Mats Engstad

The Intersection of Open Source and Digital Platforms

Master's thesis in Computer Science
Supervisor: Eric Monteiro
January 2022

Mats Engstad

The Intersection of Open Source and Digital Platforms

Master's thesis in Computer Science
Supervisor: Eric Monteiro
January 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

The use of open source software has increased in recent years, and is continuing to affect the rest of the software world to a larger degree every year. Open source code exists in every parts of the software world, from small niche projects to mainstream applications used by millions. The open source movement's influence on the technology of our world is vast which makes it an important topic for IT researchers. A lot of research has been done in the open source space, however research on the combination of open source and digital platforms is lacking. This thesis contributes to filling in this research gap by conducting a case study of the open source platform Npm (Node Package Manager) in order to better understand open source software ecosystems. This is done by qualitatively analysing important episodes in the history of Npm. Three themes emerge from the empirical data; complex dependencies, open source security, and platform governance. In a world where most successful digital platforms use a very centralized governance structure, Npm's decentralized way of controlling the platform influenced by its open source aspects, brings hope for the future of platform diversity.

Sammendrag

Bruk av åpen kildekode (open source) har økt de siste årene, og fortsetter å påvirke resten av programvareverdenen. Programvare med åpen kildekode eksisterer i alle deler av programvareverdenen, fra små nisjeprosjekter til mainstream applikasjoner brukt av millioner av mennesker. Bevegelsen innenfor åpen kildekode utøver en stor innflytelse på den teknologiske verdenen vi har i dag, derfor er dette et viktig tema for forskere innenfor IT. Mye forskning har allerede vært gjennomført angående åpen kildekode, men forskning på kombinasjonen av åpen kildekode og digitale plattformer er noe mangelfull. Denne oppgaven bidrar til å fylle inn dette hullet i akademisk litteratur, ved å gjennomføre en casestudie av Npm (Node Package Manager), en plattform for og av åpen kildekode. Dette gjøres ved en kvalitativ analyse av viktige episoder i Npms historie. Tre hovedtemaer oppstod fra de empiriske dataene; komplekse avhengigheter, sikkerhet i åpen kildekode, og styring av plattformer. I en verden hvor stort sett alle populære digitale plattformer styres av en sentralisert plattformeier, gir Npm håp for fremtidige desentraliserte plattformer styrt gjennom tankesettet til åpen kildekode.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The use of open source software in the technology of our world has seen a huge increase over the last years. Today, code written by communities of software developers working voluntarily exists for the world to use free of charge. Technology today certainly takes advantage of this offer of free and open code, for example in the area of web development where 97% of surveyed developers from 2018 reported relying on open source software in some way [2], or for companies using enterprise technology where 90 % of IT leaders in 2021 across 13 countries use enterprise open source software [3]. It is fair to say that open source software is widely used and has become mainstream in today's technology society. This was not always the case, open source software has undergone a transformation from its early stages of unorganized development done by a single developer or a small team, towards mainstream and commercially viable software with more predominant planning and design phases and business strategies [4].

The product of open source development, namely the software, spread to products and services where traditionally proprietary software dominated. In addition and just as importantly, the development model of open source software has also affected the commercial world of proprietary software development. Open source development methodologies make it easier to reduce conflicts and management overhead, even in a highly geographically distributed project [5]. As software corporations grow and become worldwide organizations, being able to continue development in a distributed setting is advantageous, and by adopting tools and coordination mechanisms from the open source community this work becomes easier. Smaller software development endeavours have also benefited from the open source movement, i.e by adopting agile development methodologies which focus on flexibility in the development process. It is fair to say that both software as a product, and the development models surrounding it today are influenced by the open source movement.

Organizing commercial marketplaces as digital platforms has been a viable way for organizations to leverage network effects to gain a competitive advantage. The use of digital platforms is fairly ingrained in our society, from healthcare to entertainment, most of us interact regularly with some sort of digital platform. To understand the popularity of platforms in our day to day lives, one can look to platform leaders in the mobile app industry such as the Apple App Store, or the entertainment industry such as YouTube. Organizing products and services as digital platforms can be an important tool to scale the business to such a level seen in technology giants such as Apple or Google.

Digital platforms also exist in the world of open source software, popular examples include the operating system Linux and the JavaScript package management tool Npm. Some open source platforms grow large enough to create software ecosystems surrounding them, which generate interesting opportunities for researchers to fill in a gap in the literature on the combination of open source software and digital platform ecosystems [6], [7]. A lot of research has been done on both open source software and digital platforms individually, but rarely on the combination of the two. Research on open source often focuses on developers' motivations to participate in open source projects, or corporations' incentives and business models when adopting open source initiatives [8]. Central parts of the literature on digital platforms include a platform's pricing structure and its actors. The nature of open source being free software makes the existing pricing models fit poorly or not at all with what can be observed on open source platforms. In addition, an open source community introduces actors different from what has previously been observed on traditional proprietary platforms. All this combined with the fact that empirical data from open source projects are readily available for researchers to dig through, makes open source platforms good candidates for conducting case studies in this field.

When looking at successful platforms today, one key attribute almost all platforms share is the presence of a dominant platform owner controlling the platform's evolution, often to achieve their business goals. On the App Store, Apple asserts its dominance through policies with a strict vetting process and an unforgiving pricing model. App developers often have very little say in how Apple should run their platform, and where they should take the platform in the future. This creates a one-sided relationship between the platform owners and the app developers, where app developers have to interact with the platform at Apple's discretion. In extreme cases this can lead to law suits as seen in the 2020 case of Apple v. Epic Games [9], where Apple was sued for its unfavourable restrictions of the apps on its App Store, as well as its pricing policy of taking a 30% cut of app revenue. Similarly, platforms such as YouTube often receives complaints from its content providers for treating them unfairly. The one-sided relationship between YouTube as a company and its users is described by some of its content providers as YouTube controlling the platform to such an extent that content providers feel overrun:

The reason YouTube treats its content creators so poorly is because they can, and they know you (or any other creator for that matter) won't do anything about it except gripe. -YouTube content provider [10]

Maybe the existence of a controlling actor such as a platform owner is necessary for a platform to be able to grow and evolve successfully without falling into chaos. This would not be an unreasonable conclusion to come to when looking at popular digital platforms today. Without a platform owner responsible for taking charge and steering the evolution of the platform, who else would do it? How else should important decisions be made? Maybe all platform decisions should be decided by user voting, but even in democracies we elect governments because ultimately someone needs to be held responsible and handle the practical side of things. This line of thinking can be extended to how we view open source projects. Just like with digital platforms, open source projects must be governed in a way to sustain its development and ensure long-term evolution. The two are also alike in the way that they connect different actors together in a collaborative environment. On the App Store, app developers collaborate with Apple, users and advertisers. On YouTube, content providers collaborate with Google, advertisers and content consumers. While in open source projects, project owners such as private firms or software foundations collaborate with developers, users and other contributors.

However, unlike digital platforms, open source projects are often governed in a way that lets its developers and users take a greater part in the decision making process. For example, open source developers generally have a lot of freedom to choose what they want to work on, and if they have an idea for some new functionality, they can take the initiative to make it and add it to the project. Open source contributors to a specific project are often described as a project's community. A community forms around a project because the focus on collaboration is so strong, as opposed to a centralized power structure seen in traditional proprietary software firms. The nature of open source also allows for developers to start their own new version of the project and reuse anything they want from the original project, often called forking a project. This can happen if a part of the community is dissatisfied with the project's evolutionary direction or its governance model. An example of a project fork happened in 2014 when a group of developers in the open source project node.js, a popular

JavaScript runtime environment, decided to fork the project and start their own version called io.js [11]. Their decision to abandon node.js and create io.js was based on being dissatisfied with the control the owners of node.js asserted in the governance of the project. The luxury of being able to branch out a new version of a project whenever developers or users want is something normal digital platforms lack. Take YouTube for example, if content providers are dissatisfied with how the platform owners govern the platform, they have limited power to influence the decision making process and are thus left with either accepting the terms of YouTube or leaving the platform.

What happens in the intersection between the open source mindset and digital platforms? Is it possible for a community of open source contributors to create and govern a digital platform without relying on an all-controlling platform owner? Given the increase in popularity of using open source as a development process and digital platforms for their economic and strategic advantages, the combination of the two is likely to occur more frequently in the future. Understanding how the nature of open source affects aspects of digital platform ecosystems, i.e. platform governance or ecosystem security, is an important part of the academic knowledge of both open source and digital platforms, and is therefore a highly relevant research topic. Given that the majority of digital platforms today have a very centralized structure, literature on decentralized digital platforms is lacking. By studying the exception to the rule, namely decentralized open source platforms, general conclusions about software ecosystems can be drawn and thus this thesis is contributing to the academic literature on digital platforms. On the basis of the background outlined in this chapter, this thesis will investigate the following research question:

How do open source specific aspects affect the evolution of software ecosystems?

This thesis performs a case study of Npm (Node Package Manager), an open source platform for JavaScript libraries (packages). Npm has seen huge growth in recent years, and was in 2018 experiencing over 4 billion package downloads per week on their platform. Because Npm is a platform mainly used for other open source packages, its community of open source contributors extends what would normally be a community only working on the platform core itself. For example, developers working on the platform core in many cases maintain their own open source packages on the platform as well. This introduces an opportunity to study how the open source community creates new actor types, and what that means for the platform ecosystem. Being a platform embracing the open source culture means that traditional platform governance strategies are ineffective to ensure the desired evolution and longevity of the platform. Understanding how the open source mindset forms these governance strategies can help us understand traditional platform governance better. This thesis' empirical data consists of episodes from the Npm project. An episode is an event of interest to the research question, i.e. the hacking of an important library on the platform. Through a qualitative data analysis of the selected episodes, this thesis aims to understand how the nature of open source affects platform ecosystem aspects.

1.1 Thesis Structure

The structure of this thesis follows a standard approach of dividing it into introduction, literature study, method, case study, discussion, and conclusion.

- Chapter 1, Introduction
  Introduces the main theme of the thesis; open source and digital platforms. A lot of research can be found on the two themes separately, but previous research combining both themes is somewhat lacking. This gap in the academic knowledge in combination with the increase in open source development worldwide are the main motivations behind the thesis's topic.

- Chapter 2, Literature Study
  Presents previous research on the main theme and explains the state of the art of work related to open source, software security and digital platforms. Open source started out as a niche development phenomenon. It has since become mainstream and its worldwide reach covers both software products and development methodologies. Software security discussions in the open source world revolve around bugs / vulnerabilities, and if its more secure to let the world go through your code with a fine-toothed comb or try to hide it as best as possible. The literature on platform governance mainly concern itself with platforms with a centralized all-controlling platform owner. This model does not always fit well when applying it to open source platforms.

- Chapter 3, Method
  In this section, arguments for and against different research methods are discussed, which leads to the decision of performing a case study of the Npm project. This specific case is chosen based on a set of case criteria, where open source, no dominant actor and platform ecosystem are important deciders. Data collection is done by selecting a set of episodes from the case, i.e. the introduction of a new functionality or the hacking of a package. Data analysis is done in a qualitative manner, where patterns and relationships in the data are analysed. Data analysis is done in stages, switching between working inductively and deductively.

- Chapter 4, Case Study
  Introduces the Npm platform and explains what parts it is made up of; registry, website, and command line interface. Gives a background of what problems Npm tries to solve in the JavaScript world, and shows how successful the tool has been over the years. Four episodes from the case is presented, where each episode consists of discussions by developers, platform owners, and users about an important happening in the ecosystem.

- Chapter 5, Discussion
  Based on the themes aggregated from the data analysis part, the discussion tries to understand the themes in the context of both previous literature and the case itself. Some discrepancies between the state of the art research and the empirical data collected in this thesis is found, especially in the are of platform governance.

- Chapter 6, Conclusion
  Summarizes the main arguments presented in the discussion and extracts the general conclusions we can make on the topic of open source software ecosystems. The open source movement is influencing software development all over the world, and perhaps its next big are of influence is digital platforms.

Chapter 2

Literature Study

2.1 Open Source Software

The term open source software refers to software that is open on the internet, free for everybody to use, modify, or distribute, it belongs to the public domain. Some of open source software's early popularity comes from the successful open source projects Linux and Apache, which caused an increased interest in open source around the year 2000 [12]. Today, the use of open source software is widespread, and every user of the internet relies in one way or another on open source software.

In earlier years, open source software was characterized by what Fitzgerald calls Free Open Source Software (FOSS) [4]. This differs from what we know as Open source software today by lacking strategic planning and sophisticated business models surrounding open source projects. Fitzgerald argues that Open source has transformed from the FOSS-style development towards what he calls Open source software 2.0. Here more sophisticated business strategies emerge and some developers are even paid to work on open source projects. The development process of Open source software 2.0 includes strategic planning to a greater extent, and more deliberate analysis and design phases are common. Every open source software project relies on an open source community to develop and maintain it. The open source community are developers and users working to enhance the product, without the expectation to receive any form of payment for it. The sense of community and the positive effects of contributing to it can in part explain the motivation behind open source developers, but does not explain why it has seen such a rapid growth in recent years according to Kogut [13]. Kogut argues that this growth indicates that traditional software development is less efficient than open source development, caused by the work needed to enforce intellectual property rights in closed proprietary software. In addition to this, open source development can leverage distributed innovation to increase efficiency and quality.

2.2 Software Security

Many aspects of software security is similar in open source projects compared to proprietary software. For example, traditional security methods such as user authentication, fire walls, and encryption are widespread in both open source and other types of software. Because open source is open for everyone to see, including hackers, the debate around bugs and vulnerabilities is especially interesting when it comes to security in open source projects.

2.2.1 Vulnerabilities

Security is a big part of the discussions around open source software in academic literature. Opening up source code for the world to see might intuitively sound less secure than keeping it hidden and away from potential hackers, but the debate goes much deeper than that, and includes notions of how to measure security, what motivates hackers, and how to fix security holes in the software. This section will investigate open source security in the academic literature, open source security here is not confined to security tools as open source products, but rather the security level of open source software in general.

The main arguments in open source security discussions boil down to "Is keeping source code hidden and secret more secure than keeping it open for the world to see, use and maintain?". This is not a trivial question to answer, because how do you define some software as more secure than other? That is the question Schryen et al. try to tackle in their paper about measuring software security [14]. They find that traditional security measurements often rely on the amount of bugs or the amount of vulnerabilities discovered in the software during its lifetime. This is not a bad starting point towards measuring security, because the existence of bugs in the code certainly introduces more opportunities for hackers to find vulnerabilities to exploit. However, Schryen et al. are skeptical of relying too much on the amount of bugs alone, because of two important implications; bugs are not all alike, and the assumption of software having a finite amount of bugs to find is wrong.

Bugs exist in all software, and range from harmless to software breaking. This means that some bugs do not introduce a security vulnerability, while others introduce multiple ways for hackers to exploit the software. Simply relying on the amount of bugs in the code to assess its security is therefore too shallow, and Schryen et al. suggest a model where the severity of the vulnerabilities are taken into account when measuring the security of the software. This is a good start to get a more granular view of quantifying security, and by combining this with the amount of time a vulnerability has been accessible will result in a better model of software security measurement shryen et al. argue. The amount of time a vulnerability exists in the code is an important measurement, a very severe vulnerability could cause a lot of harm even if it was only accessible to hackers for a short period of time. Schryen et al. highlight the lack of good models for measuring security in software, specifically for closed versus open source. They believe that the lack of good security data is one of the reasons for this, but also that a lot of data available focus on operating systems, which is not always useful when analyzing other types of software.

One important assumption the notion around measuring security by the amount of vulnerabilities discovered during a software's lifetime relies on, is that the software has a finite amount of vulnerabilities to be found. Schryen et al. argue that this is not really the case. When a bug is found to cause a security vulnerability, a patch is made to fix it, be it closed or open source. But patches themselves can introduce new bugs into the code, sometimes creating more vulnerabilities than they fix. Given this, software never really becomes "complete" in the sense that it is bug free given enough time, so a measurement of how many bugs have been found and fixed, does not necessarily tell the whole story whether or not a certain software is secure.

Even though security in software is hard to measure, the debate around open source versus closed proprietary software in a security perspective can be productive. This debate mainly revolves around security through transparency versus security through obscurity. Proponents of open source base their main arguments around the notion of the peer review process in order to discover and fix bugs, and they believe that all bugs are shallow, given enough eyeballs as Eric Raymond famously said in his paper about open source development methods [15]. Intuitively this argument does make sense, because if you open source your code and let the whole world go through it with a fine comb, one would think that a lot more bugs would be discovered compared to a team of in-house developers working on closed proprietary software. Both Schryen et al. and Payne [16] disagree with this argument, citing that the quality of the eyes looking at the code is much more important than the quantity. When it comes to open source projects, they recruit a wide variety of both experienced and inexperienced contributors, and it is not a given that any of the contributors have the security expertise required to understand that a certain bug could cause a security vulnerability. This is because software security knowledge often involves more than just understanding the programming language itself of a given project, other areas such as network protocols or cryptography also play a big part. In addition to this, Li et al. found that 70% of security vulnerabilities found in their open source case projects were caused by semantic bugs that are hard to prevent if the programmers do not have a thorough understanding of the whole system [17]. This indicates that simply having a lot of eyes on the code does not prevent security vulnerabilities to occur, but experienced programmers with security-specific knowledge and a thorough understanding of the software project is needed.

When it comes to software security, a back door is usually one of the more severe vulnerabilities software can have. This gives hackers potentially undetectable access to the program that they should not have. Schryen et al. point out that there is a lack in academic articles discussing back doors in open source software, which is especially interesting to discuss because hackers can pose as normal contributors and introduce back doors into the code. Payne argues that it is virtually impossible for hackers to introduce back doors into open source software because of the code review process and the many eyes on the code. While only a single rogue programmer could introduce a back door into closed proprietary software, the code review process would catch such an attack in an open source project he argues. Payne cites two examples to build on this argument; first, an open source "TCP Wrapper" software that got a back door introduced in its code which was discovered and patched only a day later. Second, an example from the "Interbase" database software that had a back door existing in its code for 9 years while being closed source, and when the code was published to the world as open source, the back door was discovered right away. Payne's argument that it would be virtually impossible to introduce back doors into open source code, assumes that the code would go through an open source review process before being accepted. That is however not always the case, and hackers have come up with other ways of creating back doors in open source code. Since a lot of software today rely on each other and are connected, a malicious piece of code could be introduced in one open source project and affect others. For example if the back door exists in an interpreter running the open source code, it would not be possible to detect it by simply reviewing the source code itself. Hackers can even circumvent the open source review process entirely and publish their code directly to a code base without the approval of other contributors, as seen in the hacking of the popular Npm package ESLint-scope from 2018.

Another argument in favour of open source mentioned in the literature is the flexibility around releasing a new security patch. As mentioned earlier, when it comes to security in software, time is of the essence. The longer a vulnerability is open for hackers to use, the more harm can be done. Proponents of open source security argue that open source projects are more flexible than closed proprietary software when a bug is found and should be fixed. Sometimes, the person responsible for finding the bug can simply write a patch and submit it to the open source code base directly. Closed proprietary software on the other hand often have to take business processes into consideration, and they can be halted by strict release schedules or even marketing purposes. For example, a corporation might want to wait to publish a security patch because of the implications this would have on its reputation, or corporations sometimes might want to inform only their biggest clients of a specific vulnerability before letting the rest of their clients know. These are some factors that could cause open source security patches to be implemented faster compared to closed proprietary software.

Closed proprietary software hides its source code in order to make it harder for hackers to find vulnerabilities to exploit. However, hackers still find ways to break into this type of software systems regularly, therefore security by obscurity does not always work. This is because hackers can use a system's binary code to look for vulnerabilities which is impossible to hide when the software is deployed. Although this is harder than using source code Schryen et al. argue that it is still a viable way for hackers to gain the insight they need in order to break into the system. Hiding code from the rest of the world could therefore be counter productive if hackers still find ways to break into it, because the software will not get the benefit of an open source review process.

2.3 Platform governance

Platform governance is defined by Tiwana et al. simply as *who* decides *what* in a digital multisided software platform [18]. The governance strategy of a digital platform has important implications of its evolution and its participants' actions [19]. The governance mechanisms used by platform owners are well established in the literature, however the implications of their practical implementations are not very well researched [20] [21].

A central challenge of platform governance is to be able to enforce enough control for platform owners to maintain the integrity of the platform, while giving away enough control such that content developers can provide content and innovate. Tiwana et al. [18] give three perspectives of platform governance; decision rights partitioning, control, and proprietary versus shared ownership. These categories describe mechanisms used by platform owners to govern the digital platform. Decision rights partitioning describes how the right to make important decisions are divided between the platform owners and the content developers. Schreieck et al. [20] describe this aspect of platform governance as its structure, and discuss how decision rights and ownership can be used by platform owners to achieve a certain outcome, i.e. facilitate user growth or reduce administrative work for platform owners. Schreieck et al. also include how the platform is owned in its governance structure, i.e. owned by a single firm or by a larger organization. This aspect is what Tiwana et al. call proprietary versus shared ownership. Decision rights partitioning, or a platform's governance structure, boils down to who has the power and responsibility to make important decisions for the platform.

Figure 2.1 is from Tiwana's book about platform governance [22]. He proposes a model where decision rights can be categorized as either strategic or implementational, and the power to control these lie on a spectrum from the platform owners to the app developers on the platform. He argues that decision rights are not completely controlled by only the platform owners, or only the app developers, but both actors have some degree of decision rights. By adjusting the sliders in Figure 2.1, the platform owners can decide to what degree different types of decision rights should be centralized or decentralized.

Both Tiwana et al. and Schreieck et al. include the notion of control in platform gover-

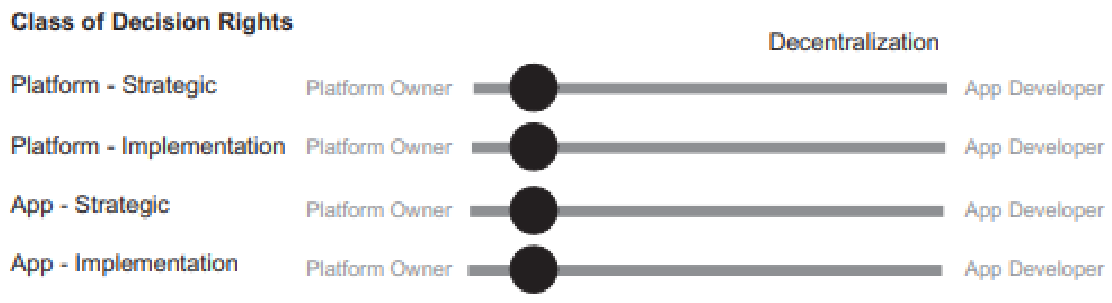

Figure 2.1: Sliders showing how decision rights can be divided between the platform owners and app developers.

nance. Controlling the platform is done through informal and formal control mechanisms used to guide developers' actions to align with the platform owners' vision. Formal control mechanisms include input and output control, as well as process control. Input control, or platform accessibility, lets platform owners decide who can participate on the platform. A high degree of developer restriction leads to higher quality in the products offered on the platform, but reduces user growth. It is not unusual that governance mechanisms come with tradeoffs between different outcomes, which is one of the main arguments of Schreieck et al. There is no one governance strategy that fits every platform, instead platform owners must choose how they implement mechanisms based on which tradeoffs they see as desirable. However, control mechanisms do not necessarily have to be a divergent zero sum relationship between platform owners and developers. In some instances their interests can both align with a given control mechanism.

Generativity is what Wareham et al. [23] describe as an ecosystem's ability to generate new content, or its output, without the need of platform owners' input. They look at platform governance as a way to find a balance between control and a desired level of generativity. Reaching the highest level of generativity possible and letting third parties in an ecosystem produce content uncontrollably is not desired as it could hurt the ecosystem in the long run by i.e. resulting in low quality of content and bad platform reputation. That is why some level of control must be achieved by the platform owners, they need to limit the behaviour of its content providers while simultaneously foster enough creative output to attract users. This creates a paradox where an ecosystem must be both stable and able to evolve. Without stability, complementors will not want to invest time and money into generating content for the ecosystem. Without evolvability the ecosystem will not be able to adapt to changes in the market, its users' needs or new technology.

Wareham et al. found three areas where the tension between stability and evolvability manifests itself in an ecosystem; output, actors, and identifications. An ecosystem's output is the content created by complementors and consumed by its users, and is the same as what Tiwana et al. and Schreieck et al. also call output. Controlling the tension between stability and evolvability in an ecosystem's output can be achieved through varying the output's variance. A high degree of variance leads to more unique products and increases variety in the products and services on the platform, while a low degree of variance results in a higher standard and homogeneity across the products and services. Through their case study, Wareham et al. found that common mechanisms used to reduce variance in output revolve around technical standards and processes that complementors must abide by. This ensures a high degree of compatibility in the ecosystem, because third party applications interface with the platform core in the same way and can potentially interface with each other as well. Because of this, the products on the platform will have a shared evolutionary trajectory and a similar level of quality. To increase the variance in output, platform owners can allow complementors to customize their products to fulfil different functional requirements, or foster specialization in different niche markets. From their case study, Wareham et al. found that this was done by having core platform functionality as open source software, letting complementors use and customize the functionality to their specific needs. Wareham et al. argue that the mechanisms used to adjust the tension between standardization and variety complement each other to enable creative freedom in the implementation of products while also ensuring that technical standards are followed.

The second area of tension highlighted by Wareham et al. is the tension between control and autonomy for the actors participating in the platform. Platform owners can control their actors by enforcing rules on how they are allowed to behave on the platform. The behaviour can be strictly controlled or allowed to be more autonomous, letting actors have a greater influence on processes when interacting with the platform. Wareham et al. look at how content generating actors submit their products to be controlled by process and output control mechanisms as a market transaction between actors and platform owners. In this transaction, actors forfeit liberty and autonomy in exchange for the perceived value the platform can offer. In the case study of Wareham et al., actors choose their desired level of control by adjusting their partner level with the platform. For example, actors can get platform certifications, dedicated platform personnel, or subscription fees. There are five levels of partnership, ranging from just a simple entry level where the only requirement of the complementor is registering their product on the platform, to the highest partner level that requires certifications, fees, good customer references, participation in customer surveys, etc. Each partner level has its own requirements that the actor must fulfil, and comes with its own set of increased values for the actor and the platform core. By choosing their desired partner level, actors have a greater autonomy in how they interact with the platform in regards to processes and control mechanisms, and to what value they seek to get from the platform.

The third tension exists in what Wareham et al. call identifications, and span from individual to collective. To ensure a more cohesive ecosystem, the platform owners must transform a set of individual, specialized and heterogeneous content into a cohesive set of functionality that promotes compatibility and reusability. If the platform achieves this, self-interested individual identifications will instead become collective identifications with a sense of community. Mechanisms to promote collectiveness involve reducing the undesirable variance found in the ecosystem, and include communal technical utilities, socialization, training opportunities and multipartner collaboration. Wareham et al. found that an important mechanism to increase community contribution is through the use of status in the ecosystem. If actors can achieve a greater legitimacy and status by participating in collective endeavours, it will serve as an incentive to contribute to community projects. Desirable variance can be promoted by promoting autonomy for actors which leads to individual identifications by i.e. using common standards to measure the quality in individual heterogeneous products. This means that actors are free to specialize and implement individual functionality while participating in the ecosystem.

Earlier research on platform governance tend to view platform governance from an authoritarian perspective, where a platform's structure is centralized and platform owners control most of the important decisions. After analyzing 30 research papers on platform governance, Manner et al. [21] found that authority-based governance is the predominant governance mechanism. Even though Schreieck et al., Manner et al. and Tiwana acknowledge that platforms with a decentralized governance structure exist, their models rely on dimensions such as pricing, market mechanisms and business models to analyze platform governance. These dimensions are not always useful when looking at platforms from an open source perspective, especially when both the platform core and services and products produced on the platform are open source. When a platform exists only of open source products it would be ineffective to use pricing as a control mechanism as Tiwana's model suggests. Other limitations with this sort of view on platform governance is that it focuses too heavily on mechanisms used to control actors on a platform, and omits the dynamics of collaboration between platform owners and other actors [24]. Martin et al. [24] analyze the democratic governance structure of a digital platform for sharing economy, and found important measures platform owners can take to implement a democratic model of platform governance. One of which includes distinguishing different types of users on the platform, i.e. users that have the right to participate in decision making processes versus users that do not. Martin et al. call these users owner-members to demonstrate that their role on the platform extends the normal user role. Even though their framework for democratic platform governance is based on platforms used in the sharing economy, some of their arguments can be generalized for digital platforms in general.

Chapter 3

Method

To gain more knowledge about open source platforms and to answer the research questions, a case study of an open source ecosystem was chosen as the best approach. Qualitative data analysis was used on important episodes from the open source platform Npm. This specific case was chosen on the grounds of being particularly interesting given that it is a platform for open source libraries, as well as having its core platform functionality developed and maintained as an open source project. In addition, the project was deemed large enough and sufficiently documented.

3.1 Research Approach

The main choice of research approach is between using a quantitative or qualitative approach to data collection and analysis. Both directions have their places in academia, and one is not necessarily considered better than the other. Instead, it all depends on the context of the research to decide whether or not a quantitative or qualitative approach is the best fit.

A quantitative research approach fits well with research that deal with numerical values in large quantities. Because statistical analyses are used to uncover new relationships and gain knowledge from the data, it is important that the amount of data is large enough to increase the validity of the statistical analyses. This approach fits well with research that try to answer "What"-, "How much"-, or "How many"-type of questions, where the researchers can use data expressed with numbers and values to answer their research questions.

On the other hand, when trying to answer questions that deal with how or why something happens or exists, a qualitative research approach fits best. When trying to gather knowledge about phenomena that can not be expressed with numbers or values, researchers can not rely on statistical analyses and must instead uncover themes and characteristics in the data that enables a qualitative analysis. The data's characteristics can be used to label an otherwise unstructured set of data, which helps researchers to connect themes that emerge.

For this thesis, a qualitative approach was deemed as the best approach to take. The research question is a "How"-type question which requires a qualitative analysis to answer. This thesis aims to gain a deeper understanding of the combination of open source and platform ecosystems in software projects. This is not easily measured with statistical analyses and must instead be answered through an investigation of selected episodes in a case study. In addition to the case study, a review of relevant literature was conducted. This was done in combination with both bottom-up and top-down data collection.

3.2 Data Collection

Empirical data was collected through a case study of the Npm project, which has extensive data openly for the world to view on GitHub [25]. The data consists of discussions about important topics among developers, users, platform owners, and package maintainers. Some of the discussions happen in forum threads found in GitHub Issues where both implementational details and sometimes broader platform strategic decisions are discussed. Other conversations happen in forums specifically made for open discussions with Npm's open source community, such as the Feedback Discussions and RFC (Request For Comments) forums. Feedback Discussions is a forum used for general feedback and discussions, where users often initiate the conversation. RFC is a process where new functionality gets discussed and refined thoroughly before being implemented, the platform owners and key developers play an important part in this process.

Given the huge amount of data available in a large open source project that has existed for years, limiting the search for data is crucial. Exactly how to accomplish limiting the search while still using a bottom-up approach can be challenging. Because bottom-up approaches rely on finding emerging themes and relationships in the data, and letting the data speak for itself, limiting the search is not a trivial task. Initially, digging through the huge open source project that Npm is, was both time consuming and challenging. To better guide the search for interesting episodes, looking for increased spikes in activity was used in the data collection process. For example, an interesting discussion requires a certain number of comments and input by different actor types. Another very useful method was to look for episodes that had been mentioned in media outside of the Npm community, i.e. news articles. The episode concerning the hacking of the ESLint-scope package was covered by news articles and blog posts, which shed light on this particular episode as an interesting starting point. The data collection in this thesis is also affected by top-down impulses. That is to say, data collection and the literature review were carried out in parallel. For example, when the theme of platform governance started to emerge as an interesting theme in the data, the literature review's focus shifted towards gaining more knowledge on this topic, while at the same time collecting more data on platform governance discussions from the case. Cycling between working inductively and deductively made it possible to find emerging themes in the data while still being able to limit the search to fit in the scope of this thesis.

3.3 Data Analysis

Data analysis is an important step in any thesis conducting a case study, which means it is important to get this process right. This thesis used a qualitative approach to data analysis, where themes, relationships, and patterns in the textual data from the case were examined in order to answer the research questions. This process was done in a series of stages defined by their inductive or deductive character. Because the data was documented in the open source project, ready to be more extensively examined and collected, the data collection and analysis could be done somewhat simultaneously. This means that as new interesting themes emerged from the inductive data analysis, more data collection was done in a top-down manner with a focus on the emerging themes. As mentioned in the data collection part, this was an important tool to help limit the search in a huge open source project, but it also helped guide the data analysis process. In the process of collecting more data about previously discovered themes, other interesting themes connected with the old ones started to emerge.

The first stage of data analysis was done in an inductive way, where the initial data was examined and categorized according to which themes emerged. These initial themes lead to a phase of reviewing relevant literature on these topics, before the next stage of data analysis began. The second stage was characterized by working deductively. In this stage the theory from the literature review was applied to the data to understand how the findings relate to prior research. Gaps in the literature on certain aspects of the themes were of particular interest, but also existing theoretical models that could help make sense of the findings. Given that empirical data rarely fits previous theoretical models one-to-one, the process of structuring the data in the context of the state-of-the-art research also included inductive impulses.

The process of working inductively in one stage, and then deductively in the next repeated itself until a final version of the themes was established. Expanding the literature review, refining the research question and collecting more data were done continuously throughout the data analysis process. Table 3.1 outlines the process of deriving themes from the empirical data collected in the case study. The data excerpts are example comments from the episodes, highlighting the essence of each label. Textual data was categorized using a set of labels constructed from both the literature review and from the data itself. From these categories, three central themes were derived which form the basis for the thesis' discussion.

Table 3.1: Themes derived from the empirical data

Data Excerpt	**Label**	**Theme**
It's not clear yet if the ESLint-scope infection was a result of a further upstream infection. There's nothing to say that the pastebin code from this incident is the same as what would be infected in other packages of authors with their credentials compromised [26]	Package Dependencies	Complex Dependencies
This should be a way higher priority to fix. yarn has shown that people need lockfiles, and npm's lockfile system (shrinkwrap) is obviously broken [27]	Ecosystem Consideration	
In a worst case scenario, this leads to even worse security practices; in best case, it catches some vulnerabilities at the cost of ongoing maintenance work [28]	Open Source Security	Security
Malicious maintainer has more direct opportunities for embedding malware in their package [29]	Vulnerabilities	
It will take a long time for the majority of package maintainers to reach this level of security awareness, but it's a project that must be started [26]	New Actor Groups	Platform Governance
These investments include the requirement of two-factor authentication (2FA) during authentication for maintainers and admins of popular packages on Npm [30]	Rule Enforcement	
Our roadmap is heavily influenced by discussions with our community that take place in our public feedback [31] repository	Decision Rights	

3.4 Choosing a Case

The empirical data used in the discussion was collected by conducting a case study of an open source software project. Choosing a suitable candidate for the project was an important decision which was made based on a set of criteria. These criteria were important to determine because studying any arbitrary open source project would not necessarily make it possible to answer the research question. Instead, by using the case criteria, a list of potential case candidates was constructed from which the chosen case was picked.

Knowing which criteria to use was not trivial, and a number of revisions for the case criteria was used before landing on the final version. For example, criteria 2 regarding dominant actors was not considered before realizing that a lot of earlier research on digital platforms included dominant actors, which was found during the literature review. Other non trivial challenges that arose revolved around criteria 3 regarding ecosystems, because how do you define a platform ecosystem? How do you know if an open source project falls under your definition or not? The final version of case criteria is given below:

- Criteria 1, Open source
  Being an open source project is the first and most important criteria. This entails that an open source community exists around the project to maintain it, including developers and users.

- Criteria 2, No Dominant Actor
  The project should be as open as possible, without the existence of a dominant actor with strict control over the project. That being said, every open source project need central actors to control the direction of the project, facilitate the development, or simply pay server costs. Often this role is taken by a software foundation or a private firm, which are regarded as owners of the open source projects. It is important for this thesis that these owners do not control the project too strictly, which could lead to sidelining the open source community.

- Criteria 3, Platform Ecosystem
  The project must have ecosystem aspects such as different actor types participating in the ecosystem, and a governance model.

- Criteria 4, Project Size
  The size of the project must be large enough to involve a decent amount of developers and users in order to study open source aspects. Collaboration between different actors in the project becomes more important as the project grows in size.

- Criteria 5, Well Documented
  To be able to collect empirical data, the project must be well documented through-

out its development life. This entails that important discussions about the projects evolution are documented and open freely for researchers to use.

The case criteria filtered out a set of potential candidates including Npm and Pip. Npm and Pip are two similar case candidates, both being open source package management tools. The large size of the Npm ecosystem compared to Pip means that Npm has a slight advantage for the case study of this thesis. Other cases considered were the .Net and Android ecosystems. Both of these would be interesting cases to look at were it not for Microsoft and Google being too large and controlling actors in the ecosystems.

The case chosen for this case study was the Npm ecosystem. Npm is a package management tool for open source JavaScript packages (libraries) mainly used in web development. In addition to hosting open source packages, the tool itself is developed and maintained as an open source project. This creates an interesting ecosystem heavily focused around the open source mindset, and offers unique aspects that can be studied in order to answer the research questions. It was also very clear early on that the contributors of Npm are very aware that they are actors in both the Npm ecosystem and the larger JavaScript ecosystem.

The Npm project is owned by a private firm, Npm Inc., however their role in the ecosystem is not as dominant compared to Microsoft or Google in .Net and Android respectively. Npm heavily relies on its open source community to contribute to both important decision making regarding its ecosystem as well as development of third-party and core functionality. As an important part of the Node.js ecosystem, Npm also falls under the umbrella of the Node Foundation created in part to better facilitate an open source community. Npm Inc.'s views on developing the platform in collaboration with an open source community is summed up by a quote from the creator of Npm, Isaac Z. Schlueter.

There is tremendous risk if the Node.js Foundation does not decisively expand its community of open source contributors. The Node.js ecosystem is larger than ever. Its continued growth depends on technical innovation, and innovation requires a healthy culture. Any project will suffer without contributions from a broad selection of its members, and any project will lose relevance if its leaders do not actively promote inclusive conduct. -Isaac Z. Schlueter

The platform that Npm has created to upload and share open source packages fits well with criteria 3, regarding platform ecosystems. All these factors combined with the fact that the Npm project is a large and mature open source project with an abundance of documentation throughout its development life, makes Npm a good choice for a case study in this thesis.

Chapter 4

Case

This thesis uses a case study for its main source of empirical data. The chosen case is the Npm (Node Package Manager) project, an open source platform and tool used to distribute open source software (packages) used in the JavaScript runtime environment Node.js. The Npm ecosystem consists of three main parts:

- The Registry
  A public database of JavaScript packages. This is the world's largest database of software [32], and is used by open source developers to upload and share their projects with the world. The database contains the actual software used for the JavaScript packages and also metadata about them. The registry also contains private packages, but the focus in this thesis will be on its larger open source part.

- The Website
  A website used to discover and manage the packages hosted in the registry. The developers of open source packages can use the website to configure their Npm developer experience, i.e. creating users and organizations, or managing access to their packages. Users of Npm use the website to search for and discover packages. Each package has its own page on the website where useful information about it is displayed such as number of weekly downloads and the project's homepage or documentation. The pages often provide guides on how to install and use the packages as well as information about how to contribute to the project.

- The Command Line Interface (CLI)
  A tool run in the terminal used by developers to interact with the Npm registry, i.e. installing or updating packages. This is the main way developers interact with the Npm ecosystem, and is an important part of being a developer in the Node.js environment. For example, after developers have browsed the website and found a package to download, they use the CLI to run the command *npm install <package name>* inside of their project to install the package.

4.1 Case Background

Npm was created in 2009 and had its first official release in 2010. The project was initially created as an open source project, free to use and contribute to by anyone. Even though the company Npm Inc. was founded in 2014 and later bought by GitHub in 2020, the project is still being developed in collaboration with an open source community. Table 4.1 outlines a few important events in Npm's lifespan. Because Npm is the default package manager for Node.js, they are two tightly coupled projects. That is why certain events in the Node.js project impact Npm.

Table 4.1: Npm timeline

2009	Npm created.
2010	Initial release of Npm.
2011	Npm reaches version 1.0.
	Node.js becomes more mainstream as larger companies such as LinkedIn and Uber start utilizing it.
2014	Npm Inc. founded.
	Node Advisory Board was created and aims to establish a more open and inclusive governance model of the Node ecosystem and its open source projects.
	Node Foundation founded.
2016	Yarn Package Manager was released, a direct competitor to Npm.
	The left-pad incident occurs.
2017	Npm version 5.0 is released fixing old bugs and promising a more robust system.
2018	Npm version 6.0 is released with a stronger focus on security.
2020	GitHub acquires Npm Inc.

The problem Npm tackles is that of sharing packaged JavaScript modules among developers, for front-end web apps, mobile apps, and the Node.js environment, which is the focus of this thesis. The problem of distributing software packages is not a trivial problem to solve, as resolving dependencies among packages can be a challenging task for a package management tool. However, the popularity of Npm has increased drastically since its beginning. Figure 4.1 shows the rapid growth of package downloads on the Npm platform from 2013 to 2018, peaking at over four billion weekly downloads in 2018. These numbers are immense and highlight the platform's importance in the JavaScript ecosystem and the internet as a whole.

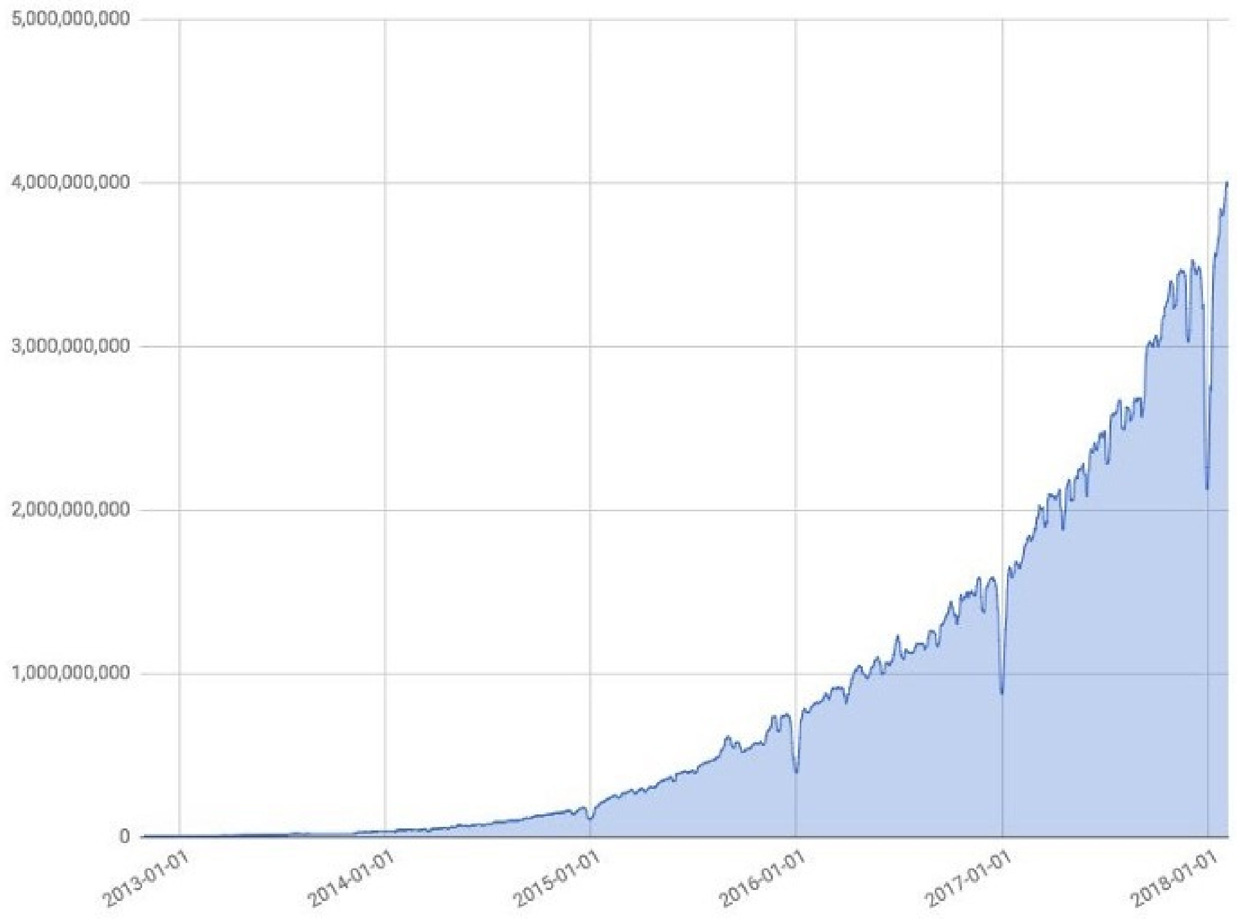

Figure 4.1: Weekly downloads of packages on the Npm platform from 2013 to 2018 [1]

4.2 Episodes

This section describes episodes chosen from the case that stood out as having potential to help analyze the research topic of this thesis. The episodes consist mostly of comments from different types of developers, platform owners, and users found on Npm's discussion forums and GitHub Issues page.

4.2.1 Introducing package-lock.json

To keep track of a project's dependencies, Npm uses files such as package.json and package-lock.json. These files define which Npm packages a project needs, their versions and rules about handling updating to newer versions. Package-lock.json was not introduced to Npm until version 5.0, in 2017. Before that, Npm used what they call shrinkwrap, which enables developers to lock the versions of a project's dependencies just as the package-lock.json file does. In 2012 discussions about resolving dependency problems not covered by shrinkwrap emerged [27].

Some projects might want different dependencies and versions, depending on factors such as operating system or being in a development or production environment. Npm had at this time functionality for both locking dependencies with shrinkwrap, and for defining optional dependencies, however a problem became apparent when projects tried to do both at the same time.

We have to shrinkwrap, since some of our dependencies are loose, and that's no good for production. Still, some of the dependencies are optional, depending on the OS. If we include a shrinkwrap file, then npm doesn't even try to install optional dependencies. And if we shrinkwrap after the optional has been installed, it's no longer optional. -package maintainer

Exactly, in current form npm-shrinkwrap is not optimal as using it installs dev dependencies when building deployment package what is far from optimal, also when hosting on azure or heroku shrinkwrapped dev deps are also being installed -package maintainer

This problem persisted up until 2016, years after first being discovered, and lead to developers being forced to work around the bug or stop using the shrinkwrap functionality altogether.

The workaround I've been doing is: Remove the entry of your optional dependency from your npm-shrinkwrap.json tree. Add it as an optionaDependencies entry in the package.json of your shrinkwrapped project. Not ideal, but easy-ish to automate, and not too difficult to manage. -package maintainer

We just ran into this problem. It loads dev dependencies in production which do not work.."precommit-hook" module for instance looks for the .git directory which doesn't exist and the whole thing fails. We've had to stop using shrinkwrap altogether. The different dependency groups should be separated in the shrinkwrap file so we can still install in production mode. -package maintainer

Other problems with shrinkwrap were also discussed around the same time, which ultimately lead to Npm looking at other ways to implement its dependency system. With the release of Npm version 5.0 in 2017 a new way of handling dependencies was introduced with the file package-lock.json, which replaced the old shrinkwrap system.

In 2016 Yarn was released. Yarn is also a package management tool for JavaScript software, and is a direct competitor to Npm. Discussions regarding the aforementioned problem of optional dependencies continued to emerge in the Npm ecosystem, also after Yarn had been released. Developers in the Npm ecosystem are very aware of what is happening in competing tools such as Yarn.

This should be a way higher priority to fix. yarn has shown that people need lockfiles, and npm's lockfile system (shrinkwrap) is obviously broken -package maintainer

FWIW, yarn has also had issues with optional dependencies -package maintainer

The new system for handling dependencies introduced with package-lock.json solved the bug with optional dependencies, but more importantly it was a way to make cross-package-manager compatibility easier because of its standardised format and usage in other package management tools, i.e. Yarn. The release notes of Npm 5 presents package-lock.json as a way to improve corss-platform compatibility [33].

A new, standardised lockfile feature meant for cross-package-manager compatibility (package-lock.json) -Npm 5 release notes

Prior to the release of Npm 5.0 and package-lock.json, a discussion of how to handle having both shrinkwrap and package-lock.json was initiated by Npm developers [34].

Here is my proposal for how shrinkwrap in npm@5 will work. We will also be introducing a new file, package-lock.json that will fill the same role as npm-shrinkwrap.json in projects that don't otherwise have a shrinkwrap. -Npm developer

With Npm 5, two ways of locking dependencies exist; shrinkwrap and package-lock.json. Older Npm versions only work with shrinkwrap, so it was important for the Npm team to keep this functionality in the Npm ecosystem to ensure backwards compatibility between old Npm versions and new packages.

4.2.2 ESLint-scope hacked

In July 2018 a popular package on the Npm platform named ESLint-scope was hacked. The hackers gained access to the Npm account of one of the developers of the package, and used this access to upload a malicious version of the package to the Npm platform. The open source communities from both Npm and ESLint quickly opened issues discussing the hacking incident regarding what went wrong and how it could be prevented in the future [26], [35], [36]. An incident report [37] and a postmortem [38] of the event was published by the Npm and ESLint team after the episode.

The malicious version of ESLint ran a piece of code upon installation stored on a remote server which stole the user's login credentials such as access tokens and sent them to the hackers. Because the ESLint package is a very popular package with over 400 other dependent packages and millions of weekly downloads, the virus had the potential to spread quickly across the Npm ecosystem. The initial hacked Npm account that made this attack possible was compromised because the owner did not use two-factor authentication on the account.

The main concern of Npm developers is the threat of the virus spreading rapidly across the ecosystem and infecting other large packages. Securing the ecosystem is a recurring theme in these discussions. The developers remember what happened in 2016 with the left-pad incident, where major parts of the internet was taken down due to a small seemingly insignificant package was taken down from the Npm platform. The ripple effect of cascading errors this small package caused in the Npm ecosystem is similar to what could happen during this episode of the ESLint hack.

This could theoretically be a self-replicating virus affecting all packages of all authors whose credentials were compromised, and then all packages that depend on those packages, and so on. The virus could also then change its behavior to do more than leak credentials... Just because that's "all" it did here and just because the pastebin has been removed, that doesn't mean that's what it would do to downstream affected packages.
-ESLint developer

This attack was successful because the hackers could upload the malicious package directly to the Npm platform without going through a GitHub review process. Some of the developers note that requiring a public review process before publishing to the Npm platform would reduce the chances of an attack like this happening in the future.

As a matter of fact, there is no release tag for 3.7.2 on GitHub, so I think it would be great to consider double checking with GitHub repository before publishing any code. This would at least limit the possibility of uploading the malicious code to Npm without having GitHub credentials to tag the release/version. -Npm developer

Other suggestions on security measures include enforcing stricter security rules for large and popular packages, that would affect the whole Npm ecosystem in the event of an attack. These rules could be forcing developers of large packages to use two-factor authentication or having manual audits of the code. One discussion from this episode suggests promoting packages that are made by developers using two-factor authentication. The idea being that developers will be encouraged to think more carefully about their security and enable two-factor authentication if the Npm platform rewards this in some way. In fact, two-factor authentication on accounts is the most mentioned measure from this episode that the Npm ecosystem can take to protect itself from hackers.

The maintainer whose account was compromised had reused their npm password on several other sites and did not have two-factor authentication enabled on their npm account. We, the ESLint team, are sorry for allowing this to happen. We hope that other package maintainers can learn from our mistakes and improve the security of the whole npm ecosystem -ESLint team

The Npm team unpublished the infected version of the ESLint package from the platform to prevent further damage. In addition to this, they revoked all access tokens on the entire platform issued before the attack, ensuring that any stolen access tokens would be useless. In the aftermath of this episode, the Npm team also made it possible for package owners to enforce two-factor authentication on any account that has write access to the package. Following GitHub's acquisition of Npm, new security measures are being implemented in 2022. These include forcing popular packages to enable two-factor authentication.

4.2.3 CVE-Reporting

Common Vulnerabilities and Exposures, or CVE, is a publicly available list of known security vulnerabilities in software systems. CVE reporting in the Npm ecosystem is an important tool developers use to assess the security levels of open source packages they might want to use. This includes both normal developers looking to use Npm packages in their projects, as well as package maintainers wanting to use Npm packages in their own Npm-package. The episode from 2020 regarding CVE reporting revolves around the level of transparency the ecosystem should have about these security threats [28], [29]. When a new CVE report finds that an Npm-package has a vulnerability, the users of this package get notified. This in itself is not a bad idea, however, the CVE reporting system is not perfect and makes the process in the Npm ecosystem flawed. That is because some of the reported vulnerabilities are actually false positives, which over time generates a tension between the end users and the package maintainers. Package maintainers want the ability to hide these security reports about their packages, if they are false positives. If a user sees that a new security vulnerability is discovered in one of the packages they use, they can be left with the choice of trusting the package maintainer's word that this is in fact a false positive. This problem can be solved by letting package maintainers, the people closest to the code, have the control over which CVE reports make it through to the end users on the platform.

I didn't come up with a concrete proposal; I've just had multiple conversations where I wanted, as a maintainer, to be able to say "CVE (Common Vulnerabilities and Exposures) XXX does not apply to my package", and then ensure that "my package" is never the reason a user sees a warning about that CVE. -Npm package maintainer

The idea of giving the package maintainers all the control is not met without resistance. Other Npm users in the discussion are quick to point out the potential attack vector for hackers to just lie about the severity of a CVE and hide it from users.

There's also a potential for 'lying' or mistakes if a maintainer provides this information for their package (though unlikely, I see that as a particularly attractive entry point for an attacker if it existed) -Npm developer

Although the majority of package maintainers have good intentions, it only takes a single infected package to potentially spread a virus to other packages. That is why this threat is taken very seriously.

Malicious maintainer has more direct opportunities for embedding malware in their package, IMO. -Npm developer and security expert

How would you stop a malicious maintainer from hiding a CVE report for the purposes of leaving users vulnerable? -Npm developer

The problem of hiding CVE-reports is a discussion around security for the ecosystem, trust in the open source developers, and convenience for package maintainers. The problem is not only contained to malicious package maintainers, because even developers with good intentions might not have the required software security knowledge needed to know whether a reported CVE introduces a security threat in their package or not.

4.2.4 Npm Public Roadmap

In 2020 the Npm team introduced the Npm Public Roadmap as well as a new feedback process for its users. The roadmap is meant to communicate to the users what is currently being worked on by the Npm team as well as what is scheduled to be worked on in the future. In their blog post about the release of the new public roadmap repository, the Npm team highlight how they welcome transparency and cooperation with the community [39].

We know that maintaining open lines of communication and bringing transparency to our decisions is integral for the future of npm and JavaScript. -Npm developer

Transparency alone is not the same as involving the community in important decision making processes. That is why the public roadmap is based on discussions the community have about new functionality, and suggestions from users or other open source developers are taken seriously and formally discussed in the Npm ecosystem [31].

Our roadmap is heavily influenced by discussions with our community that take place in our public feedback repository. -Npm developer

An example from the case study showcasing how community discussions influence Npm's public road map is found in the issue of displaying package downloads distributed over different versions. This discussion started on Twitter in October 2020, where a user wanted to know if there was a way to see a package's downloads for specific versions of that package. At that time, users could easily see package downloads over different time periods on a package's home page, but this was an aggregation of all downloads across every version of a package. The Twitter conversation resulted in a formal issue being opened on Npm's GitHub in its section for giving feedback.

Show package usage by version. Currently the download statistics allow to see overall adoption of library. But this is not good enough when it comes to understanding adoption of different package versions. The use cases range from getting some understanding of how your beta is performing, to making decisions about shipping a security fix on an older release. Also, having a graph of version mapping and downloads between modules would be massive for better understanding of ecosystem relationships. -Package maintainer

Other package maintainers liked this idea and agreed that it would be useful information about a package to display to users. About a month later, in November 2020 this functionality was added to the public roadmap and scheduled to be worked on in Q1 2021. This is just one example of how the open source community of Npm influences the strategic decisions on the platform, and thus its evolutionary trajectory.

4.2.5 Two-Factor Authentication

The use of two-factor authentication on user accounts is an important security measure any system can take. Two-factor authentication is also used to protect accounts on the Npm platform. However, this was not always possible, and certainly not always required. Npm introduces two-factor authentication in 2017, but as an optional security measure. A year later, in 2018 multiple discussions around the enforcement of two-factor authentication emerged [40], [41]. The precursor for both of these examples was the incident regarding the hacking of the ESLint-scope package, caused by the lack of two-factor authentication on one of its developer's account.

Today's eslint fiasco shows that 2FA can be a powerful tool for stopping unauthorized package publishes. When a package author choose to not use 2FA and gets compromised as a result, the impact don't just affect the author, it impacts everyone depending on that package. -Package maintainer

In light of the growing compromises of NPM packages and malicious actors trying to mass compromise, I've drafted a proposal to enforce 2FA on attempted publish of a package. -Npm user

Users of Npm want changes to how two-factor authentication is handled on the platform. Some want it to be required by all developers, others want to be able to know if a package they use is protected by two-factor authentication or not. The Npm platform could be more transparent about individual packages' security levels by for example showcasing two-factor authentication status on a package's page on the Npm registry website.

Include prominently on the npm website whether a package is published by an author using 2FA and create a repo badge to advertise this. -Package maintainer

Another way to increase security without strictly forcing the use of two-factor authentication is to let users control a setting to only allow packages protected by two-factor authentication to be downloaded into their project. If this becomes widespread among Npm users, package maintainers would be highly incentivized to enable two-factor authentication.

Allow me to specify "Only trust 2FA enabled packages" within my package.json . If a non-2FA dependency is encountered barf loudly so I can hound them and/or remove the dependency. -Npm user

Requiring all package maintainers and Npm developers to use two-factor authentication would certainly drastically increase ecosystem security, but at the same time it is a very controlling enforcement that would affect thousands of individuals. Everyone agrees that increased security is a good thing, however often in the software world one can not get something for free without giving up something else. In the case of two-factor authentication, some users are worried about its implications on automated build and deploy systems that are set up make the process of adding new functionality to packages and deploying them easier.

There are some reasonable cons that could be present, if implemented: 1) Deployment over CI/CD May need to be altered to use one-time pass codes, it would require a master token, a child token. 2) The time to deploy could be increased by a minute or two, however, this doubles as a pro, because the time it takes is merely having you verify a 2FA token to publish. -Npm user

The middle ground approach discussed is to only force the most popular packages to use two-factor authentication. Packages with millions of downloads every week or with a high number of other dependent packages have a huge impact on the ecosystem. Thus, it would be reasonable to suspect that they are larger targets for hackers.

Require packages with enough downloads (I'm not familiar enough with the ecosystem to say what an appropriate amount would be) to use 2FA to further publish packages. -Package maintainer

The middle ground approach is in fact what took place three years later, in 2021. GitHub acquired Npm in 2020, and has been working to increase the security in the Npm ecosystem. GitHub has decided that a selection of popular packages must use two-factor authentication on their developer accounts, starting in 2022 [30]. This decision follows a time of periodical attacks on Npm package maintainers' accounts with the goal of injecting malicious code into public packages.

We periodically see incidents on the registry where npm accounts are compromised by malicious actors and then used to insert malicious code into popular packages to which *these accounts have access. -Chief Security Officer at GitHub*

Publishing malicious code to the Npm registry directly from their own accounts is a valid tactic hackers can use, however the impact of such an attack would not be as severe as an account takeover of a developer account with publishing rights to a popular package. That is because accounts of popular packages can potentially reach millions of users, even users of other packages that depend on the originally infected package.

Even though high-impact account takeovers are relatively infrequent, when compared to direct malware published from attackers using their own accounts, account takeovers can be wide reaching when targeted at maintainers of popular packages. -Chief Security Officer at GitHub

To combat account takeovers, accounts with publishing rights to popular packages on the Npm platform will be forced to enable two-factor authentication starting in 2022.

we will begin to require two-factor authentication (2FA) during authentication for maintainers and admins of popular packages on npm, starting with a cohort of top packages in the first quarter of 2022. -Chief Security Officer at GitHub

Chapter 5

Discussion

The thesis' discussion revolves around three themes emerging from the empirical data collected in the case study. Table 5.1 shows which episodes or phenomena that belong to each theme. Some themes are overlapping, meaning that certain episodes belong to two themes, i.e. CVE reporting and Two-factor authentication.

Table 5.1: Themes in the Npm project

Dependencies	
	package.json
	package-lock.json
Security	
	Two-factor authentication
	Vulnerabilities
	ESLint-scope hacked
	CVE reporting
Platform Governance	
	Npm public roadmap
	Enforcing security measurements
	Top packages forced to use 2FA
	CVE reporting

5.1 Dependencies

The Npm ecosystem is an intertwined network of open source libraries, developers, maintainers, and users. The open source libraries published on the platform, called packages, often make use of other open source libraries around the web, or from within the Npm ecosystem itself. Take for instance a simple package such as the library named *color*, made to handle coloring in CSS easier. To use this package, the user has to install two other packages named *color-convert* and *color-name*, each of which have their own dependencies that also need installing and so on. This is often called a package's dependency tree, which is illustrated for the package *color* in Figure 5.1. Calling it a dependency tree can sometimes be misleading, as it can imply a reasonably simple structure of linear dependencies.

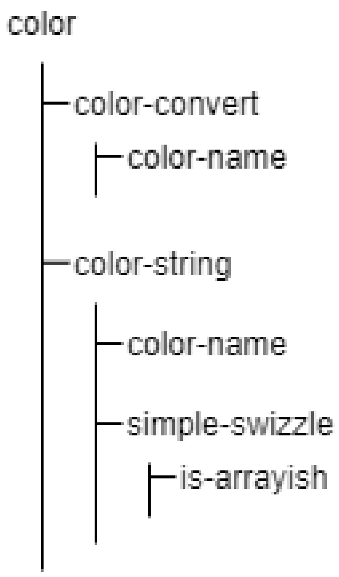

Figure 5.1: Dependency tree of the color package.

For packages with more complex dependency relationships, the structure representing its dependencies are often better described with a graph. This is because as the list of dependencies grows, it is more common to see that packages in the dependency tree share some of their dependencies. Figure 5.2 shows a graph where the nodes are packages and the vertices indicate a dependency from one package to another [42]. This is the dependency graph of an arbitrary package called *waterline*, and is included here to highlight how the complexity of the package relationships quickly becomes overwhelming as the number of nodes in a dependency graph increases. The graph also displays the specific version of each package to indicate that packages exist as different versions of themselves. Sometimes other packages might need specific versions of its dependent packages to work properly, i.e. a package might not work with the newest release of one of its dependent packages, but instead needs an earlier version. When looking at a dependency graph, it becomes easier to grasp how a small change in one package can propagate through the network and cause unforeseen changes in other packages, sometimes many levels up the dependency tree.

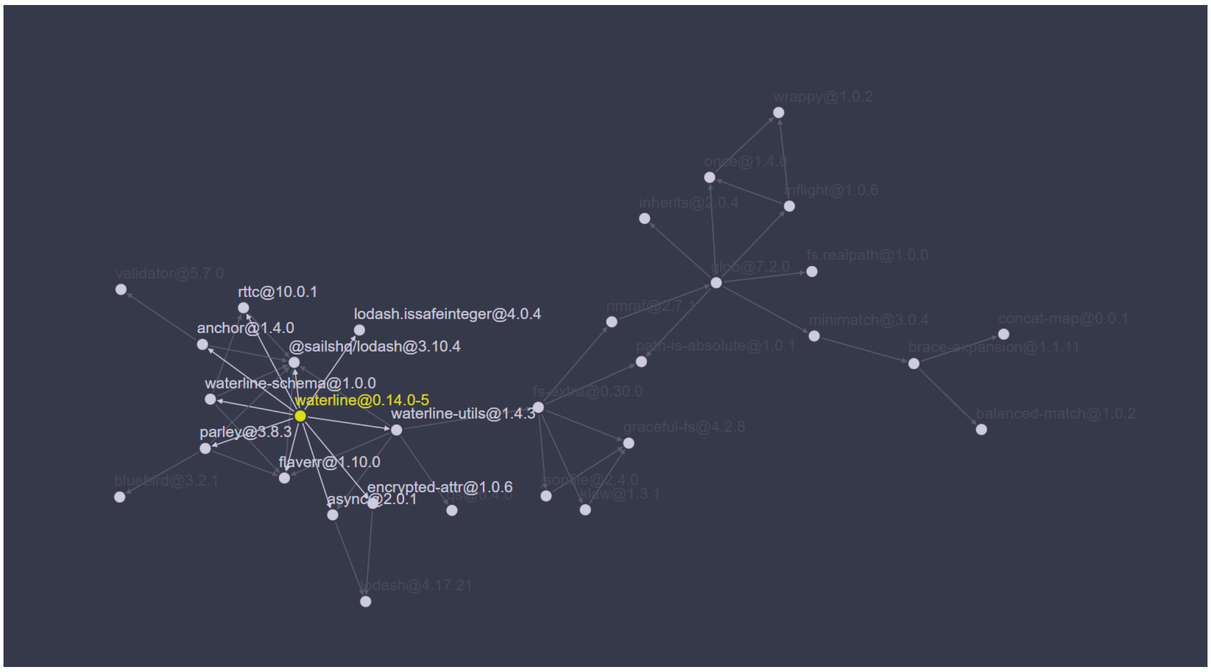

Figure 5.2: Dependency graph of the waterline package. Package names in white indicate a direct dependency of the waterline package, while names in gray indicate a dependency of a dependency.

One challenge Npm tries to tackle is the complexity of dealing with interdependent packages, and one way of dealing with this is through the package files package.json and package-lock.json. These files keep track of a project's dependencies, their versions, and rules for updating them. Package.json was used from the very start of the Npm project, and lists all dependencies a project has. A project might use many other open source libraries and thus have many dependencies, resulting in too much data to be reasonably published to and downloaded from a source control tool. That is why package.json exists, by only publishing a list of a project's dependencies, others can use this list to download the packages they are missing locally in order to run the project. If a developer adds functionality to the project that relies on a new open source library, they just add the name of that library to the list in package.json and publish it to the source control, instead of publishing the whole new library with its source code. This obviously increases performance and usability of the tool, but at the same time it introduces a problem relating to package versioning. Because if you want the same project to run on different machines, the names of the dependent packages is not enough for the project to run consistently across different environments, especially across a long time frame. That is because the open source packages are constantly worked on and improved upon on, and new releases are published fixing bugs or introducing new backwards incompatible functionality. Thus, if you install a package on one machine, and some time later install the same package on a different machine, they might produce different results when used with your project. Npm solves this by including rules for package versions in the package.json file, i.e. a listed dependency might be paired with a versioning field looking like ˆ2.0.2. This tells the system that for this specific dependency, version 2.0.2 should be used, and indicated by the ˆ symbol, the system can install any new versions of this package as long as it is a 2.0.x version. This uses semantic versioning where developers have rules for what constitutes a new version based on the new functionality that is added, i.e. in the Npm ecosystem keeping a package version on the form 2.0.x ensures that only backward compatible bug fixes are added to the package.

The way of handling package versioning with a package.json file as described above was used by the Npm tool from the very start in 2010, and is in fact the standard way in the JavaScript ecosystem being used by other popular package management tools, i.e. Yarn. However, this solution only addresses parts of the versioning problem and does for instance not guarantee consistency of dependencies across environments since it allows different versions of dependency packages to exist in a project in different environments. This is why lock files are generally used by package managers in the JavaScript ecosystem today, lock files can be viewed as a snapshot of a projects dependencies and their exact versions. Npm introduced its now widely used package-lock.json file in 2017, but discussions regarding locking dependencies emerged already in 2011 on the Npm GitHub. The need to lock dependencies is and was not unique to the Npm tool, and even in 2011 the developers at Npm looked to other open source projects for inspiration on how to solve this issue. External impulses like this is common to see in the discussions Npm developers are having throughout the development of the Npm tool.

An example of external impulses influencing the Npm developers is in the release of Npm 5 which in many ways is regarded as an answer to the competing package manager Yarn. In Npm 5 locking dependencies is done through a package-lock.json file, very similar to how its done in Yarn with its yarn.lock file. Even though Npm already had functionality to lock dependencies through something called shrinkwrap, they decided to mirror this functionality in the new package-lock.json, with reasoning such as making it easier for new developers to understand its intended functionality and not having to support later versions of Npm with the old shrinkwrap. The argument of making it easier for new developers is an interesting one, because it often cites familiarity of other tools external to Npm. For instance, just the simple choice of naming the file package-lock.json was up for debate numerous times in its development, where developers arguing for a name closer to that of Yarn's yarn.lock in order to make it clearer what this file is actually for. Just by including the word lock in the filename it becomes easier to see that this file has something to do with locking dependencies, instead of a more ambiguous name such as shrinkwrap. This familiarity principle is also used by Npm developers when giving reasons for mirroring the shrinkwrap functionality to the new package-lock.json in the first place. When developers new to the Npm ecosystem see the lock file they instantly recognize it from other package management tools such as Yarn or Composer, which lets them easier understand its role in the code base, and ultimately makes the onboarding experience for newcomers to the tool better.

5.2 Security

Security in software development is to some degree always a relevant point of discussion. This is also the case for the Npm project, where security is a reoccurring theme among developers. As seen in the literature review, security in open source software projects is a widely debated topic in academic articles. However, a lot of research done focuses on operating systems or server technologies, and a lack of research on open source projects facilitating its own platform ecosystem such as Npm exists.

For Npm, security does not only mean ensuring that no vulnerabilities exist in the code for the Npm tool, but also that no malicious code exists in its ecosystem of open source packages. The former is a theme that does not often come up in the Npm project. Code reviews are done rigorously and often with experienced developers that have worked on the project for a long time. In addition, when having discussions about how to implement certain functionality, key developers that are familiar with that part of the code base are included for guidance. This is possibly one of the reasons why the notion of making the Npm tool specific code secure rarely comes up, it is almost a given that this will be the case. Given that this is exactly one of the arguments presented by Li et al. [17] in order to make open source software less likely to contain security vulnerabilities, it stands to reason to think that the Npm tool itself is secure enough for its purpose.

The discussion in the Npm project regarding security often revolves around how to make the Npm ecosystem secure. This is an ecosystem of open source packages where a vulnerability in one package can spread across the network of dependent packages, like a computer virus jumping from machine to machine, causing a cascading effect of insecure packages. A potential attack vector for hackers to use against open source communities is to pose as a normal contributor and inject malicious code into an open source project. This risk exists in the Npm ecosystem just as it does in other open source projects, however, for hackers to gain the most they usually have to target popular open source projects with lots of users. Big projects like these often have a better code review process before new code is merged into the code base, making it less likely for hackers to be able to sneak their malicious code in. In the Npm ecosystem, the situation is a bit different. Since packages use each other, hackers can find a popular package that is widely used and thus probably has a large open source community. But instead of targeting that specific package, they can target its dependencies, that might be smaller projects focusing on niche functionality and probably has a much smaller community maintaining it. If none of its dependencies are potential candidates for hacking, they can go after its dependencies' dependencies, and so on. Take for instance the waterline package from Figure 5.2, this project has an open source community of about 100 contributors but one of its dependencies, the parley package, only has 6. It could be way easier for a potential hacker to bypass the smaller project's code review process. Changing the parley package could result in a bug in the waterline package that introduces a vulnerability for the hacker to exploit. Another implication of the way the Npm ecosystem works is that hackers can target newly made packages, which according to maillart et al. have more easily discoverable bugs [43].

...as the program ages (and therefore, the probability of finding a vulnerability decreases), switch to newer "easier" programs. -Maillart et al. [43]

Even though the Npm platform lets users download open source packages, the code is only open source on external platforms, i.e. GitHub, no code is exposed on the Npm platform itself. This introduces another attack vector for hackers; bypassing the version control tooling and injecting malicious code straight into the Npm registry which is the source where users download packages from. This is a good example of how the Npm project showcases that open source security is more nuanced than Payne's [16] argument about introducing back doors into open source software being virtually impossible makes it out to be. This type of attack works because there is no way of knowing if the code a user downloads from Npm is in fact the same code that is open sourced in the project's GitHub repository. If a hacker gains access to the Npm account of a package maintainer, they can potentially upload their own code to the Npm registry infecting that package's users. This would not be a problem if users just checked with a project's source control before downloading a new version of a package, verifying that the new version that was published to the Npm registry does in fact exist as a new version on GitHub as well. In practice though, very few if any users would manually go through all their packages every time they run the Npm command that automatically updates all their packages according to the rules set in the package.json mentioned earlier. And since their packages combined often have lots of dependencies, they would have to check all of those as well in order to be completely sure that the code they download is code that exists as open source on i.e. GitHub. This seems as a vulnerability of the Npm platform with high potential to cause problems, therefore measures such as two factor authentication and tagging Npm releases with GitHub releases were done in order to prevent an attack like this from happening. This proved not to be enough however, as only a couple of months later the package ESLint-scope was hacked.

When ESLint-scope was hacked in 2018, attackers gained access to a developer's Npm account and used it to publish their own malicious version of the package directly to the Npm registry. The malicious code would steal login credentials of every user of the package, which was seen as a huge breach given that ESLint-scope had around 2,5 million weekly downloads at the time. In addition, the package had 17 other Npm packages that depended on it, introducing the possibility for the hackers' code to spread to other packages and propagate through the Npm ecosystem.

it's not clear yet if the eslint-scope infection was a result of a further upstream infection. The virus could have modified itself (or have been manually modified, or use a different version entirely) so as to be unrecognizable from the original. There's nothing to say that the pastebin code from this incident is the same as what would be infected in other packages of authors with their credentials compromised. -Npm developer

Another way this hack could spread across Npm is caused by the fact that package maintainers are very likely also users of Npm. This means that if any package maintainers used ESLint-scope, they could have their login credentials to Npm stolen and the hackers would then have a new package totally unrelated to ESLint to override with malicious code. An attack like this, with different ways of spreading across the ecosystem that would be very hard to track, could pose a huge threat to the integrity of Npm. This is why many solutions proposed to fix this incident include measures that involve the whole Npm ecosystem, not just the infected packages.

The key is revoking all npmrc tokens globally before the attacker takes action. -Npm user

It seems like the reasonable thing to do would be: Temporarily halt all package publi-

cations. Search the entire registry for references to this virus and unpublish any infected packages. Globally revoke all authentication tokens. -Npm package maintainer

A larger focus on security had been starting to emerge in the Npm project prior to this event, two-factor authentication on Npm accounts was for instance introduced a year earlier in 2017. And in April 2018 Npm version 6 was released, promising a bigger emphasis on security. For the ESLint incident, the cause of the hack can be viewed as a user error, given that the compromised account did not use two factor authentication. On the other hand, the Npm tool lets anyone publish directly to the Npm registry from their local machine, if this was not the case and instead code was forced to go through an open source platform such as GitHub before entering the Npm registry an attack like this would not be possible in the first place. The question of how to implement security then becomes a question of how Npm want to govern their platform, and the ESLint incident sparked discussions about forcing security measures upon the Npm developers, i.e. requiring two factor authentication or making it harder to publish their code to the Npm registry. It is hard to argue against increasing security in a product, but increased security always comes with a cost. This could for instance be an increase in complexity for automated systems such as continuous integration, which is a concern from package maintainers in the Npm project in the discussions revolving around enforcement of security measures.

As described in the literature review, measuring security in software can be hard. However, the Npm project tries to present users with a security measurement using its auditing feature. This feature scans a project for known vulnerabilities and generates a report that shows each vulnerability a project has paired with a score of how severe a vulnerability is. The scores are simply four categories ranging from low to critical, and the report does not contain any more information regarding the type of vulnerability or other measurements. The audit feature of Npm does somewhat align with the security measurement model described by Schryen et al. [14] in that it includes the severity of each vulnerability in its report. However, one interesting topic of discussion appearing in Npm regarding security measurement is the desire to hide or ignore vulnerabilities given a low severity score. Given that Npm's security measurement allows for a vulnerability to be classified as low severity, many developers want to hide these from their users. This is especially true for package maintainers, that report being overwhelmed with the amount of security warnings for their projects.

Today a small group of maintainers get bombarded with issues and dependency update PRs when a new CVE is reported on popular libraries. Many, if not most, of these are false positives from a vulnerability perspective. This level of noise creates distrust *between security companies/researchers, maintainers, and end users. It also frustrates users and maintainers due to the shear volume of work it creates for them. In a worst case scenario, this leads to even worse security practices; in best case, it catches some vulnerabilities at the cost of ongoing maintenance work. -Npm package maintainer*

This highlights one problem with measuring software security, specifically the labeling of vulnerabilities, because who decides which severity level a reported vulnerability should have? A suggestion from the Npm discussions is to let the people closest to the package in question decide whether or not a vulnerability is severe or not, which makes sense because they know the code best. However, a problem with this logic arises if we recall the arguments from Li et al. [17] and Payne [16] which highlight the importance of security specific knowledge in order to assess software security.

Security flaws are not like ordinary bugs. They are subtle and complex, and often require specialized or in-depth knowledge to identify. -Payne [16]

Package maintainers do not necessarily have the expertise required to make a good assessment of a discovered vulnerability, and thus it could be labeled as having a low severity when in reality it should have been labeled critical. This only becomes more complex and difficult for package maintainers when considering a vulnerability warning for a package could potentially be traced back several levels in its dependency tree. Nonetheless, it is still interesting to note the push from some package maintainers to receive more autonomy and decision power over the measurement of security in their own packages.

5.3 Platform Governance

The Npm platform is governed by its owners, the private firm Npm Inc. They hold most of the strategic and some implementational decision rights related to the evolution of the platform, or what Tiwana calls Platform decisions. It is not uncommon that the platform owners are responsible for majority of platform decisions rights, but where the Npm platform differs from many other platforms is in its open source development of the platform core. This shifts a lot of implementational decision rights towards the community and away from the platform owners, even though they themselves have the last say in what gets implemented or not. Npm as platform owners rely on its open source community to improve upon and contribute to the platform core, but must also ensure the longevity and stability of their platform at the same time. Relying on the work of open source developers means that Npm Inc. must share the strategic platform decision rights with its community in order to facilitate its open source development. The literature often focuses on using platform governance to control content providers in order to facilitate creative output and innovation. For example, Tiwana et al. view platform governance mechanisms as tools to control module developers in a way that fosters innovation in the development of their modules, or third-party apps.

A central governance challenge is that a platform owner must retain sufficient control to ensure the integrity of the platform while relinquish ing enough control to encourage innovation by the platform's module developers. -Tiwana et al. [18]

This way of looking at platform governance is only partly aligned with what we see on the Npm platform. The platform owners of Npm certainly must facilitate innovation of the platform's creative output, however the creative output is not only limited to third party apps but includes the development of the core platform functionalities. Npm Inc. must govern its platform in a way that facilitates an open source community working on these core functionalities. This adds an extra layer of platform governance that is not discussed much in the digital platform literature.

One explicit way Npm includes its open source community in decision making is through the Npm public roadmap. This is a list of features for the npm core platform that is being worked on or scheduled to be implemented in the future. The roadmap is organized into quarterly slots, each with their assigned issues to be worked on in the given time frame. The public roadmap is not only a way for the platform owners to be transparent about the future direction of the platform, but is also heavily influenced by the discussions in the Npm community. This means that some of the strategic platform decisions of Npm is heavily influenced by its community. Recalling Tiwana's way of looking at decision rights partitioning in Figure 2.1, we can see that there are no sliders for a platform's community, instead decision rights exist only on a line and is limited to platform owners and app developers. This way of looking at decision rights partitioning does not fit very well with what we see on the Npm platform. Figure 5.3 expands on Tiwana's sliders by introducing a third dimension where the slider can go; a platform's community. The figure shows how strategic platform decisions are divided between the platform owners, package maintainers (app developers), and the open source community surrounding Npm. Even though Npm Inc. bases its public road map on community discussions, they as platform owners ultimately have the final say in deciding what future direction to take the platform in. That is why the "slider" is moved closer towards platform owners in Figure 5.3.

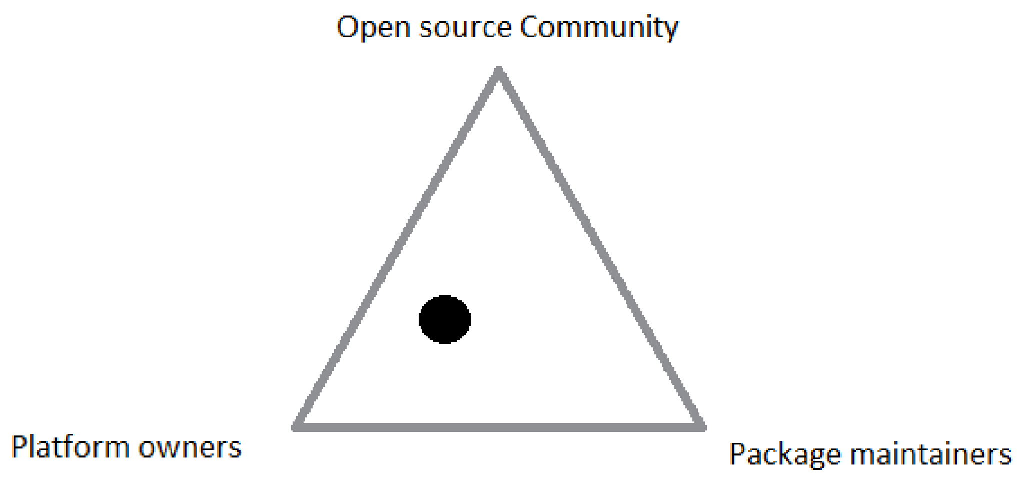

Figure 5.3: Decision rights partitioning of strategic platform decisions on the Npm platform.

Wareham et al. found that opening up a platform's source code was a way to increase desired variability in a platform's output [23]. They argue that when the platform core is open source, third party developers can customize it to fit their functional requirements better, leading to more specialization and variety in the products on the platform. This is somewhat the case for the Npm platform as well, seeing as its core is also open source. However, much of the actual code for new functionality is done by Npm staff or open source contributors focusing on the Npm platform core instead of package development. Instead, package maintainers influence the core functionality more by taking part in discussions and reporting bugs, which in it self can be classified as customizing the core functionality. It does happen that new core functionality is implemented by package maintainers, but not to the same degree that Wareham et al. found in their case study, and often this is done by package maintainers that also frequently contribute to the core functionality as well. The type of actors that have this dual role of not only producing packages for the platform but also developing the core is an actor group without much cover in the academic literature. Actors with dual roles have previously been discussed, i.e. the case of the owner-members found on popular sharing economy platforms by Martin et al. [24].

It might be helpful to draw a distinction between: users with a passing or limited interest in the platform – who do not have rights to participate in democratic decision-making processes; and the more engaged owner-members of the platform - who have *rights to participate in democratic decision-making processes. -Martin et al. [24]*

This is certainly a good example of collaboration of between platform owners and users, but not on the same level as in the Npm ecosystem. Given the existence of the public roadmap where every Npm user can influence the decision making process, the description from Martin et al. would classify every Npm user as an owner-user, which makes the distinction obsolete. In addition, Npm users have the rights to participate in the implementation of new core platform functionalities and maintenance of existing ones. This raises the collaboration between platform owners and other actors to a level not seen in any of the case studies from the literature.

The tension between control and autonomy for platform actors introduced by Wareham et al. has manifested itself on the Npm platform in recent discussions regarding enforcement of security measurements for its actors. Specifically in discussions around CVE (Common Vulnerabilities and Exposures) reporting and enforcing two-factor authentication. User authentication has been an important topic in the Npm ecosystem ever since the hacking of ESLint-scope in 2018 up until today, and other attacks exploiting the lack of two-factor authentication has been done in the mean time. For example, in November 2021 a report from a bug bounty program found a way to publish content to an Npm package using an account without the proper authorization, which is very reminiscent of the attack on ESLint-scope a couple of years earlier. The situation in the Npm ecosystem has lead to the decision that popular packages with a high number of downloads or dependent packages will be forced to use two-factor authentication on their developer accounts. This decision should not come as a shock to the Npm ecosystem, as many discussions revolving around two-factor authentication emerged around 2018, after the ESLint-scope incident.

I'd like to flag all my packages as requiring 2FA in order to publish them. Basically, don't make it a user setting but rather a package setting. If a person with publish rights to the package does not have 2FA set up and tries to publish; disallow the publish. Would that be possible? -Package maintainer

Feature Request: Allow me to specify "Only trust 2FA enabled packages" within my package.json . If a non-2FA dependency is encountered barf loudly so I can hound them and/or remove the dependency. -Npm user

Forcing only popular packages to use two-factor authentication could be seen as a first step to introduce partner levels on the Npm platform. Wareham et al. found that partner levels between the platform owner and third party developers come with different sets of requirements the developers must fulfil, one of which could be to require strict authentication of the developers. In return, a higher level of partnership should result in more value created for the developers.

To obtain higher certification status, partners must typically demonstrate a variety of achievements (to obtain points) or willingly submit to greater levels of control over their processes and outputs. -Wareham et al. [23]

In the case of the package maintainers being forced to use two-factor authentication, the value comes mostly in the form of increased validity by being recognized by the platform owners as more secure. A big difference in what Wareham et al. found and Npm, is that actors in Npm can not choose a desired partner level as they could freely do in the case from Wareham et al. This distinction shifts the tension more towards control instead of autonomy in the particular example of two-factor authentication. Wareham et al. describes an actor's choice of partner level as a market transaction with the platform owners, where they give up control in exchange for some perceived value. Since the top Npm packages have no choice but to use two-factor authentication, it feels more like an enforcement of a standard they must abide by. Another area where we can see a similar tension between autonomy and control in regards to security measurements is the discussions of CVE reports and the implications of giving too much actor autonomy has on ecosystem security. As discussed earlier, it is not a trivial task to decide how to enforce rules about ecosystem security, but interestingly many Npm users and package maintainers favour control over autonomy and are willing to submit to a higher level of control over their processes to achieve the common goal of increased ecosystem security.

Chapter 6

Conclusion

Through studying the Npm project, wider conclusions about both open source and software ecosystems in general can be drawn. The thesis started out by asking the following research question:

How do open source specific aspects affect the evolution of software ecosystems?

The intersection of the two themes offers interesting knowledge that can be applied to both themes individually, and highlights how researchers can understand software ecosystems through open source aspects. As the open source movement continues to expand, applying knowledge learned from studying open source projects in order to understand other areas of research can become more useful in the future.

Open source software has shifted from being a distant and obscure development process towards becoming mainstream and commercially viable software that supports the whole world. In this shift, open source projects and communities have become more interconnected, and instead of existing as individual projects focusing inwards, they cooperate among themselves to a greater degree. The academic debate around open source often takes on the question of what it is, and how it distinguishes itself from traditional proprietary software. This line between open source and everything else is becoming harder to define, as the open source movement continues to influence the rest of the software development world. For example, the matter of governance in open source projects have similarities also seen in traditional software development, especially regarding digital platforms. In addition, a lot of the software in today's world, proprietary or not, relies on open source in some way or form. This further obfuscates the boundary between what is and is not open source software. Take for example proprietary software that ultimately need open source libraries in order to function. Those projects themselves are not open source because developers can not freely use and contribute to them, however, in the end they still rely on the work of open source communities.

Open source projects are becoming more interconnected, i.e. libraries depending on each other, which starts to bear a resemblance to how modules in traditional software projects interact with each other. The challenges seen in open source projects today, are similar to challenges that complex proprietary software projects also face. Security for instance, is a major concern in the open source world. The threat of a virus spreading across an ecosystem of open source projects is similar to how a security vulnerability could spread across other software ecosystems.

The majority of successful digital platforms today are dominated by a governance structure defined by a centralized all-controlling platform owner. This thesis' case study of a digital platform embracing the culture of open source, and the decentralized governance model this entails, shows that the all-controlling platform owner might not be a strict necessity for a platform's success. Because of the limited research on decentralised digital platforms, this thesis contributes to the academic knowledge on digital platforms and software ecosystems. In a world which is becoming increasingly aware of the benefits of decentralization, maybe the trend for future digital platforms will shift towards embracing openness and community governance instead of an all-controlling platform owner. Maybe this is the next area where the open source movement will influence the rest of the software world.

6.1 Limitations

This thesis has explored the topic of open source platforms, but is not without its limitations. One weak point of the thesis is the lack of interviews with developers or users of the Npm platform, which could have improved the empirical basis used in the analysis. Because the Npm project is a huge open source project with hundreds of contributors and thousands of comments and discussion threads, only a very small subset of the total empirical data available was collected. This is expected given the scope of the thesis and the nature of conducting case studies, but is nonetheless a valid limitation to mention.

6.2 Further Work

As the popularity of open source and digital platforms grows, it is likely that the combination of the two will continue to be a relevant topic for future researchers. To increase the validity of the findings of this thesis, similar case studies could be conducted on other open source platforms. Another interesting area to study would be to look at how the ownership structure of open source projects affects their evolution. Some open source projects have big private corporations backing them, while others employ a more democratic ownership structure. It could be interesting to study these organizational differences.

Bibliography

[1] "Blog post, npm weekly 133." https://medium.com/npm-inc/npm-weekly-133-billions-of-packages-downloaded-621a5196593, 2018.

[2] "Attitudes to security in the javascript community." `https://medium.com/npm-inc/security-in-the-js-community-4bac032e553b`, 2018. Accessed on 26.05.2021.

[3] R. Hat, "The state of enterprise open source," 2021.

[4] B. Fitzgerald, "The transformation of open source software," *MIS quarterly*, pp. 587–598, 2006.

[5] V. K. Gurbani, A. Garvert, and J. D. Herbsleb, "A case study of a corporate open source development model," in *Proceedings of the 28th international conference on Software engineering*, pp. 472–481, 2006.

[6] O. Franco-Bedoya, D. Ameller, D. Costal, and X. Franch, "Open source software ecosystems: A systematic mapping," *Information and software technology*, vol. 91, pp. 160–185, 2017.

[7] M. De Reuver, C. Sørensen, and R. C. Basole, "The digital platform: a research agenda," *Journal of Information Technology*, vol. 33, no. 2, pp. 124–135, 2018.

[8] N. Economides and E. Katsamakas, "Two-sided competition of proprietary vs. open source technology platforms and the implications for the software industry," *Management science*, vol. 52, no. 7, pp. 1057–1071, 2006.

[9] "Epic games v. apple." `https://en.wikipedia.org/wiki/Epic_Games_v._Apple`, 2020.

[10] "Why does youtube treat it's content creators so poorly?." `https://www.quora.com/Why-does-YouTube-treat-its-content-creators-so-poorly`, 2016.

[11] P. Krill, "Qa: Why io.js decided to fork node.js." `https://www.infoworld.com/article/2855057/why-iojs-decided-to-fork-nodejs.html`, 2014.

[12] A. Fuggetta, "Open source software—-an evaluation," *Journal of Systems and software*, vol. 66, no. 1, pp. 77–90, 2003.

[13] B. Kogut and A. Metiu, "Open-source software development and distributed innovation," *Oxford review of economic policy*, vol. 17, no. 2, pp. 248–264, 2001.

[14] G. Schryen and R. Kadura, "Open source vs. closed source software: towards measuring security," in *Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 2016–2023, 2009.

[15] E. Raymond, "The cathedral and the bazaar," *Knowledge, Technology & Policy*, vol. 12, no. 3, pp. 23–49, 1999.

[16] C. Payne, "On the security of open source software," *Information systems journal*, vol. 12, no. 1, pp. 61–78, 2002.

[17] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now? an empirical study of bug characteristics in modern open source software," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pp. 25–33, 2006.

[18] A. Tiwana, B. Konsynski, and A. A. Bush, "Research commentary—platform evolution: Coevolution of platform architecture, governance, and environmental dynamics," *Information systems research*, vol. 21, no. 4, pp. 675–687, 2010.

[19] P. Song, L. Xue, A. Rai, and C. Zhang, "The ecosystem of software platform: A study of asymmetric cross-side network effects and platform governance," *Mis Quarterly*, vol. 42, no. 1, pp. 121–142, 2018.

[20] M. Schreieck, A. Hein, M. Wiesche, and H. Krcmar, "The challenge of governing digital platform ecosystems," *Digital marketplaces unleashed*, pp. 527–538, 2018.

[21] J. Manner, D. Nienaber, M. Schermann, and H. Krcmar, "Governance for mobile service platforms: a literature review and research agenda.," in *ICMB*, p. 14, 2012.

[22] A. Tiwana, *Platform ecosystems: Aligning architecture, governance, and strategy*. Newnes, 2013.

[23] J. Wareham, P. B. Fox, and J. L. Cano Giner, "Technology ecosystem governance," *Organization science*, vol. 25, no. 4, pp. 1195–1215, 2014.

[24] C. J. Martin, P. Upham, and R. Klapper, "Democratising platform governance in the sharing economy: An analytical framework and initial empirical insights," *Journal of Cleaner Production*, vol. 166, pp. 1395–1406, 2017.

[25] "Node package manager (npm) github page." `https://github.com/npm`, 2021.

[26] "Eslint-scope 3.7.2 has been hacked 21202." `https://github.com/npm/npm/issues/21202`, 2018. Accessed on 04.05.2021.

[27] "shrinkwrap and optionaldependencies 2679." `https://github.com/npm/npm/issues/2679`, 2012.

[28] "Reduce the noise, work, and frustration from cve reporting 62." `https://github.com/npm/feedback/discussions/62`, 2020. Accessed on 04.05.2021.

[29] "Suggest ignoring a vulnerability by the package maintainer 386." `https://github.com/nodejs/package-maintenance/issues/386`, 2020. Accessed on 04.05.2021.

[30] M. Hanley, "Github's commitment to npm ecosystem security." `https://github.blog/2021-11-15-githubs-commitment-to-npm-ecosystem-security/`, 2021.

[31] "npm public roadmap." `https://github.com/npm/roadmap`, 2020.

[32] "About node package manager." `https://docs.npmjs.com/about-npm`, 2021.

[33] "Npm release notes, v5.0.0." `https://blog.npmjs.org/post/161081169345/v500`, 2017.

[34] "spec: Describe npm-shrinkwrap.json and package-lock.json 16441." `https://github.com/npm/npm/pull/16441`, 2017.

[35] "Virus in eslint-scope? 39." `https://github.com/eslint/eslint-scope/issues/39`, 2018.

[36] "eslint-scope attack." `https://gist.github.com/hzoo/51cb84afdc50b14bffa6c6dc49826b3e`, 2018.

[37] "Postmortem for malicious packages published on july 12th, 2018." `https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes`, 2018.

[38] "Compromised version of eslint-scope published." `https://status.npmjs.org/incidents/dn7c1fgrr7ng`, 2018.

[39] M. Borins, "Introducing the npm public roadmap and a new feedback process." `https://github.blog/2020-10-22-introducing-the-npm-public-roadmap-and-a-new-feedback-process/`, 2020.

[40] "Require or at least broadcast package 2fa." `https://npm.community/t/require-or-at-least-broadcast-package-2fa/458`, 2018.

[41] "Enforce 2fa on publish." `https://github.com/npm/npm/issues/21207`, 2018.

[42] "Visualization of npm dependencies." `http://npm.anvaka.com/#/view/2d/waterline/0.14.0-5`, 2021. Accessed on 04.10.2021.

[43] T. Maillart, M. Zhao, J. Grossklags, and J. Chuang, "Given enough eyeballs, all bugs are shallow?," in *Workshop on the Economics of Information Security*, 2016.