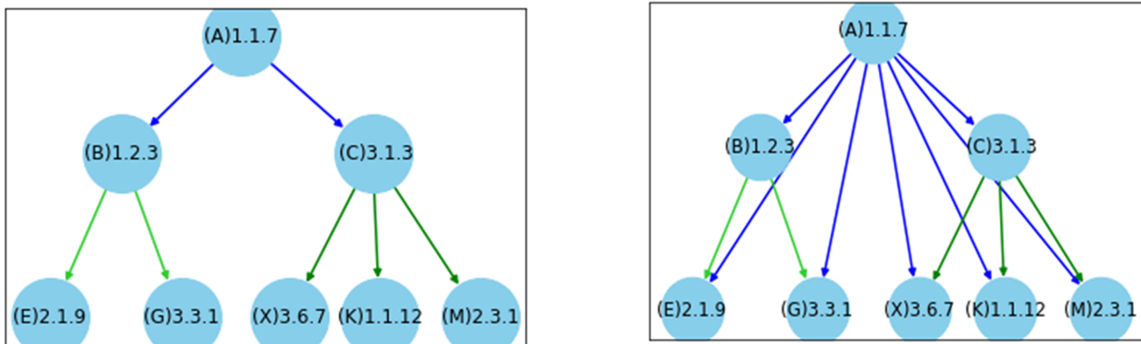
The transitive closure(TC) of a network is a concept used in graph theory and refers to the reachability of nodes in a graph. In other words, it represents the smallest set of edges such that there is a path from each node to every other node in the graph. In a directed graph, the transitive closure can be obtained by computing the transitive reduction, which is the smallest graph that preserves all the reachability information of the original graph. Below, the transitive closure is defined:

The transitive closure of $\mathscr{G} = (\mathscr{N}, \mathscr{E})$ is a graph $\mathscr{G}+ = (\mathscr{N}, \mathscr{E}+)$ such that for all i, j in $\mathscr{E}$, there is an edge (i, j) in $\mathscr{E}+$ if and only if there is a path from i to j in $\mathscr{G}$. [1]

An example of such transformation is represented in Figure 3.1 below. Package (A)`1.1.7` directly depends on two other packages and indirectly on five packages. Alternatively, Package (E) `2.1.9` has one reverse dependency, Package (B) `1.2.3`. In the TC case (graph on the right), Package (E) `2.1.9` has two reverse dependencies, with Package (A)`1.1.7` as a transitive dependent. The transitive closure of the graph allows for all dependencies, direct and indirect, to be considered.

[1] Transitive closure defined by the Python package networkx.

Figure 3.3: Direct Dependencies vs. Transitive Closure



Notes: The network includes eight versioned packages(nodes) and their dependencies(edges). On the left, only direct dependencies are considered, while on the right, both direct and transitive dependencies are considered. Package (A) 1.1.7 has two dependencies in the left graph, but in the TC case Package(A) 1.1.7 has seven dependencies, two direct dependencies and five transitive dependencies. Alternatively, in the left graph, Package (E) 2.1.9 has one reverse dependency, Package (B) 1.2.3. In the TC case, Package (E) 2.1.9 has two reverse dependencies, with Package (A).1.1.7 as a transitive dependent.

3.4. DATA DESCRIPTION

3.4.1. DATA

This paper employs a replicated dataset from an empirical study conducted by Decan et al., 2019.[2] This dataset includes all the metadata on package releases and dependencies in the npm package manager retrieved from the publicly accessible repository libraries.io (Nesbitt & Nickolls, 2017). Data is released under the Creative Commons Attribution-ShareAlike 4.0 International License.[3] The primary analysis sample includes all published packages between the period 1 January 2012 to 1 January 2017. Notably, the metadata for each package includes a unique package ID, a version number, dependency requirements, publication timestamps, as well as version constraints and the history of all version numbers for the sample period. After the data cleaning process, the dataset employed in this thesis has a total of 285,652 unique packages. For the sample period, this translated into a total of 1,576,947 package versions and 24,020 200 direct dependencies.

One benefit of employing this duplicated data set is that it distinguishes and deliberately disregards packages that cause interference and are not beneficial for the analysis. Within the npm ecosystem, there are five different types of dependencies. These are categorised as production,

[2] Decan et al., 2019 analyse the differences and similarities between seven packaging ecosystems including: Cargo for Rust, CPAN for Perl, Cran for R, npm for Javascript, NuGet for the NET.platform, Packagist for PHP, and RubyGems for Ruby.

[3] See https://creativecommons.org/licenses/by-sa/4.0/

development, peer, optional, and bundled dependencies. This dataset restricts dependencies to only those that are required to install and execute the package. Moreover, dependencies that target packages hosted directly on the web or Git repositories are excluded. This accounts for approximately less than 2.5% of all dependencies in npm (Decan et al., 2019). Dependencies tagged as "alpha", "beta", and "rc" are unstable packages that are not reliable enough to be used in production and therefore are not considered in this network study. These packages account for approximately 3.63% of the dataset.

3.4.2. CONSTRUCTING NETWORK ANALYSIS OBJECTS

The formation of the dependency graph is implemented using an edge list. An edge list is a single data structure representing the connections between data item values in the form of a sequence of interconnected nodes. This thesis constructs the structure for the network analysis by using a *source* data item and a *target* data item. For each value of the source data item, a node is generated, and a connection is established from that node to the node corresponding to the value of the target data item.

For instance, Package A depends on Package B, so Package A represents the source data item, and Package B, as the dependency, represents the target data item. When the same edge connects two vertices, they are said to be neighbouring. Recall that a dependence arises if a project requires certain functionality, and instead of building this functionality herself, the coder decides to use the code provided in another package. Since the network is directed, the edges of the dependency graph represent a list of ordered pairs. Each pair represents a single edge and comprises two unique IDs of the nodes involved: the source ID and the target ID. Thus, this analysis is limited to packages with at least one dependency.

To render an edge list that accounts for all versioned dependency relations, a set of rules and functions were developed to transform SemVer expressions into corresponding version ranges. Thus, the variable of interest for data pre-processing is the constraint variable. In addition, two key variables are constructed called *Version Converted* and *Version Range*. The *Version Converted* variable is a function that transforms each version string `x.y.z` into an integer using the formula below:

$$\textit{Version Converted} = x \cdot 10^6 + y \cdot 10^3 + z \tag{3.3}$$

It is important to note that this method may be flawed as SemVer does not have a size limit on the version string (Preston-Werner, 2021). For instance, it is at least logically possible that three-digit values may exist for major, minor and patch values. This implies that a package with the

following version strings, `2.0.0` and `1.1000.0`, will both convert to 2000000. However, due to the historical reality of npm's software development and the implications of package changes, this study assumes the incidence of this occurrence is low.[4]

A set of rules is developed in line with npm's semantic guidelines such that the constraint variable is interpretative. Using the *Version Converted* variable, an upper and lower bound version value for a required dependency is determined. If a dependency's constraint matches a regular expression in the rules, then the function returns a minimum and maximum integer value, which is stored in the *Version Range* variable. A function is performed that filters the list of versioned dependencies to contain only the versions within the compatible range for a given package. A new edge is created for every match and added to the list of ordered pairs. Once all possible dependency versions within the sample are accounted for, source and target node identifier rules are applied to the dataset. In turn, the constraint variable is expanded to construct a filtered edge list of versioned dependencies.

Several issues arise in constructing an edge list due to anomalies in the data. These anomalies are described below, starting with the generation of version ranges. When the SemVer rules did not apply to the given constraint due to inconsistent semantic notation, manual transformations were performed to produce the approximate regular expression. After that, unclassified dependency versions were removed from the dataset. For example, 3% of packages had versions defined with irrelevant names instead of version strings. While in the final process of constructing an edge list, it was revealed that there were twelve packages which contained more than sixteen version string characters. These data points halt the edge formulating process due to an overflow integer error. Moreover, such version strings do not reflect the meaningfulness of npm's positional notation and were subsequently removed.

The processed edge list consists of the following variables: source-ID, target-ID, package, version, project timestamp, dependency, dependency version, constraint, and dependency timestamp. Pre-processing is a necessary step in storing the data in a relational table. To analyse the evolution of the network, the dataset is parsed over five years, from the beginning of 2012 to the beginning of 2017.

According to Wittern et al. (2016), 80% of packages in the npm ecosystem have at least one direct dependency package. In this thesis, the network only consists of versioned packages that either possess a direct or reverse dependency. In accordance with prior studies, isolated clusters of

[4] Preston-Werner(2021) a cofounder of Github. He states that JavaScript coders should use good judgment as, *"A 255-character version string is probably overkill, for example. Also, specific systems may impose their own limits on the size of the string."*

connected nodes that exist on the network's periphery are disregarded by limiting the npm network to its giant weakly connected component (WCC). In my consideration, I have disregarded weighted links, thereby assuming that all links have a weight of one. Additionally, to reflect the transitive dependencies within the network, the transitive closure of the network is considered.

3.4.3. COMPUTATIONAL COMPLEXITY

The computational complexity of constructing versioned dependency graphs means expanding approximately 13,997,806 edges to include all possible version relations is practically infeasible using an advanced i5-core processor. Thus, computations were performed using the facilities provided by the University of Cape Town's ICTS High-Performance Computing (UCT HPC) team: hpc.uct.ac.za. This allows for computations to be performed across 40 cores simultaneously. Processing constraints into versioned dependencies requires a considerable amount of time. For instance, on average, processing 50,000 aggregated edges using the provided facilities took approximately 10 hours using 40 cores in parallel.

The case in which an abundance of data streamed into any network analysis software is a particular concern as the algorithms may be unable to process the data or even halt the process due to a shortage of storage. Thus, once a versioned edge list is created, the data is parsed into time intervals by year, from 2012 to the beginning of 2017. Packages are restricted to the year they were published, while their dependencies are limited to the year of interest and any period before. This is done to ease computational complexity. Here, the depth of the dependency tree may be limited in terms of transitive dependencies because the network does not account for transitive dependencies published before the year of interest. Since packages published in 2012 may depend on packages created before the sample period, the earliest timestamp for a dependency package is dated from the beginning of 2010.

3.4.4. GRAPH-BASED VISUALISATIONS

The sheer size of the npm network and the frequent release of libraries pose many challenges to constructing a dependency graph. Performing visual program techniques at an ecosystem level to obtain an accurate versioned dependency graph is computationally expensive. Additionally, restricted periods for published packages present an approximate network representation, potentially leading to false interpretations. This paper uses Gephi for graph-based visualisations. As an open form of graph software, it provides easy and broad access to network data. It enhances the visual experience using layout algorithms, navigation and interaction methods, and incremental manipulating and clustering mechanisms. Despite such benefits, there are short-

comings to such techniques. This includes:

1. The difficulties in visualising and comprehending large graphs. For instance, a graph with over fifty thousand nodes causes a performance bottleneck of the platform and decreases the viewability significantly. In general, comprehension and detailed data analysis in graph structures are most accessible when the size of the displayed graph is small.

2. The efficiency of graph layout algorithms may be scaled up to 100,000 nodes and 1 million edges, not beyond that.

3. The time complexity for visualisation, interaction and update of a graph is relatively high and increases with an increase in the graph size.

4

RESULTS & DISCUSSION

4.1. INTRODUCTION

Open-source software systems are engineered to be functional and evolvable(Myers, 2004). These key characteristics suggest particular forms of network organisation. Analysing the degree distributions and node mixing patterns is one way to understand connectivity and the underlying architecture of a complex network. In this chapter, I tackle the subject of the complexity and structure of the npm ecosystem from the perspective of complex network theory. More precisely, by examining the degree distributions, I investigate whether the npm dependency network exhibits scale-free and small-world behaviour. Previous research has indicated that software dependency networks have a power-law nature. However, these studies generally focus on classes or packages instead of versioned packages.

Standard software engineering conventions govern the direction of package collaboration, and thus, preserving edge directionality is imperative for uncovering software network features. For the case of the npm network, the in-degree distribution is equivalent to the degree of package reusability, while the out-degree distribution reflects package complexity. Since mixing patterns are governed by several parameters, including maximum degree, average path length, average degree and the scaling exponent, this chapter is also concerned with assortativity in the npm network.

The remainder of this chapter is organised as follows: In section 4.2, I present the basic properties of the full npm network to illustrate its rich development and evolution history. In section 4.3, I use complex network theory to determine the topological properties of npm's largest weakly connected component and the corresponding transitive closure for each year in the sam-

ple period in an attempt to better abstract and characterise JavaScript's software development process. The implications of these large-scale network parameters in the context of software design are examined. Section 4.4 analyses the value of the most influential nodes within the network based on degree connectivity. Graph-based visualisations are provided where possible to present an appropriate level of network abstraction.

4.2. OVERVIEW OF THE NPM ECOSYSTEM

JavaScript's package manager is a central repository comprising many interconnected versions. While the npm ecosystem appears connected, greater insight can be gained by partitioning the network by time intervals. To begin, I present some basic network characteristics of the entire npm ecosystem to shed light on the network structure and how it changes over time. Marco-characteristics of a complex network refer to the density of connections or the segregation among nodes, while micro-characteristics analyse individual nodes and their relation to other nodes. **Table 4.1** below describes the marco-characteristics of npm's ecosystem. The complex network analysis expands in the following section to assess the degree distribution and degree correlation.

NODES

Since its inception in 2010, the npm ecosystem has evolved from a limited assortment of packages to become the world's most extensive open-source software ecosystem(Zimmermann et al., 2019). **Table 4.1** shows the evolution of the number of versioned packages (nodes) available on npm and the number of direct dependencies (edges) from 2012 to 2016. Since this thesis proposes a versioned-dependency approach, it is important to distinguish the proportion of the observed npm network growth from package release growth versus the growth of unique packages. Recall that a package is a computer program that completes a specific task, whereas a package release describes a particular package version.

Figure 4.1 presents the difference in the growth of unique packages and package releases using cumulative monthly snapshots. I compute the growth of unique packages by accounting for the first stable version of a package while excluding its later versions. The number of unique packages and package releases grows over time, though the magnitude of growth differs. For the sample period, approximately 157 unique packages were published per day on average. The number of package versions is considerably large, with an average of approximately 852 package versions released daily. By the end of 2012, an average package had five versions. This increased to seven by the end of 2016.

The package release growth reflects the popularity of JavaScript and the active contributor com-

Table 4.1: Basic Properties of the full npm network for each year (2012-2016)

Table 4.1					
Variable	**2012**	**2013**	**2014**	**2015**	**2016**
#Number of Nodes	56,316	149,162	333,206	604,663	1,576 947
#Number of Edges	1,060 573	3,450 271	7,875 114	16,245 649	24,020 200
Density	0.0003	0.0002	0.0001	0.0001	0.0000
Weakly Connected Components	272	690	1043	1626	1442
Strongly Connected Components	56,246	148,790	332,362	603,671	1,575 346

Notes: The table includes network measures that indicate the number of versioned packages (nodes), number of direct dependencies(edges), density (ratio of possible to actual dependency relations), and the number of strongly and weakly connected components in the full network.

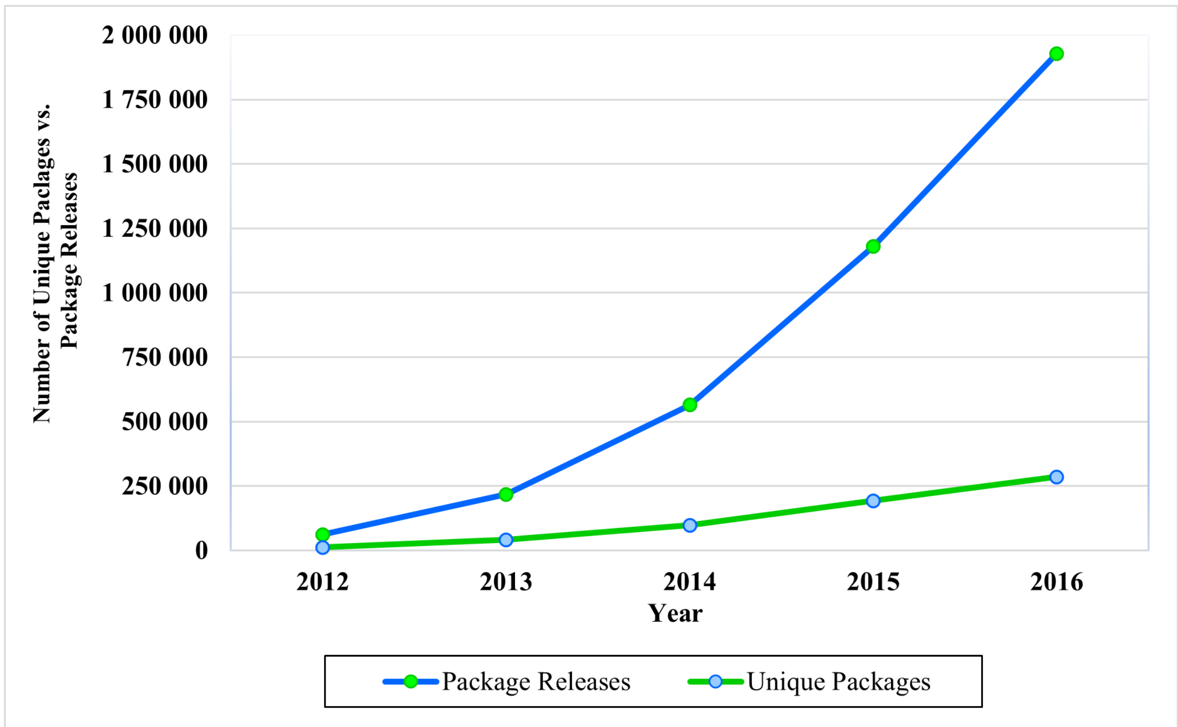
munity of the npm ecosystem. There is an expectation that the number of unique packages steadily increases over time to meet the novel functionality demand of the developer community (Decan et al., 2019). However, the requirement to accommodate change significantly constitutes a substantial motivation in software engineering (Myers, 2004). Therefore, accounting for the modifications to existing packages supports the open-source software ultimata of functionality and evolvability. Version releases prevent coders from committing to poorly constructed code and pervasive code reorganisation. An important question, therefore, is how the version-level software structure conspires in the large to form the macrostructure of the npm ecosystem.

EDGES

The availability of many versioned packages indicates a rich pool of software components available for reuse. **Table 4.1** shows how the number of direct dependencies evolves. The ratio of direct dependencies to versioned packages increased from 18.8 in 2012 to 26.9 in 2015. Interestingly, the direct dependency ratio decreased in 2016 to 15.2, likely due to the changes to the npm ecosystem after the widespread software breakage caused by the removal of the *left-pad* package (Decan et al., 2019). A noteworthy observation is that a directly proportional increase in direct dependencies results in a disproportionately larger increase in transitive dependencies. Recall that a project has transitive dependencies on packages when the direct dependencies require their own set of dependency packages to function. From 2012 to 2015, the number of transitive dependencies of an average package increased from 32.5 to 246.2 before decreasing to 111.9 in 2016.

A coder explicitly declares the direct dependencies of her project. However, she is less aware of the transitive dependencies as they typically resolve transparently. **Figure 4.2** illustrates the ra-

Figure 4.1: The Growth of Unique Packages vs Package Releases over Time (2012-2016)



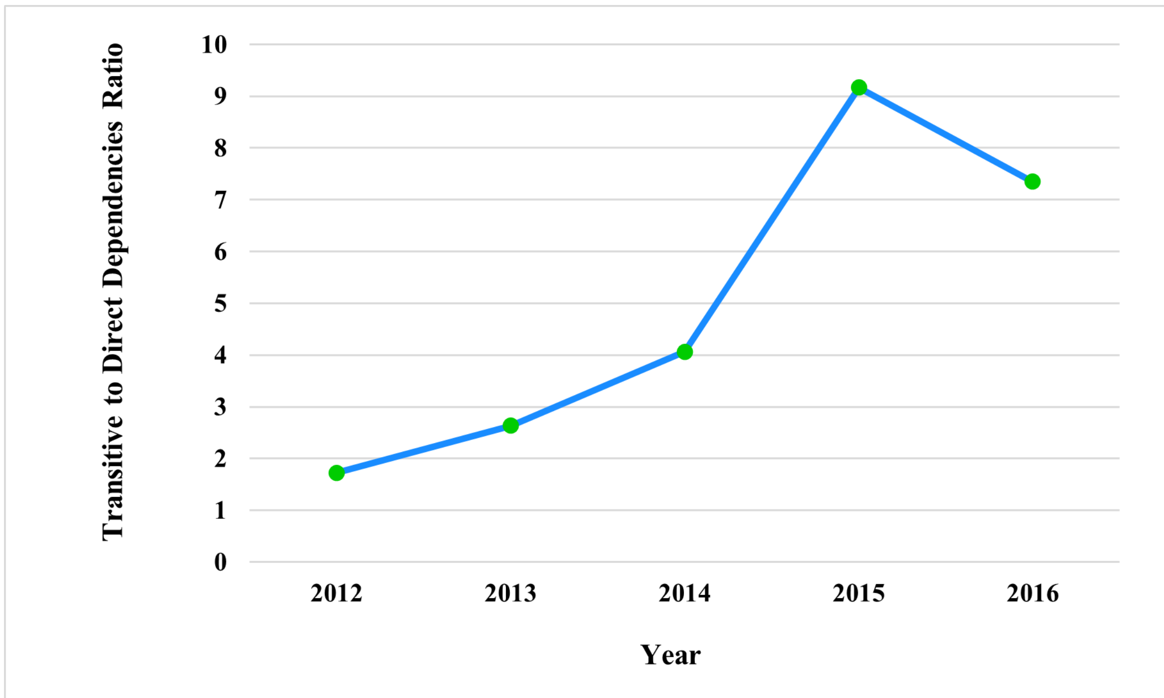
Notes: The growth is calculated using cumulative snapshots for each year of the sample period. A unique package is defined as a newly published package that completes a specific task, while a package release refers to a certain package version.

tio evolution between the total number of transitive dependencies and direct dependencies. For instance, this ratio demonstrates that a coder in 2015, on average, implicitly trusts nine transitive dependencies for every direct dependency included in her project. Transitive dependencies may span through multiple levels of dependencies (Zimmermann et al., 2019). It is imperative to note that the method used to construct the npm network in this thesis excludes transitive dependencies when the required dependency package was published before the year of interest. This measurement error might then artificially decrease the depth of the dependency tree and, therefore, decrease the number of observed transitive dependencies.

DENSITY

Network density represents the proportion between the edges present in a graph and the maximum number of edges the graph can possess. It is one basic macro-metric which captures the notion of diffusion and contagion within a network (Jackson, 2014). Below is the formula used to calculate the density of a directed graph, where n is the number of nodes and m is the number

Figure 4.2: Transitive to Direct Dependencies Ratio Trend over Time (2012-2016)



Notes: The transitive to direct dependency ratio increased from 2012 to 2015. The graph shows a slight downward trend between 2015 and 2016, likely due to the 'left-pad' incident. The ratio is calculated on a cumulative yearly basis.

of edges in $\mathscr{G}$:

$$Density = \frac{m}{n(n-1)} \tag{4.1}$$

Denser networks, in terms of average connections per node, lead to more extensive contagion, all else held equal. The density statistic from **Table 4.1** indicates that the npm network remains sparse over the sample period. This outcome is anticipated in both real-world and software networks (Qin et al., 2008). Furthermore, sparsity reduces the computational complexity of estimations and suggests plausible statistical properties (Kula et al., 2017).

CONNECTED COMPONENTS

An undirected graph's connected component is the maximal collection of nodes that can be reached equally by traversing its edges. While the component structure of directed networks can be divided into weakly connected components (WCC) and strongly connected components (SCC). The connected components in an undirected graph, where paths are assumed to be bidirectional, closely resemble the weakly connected components. Real-world, directed networks typically have one giant weakly connected component that subsumes most of the network, while

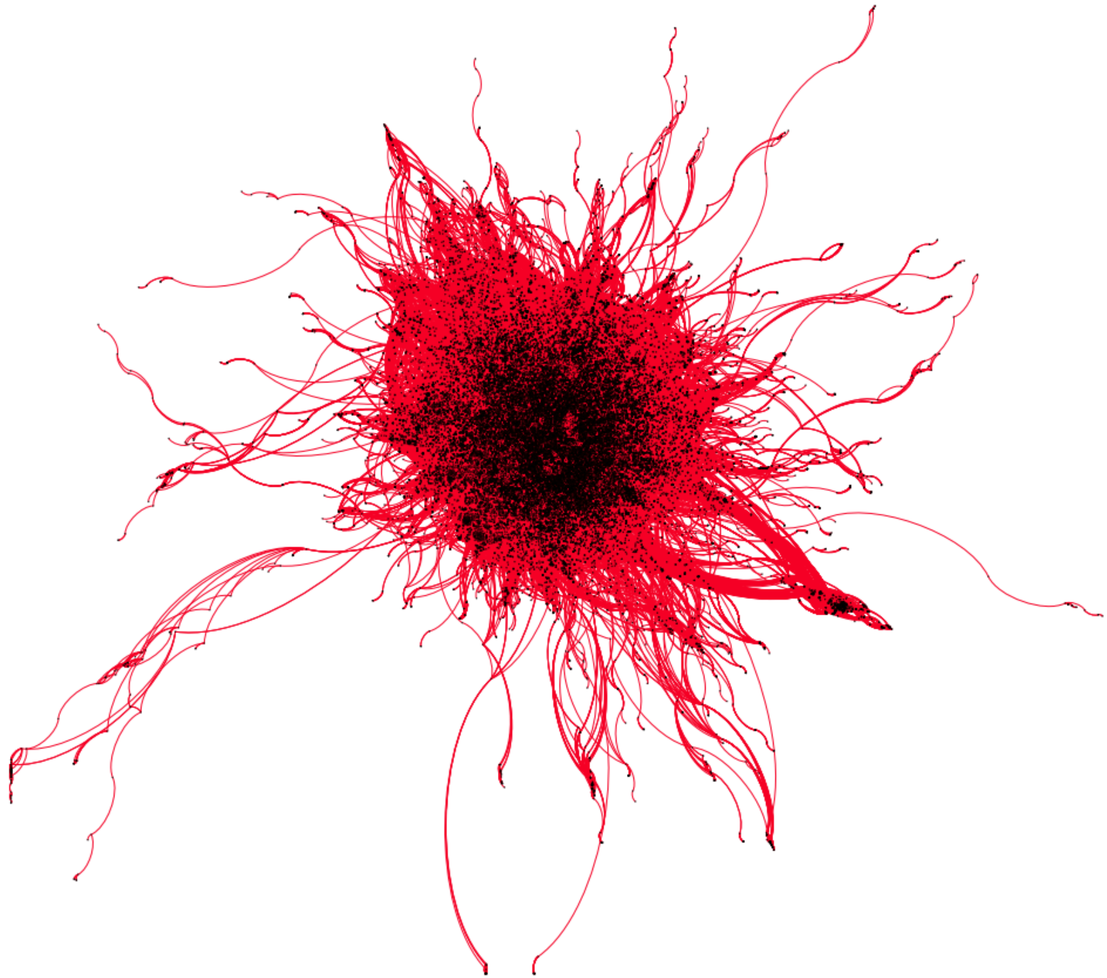
the rest of the network is divided into many smaller disconnected components. Strongly connected components are subgraphs in which every node is reachable from every other node by following directed edges. Therefore, every node in a directed graph is part of a weakly connected component. However, not all nodes belong to a strongly connected component.

Connected component analysis reveals trends over the sample period, summarised in **Table 4.3**. Each year consists of a single dominant WCC, comprising many of the total nodes in the system (ranging from 96.89 -99.32% over the sample period) and a few remaining WCCs. A graph-based visualisation of the largest WCC for the npm ecosystem 2012 is shown below in **Figure 4.3**. The dependency graph is a snapshot of the network at the end of the year comprising published packages and their versioned dependencies. Recall that dependency packages may derive from earlier periods, dating back to 2010, while published packages are restricted to the year of interest. **Figure 4.3** consists of 54 567 nodes and 1 051 964 edges.

In contrast, a few nodes, less than 1-5% over five years, belong to the largest SCC component. The absence of significant membership in SCCs in software networks is distinct from that observed in other directed networks such as WWW and numerous metabolic networks (Barabási, 2009). Software systems have a hierarchical structure, subdividing the system into logical modules or subsystems at various levels (Abadi & Cardelli, 2012). SCCs represent mutually reachable subgraphs. However, the hierarchical directionality inherent in software development design makes these mutually reachable clusters improbable and incongruous from the software development standpoint (Myers, 2004) Therefore, the rest of this chapter focuses on the largest WCC and the corresponding transitive closure (TC).

Table 4.2 below demonstrates the properties of the full npm network, the largest WCC, and its corresponding TC for each year. Specifically, it shows the changes in the number of nodes, edges, average degree, and the standard deviation of the in- and out-degree. In general, the average degree per node in terms of direct and indirect dependencies increases tenfold for the transitive closure case. The high variability associated with the in and out-degree mean may be observed due to the presence of hubs typically found in scale-free networks. **Table 4.2** highlights the implications of complex network growth and its important role in shaping a network's degree distribution.

Figure 4.3: Early sample of the 2012 npm network



Notes: This is a snapshot of the largest weakly connected component of the npm network at the end of December 2012. Nodes are packages, and a directed link indicates a versioned package depends on another versioned package for functionality. The network consists of 54,567 nodes and 1,051 964 edges. This graph was created with Gephi software using the Force Atlas 2 Algorithm.

Table 4.2: Properties of the npm dependency network over Time (2012-2016)

2012					
Network	**Nodes**	**Edges**	**Avg.Degree**	**SD In-Degree**	**SD Out-Degree**
Full npm	56 316	1,060 573	18.833	111.031	41.282
Giant WCC	54 567	1,051 964	19.278	112.731	41.766
TC	54 567	2,879 573	52.771	369.294	95.809

2013					
Network	**Nodes**	**Edges**	**Avg.Degree**	**SD In-Degree**	**SD Out-Degree**
Full npm	149 162	3,450 271	23.131	167.562	61.448
Giant WCC	145 179	3,443 044	23.716	169.805	62.179
TC	145 179	12,525 641	86.277	803.737	167.295

2014					
Network	**Nodes**	**Edges**	**Avg.Degree**	**SD In-Degree**	**SD Out-Degree**
Full npm	333 206	7,875 114	23.634	202.231	66.594
Giant WCC	326 725	7,861 842	24.063	204.294	67.176
TC	326 725	39,846 382	121.957	1459.284	239.596

2015					
Network	**Nodes**	**Edges**	**Avg.Degree**	**SD In-Degree**	**SD Out-Degree**
Full npm	604 663	16,245 649	26.867	313.744	72.976
Giant WCC	592 662	16,105 017	27.174	316.814	72.976
TC	592 662	165,066 780	278.518	3691.959	566.338

2016					
Network	**Nodes**	**Edges**	**Avg.Degree**	**SD In-Degree**	**SD Out-Degree**
Full npm	1,576 947	24,020 200	15.232	275.801	118.575
Giant WCC	1,566 287	23,981 444	15.311	276.734	118.972
TC	1,566 287	200,455 030	127.981	2583.638	469.330

Notes: Three cases are considered: (I) Full npm network, (II) The giant weakly connected component (WCC) (III)The corresponding transitive closure of the giant WCC. The average in- and out-degree are equal. The standard deviation (SD) for the in-degree and out-degree are considered.

4.3. STRUCTURE AND COMPLEXITY

4.3.1. DEGREE DISTRIBUTIONS

Degree distributions provide insight into the connectivity of nodes in a graph and reflect the basic information about the network's organisation. Formally, the in-degree and out-degree distributions, represented by $P^{in}(k)$ and $P^{out}(k)$ respectively, indicate the likelihood of finding a node with a specified in-degree or out-degree k in a given graph. Degree distributions illustrate the difference between complex networks from simple random graphs. Numerous real-world networks exhibit a 'scale-free' degree distribution in which all values are expected to occur without characteristic size or scale. This suggests that $P(k)$ obeys a power law over an extended range of degrees:

$$P(K)\ k^{-\lambda} \tag{4.2}$$

With $P(k)$ as the probability of a certain degree k and λ as the scale-free exponent, with $\lambda > 1$. Conversely, in simple random graphs, each edge is present or absent with equal probability. Thus, random graphs generally follow a Poisson distribution in the limit of a large graph size. Scale-free networks are characterised by a logarithmically growing average path length and an approximately algebraically decaying distribution of vertex connectivity (Barabási and Albert, 1999; Georg, 2012). Barabási and Albert (1999), originally introduced them to describe real-life networks such as the World Wide Web and social networks.

To generate a scale-free network, there are initially n nodes and no edges connecting them. At every time step t, a new vertex v with m edges is added to the network until the total number of nodes is reached (Barabási, 2009). These edges are connected to existing vertices with a probability proportional to the degree of the nodes in the network (Barabási, 2009). As the network evolves, a small number of nodes progressively obtain a higher node degree and, subsequently, a higher probability of connecting to newly introduced nodes. This preferential bias in connectivity, termed the *'preferential attachment'* mechanism, creates power law distributions. [1]It reflects the fact that packages with a substantial number of reverse dependencies (high in-degree) have a higher probability of being dependencies for newly published packages and hence establish central hubs in the network. These hubs are critical to spreading information, as they are versioned packages with numerous programs that depend on them for proper functioning.

The coexistence of network growth and the preferential attachment mechanism generate scale-

[1]Price (1976)studied the occurrence of power law in citation networks. He proposed a mechanism called the Principle of Cumulative Advantage, which suggests the creation of winner-takes-all dynamics, where a few individuals or entities come to dominate their respective fields while others fall behind. The cumulative advantage principle can be observed in various fields, including economics, social networks, and cultural production (Jackson, 2014).

free networks. Indeed, most real-world networks are scale-free, including biological (protein interactions), technological (the internet at a router level) and social networks (Barabási, 2009). Scale-free networks have enhanced robustness in the face of random node failure and random damage (Albert et al., 2000; Dorogovtsev et al., 2003). One would have to remove a significant portion of nodes to destroy or fragment such networks randomly. This inherent robustness partially explains why scale-free architecture is ubiquitous in many evolved networks found in nature. This also suggests that targeted attacks must be explicitly designed to destroy such networks effectively. For instance, an npm maintainer removed 250 packages from the npm repository in March 2016 as a form of protest. However, of those 250 packages, only the removal of the small utility *left-pad* package led to famous widespread software breakage as many packages directly or indirectly relied on it. Therefore, a topological analysis of the network may identify packages vulnerable to targeted attacks that would lead to large-scale perturbation.

In directed networks, it is possible for the in-degree distribution to be scale-free while the out-degree distribution is not, or vice versa. I examine the in- and out-degree distribution of the largest WCC and its corresponding transitive closure over the sample period. For each year, I compute the probability density functions (PDFs) and complementary cumulative frequency distributions (CDDFs) and then plot the logarithms of these distributions in **Figures 4.2** and **4.3**.[2]. Note, a CCDF scales at $\lambda - 1$, hence the shallower appearance (Alstott et al., 2014). All distributions exhibit an approximate power-law scaling region, which can be directly observed in a log-log plot with a straight slope line λ followed by a faster decay at large k (Alstott et al., 2014). The legends in **Figures 4.2** and **4.3** indicate the values of the scale-free exponents.

The distribution yields to power law after k_{min}, a cut-off introduced to deal with the continuum limit. The k_{min} is a parameter used in the estimation of the power law distribution of a network. It is defined as the value that minimises the Kolmogorov-Smirnov distance (D) between the observed data and the power law fit. The Kolmogorov-Smirnov distance measures the difference between two continuous cumulative distribution functions, and it is used to determine the goodness-of-fit of the power law to the data (Alstott et al., 2014). Thus, in this thesis, I use the scale-free exponent and the k_{min} parameters to define the degree distributions of scale-free networks. **Tables 4.3** and **4.4** summarise the in- and out-degree distribution parameters, respectively.

Interestingly, the npm graphs demonstrate a persistent imbalance between the in-degree and out-degree distributions, characterised by the in-degree distributions extending towards larger

[2]It is common to use CCDFs to display heavy-tailed distributions. However, if the distribution has prominent peaks in the tail region, they may be more evident when visualised as a PDF rather than a CCDF. Furthermore, if there is an upper limit on the distribution, the behaviour of PDFs and CCDFs may differ. (Alstott et al., 2014))

values of k. This implies that nodes with a greater number of incoming links are more prevalent than nodes with a greater number of outgoing links.**Figure 4.4** illustrates this difference, where the out-degree distributions are heavily truncated compared to the in- and overall-degree distributions. For the WCC and TC in-distributions (**Figure 4.2**), the empirical data points generally fall around the power-law fitted lines without apparent outliers. In the WCC in-degree distribution graphs, as the npm network grows, the scale-exponent increases over time from $\lambda_{in} \approx 1.95$ in 2012 to $\lambda_{in} \approx 2.16$ by the end of 2016.

A strong requirement for the scale-free hypothesis values between the range of $2 < \lambda < 3$, which suggests the distribution's mean is finite, and its variance is infinite, asymptotically. That is, a randomly chosen node in the 2016 network could have a small degree, as 95 per cent of nodes have an in-degree less than 35, yet a random node may also yield thousands of incoming links. For instance, in the 2016 network, the largest package in terms of in-degrees had 65,223 versioned packages directly requiring it for functionality. It is also evident that the growing number of nodes accelerates the rise in maximum degree, especially in the TC case. The super-linear growth of the maximum degrees suggests that new nodes tend to connect to already highly connected nodes.

The scale-free exponent for the TC in-degree distribution remains below two each year, which is a relatively unusual value in the scale-free literature (Broido & Clauset, 2019). This indicates the presence of a superlinear preferential attachment mechanism where the biggest transitive hubs grow linearly with the system size, subsequently influencing the network into a hub and spoke configuration in which most nodes link to a few central nodes. Furthermore, the TC in-degree distribution yields to power law at a relatively small k_{min} and extends over a considerable k range compared to all other degree distributions. For $1 < \lambda < 2$, the tendency to link to highly connected nodes is present, yet the bias is weaker relative to the $2 < \lambda < 3$ ultra-small world regimes.

.

Table 4.3: In-Degree Distribution Properties over Time (2012-2016)

WCC					
	2012	**2013**	**2014**	**2015**	**2016**
$max(k_{in})$	5344	14 584	34 863	62 499	65 223
λ^{in}	1.95	1.72	1.97	2.10	2.16
σ^{in}	0.014	0.005	0.008	0.011	0.010
$\hat{k}_{min}$	40	13	92	261	251

TC

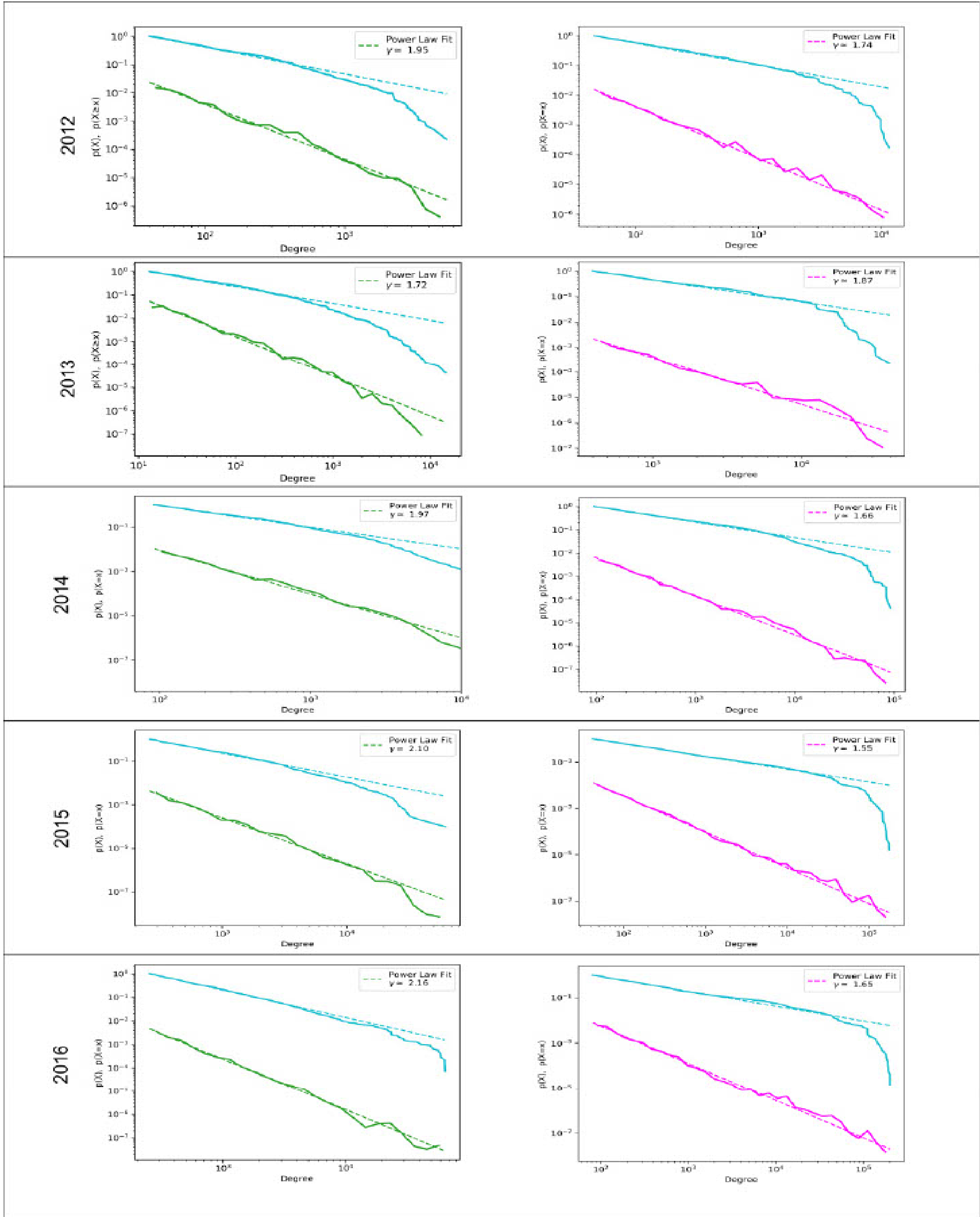
	2012	**2013**	**2014**	**2015**	**2016**
$max(k_{in})$	11 560	39 326	92 380	175 888	201 322
λ^{in}	1.74	1.87	1.66	1.55	1.65
σ^{in}	0.010	0.013	0.004	0.002	0.002
$\hat{k}_{min}$	46	400	95	43	81

Notes: The table provides properties of the in-degree distribution for the largest WCC and the corresponding TC for each year over the sample period. $max(k_{in})$ reflects the node with the largest incoming links, λ^{in} is the scaling exponent, σ^{in} is the standard error of the scaling exponent, and $\hat{k}_{min}$ is the minimum degree k value for which the power law holds.

The WCC out-degree scaling exponent decreases over time from $\lambda_{out} \approx 2.95$ in 2012 to $\lambda_{out} \approx 2.59$ in 2016, thus exhibiting a strong case for power-law behaviour. However, the power-law regions are relatively small for the out-degree distributions. Complex programs with too many dependencies are less reliable and difficult to maintain. This software design principle suggests an upper bounding effect on the power law scaling for the out-degree distribution. **Figure 4.3** demonstrates this as the distributions show an abrupt cut-off towards the right of the plot, and the upper tail exhibits apparent deviations from the power law scaling.

Between 2012 and 2015, the TC out-degree distributions have a value of $\lambda_{out} > 3$, which indicates that the degree distributions decay sufficiently fast to generate smaller and less numerous hubs, the 2012 network being the strongest case in this instance. However, the TC scale-exponent decreases over time from $\lambda_{out} \approx 4.04$ in 2012 to $\lambda_{out} \approx 2.43$ in 2016, thus exhibiting a gradual decrease in the distance between nodes located within the power law region. This evolution towards the ultra-small world regime within the transitive closure networks highlights how mid-level functioning dependency hubs act as bridges between low-level and high-level functioning packages.

Figure 4.4: Probability Density Function & Complementary Cumulative Distribution Function of In-Degrees (2012-2016).



Notes: Probability Density Function ($p(X)$, green for WCC and pink for TC) and Complementary Cumulative Distribution Function ($p(X)x \geq x$, blue) of In-Degrees (2012-2016). Dotted lines indicate power-law fits (straight lines on log-log scales) to histogram data in regions indicated. The legends show the values of the power-law exponents γ^{in} in for each fit where $p_k\ k^{-\gamma}$

Table 4.4: Out-Degree Distribution Properties over Time

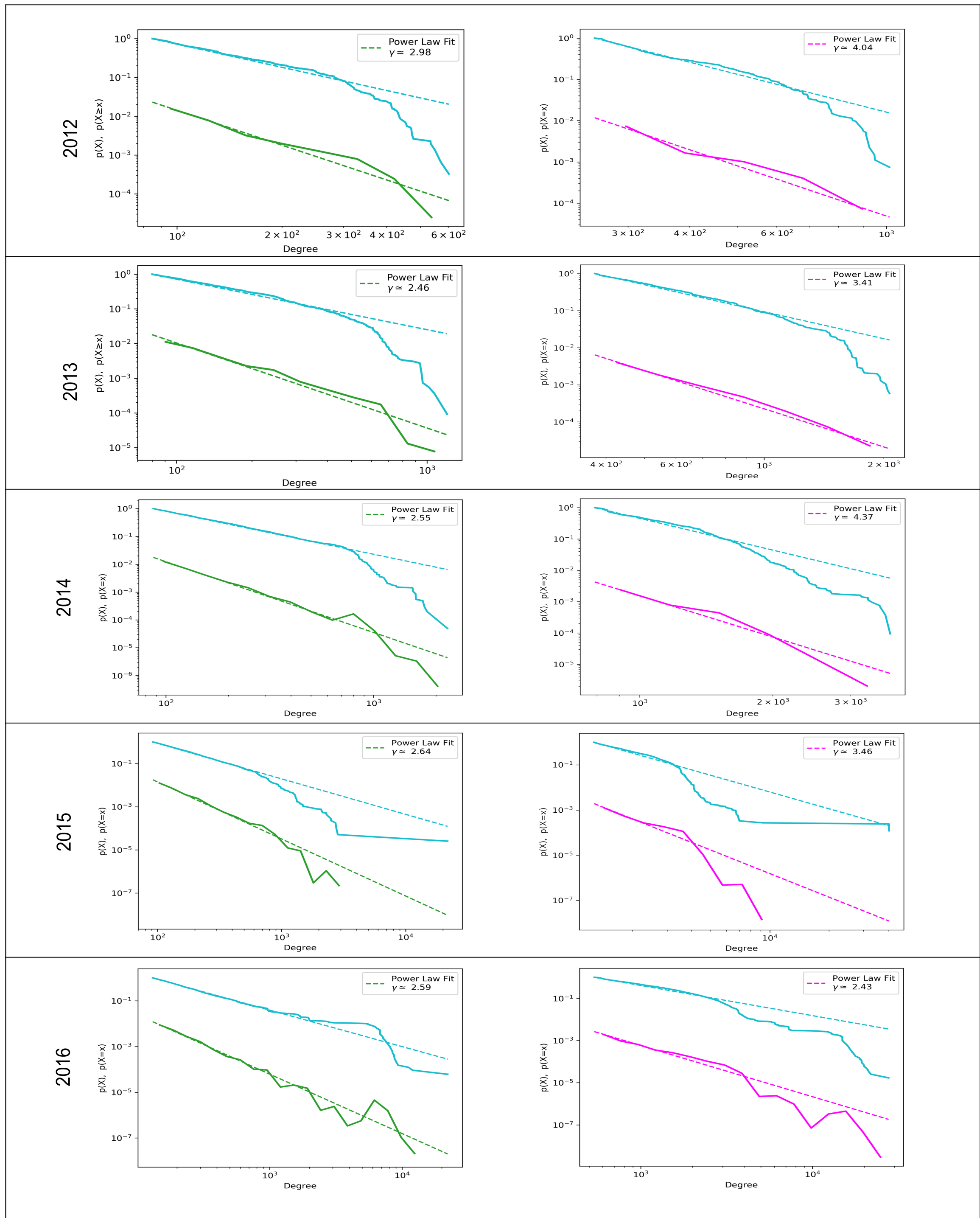
WCC					
	2012	**2013**	**2014**	**2015**	**2016**
$max(k_{out})$	603	1203	2271	21 795	22 139
λ^{out}	2.98	2.46	2.55	2.64	2.59
σ^{out}	0.036	0.014	0.011	0.008	0.009
$\hat{k}_{min}$	85	80	87	92	131
TC					
	2012	**2013**	**2014**	**2015**	**2016**
$max(k_{out})$	1015	2069	3687	40462	27779
λ^{out}	4.04	3.41	4.37	3.46	2.43
σ^{out}	0.059	0.026	0.033	0.014	0.004
$\hat{k}_{min}$	257	375	536	791	1274

Notes: The table provides properties of the out-degree distribution for the largest WCC and the corresponding TC for each year over the sample period. $max(k_{out})$ reflects the node with the largest outgoing links, λ^{out} is the scaling exponent, σ^{out} is the standard error of the scaling exponent, and $\hat{k}_{min}$ is the minimum out-degree k value for which the power law holds.

The analysis of the out-degree reflects the number of packages required for another package to function. Nodes with a limited out-degree represent simple projects since they do not inherit many methods/functions from other packages. Conversely, nodes with a substantial out-degree tend to be more complex and sophisticated as they inherit functionality from multiple sources. Hence, out-degree distributions that are heavy-tailed or scale-free indicate a broad range of complexities. Meanwhile, versioned packages with a high in-degree are frequently utilised in different contexts, while those with a low in-degree are not. On the other hand, versioned packages that are neither substantially reused nor heavily constructed from other elements are considered less interesting.

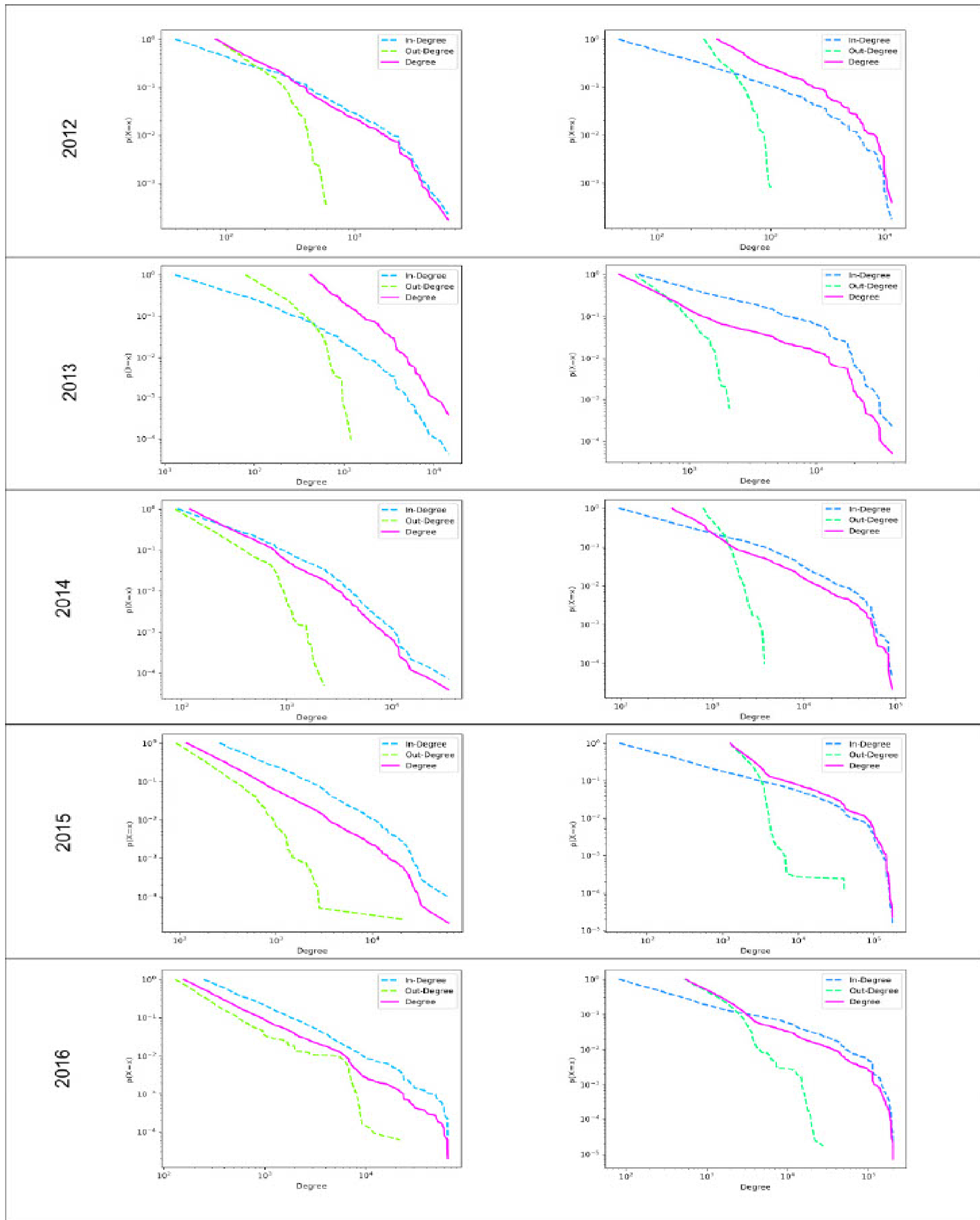
As expected, the highest degrees in the npm network correspond to incoming links, reflecting the common software engineering practice of reuse. The primary aim of software development is to establish an optimal system of dependencies so that core components can be used repeatedly in different situations to carry out fundamental tasks (Myers, 2004). Core components are designed with minimally constraining specialisations so that they can be included in higher functional layers to combine or build upon those fundamental tasks.

Figure 4.5: Probability Density Function & Complementary Cumulative Distribution Function of Out-Degrees



Notes: Probability Density Function ($p(X)$, green for WCC and pink for TC) and Complementary Cumulative Distribution Function ($p(X)x \geq x$, blue) of Out-Degrees (2012-2016). Dotted lines indicate power-law fits (straight lines on log-log scales) to histogram data in regions indicated. The legends show the values of the power-law exponents γ^{out} in for each fit where $p_k\ k^{-\gamma}$

Figure 4.6: Comparison Between The Degree, In-Degree and Out-Degree CCDFs (2012-2016)



Notes: Complementary Cumulative Distribution Function ($p(X)x \geq x$) for: the in-degree (blue, dotted), the out-degree (green, dotted), and the total degree distribution (pink, solid line). Note the graphs on the left represent the giant WCC, with their corresponding TC graphs on the right. It is evident in the graphs show the out-degree distribution is heavily truncated versus the in-degree distribution. This trend is consistent for all the years in the sampler and is also reflected in the transitive closure case.

4.3.2. Alternative Distributions

The power distribution is fit to obtain the parameters λ and $\hat{k}_{min}$. However, these parameters alone do not provide insight into the accuracy of the fit (Alstott et al., 2014). To determine the statistical credibility of the fitted model, a standard goodness-of-fit test is applied, which returns a standard p-value. A pairwise comparison using the normalised log-likelihood ratio test (LRT) is used to evaluate each power-law model to non-scale free alternative models (Alstott et al., 2014). The alternatives used here are the (i) exponential, (ii) power-law with exponential cut-off and (iii) log-normal distributions (**Tables 4.6** and **4.7**) (Amaral et al., 2000; Buzsáki and Mizuseki, 2014). The validity of the use of LRT for the alternative degree distribution models has been previously established (Clauset et al., 2009; Vuong, 1989). The maximum likelihood is restricted to $k_{min} \leq k$, which makes the models directly comparable, subsequently generating a slight bias in favour of the power law, as the optimal choice of k_{min} for an alternative distribution may not be the optimal choice for the power law (Clauset et al., 2009).

It is noteworthy that the outcome of this test offers indirect support for the scale-free hypothesis, as a power-law model can be favoured over certain alternatives, even if the power-law itself is not a statistically valid data model (Alstott et al., 2014). The power-law model and the alternatives are evaluated and compared through the use of a log-likelihood ratio, which serves as the test statistic $R = L_{PL} - L_{ALT}$ where L_{PL} is the log-likelihood of the power-law model and L_{ALT} is the log-likelihood of a particular alternative model. The sign of R indicates which model is a better fit for the data: the power law ($R > 0$), the alternative $R < 0$ or neither ($R = 0$). The log-normal distribution = Normal (μ, σ) is characterised by the parameters (mean and standard deviation) of the variable's natural logarithm. This distribution is highly skewed for large values of σ and features a fat tail for large k values.

Tables 4.5 and **4.6** show that most in- and out-degree distributions favour a power law fit over the exponential distribution, as the likelihood ratios are positive and significant. It is evident that the accompaniment of an exponential cut-off is regarded as a supportive correction to the power law fitting. In this case, the npm network indicates that finite-size effects are common for both in and out-degree distributions. Since this thesis distinguishes between versions, it is understandable that a versioned package may have a limited in-degree in the face of evolving functionality. In real networks, the intrinsic qualities of a node can influence the rate at which it acquires links. That is, a simple utility package may be reused in many contexts, but when an updated version is published onto the registry, the developer community may opt for the latest version over the older version simply because it provides improved functionality. Thus, a versioned package may face a threshold of incoming links when in competition with its later releases. This is in line with Decan et al.(2019)'s work, which highlights that young packages,

in terms of time elapsed since their first release, are more subject to frequent updates than mature packages. The potential bias in connectivity to newly published packages contrasts with the Barabasi-Albert model, which leads to scenarios where the late nodes will never turn into the largest hubs.

Notably, versions may differentiate between unique states of a package, yet the functionality offering is fundamentally the same. Suppose there are two packages which provide similar functionality. The package with a high in-degree (a popular dependency) indicates robustness. It signals to the coder to be the better-performing package with its wide reuse compared to the similar package with a low in-degree. This cumulative advantage is also driven by external factors such as community reviews, reading issues on Github, frequency updates and maintainer reputation. However, such considerations are beyond the scope of this thesis.

Table 4.5: Normalised Loglikelihood Test Results for Non-Scale Free Alternatives for In-Degree Distributions for WCC & TC

k_{in}						
	WCC Exponential	**WCC Truncated**	**WCC Log-Normal**	**TC Exponential**	**TC Truncated**	**TC Log-Normal**
2012	15.251	-66.940	-6.556	26.388	-66.939	-6.556
p-value	0.000	0.000	0.000	0.000	0.000	0.000
2013	37.945	-20.481	-16.601	23.210	-10.671	-5.935
p-value	0.000	0.000	0.000	0.000	0.000	0.000
2014	23.18	-12.512	-10.483	51.461	-21.324	12.753
p-value	0.000	0.000	0.000	0.000	0.000	0.000
2015	20.512	-7.675	-13.994	122.40	-35.663	-13.730
p-value	0.000	0.000	0.000	0.000	0.000	0.000
2016	20.562	-4.723	-2.774	117.930	-27.704	-3.328
p-value	0.000	0.000	0.000	0.000	0.000	0.000

Notes: The loglikelihood ratio test is used to evaluate the comparison of the power law hypothesis with exponential, truncated power law, and lognormal distributions. The results of the test include the normalised ratio and the corresponding p-value. A small p-value indicates that the direction of change is not solely due to random variation. In contrast, a large p-value indicates that the statistic cannot distinguish between the different distributions.

The exponential distribution is favoured over the power law for the 2014 and 2015 transitive out-degree distributions. The exponential distribution exhibits a thin tail and relatively low variance and is the absolute minimum alternative because the definition of "heavy-tail" is against

the “light-tailed” exponential behaviour. When the scaling exponent is $\lambda > 3$, the distribution has a relatively thin tail. Therefore, these outcomes are in accordance with the broad distribution of the networks' scaling exponent. The log-normal is a broad distribution that can exhibit heavy tails but is nevertheless not scale-free. From an empirical perspective, the log-normal fit is at least as good as the power law fit for the majority of degree distributions, indicating that the npm networks may, in reality, be log-normal networks. Importantly, the heavy-tailed, log-normal distribution maintains the conclusion that a broad spectrum of complexities or reuse exists within the npm network.

Table 4.6: Normalised Loglikelihood Test Results for Non-Scale Free Alternatives for Out-Degree Distributions for WCC & TC (2012-2016)

k_{out}						
	WCC Exponential	**WCC Truncated**	**WCC Log-Normal**	**TC Exponential**	**TC Truncated**	**TC Log-Normal**
2012	0.679	-61.342	-6.584	2.289	-18.598	-3.567
p-value	0.498	0.000	0.000	0.022	0.000	0.000
2013	3.600	-18.502	-14.901	0.649	-11.150	-8.980
p-value	0.0003	0.000	0.000	0.516	0.000	0.000
2014	27.189	-14.512	-9.514	-4.823	-10.781	-9.543
p-value	0.000	0.000	0.000	0.000	0.000	0.000
2015	16.976	-10.941	-5.188	-4.803	-10.843	-18.962
p-value	0.000	0.000	0.000	0.000	0.000	0.000
2016	32.871	-0.001	6.257	14.170	-41.058	-43.892
p-value	0.000	0.999	0.000	0.000	0.000	0.000

Notes: The loglikelihood ratio test is used to evaluate the comparison of the power law hypothesis with exponential, truncated power law, and lognormal distributions. The results of the test include the normalised ratio and the corresponding p-value. A small p-value indicates that the direction of change is not solely due to random variation, while a large p-value indicates that the statistic cannot distinguish between the different distributions.

Overall, the balance of evidence for or against scale-free structure varies by network year and degree type. Over the sample period, the npm network may not exhibit direct statistical evidence of scale-free distributions. However, for the degree distributions where the scaling exponent falls within the $\in (2,3)$ range, the power law is a better model of the degrees than the alternatives. This would include all years for the largest WCC out-degree distribution, the WCC in-degree distributions for 2014, 2015, and 2016; and the TC out-degree distribution for 2016. This is in line with other technological and software networks which exhibit scale-free structure (Broido and

Clauset, 2019; Mora-Cantallops et al., 2020).

A pattern of historical preference amongst community users may lead to specific packages, regardless of version, remaining popular throughout evolution. This thesis argues that heavy-tailed distributions may be explained by a generative mechanism similar to the preferential attachment mechanism. New packages or standard programs preferentially declare popular (large in-degree) packages as dependencies (providers of fundamental functionality), subsequently forming super-hubs. The caveat here is that it is possible in a software network for later versions (new nodes) to become super-hubs as they represent enhanced functionality.

4.3.3. Small-World Behaviour

The small-world phenomenon implies that the distance between two randomly chosen nodes in a network is short. Small-world behaviour also relates to the clustering coefficient, indicating a network's cluster tendency. It is a metric which evaluates the extent to which the nodes in the neighbourhood of a certain node are connected. Watts and Strogatz (1998) define an algorithm that generates a network that exhibits a large clustering coefficient and a small average path length.

Table 4.7 below captures the clustering coefficient (c) and average path length (l) for the npm network over time and compares it with the respective Watts-Strogatz model. It is clear from the average path length that the npm ecosystem behaves as a small-world network. The decrease in the npm clustering coefficient in 2016 relative to the previous four years can be explained by the removal of the `‘left-pad’` package incident, which led to widespread breakage.

The presence of hubs created from scale-free networks impacts the small-world property. A network is said to have an ultra-small world regime when the scaling exponent is $2 < \lambda < 3$, as the presence of hubs within the network radically reduces the path length. This is because the significant linking to a large number of small-degree nodes creates short distances between them. In other words, hubs shrink the distance between nodes. When $\lambda > 3$, the hubs in the network are not significantly large and numerous to have a substantial impact on the distance between the nodes.

Table 4.7: Small-World Properties of the npm network over Time vs WS-Model

Table 4.7					
Variable	**2012**	**2013**	**2014**	**2015**	**2016**
c	0.021	0.021	0.027	0.24	0.013
l	2.030	2.230	2.636	3.463	3.198
WS c	0.023	0.024	0.024	0.024	0.023
WS l	4.101	4.071	4.403	4.504	5.774

Notes: c is the clustering coefficient and l is the average path length. Network statistics are calculated for the giant WCC over the sample period (2012-2016). The corresponding Watts-Strogatz network statistics are calculated for comparison.

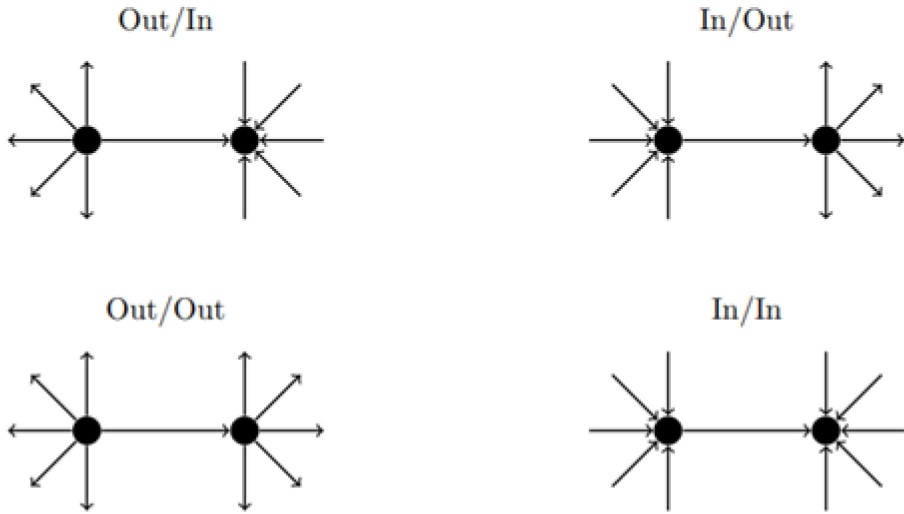
4.3.4. Assortativity

Assortativity refers to the extent to which similar nodes are connected in a network (Newman, 2003). When nodes with a large degree have a connection preference for nodes of similar large degrees, the network is said to have a positive degree-degree correlation (assortative). If highly connected nodes are more likely to make links with isolated, less connected nodes, it is said that the nodes within the network mix disassortatively. Degree assortativity is quantified by computing the Pearson correlation coefficient of the degrees at each end of the links in the network (Newman, 2003). In a directed network, the relationship between in-degree and out-degree results in four degree correlation profiles illustrated in **Figure 4.7** below (Newman, 2002).

Figure 4.8 demonstrates the relationship between in-degrees and out-degrees in the npm network for each year, where every node in each graph is represented by its (k^{out}, k^{in}) pair (Newman, 2002). The graphs on the left account for direct relations within the largest WCC, while the graphs on the right depict the corresponding transitive network. As is clear from **Figure 4.8**, k^{in} and k^{out} are negatively correlated, especially for the large degree nodes.

Moreover, it is evident that both the in-degree and out-degree are quite heterogeneous. Shown in **Table 4.7**, from 2012 to 2016, the percentage of nodes with $k_{in} = 0$ remain within the approximate range of 55 to 65 per cent of the npm network. This indicates that a significant majority of packages are interdependent, as they require other packages for their proper functioning, and that this dependence is growing more pronounced over time. Conversely, the proportion of strictly required packages where $k_{out} = 0$ remains quite stable over the sample period, ranging approximately between 19 and 24 per cent of the network. These independent packages represent the basic building blocks of the ecosystem as they are designed to execute a specific task yet

Figure 4.7: Assortativity Types in Directed Graphs



(Image source: Hoorn et al.,(2013).

are broadly reusable in various applications.

Trends evident in **Figure 4.8** are confirmed through the calculation of the linear Pearson correlation coefficients between the sets k_{in} and k_{out} for those nodes with either a large in-degree or large out-degree. $e_{jk}^{in,out}$ defines the joint probability distribution of finding a directed link from a source node of j_{out} out-degree and to a target node of k_{in} in-degree (Newman, 2002). The probability is normalised, i.e.:

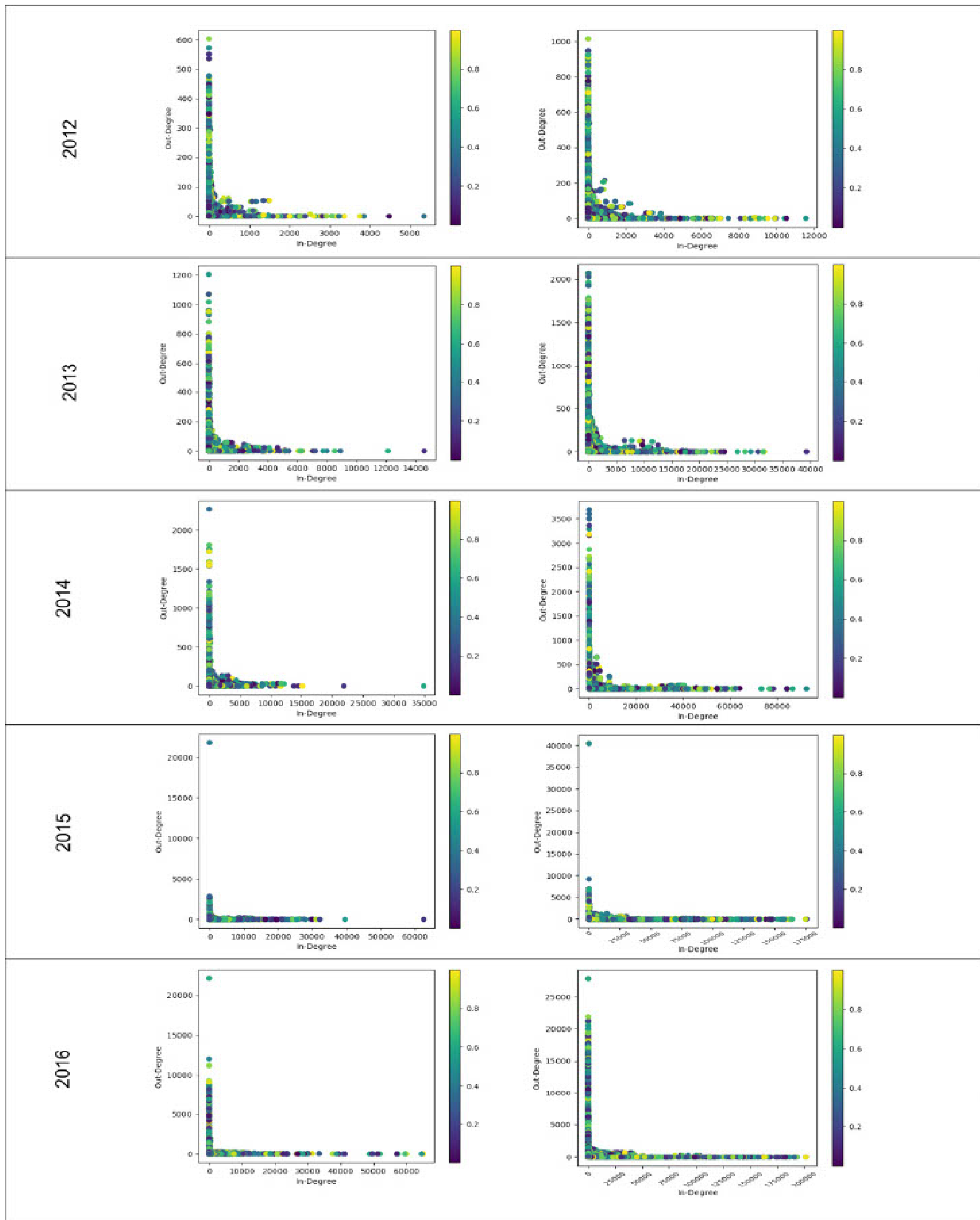
$$\sum_{jout} e_{jk}^{out,in} = q_k^{in} \tag{4.3}$$

$$\sum_{kin} e_{jk}^{out,in} = q_j^{out} \tag{4.4}$$

$$\sum_{jout,kin} e_{jk}^{out,in} = 1 \tag{4.5}$$

The network displays correlations if $e_{jk}^{in,out}$ deviates from the random expectation Newman, 2002. Here, the direct network assortativity coefficient r is defined as:

Figure 4.8: Scatter plot of the number of incoming links versus outgoing links



Notes: Figure 4.8 shows the relationship between in-degree and out-degrees, where every node in each graph is represented by it (k^{out}, k^{in}) pair. The scatter plots on the left represent the giant WCC with the corresponding transitive closure on the right.

$$r_{in,out} = \frac{\sum_{jk} jk(e^{in,out}_{jk} - q^{in}_{\leftarrow j} q^{out}_{\rightarrow k})}{\sigma^{in}_{q\leftarrow} \sigma^{out}_{q\rightarrow}} \tag{4.6}$$

where $e^{in,out}_{jk}$ is the joint degree distribution, $q^{in}_{\leftarrow j}$ is the probability of finding a node with in-degree j by following a random link backwards and $q^{out}_{\rightarrow k}$ is the probability of finding and out-degree with degree k by following a random link forward (Newman, 2002). $\sigma^{in}_{q\leftarrow} \sigma^{out}_{q\rightarrow}$ are the standard deviations of the respective distributions(Newman, 2002). This definition accounts for the correlation between the in-degrees of the source nodes and the out-degrees of the target nodes.

Here,$-1 \leqq r \leqq 1$, whereby $r = 1$signifies perfect assortativity, while $r = -1$ implies perfect disassortativity. The network has no assortativity when $r = 0$ which indicates any node can randomly connect to any other node otherwise known as a neutral network. Notably, perfect scalar disassortativity ($r = -1$) in real-world networks is extremely rare. In general, the minimum value of the Pearson correlation coefficient, r_{min}, satisfies the inequality $-1 \leqq r_{min} < 0$, indicating that a perfectly disassortative network is closer to a randomly mixed network than a perfectly assortative network. When different types of vertices are present, random mixing typically pairs dissimilar vertices, which is similar to disassortative mixing. As a result, it is common for most disassortative networks to have values close to $r = 0$, as opposed to assortative networks.

Table 4.8: Degree Correlation Coefficients for WCC and TC For Each Year

	2012	**2013**	**2014**	**2015**	**2016**
% $k_{in} = 0$	56.83	58.49	60.38	60.08	64.93
% $k_{out} = 0$	23.39	21.9	19.66	20.56	21.34
Degree Correlation Coefficients for WCC					
Out-In	-0.129	-0.105	-0.104	-0.068	-0.160
Out-Out	-0.010	-0.024	-0.035	-0.007	-0.050
In-In	-0.001	-0.007	-0.012	-0.010	-0.004
In-Out	-0.016	-0.003	-0.012	-0.007	0.001
Degree Correlation Coefficients for TC					
Out-In	-0.129	-0.153	-	-	-
Out-Out	0.145	0.150	-	-	-
In-In	0.036	0.017	-	-	-
In-Out	-0.013	0.009	-	-	-

Notes: The table shows the percentage of nodes with zero incoming links and the percentage of nodes with zero outgoing links, respectively. Four correlation profiles are considered the degree mixing coefficients for each year relating to the in- and out-degrees are provided.

Table 4.7 shows the values of the correlation coefficient for each year of the npm network. Note the correlation coefficient for the transitive network graph could only be calculated for the years 2012 and 2013 due to a computational memory error. However, they are included here in the analysis as a reference to demonstrate the changes that the assortativity coefficient may undergo when transitive dependencies are considered. Importantly, it is possible that Pearson's correlation coefficient is scaled down by the high variance in the respective degree sequences (van der Hoorn & Litvak, 2013). The following was observed from the degree correlation analysis.

Firstly, while three of npm's WCC network correlation profiles document negligible correlations ($r \approx 0$), there is a relatively stronger disassortativity tendency between out- and in-degrees for nodes with large degrees. This disassortativity for the $r_{out,in}$ profile suggests that hubs of high out-degree nodes prefer to link to low in-degree nodes, or hubs of in-degree nodes tend to be connected to low out-degree nodes, keeping the directionality of the links in mind. However, the presence of disassortative hubs does not preclude hubs from being interconnected. It merely means that a significant share of the links of these hubs is connected to comparatively peripheral nodes (Piraveenan, 2010). [3] When transitive dependencies are considered for 2012 and 2013, the in-out assortativity coefficient is $r \approx -0.129$ and $r \approx -0.153$, respectively.

[3] Figure 4.3 depicts this relationship.

Here, negative assortativity is expected as it supports connectivity between diverse elements in the network, an important feature for producing complex programs. Consequently, a clear distinction can largely be made between the large-scale providers of functionality represented by required packages with a high in-degree and the large-scale consumers, represented by high-level dependent packages with a high out-degree. As previously stated, simple software components are frequently reused, primarily because they are applicable in multiple contexts, whereas complex programs are specialised and thus applicable in limited contexts.

There is a weaker signature of disassortative mixing when out-degrees and in-degrees are considered separately. Over time, the out-assortativity coefficient becomes neutrally assortative ($r \approx 0$), indicating that nodes with small and high out-degrees connect to each other randomly. Even though the level of out-assortativity is preserved over the years for the WCC case, the transitive closure graph shifts the values considerably towards the positive assortative side where $r \approx 0.145$ in 2012 and $r \approx 0.150$ in 2013. An assortative network by degree is characterised by a core-periphery structure, with a core of high out-degree nodes surrounded by a less dense periphery of nodes with low degrees.

A possible explanation for the assortativity among out-degrees within the npm network is partly due to the hierarchical layers of functionality inherent in software design. For instance, a complex program with many declared dependencies (a node with a large out-degree) does not directly collaborate with very low-level packages. Instead, it is built up of mid-level packages, which themselves require low-level software packages. This demonstrates that it is possible to have an embedded transitive network pattern retaining out-degree assortative hubs when the network is non-assortative for the same correlation profile at a direct dependency level.

Similarly, the in-assortativity converges towards random mixing for the WCC case, yet when the transitive dependencies are considered, again, the values shift towards weak positive values of assortativity. This reflects the tendency of basic-task functions with a large in-degree to collaborate at the base of a hierarchy of functional layers. The progression towards neutral assortativity possibly reflects an artefact of the elements in the network becoming closely coupled over evolutionary time.

Degree correlation is an important component of topological analysis for a number of reasons. A network can be classified based on the mixing pattern of nodes such that common topological traits can be identified in networks from various domains (i.e., biological and technological), or networks from a similar domain can be distinguished based on topology (i.e., the dependency networks of CRAN, PyPi etc.) Furthermore, understanding mixing patterns provides insight into

the evolution or design of networks and may inform the design of growth models which attempt to duplicate the desired mixing patterns. Connecting patterns between nodes may highlight key functionalities between nodes and sheds light on how to defend a network from attacks.

4.4. INFLUENTIAL VERSIONED PACKAGES

It is possible to measure the vulnerability of the ecosystem by finding the most critical nodes within the network. To determine the relative importance of a node within a network, centrality metrics may be used. These metrics include betweenness centrality, closeness centrality and degree centrality. There are a few centrality metrics that can be used to determine a node's influence. Closeness centrality indicates the average distance of a node to all other nodes. While the betweenness centrality metric indicates inter-node influence by measuring how often paths between nodes must traverse a given node. Betweenness centrality could not be calculated because of its high computational costs and closeness centrality produced negligible results. Degree centrality is the simplest connectivity metric, which counts the number of edges attached to the node. It is an appropriate measure to find highly connected or popular nodes that, in case of failure, would impact a larger number of nodes.

It is imperative to note that software contagion propagates through any number of levels of dependency, not only direct ones. Thus, when determining critical nodes within a software network based on degree centrality, one must consider both the direct and transitive links of a node. The normalised degree centrality DC_i computed for each versioned package as:

$$DC_i = \frac{k_i}{n-1} \tag{4.7}$$

With k_i being the degree of node i,n the total number of nodes in the network and DC_i [0,1]. In the table below (**Table 4.8)** contains the top 10 influential versioned packages for each year. For each versioned package, both the direct dependencies (DD) and transitive dependencies (TD) are represented, combined with their degree centrality values (DCi). All listed versioned packages have zero out-degrees, thus they represent basic-service functions. It is worth noting how several packages have limited direct dependencies, but their transitive dependencies are up to three orders of magnitude or larger. In fact, in 2016, the **"ansi-regex"** package version `2.0.0` had only 178 direct dependencies yet had over 200 000 transitive dependencies.

A high-out degree centrality value indicates that a package relies heavily on other versioned packages for functionality. From 2012 to 2015, the degree of centrality increased as the ecosystem grew. Recall this thesis uses static dependency graphs to map dependency relations. Therefore, the values do not reflect runtime realities when only one versioned package is called per dependency requirement. However, degree centrality does capture the dependency popularity of a versioned package within an ecosystem. This thesis proposes additional considerations when determining the influence a critical package has on the vulnerability of the ecosystem. First, various packages can be replaced fairly easily in the case of network disruption with an-

other package due to redundant resources inherent in the OSS distribution processing systems (Myers, 2004). Therefore, the real cost of removing a package depends on the ability to replace it with another package. Alternatively, if there is a bug in the package, then that package is compromised. Including a compromised package leads to further compromisation down the dependency chain. An area for future research would be investigating cascading failure in the npm network with a comprehensive software bug propagation framework.

Table 4.9: Top 10 Most Influential Versioned Packages in The npm Network Based on Degree Centrality from 2012-2016

Top 10 Influential Nodes in **2012**				
Package	**Version**	**DD**	**TD**	**DC_i**
uglify-js	1.2.5	1305	11 560	0.212
async	0.1.22	4477	10 532	0.193
uglify-js	1.2.6	1235	10 417	0.191
uglify-js	1.2.0	925	9980	0.183
uglify-js	1.2.4	870	9927	0.182
uglify-js	1.2.3	873	9926	0.182
async	0.1.23	3747	9917	0.182
uglify-js	1.2.2	852	9907	0.182
uglify-js	1.2.1	850	9906	0.182
async	0.1.18	3179	9553	0.175

Top 10 Influential Nodes in **2013**				
Package	**Version**	**DD**	**TD**	**DC_i**
async	0.2.9	14 584	39 326	0.271
mime	1.2.9	2723	31 726	0.219
async	0.2.8	8536	31 508	0.217
mime	1.2.11	2919	31 508	0.217
mime	1.2.10	2424	30 598	0.211
async	0.2.7	7738	29 238	0.201
mkdirp	0.3.5	6169	28 236	0.195
underscore	1.4.4	12135	26 899	0.185
qs	0.6.5	439	24 482	0.169
async	0.1.22	6260	24 353	0.168

Top 10 Influential Nodes in **2014**				
Package	**Version**	**DD**	**TD**	**DC$_i$**
inherits	2.0.1	2397	92 480	0.283
isarray	0.0.1	236	86 561	0.265
core-util-is	1.0.1	183	84 438	0.258
core-util-is	1.0.0	22	84 187	0.258
string_decoder	0.10.25	57	84 179	0.258
string_decoder	0.10.31	112	84 179	0.258
string_decoder	0.10.24	48	84179	0.258
async	0.9.0	21867	78 398	0.234
mime	1.2.11	5543	77 114	0.236
minimist	0.0.8	1945	76 648	0.235

Top 10 Influential Nodes in **2015**				
Package	**Version**	**DD**	**TD**	**DC$_i$**
inherits	2.0.1	3429	175 888	0.297
escape-string-regexp	1.0.4	533	174 971	0.295
ansi-rege	2.0.0	94	15 031	0.277
ms	0.7.1	867	163 424	0.276
debug	2.2.0	24 416	162 305	0.274
strip-ansi	3.0.0	803	162 236	0.274
escape-string-regexp	1.0.2	216	161 765	0.273
escape-string-regexp	1.0.3	584	160 152	0.270
isarray	0.0.1	418	159 225	0.269
ansi-styles	2.1.0	141	158 883	0.268

Top 10 Influential Nodes in **2016**				
Package	**Version**	**DD**	**TD**	**DC$_i$**
ansi-regex	2.0.0	178	201 322	0.129
strip-ansi	3.0.1	1976	201 286	0.129
strip-ansi	3.0.0	557	201220	0.128
escape-string-regexp	1.0.5	1132	193 318	0.123
supports-color	3.1.2	535	192 874	0.123
escape-string-regexp	1.0.2	185	192 713	0.123
escape-string-regexp	1.0.4	511	191 141	0.122
escape-string-regexp	1.0.3	442	191058	0.122
ansi-styles	2.2.1	261	190 850	0.122
chalk	1.1.3	261	190 726	0.123

Notes: For each versioned package, both the direct dependencies (DD) and transitive dependencies (TD) are represented, combined with their degree centrality values (DCi).

4.5. SUMMARY

JavaScript's package manager ecosystem constantly evolves as new packages are added and existing packages are updated, making it a dynamic and complex system. By analysing the npm ecosystem from the perspective of network science, one can gain insights into various aspects of package manager ecosystems, such as the distribution of dependencies, the connectivity of packages, and the potential for cascading failures.

The results presented in this section highlight that preserving edge directionality is important in analysing the degree distribution and degree correlation between software components as it helps to maintain the meaning and context of the dependencies within the network. In this thesis, maintaining edge directionality demonstrated a network of the differences between in- and out-degree distributions, the anti-correlation between large in-degree and out large out-degree, and the positive assortative mixing among out-degrees. Any endeavours to comprehend network topologies based on software engineering principles or processes without acknowledging the asymmetry of connections between nodes will be insufficient.

The value of the exponent γ and the deviations from the power law provide valuable information on the mechanisms driving the underlying formation of the network, such as small degree saturation and high degree cut-offs due to finite size effects. The findings show that the npm network displays scale-free behaviour for both in and out-degree distributions.

This section demonstrates how complex network science can be used to identify packages that play a critical role in the ecosystem due to their high number of reverse dependencies. These hubs can be considered potential points of failure and can be monitored closely to minimise the risk of cascading failures.

5
Conclusion

This thesis has examined the presence of scale-free and small-world qualities in npm's dependency network from the beginning of 2012 to the end of 2016. It also investigated assortative mixing and identified critical nodes within the network over the sampler period. Unlike most package manager studies, the complex analysis has been undertaken at a version level. This concluding chapter recapitulates the findings of this thesis and outlines avenues for future research.

5.1. Summary of Contributions

In this thesis, I have examined several aspects of npm's package dependency network, inspired by questions of complex networks and software engineering. This thesis has three primary contributions guided by research questions. First, JavaScript's popularity and active contributor community have led to substantial growth. Approximately 852 package versions were released daily over the sample period, and the number of transitive dependencies of an average package increased from 33 in 2012 to 246 in 2015 before decreasing to 112 in 2016. The decrease in growth can be attributed to the `left-pad` incident, and overall, this thesis finds that the structure of the npm network significantly changed after March 2016.

The second contribution relates to the structure and complexity of the network based on degree distribution and degree correlation. Results show that basic-service packages heavily dominate the in-degree distribution while more specialised, complex packages occupy the heavy tail of the out-degree distribution. Similar to other findings in the open-source software literature, the npm network exhibits scale-free and heavy-tailed distributions and small-world behaviour.

However, to the author's knowledge, no study at this point in time has analysed the npm package ecosystem at a version level from the perspective of complex network science. The disparity between the two is distinctly pronounced. Furthermore, the mixing patterns indicate the hierarchical structure of software design influences the npm network topology. Software design methods strive to reduce the specificity of interactions while promoting broad reusability, and the scale-free characteristic of software collaboration networks may embody this type of compromise on a large scale.

Thirdly, using degree centrality, one can identify critical versioned packages in the network. The removal of the `left-pad` had a high impact not because many others directly used it but indirectly through transitive dependencies. The findings show that in 2016, one versioned package had over 200 000 transitive. Thus, as the npm network grows, the magnitude of potential cascading failure increases significantly.

Importantly, the findings of this thesis highlight the significance of maintaining the directionality of edges in the npm network, as it uncovers several network characteristics, such as the disparities between in-degree and out-degree and the positive and negative assortative mixing.

In conclusion, representing the dependency graph of an npm ecosystem at a version level is important for complex network analysis as it accounts for the impact different versions of packages have on the network structure, topology, and evolution, including the distribution of nodes, edges, mixing patterns and centrality measures. Moreover, it is imperative to note that varying editions of packages possess diverse vulnerabilities and security vulnerabilities. By constructing the npm network at a version level, it becomes possible to accurately examine the dissemination of vulnerabilities and security risks within the network and determine packages that could be crucial to the network's security. This information can be useful for software developers, network administrators, and security analysts who wish better to understand the relationships between versioned packages and dependencies and make informed decisions about package management.

5.1.1. FUTURE RESEARCH

More work is needed to understand npm's rapidly evolving network. Some areas of future research include:

- The application of complex network analysis on the npm ecosystem to identify community structures with the npm network and evaluate how they influence the network's topol-

ogy.

- Apply complex network analysis to the contributor and maintainer community participating in the npm ecosystem.

- Developing a model using complex network parameters to understand bug propagation in line with software principles.

- The application of network analysis to other OSS package manager ecosystems at a version level.

"Clouds are not spheres, Mandelbrot is fond of saying. Mountains are not cones. Lightning does not travel in a straight line. The new geometry mirrors a universe that is rough, not rounded, scabrous, not smooth. It is a geometry of the pitted, pocked, and broken up, the twisted, tangled, and intertwined. The understanding of nature's complexity awaited a suspicion that the complexity was not just random, not just accident. It required a faith that the interesting feature of a lightning bolt's path, for example, was not its direction, but rather the distribution of zigs and zags. Mandelbrot's work made a claim about the world, and the claim was that such odd shapes carry meaning. The pits and tangles are more than blemishes distorting the classic shapes of Euclidian geometry. They are often the keys to the essence of a thing."
- James Gleick. Chaos Making a New Science

References

Abadi, M., & Cardelli, L. (2012). *A theory of objects*. Springer Science & Business Media.

Albert, R., Jeong, H., & Barabási, A.-L. (2000). Error and attack tolerance of complex networks. *nature*, *406*(6794), 378–382.

Alstott, J., Bullmore, E., & Plenz, D. (2014). Powerlaw: A python package for analysis of heavy-tailed distributions. *PloS one*, *9*(1), e85777.

Amaral, L. A. N., Scala, A., Barthelemy, M., & Stanley, H. E. (2000). Classes of small-world networks. *Proceedings of the national academy of sciences*, *97*(21), 11149–11152.

Barabási, A.-L., Albert, R., & Jeong, H. (2000). Emergence of scaling in random networks. *Nature*, *406*(6794), 378–382. https://doi.org/10.1038/35020579

Barabási, A.-L. (2009). Scale-free networks: A decade and beyond. *science*, *325*(5939), 412–413.

Barabási, A.-L. (n.d.). Network science. http://barabasi.com/networksciencebook/

Barabási, A.-L., & Albert, R. (1999). Emergence of scaling in random networks. *science*, *286*(5439), 509–512.

Broido, A. D., & Clauset, A. (2019). Scale-free networks are rare. *Nature communications*, *10*(1), 1–10.

Brooks, F. P. (1978). *The mythical man-month: Essays on softw* (1st). Addison-Wesley Longman Publishing Co., Inc.

Buzsáki, G., & Mizuseki, K. (2014). The log-dynamic brain: How skewed distributions affect network operations. *Nature Reviews Neuroscience*, *15*(4), 264–278.

Clauset, A., Shalizi, C. R., & Newman, M. E. (2009). Power-law distributions in empirical data. *SIAM review*, *51*(4), 661–703.

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.

Dorogovtsev, S. N., Mendes, J. F., & Dorogovtsev, S. N. (2003). *Evolution of networks: From biological nets to the internet and www*. Oxford university press.

Durlauf, S. N., & Blume, L. E. (2008). The economy as a complex system. In B. LeBaron & L. Tesfatsion (Eds.), *Handbook of computational economics* (pp. 1061–1147). Elsevier. https://doi.org/10.1016/S1574-0021(08)00026-0

Elliott, M., Golub, B., & Jackson, M. O. (2014). Financial networks and contagion. *American Economic Review*, *104*(10), 3115–3153.

Georg, C.-P. (2012). The effect of the interbank network structure on contagion and common shocks. *Journal of Financial Stability*, *8*(2), 89–107.

Goyal, S. (2009). Connections: An introduction to the economics of networks.

Hines, P., Balasubramaniam, K., & Sanchez, E. C. (2009). Cascading failures in power grids. *Ieee Potentials*, *28*(5), 24–30.

Jackson, M. (2014). *Social and economic networks*. Princeton University Press.

Kikas, R., Gousios, G., Dumas, M., & Pfahl, D. (2017). Structure and evolution of package dependency networks, 102–112.

King, A. (2016). Understanding the npm dependency model. https://lexi-lambda.github.io/blog/2016/08/24/understanding-the-npm-dependency-model/

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*.

Luna, F., & Stefansson, B. (2012). *Economic simulations in swarm: Agent-based modelling and object oriented programming*. Springer US. https://books.google.co.za/books?id=oqDhBwAAQBAJ

Mora-Cantallops, M., Sánchez-Alonso, S., & Garcıa-Barriocanal, E. (2020). A complex network analysis of the comprehensive r archive network (cran) package ecosystem. *Journal of Systems and Software*, *170*, 110744.

Myers, C. R. (2004). Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical review E*, *68*(4), 046116.

Nesbitt, A., & Nickolls, B. (2017). *Libraries.io Open Source Repository and Dependency Metadata* (Version 1.0.0). Zenodo. https://doi.org/10.5281/zenodo.808273

Newman, M. E. J. (2002). Assortative mixing in networks. *Physical Review Letters*, *89*(20), 208701. https://doi.org/10.1103/PhysRevLett.89.208701

Newman, M. E. J. (2003). Mixing patterns in networks. *Physical Review E*, *67*(2), 026126. https://doi.org/10.1103/PhysRevE.67.026126

Newman, M. E. J., Barabási, A.-L., & Watts, D. J. (2008). Random graph models of social networks. *Journal of Statistical Mechanics: Theory and Experiment*, *2008*(10), P10008. https://doi.org/10.1088/1742-5468/2008/10/P10008

Piraveenan, M. (2010). Topological analysis of complex networks using assortativity. *Academia, University of Sydney*.

Preston-Werner, T. (2021). Semantic versioning 2.0.0.

Qin, Z., Zheng, X., & Xing, J. (2008). *Introduction to software architecture*. Springer.

Raemaekers, S., Van Deursen, A., & Visser, J. (2014). Semantic versioning versus breaking changes: A study of the maven repository. *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 215–224.

Šubelj, L., & Bajec, M. (2012). Software systems through complex networks science: Review, analysis and applications. *Proceedings of the First International Workshop on Software Mining*, 9–16.

van der Hoorn, P., & Litvak, N. (2013). Degree-degree correlations in directed networks with heavy-tailed degrees. *arXiv preprint arXiv:1310.6528*.

Vuong, Q. H. (1989). Likelihood ratio tests for model selection and non-nested hypotheses. *Econometrica: Journal of the Econometric Society*, 307–333.

Watts, D. J., & Strogatz, S. H. (1998). Collective dynamics of 'small-world'networks. *nature*, *393*(6684), 440–442.

Wittern, E., Suter, P., & Rajagopalan, S. (2016). A look at the dynamics of the javascript package ecosystem. *Proceedings of the 13th International Conference on Mining Software Repositories*, 351–361.

Yang, Y., Nishikawa, T., & Motter, A. E. (2017). Small vulnerable sets determine large network cascades in power grids. *Science*, *358*(6365), eaan3184.

Zimmermann, M., Staicu, C.-A., Tenny, C., & Pradel, M. (2019). Small world with high risks: A study of security threats in the npm ecosystem. *28th USENIX Security Symposium (USENIX Security 19)*, 995–1010.