UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Liisa Sakerman

Analysis of Third-Party Dependencies – A Systematic Literature Review

Master's Thesis (30 ECTS)

Supervisor: Kristiina Rahkema, MSc

Tartu 2022

Analysis of Third-Party Dependencies – A Systematic Literature Review

Abstract:

The aim of this thesis is to provide an aggregate view of the relevant studies done in the field of third-party dependency analysis. Developers often use and rely on third-party libraries in their projects and package managers help to handle and keep track of those dependencies. This paper presents a systematic literature review in the domain and creates an overview of the contributions of the empirical studies. Most of the studies focused in their aims on the third-party dependency maintenance aspects and their security implications. The problems they discussed were related to these aspects as well, with suggestions to incorporate more automated tool support to aid with the maintenance. Such tools were also developed in the scope of some of the studies. Studies were data-heavy, where the metadata was mined from open-source databases or package manager repositories – most investigated package managers were Maven and npm. For future work it was suggested to carry out the existing research for other package managers, extend the research to the mobile domain and complement quantitative approaches with qualitative methods.

Keywords: Third-party dependencies, software library ecosystems, systematic literature review

CERCS: P170 Computer science, numerical analysis, systems, control

Kolmandate Osapoolte Teekide Analüüs – Süstemaatiline Kirjanduse Ülevaade

Lühikokkuvõte:

Magistritöö eesmärk on anda ülevaade asjakohastest teadustöödest kolmandate osapoolte teekide analüüsi teemal. Arendajad kasutavad tihti oma tarkvarasüsteemides kolmandate osapoolte teeke ja tuginevad paketihalduritele, mis aitavad sõltuvusi hallata ja jälgida. Töö annab süstemaatilise ülevaate kirjandusest antud valdkonnas ja loob ülevaate empiiriliste uuringute panusest. Enamik uuringuid keskendus oma eesmärkides kolmandate osapoole teekide korrashoiu aspektidele ning nendega seotud turvariskidele. Töödes väljatoodud probleemid olid samuti seotud nende aspektidega, koos ettepanekutega lisada hoolduse abistamiseks rohkem tuge automatiseeritud tööriistade kaudu. Selliseid tööriistu töötati välja ka mõne uuringu raames. Uuringud olid andmemahukad, kus metaandmed kaevandati vabakasutusega andmebaasidest või paketihaldurite hoidlatest – enim uuritud

paketihaldurid olid Maven ja npm. Edaspidiseks tööks pakuti välja viia läbi olemasolevad uuringud teiste paketihaldurite raames, laiendada uuringuid mobiilivaldkonnale ning täiendada kvantitatiivseid tehnikaid kvalitatiivsete meetoditega.

Võtmesõnad: Kolmandate osapoolte teegid, tarkvara teekide ökosüsteem, süstemaatiline kirjanduse ülevaade

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Table of Contents

Introduction

It is a common practice nowadays for developers to use and rely on third-party libraries in their projects [1]. Third-party libraries provide a specific functionality and are useful for developers to avoid rewriting code for common solutions [1]. A dependency arises when an application is dependent on some third-party software and package managers can help with handling and keeping track of those dependencies [2]. First package managers emerged in late 1990s [3, 4] but most were developed after the success of JavaScript's Node Package Manager (npm), created in 2010. The usage of package managers and third-party libraries is not a new practice and due to its popularity a lot of research has been done in this domain. For example, one area of interest has been researching, how using third-party libraries can introduce security risks [5, 6].

There does not, however, exist a comprehensive study that summarizes these papers. This thesis aims to bridge the gap by conducting a systematic literature review and providing an up-to-date overview of the empirical work done on the topic of third-party library analysis. This kind of review is important, as it summarizes the current collective knowledge on the subject and provides a balanced view including already established ways as well as conflicting findings. The review is essential for mapping what is already known about the third-party dependency analysis and finding out knowledge gaps. That way it is possible to identify prospects for future research.

Therefore, the aim of this thesis is to provide an aggregate view of the relevant studies done in the field of third-party dependency analysis. This paper looks into what exactly has been studied in the domain by:

- identifying main data sources used, problems investigated, their solutions, third-party package managers mentioned, methods applied and new tools built
- summarizing the future work proposals.

This creates an overview of the contributions of the empirical studies.

The following research questions (RQs) were posed and are answered in this thesis:

RQ1. What are the main research objectives?

RQ2. What are the main data sources used for third-party dependency analysis related studies?

RQ3. Which methods have been used to study third-party dependencies?

RQ4. What are the main problems found with third-party dependency usage?

RQ5. How can the problems related to third-party dependencies be minimized?

RQ6. Which third-party package managers were used in the studies?

RQ7. Which new tools have been created in the scope of third-party dependency analysis related studies?

RQ8. What would be the future work in the topic of third-party dependency analysis?

The first chapter of this thesis describes the idea of third-party software components, package managers and summarizes related work on the topic. The second chapter gives overview of the methodology, describing the systematic literature review process. The third chapter presents the results. The fourth chapter contains discussion and the final chapter presents the conclusions.

1. Background and Related Work

This chapter gives an overview of relevant background information on the topic of third-party libraries and package managers; related previous work is presented as well.

1.1. Third-Party Software Libraries

Third-party software libraries are reusable components developed by someone other than the original vendor and that are not the system's own elements [7]. The idea of third-party libraries was introduced with the concept of component-based software, where a system is reduced to separately developed software components, as opposed to a monolithic system [8]. In more recent times with numerous open-source materials available, developers do not need to "reinvent the wheel" anymore, but rather can use already existing libraries in their code that fulfil their needs or solve common problems, to avoid code duplication or over-engineering [1]. Third-party libraries provide tailor-made functionalities [9]; developers only need to implement the needed pieces and embed them into their own code [10], which saves the development time and cost. Another advantage to using libraries is that the code is already tested and usually does not affect the project's overall quality and stability [11].

Extensive library usage can however bring numerous problems. These can include issues when having to switch libraries [12], frequent update needs [12], "dependency hell" [12], lack of support [13] or security issues [5, 6]. Even deciding which dependencies exactly to use can be troublesome for some developers [1]. One biggest disadvantage to the usage of third-party libraries is that the developers have to fulfil the code maintenance tasks, such as having to decide if and when to update the current library dependencies [14]. Lack of maintenance often causes third-party libraries to rely on vulnerable code, according to Kula et al. [12] 81.5% of the studied systems still kept their outdated dependencies because updating to a newer version is seen as extra workload and responsibility.

1.2. Package Managers

Projects can have a multitude of dependencies and dependencies can have sub-dependencies of their own [2]. For bigger projects, handling the dependencies manually can become troublesome and package manager systems can help with installing new dependencies and managing existing ones [2].

First systems dpkg/apt [4] and RPM [3], that can conceptually be called "package managers", were created in 1990s. Today, the more popular package managers are for

example npm for JavaScript and PyPI for Python, but almost every widespread programming language has their own package manager [15]. Since its creation in 2010, npm has grown into a collection of over 2 030 000 third-party libraries, as of October 2021 and after npm's launch and success there has been a new package manager released almost every year since [16].

1.3. Related Work

The topic of third-party dependency analysis has been relevant since the 1990s. Since then, a multitude of papers has been published that touch upon different aspects of third-party components, package managers and dependency issues. There has not however been a comprehensive systematic literature review carried out to gather all the existing knowledge. This is what the current thesis aims to do.

Zhan et al. [17] conducted a first systematic literature review on Android third-party library (TPL) related studies. They collected 74 primary research papers from 2012-2020 closely related to Android third-party library analysis and summarized the taxonomy of existing studies into four dimensions: research objectives, targeted libraries, type of TPLs and type of program analysis. From the research objectives point of view, one third of the collected papers focused on the security issue analysis on TPLs. For targeted libraries, the research papers were almost divided to half: 47% of the papers were focused on ad libraries, while the rest analyzed no-ad libraries. Almost all studies concentrated on Java libraries and half of the studies used static code analysis.

Third-party dependency usage has been studied a lot in the context of security. Umm-e-Laila et al. [18] studied the third-party software adoption factors for critical IT infrastructure by conducting a review of the relevant literature and proposed a framework to help the industry to have increased confidence in third-party library usage. Zimmermann et al. [5] systematically studied the security risks in the npm ecosystem. The openness of npm ecosystem has caused security risks, namely incidents of single libraries breaking or attacking the software running on millions of computers. They found that single libraries could indeed impact large parts of the entire ecosystem and lack of maintenance could cause dependencies on vulnerable code. They concluded that the recent security incidents are not merely individual cases but likely the first signs of a larger problem. Lauinger et al. [6] conducted a comprehensive study on JavaScript library usage and resulting security

implications. They concluded as one of their findings that more than 37% of the studied websites use at least one library with a known vulnerability.

Inspired by the problem of software projects depending on vulnerable code, it has also been a topic to investigate, how and when developers update their library dependencies. Kula et al. [12] found that updates of dependencies are not regularly practiced, 81.5% of their studied systems remained with an outdated dependency, additionally oftentimes developers are not aware of the problems and risks. Pashchenko et al. [13] and [19] answers the question, as to why developers keep the known vulnerabilities. The most common reason is that developers usually lack the resources to cope with possible breaking changes, and therefore avoid updating dependencies unless the vulnerability is not a severe security risk, widely known or doesn’t require significant efforts.

These were just a few of more researched topics, and it will be beneficial to look into the papers more to identify other research areas as well.

2. Research Methodology

The thesis follows the guidelines for a systematic literature review (SLR) set by Kitchenham et al. [20]. SLR follows clear methodological steps to research literature [20] which allows the summarization of the relevant literature, and is also suitable for finding research gaps that, in the end, can be considered for future research. SLR can be divided into three main phases: planning, conducting and reporting. The review was performed by identifying appropriate reports and extracting the data according to the research questions.

2.1. Planning the Review

This chapter will present the planning process of the systematic literature review.

2.1.1. Scope and Research Questions

The first step in a systematic review is to define the research scope. This thesis aims to search for papers, that report on studies related to third-party dependency analysis. The research questions set according to the aim are:

RQ1. What are the main research objectives?

Justification. Identifying the main topics of interest outlines the work done. It gives the researchers an overview of already covered areas, helps them pinpoint lacks and challenges.

RQ2. What are the main data sources used for third-party dependency analysis related studies?

Justification. It is useful to map out the data sources used, to prove the soundness of the studies, allow for replication, help with finding data sources for future research and bring out possible gaps in the starting points.

RQ3. Which methods have been used to study third-party dependencies?

Justification. Identifying the methods used allows for replication, assessing their limitations and finding prospects for further research.

RQ4. What are the main problems related with third-party dependency usage?

Justification. As discussed previously, third-party library usage is a double edged sword that can have many setbacks. Numerous papers have analyzed these problems and pinpointing them is needed in order to find possibilities for minimizing them.

RQ5. How can the problems related to third-party dependencies be minimized?

Justification. With the problem analysis also come suggestions on how to mitigate them. Collecting the solutions can better help practitioners to tackle the known problems about third-party components, draw more attention to them and be helpful for practitioners who are in need of a collective source of solutions.

RQ6. Which third-party package managers were used in the studies?

Justification. It will be interesting to know which package managers have been more popular among the researchers, and which have not yet been used in empirical studies.

RQ7. Which new tools have been created in the scope of third-party dependency analysis related studies?

Justification. It would be useful to collect for practitioners' tools, that have been created for activities related to third-party libraries or as solutions for some limitations with dependency analysis.

RQ8. What would be the future work in the topic of third-party dependency analysis?

Justification. To give ideas for future research, suggestions from the existing articles will be collected.

2.1.2. Search Strategy

Search Terms. Search strings were formulated from keywords based on the scope and research questions by identifying the main topics and terms and finding synonymous words. The main idea of the thesis is to find publications, that use third-party libraries, therefore one of the keywords was "third-party"; in some papers, "third-party" can also be called "open-source". The first group of keywords is: g_1={"third-party", "open-source"}. The second group of synonymous keywords refers to the library aspect: g_2={"library", "package", "component", "software"}. And finally, to get the most accurate results, the third group will limit the results to dependency analysis: g_3={"dependency"}. The search string was constructed using the three groups as follows: *search term* = $g_1 \cap g_2 \cap g_3$, resulting in string: *(third-party OR open-source) AND (library OR package OR component OR software) AND (dependency)*. The search string took into consideration the Ebsco Discovery Service library database's specificity: database also found the plural versions of the keywords and for hyphenated words searched also the same phrase without the hyphen.

The search string was then tested to see, if it found all seven papers from a starting set [5, 12, 21, 22, 23, 24, 25], which was given to the author from her supervisor before any search,

as a benchmark. After testing this search string, the term "ecosystem" was also added to it. This was done to indicate the library ecosystem side of the topic, as some papers don't explicitly refer to *third-parties* or *open-source* components, but rather to the package manager ecosystem as a whole. Therefore, the new search string was the following: *(third-party OR open-source OR ecosystem) AND (library OR package OR component OR software) AND (dependency).*

This string was able to find five [12, 21, 22, 23, 25] of the seven papers from the starting list, however the missing two papers [5, 24] were not present in the chosen digital library at all, so it is understandable, why those two were not found. Therefore, the search string was suitable, as it found five of the five papers that could have been found.

Digital Libraries. The digital library chosen for this study was Ebsco Discovery Service [26]. This database includes results from numerous different digital libraries such as IEEE Xplore and ACM and was for this reason chosen for the main search pool. The full list of content providers present in Ebsco Discovery Service is brought in Appendix I. As it was discovered, that Ebsco did not have two papers from the starting list of papers, it was decided to carry out an additional search on Google Scholar [27]. The search string for Google Scholar had to be modified to also include plural versions of the keywords, as Google Scholar otherwise would not find them. As using the wildcard character "*" yielded too many irrelevant results, this was the resulting search string for Google Scholar: *(third-party OR third-parties OR open-source OR open-sources OR ecosystem OR ecosystems) AND (library OR libraries OR package OR packages OR component OR components OR software) AND (dependency or dependencies).*

2.1.3. Selection Criteria

For the study selection exclusion and inclusion criteria were set. Exact duplicates were removed from the search list by Ebsco Discovery Service itself.

The exclusion criteria (EC):

EC1: Papers that did not have the Ebsco Discovery Service's Source Type „Academic Journals", „Conference Materials", „Reports" or „Dissertations/Theses". Source Types „Magazines", „Books", „Trade Publications", „News" and „Reviews" were discarded as they are not peer-reviewed and would likely not contribute to the empirical research.

EC2: Papers written in some other language than English.

EC3: Papers that are not related to third-party dependency analysis in software domain. Occasionally, the keywords would find papers, that did not belong in the computer science field which were then discarded.

EC4: Papers for which the full text was not found.

Inclusion criteria (IC):

IC1: Papers analysing topics related to the third-party dependency analysis and management, e.g. how the dependencies are introduced, updated, maintained in a project.

IC2: Papers analysing dependencies of whole package registry ecosystems.

IC3: Papers discussing the problems related to third-party dependencies.

IC4: Papers suggesting solutions to solving problems related to third-party dependencies.

IC5: Papers creating a new tool related to third-party dependency analysis.

Additional quality criteria (QC) were set to assess the quality of the studies:

QC1: Are the aims, methodologies and results of the study clearly stated?

QC2: Are the main goals of the paper third-party dependency analysis related? Many articles may touch upon the topic of third-party dependencies, but carry out a study with different aims.

QC3: Is the paper extensive? According to the international academic rules, publications less than five pages in a double-column format or less than seven pages in a single column format are considered to be short papers [17] and so were not included in this research as they would likely be only ideas of research and not yet fully carried out.

2.1.4. Data Extraction Table

Data extraction table allows for a methodological and consistent way of data collection. Template for data extraction form is presented in Table 1. As for identification data items, article title, authors' names and publication year were collected to allow for easy identification. According to the respective research questions, data sources, research objectives, problems, mitigation techniques, package managers, methods, new tools and future work ideas were collected as data items. The filled data extraction table can be found attached under extras.

Table 1. Data extraction table template.

Data Item	**Value**	**RQ**
Article Title	Name of the article	Identification Data
Author name(s)	Set of names of the authors	
Publication year	Year when the study was published	
Venue	Name of the venue, where the article was published	
Data Sources	Main data sources used in the study	RQ1
Research Objectives	Main research objectives of the study	RQ2
Problems	Main problems found with third-party dependency usage	RQ3
Mitigation Techniques	Solutions to the mentioned problems	RQ4
Package Managers	Third-party package managers used in the study	RQ5
Methods	Methods used in the study	RQ6
New Tools	New tools created in the scope of the study	RQ7
Future Works	Future work suggestions	RQ8

2.1.5. Data Synthesis

Each collected paper in the final list was reviewed and read through, and corresponding data fields were filled in the data extraction table. Research questions were then answered with the data collected from the papers by carrying out descriptive synthesis – for each research question common trends were found and results were summarized. The technique used for the data synthesis was grounded coding [28], where the codes aka categories were derived directly from the data extraction table. This approach was used for RQ1, RQ3, RQ4, RQ5 and RQ8, where the text extracted from the articles that answered the research question was coded. An example explaining this process in presented in Appendix II for RQ1, the same was done for all research questions that needed coding. As RQ2, RQ6 and RQ7 already included the answers to the research questions in the data extraction table, no coding was necessary for them.

2.2. Conducting the Review

This chapter describes the search process and selection criteria application with first results of demographics.

2.2.1. Executing the Search

Search was first conducted on the Ebsco Discovery Service using the constructed search string: *(third-party OR open-source OR ecosystem) AND (library OR package OR component OR software) AND (dependency).* This search yielded 4043 hits as of 04.12.2021. No time period restriction was set on the search to not exclude any relevant papers. Ebsco Discovery Service also offers to run the search on specific fields of the papers' metadata. This was not used and no specific field was selected, as to not limit the possible results.

2.2.2. Applying the Selection Criteria

For the found papers the exclusion and inclusion criteria where applied for elimination, following a top to bottom approach. If a paper met an exclusion criterion or it failed an inclusion criterion, it was discarded without considering further criteria. The process is illustrated on Figure 1.

First, the exclusion criteria were applied. Papers were filtered according to the Source Type filtering of Ebsco Discovery Service and papers that did not have the Source Type „Academic Journals", „Conference Materials", „Reports" or „Dissertations/Theses" were discarded – 2396 hits remained.

The second exclusion criterion was applied; non-English papers were discarded – 2177 hits remained.

The third exclusion criterion was applied; papers that were not related to third-party dependency analysis in software domain were discarded – 320 hits remained. For this step, the papers' titles and, at times, abstracts were reviewed. Titles that clearly were irrelevant were discarded. In case of any doubt whether the study would be relevant or not, it was included to analyse more thoroughly in the next stage

The fourth exclusion criteria were applied and one paper was discarded as it did not have the full text publicly available – 319 hits remained.

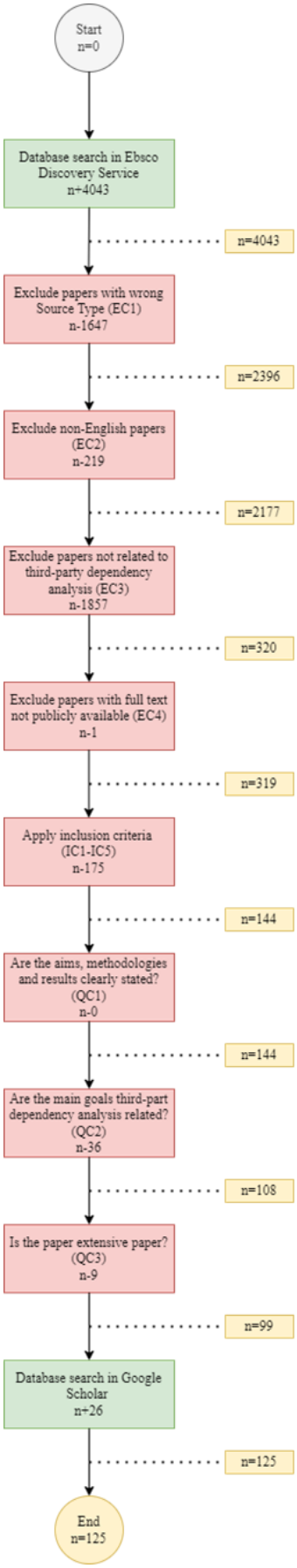

Figure 1. Study selection process.

Next, the inclusion criteria were applied to determine, which papers to include in the study. For this, the papers were reviewed by reading the introduction, methodology and conclusions. In case of uncertainty, the full text was read. Afterwards, 144 hits remained. Lastly, quality criteria were applied to assess the quality of the articles. The first quality criterion did not discard any papers. This shows good quality of today's research on the topic as the aims, methodologies and results of the studies were stated clearly.

The second quality criterion was applied. There existed papers, that talked about the topic of third-party analysis in their introduction or literature review, but their main goals were related to some other topic. In that case such articles were discarded as this thesis aims to study papers, which main purpose is in the third-party library domain. 108 papers remained.

The third quality criterion discarded 9 papers; eliminated articles were short papers and did not classify as extensive papers that this paper aims to find. Therefore, the final number of papers from primary search was 99.

Next, to complement the Ebsco Discovery Service search, further search was conducted on Google Scholar, to find an additional list of papers. The previously defined search string was used: *(third-party OR third-parties OR open-source OR open-sources OR ecosystem OR ecosystems) AND (library OR libraries OR package OR packages OR component OR components OR software) AND (dependency or dependencies).* This resulted in about 56300 hits, but Google Scholar only shows the top 1000 most relevant results. From those 1000, additional 26 papers were selected by already applying all the study selection criteria. The two papers [5, 24], which were missing from the Ebsco Discovery Service, but were presented in the starting set of articles, were also found from this search. Therefore, the final list of papers to be analyzed consisted of 125 articles.

2.2.3. Demographics

Figure 2 shows the yearly distribution of the selected papers' publication years. The earliest paper in this systematic literature review is a master's thesis from 2002 [29], that already addressed problems related to automatic third-party dependency resolution in the Debian ecosystem. Next notable papers in 2007 [30] and 2009 [31] touched upon Debian as well. From mid-2010s as new package managers were starting to emerge, the topic gained more popularity. The interest has been growing in the latest years showing an increasing trend, with 30 papers published in 2021. Even though the search for publications was done at the end of 2021, 4 papers to be published in 2022 were already found as well.

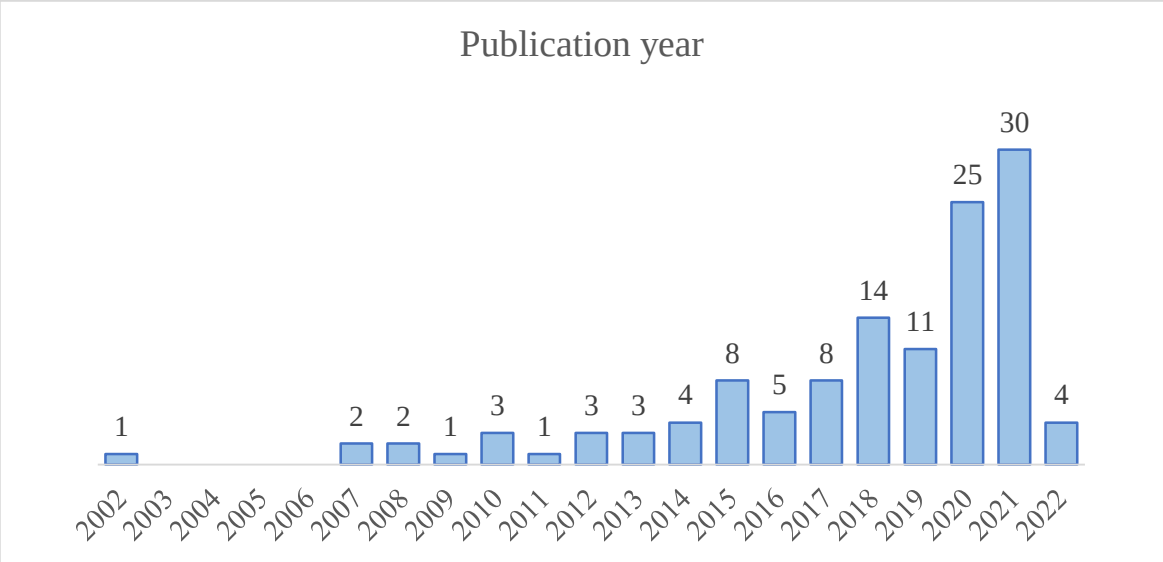

Figure 2. Yearly distribution of published papers.

Venues, where the articles were published in, are pictured on Figure 3. Two major venues where the chosen articles were published were IEEE Transactions on Software Engineering and Empirical Software Engineering – both had published 12 papers of this reviews' selection. Another bigger journal was Journal of Systems and Software with 6 publications. More domineering conferences in this selection were IEEE/ACM International Conference on Software Engineering with eight papers, and IEEE International Conference on Software Analysis, Evolution and Reengineering and IEEE/ACM International Conference on Mining Software Repositories both with seven papers. There were also three papers from workshops and three master's thesis included. 44 publications were from individual venues.

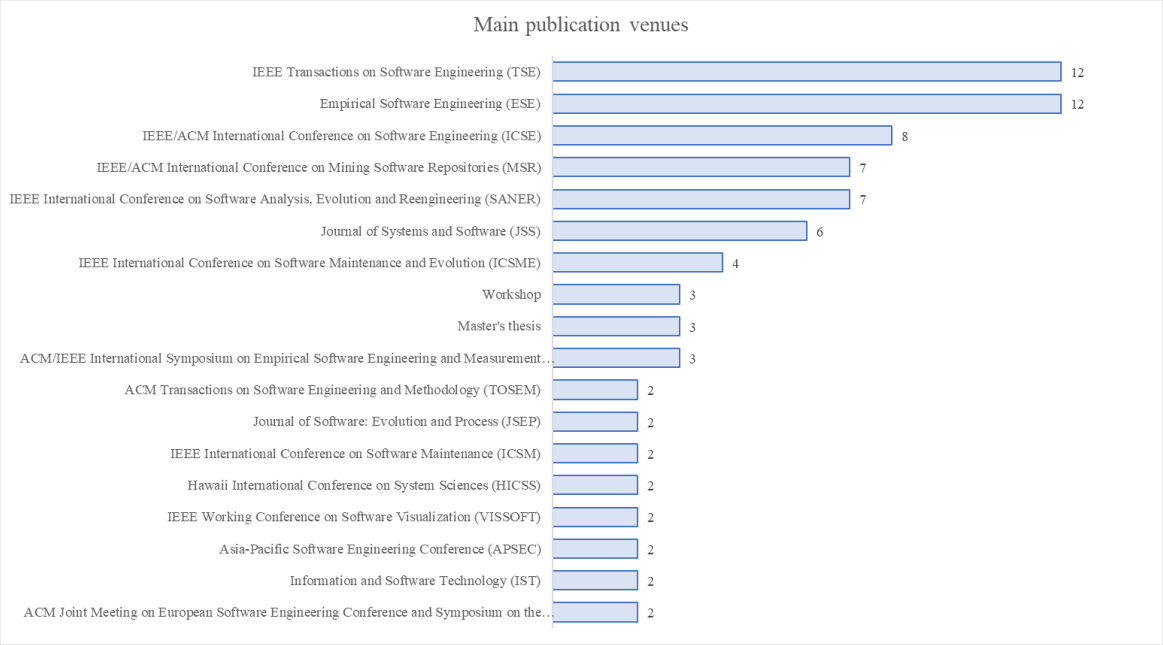

Figure 3. Main publication venues.

3. Results

In this chapter, the data collected from the studied articles is presented and each research question is answered. The terminology used in this thesis is explained in Appendix III.

3.1. Research Question 1: Research Objectives

Studied articles' research goals were extracted and analysed to understand what kind of papers have been written, what has been already researched and which aspects are lacking. The research objectives can be classified into four categories as seen in Table 2: third-party dependency maintenance related studies; studies focusing on the security aspects; studies that explored the social aspects of third-party dependency usage; and studies focusing on researching whole ecosystems.

Table 2. Categorization of research objectives.

	Research Objectives	Studies	Package Managers
Maintenance	Updating/ downgrading dependencies	[12], [14], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44]	Debian, Maven, npm
	Dependency/ conflict resolution	[13], [29], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57], [58], [59], [60], [61], [62], [63], [64]	Alire, Bower, Cabal, Cargo, Clojars, CPAN, CRAN, Debian, dnf, Go, Maven, npm, NuGet, Opam, Packagist, Paket, PyPI, RPM, RubyGems, Spack,
	Detecting/ visualizing dependencies or libraries in projects	[65], [66], [67], [68], [69], [70], [71], [72], [73]	Maven, npm, Spack
	Dependency versioning	[74], [75]	Atom, Cargo, CPAN, CRAN, Dub, Elm, Haxelib, Hex, Homebrew, Maven, npm, NuGet, Packagist, Pub, Puppet, PyPI, RubyGems
	Bloated/ imitated dependencies	[76], [77], [78], [79]	Maven
	Technical lag	[25], [80], [81], [82]	Atom, Cargo, Dub, Elm, Haxelib, Hex, Maven, npm, NuGet, Packagist, Pub, Puppet, PyPI, RubyGems
	Licensing	[83], [84], [85]	npm

Security	Library vulnerabilities	[5], [6], [12], [13], [17], [24], [34], [39], [86], [87], [88], [89], [90], [91], [92], [93], [94], [95], [96]	Bower, Maven, npm, NuGet, PyPI, RubyGems
	Detection and mitigation	[35], [97], [98], [99], [100], [101], [102], [103], [104], [105], [106], [107], [108]	Cargo, Maven, npm, Xargo
Social	Technology adoption	[18], [43], [101], [102], [109], [110], [111], [112]	CRAN, Maven, npm
	Recommendation systems	[1], [36], [113], [114], [115]	Composer, Maven, npm, PyPI
	Interactions	[114], [116], [117]	Atom, Bioconductor, Cabal, Cargo, CocoaPods, CRAN, CPAN, Go, Hex, Luarocks, Maven, npm, NuGet, Packagist, PyPI, RubyGems, Stack
	Developers' behaviour	[12], [13], [14], [75], [109], [111], [117], [118]	Atom, Bioconductor, Cabal, Cargo, CocoaPods, CRAN, CPAN, Dub, Elm, Go, Haxelib, Hex, Homebrew, Luarocks, Maven, npm, NuGet, Packagist, Pub, Puppet, PyPI, RubyGems, Stack
Ecosystem	Evolution of dependencies/ libraries/ communities	[23], [119], [33], [112], [120], [121], [122], [123]	Cargo, CRAN, Elm, Maven, npm, PyPI, RubyGems
	Ecosystem evaluation	[17], [117], [124], [125], [126], [127], [128]	Atom, Bioconductor, Cabal, Cargo, CocoaPods, CRAN, CPAN, Go, Hex, Ivy, Luarocks, Maven, npm, NuGet, Packagist, Portage, PyPI, RubyGems, Stack
	Dependency networks	[21], [22], [82], [87], [121], [129], [130]	Cargo, CPAN, CRAN, Elm, Maven, npm, NuGet, Packagist, PyPI, RubyGems
	Library evaluation	[131], [132], [133], [134], [135], [136]	Cargo, CRAN, Debian, npm, Packagist, PyPI, RubyGems
	Projects' connectedness	[30], [31], [94], [125], [137], [138], [139], [140]	Atom, Bioconductor, Cargo, CPAN, CRAN, Debian, Homebrew, Luarocks, Maven, npm, Opam, Packagist, PyPI, RubyGems

3.1.1. Maintenance

The maintenance-related papers focused on studying aspects of updating and downgrading the third-party dependencies, conflict resolution, visualizations, dependency versioning, bloated dependencies, technical lag and licensing topics.

Updating/ Downgrading Dependencies

Much of the dependency maintenance research focused on the aspects and problems of updating and downgrading the project's third-party dependencies. These articles studied when and how the components and their dependencies are upgraded and why sometimes dependency downgrades are needed.

Di Ruscio et al. [32] aimed to propose a model-driven approach for the Debian ecosystem to prevent system configuration faults before performing the upgrade. This approach was presented as a fault detector component of EVOSS (EVolution of free and Open Source Software – a model-based approach to support the upgrade of free and open-source software systems [141]) to find faults that normally remain unpredicted, like missing packages or services upon upgrades. Similarly, Ferreira et al. [35] enforced a permission system to protect applications against malicious updates from npm packages.

Bavota et al. [33] studied the whole ecosystem of Apache and how they manage their upgrades while Kula et al. [43] investigated how willingly the Apache ecosystem upgrades and adopts the latest Maven releases.

Salza et al. [14] investigated when, how and why developers update their dependencies in the mobile domain and Kula et al. [12] sought to find out the same for Maven projects. Huang et al. [38] and Nguyen et al. [44] proposed solutions that help Android developers to keep their project dependencies up-to-date.

Zapata et al. [34] explored how projects react to vulnerability library updates. Berhe et al. [36] wanted to reduce the risk of breaking updates and designed an approach for update scheduling, much like Cox et al. [39], who aimed to make updates more transparent by measuring the difference between the used version of a dependency and the newest version of a dependency in a system.

There have been automated solutions for dependency updates proposed and Hejderup et al. [41] set out to understand how reliable these automated solutions are. Automated pull

requests have also been proposed to encourage developers to update their dependencies and Mirhosseini et al. [42] evaluated the effectiveness of this approach.

On the other hand, two papers focused on the library downgrades: Suwa et al. [37] studied the factors leading to rollbacks in the Java community and Cogo et al. [40] investigated the downgrades in the npm ecosystem.

Dependency/ Conflict resolution

A lot of the papers were aimed at studying the third-party dependency resolutions and conflict resolving possibilities in the projects.

Some papers have proposed new solutions for communities, that are in a need of dependency management systems. Mosteo [45] for example presented a library manager for the Ada ecosystem and Green et al. [51] described the development of SpackDev multi-package build system for Spack community.

Automated solutions have been proposed to help developers with the dependency management: Oshher et al. [46] presented a method to automatically resolve third-party dependencies and Atzenhofer et al. [47] developed a tool to automatically resolve missing third-party dependencies. Macho et al. [62] proposed an automated approach for repairing dependency-related build breakages. On the other hand, automated dependency resolution can be problematic when it comes to transitive dependencies. Jezek et al. [59] described this issue and proposed a new approach based on static type checking.

Dependency conflicts have been a specific concern in the dependency management. Wang et al. [52] studied real-world dependency conflict issues in the Maven ecosystem; Wang et al. [55] and Mukherjee et al. [61] studied them in Python and Javan Jafari et al. [58] documented the recurring dependency issues in the npm ecosystem. Wang et al. [48] investigated if Maven dependency conflict issues can be reproduced with automated test generation while imitating the failure inducing conditions, Wang et al. [53] tried to find out if dependency conflicts can cause semantic conflicts and adapted a test generation technique as well. In [57] inter-repository package dependency problems management was discussed in relation to how GitHub influences the R ecosystem.

Compatibility is an aspect to pay attention to while using third-party libraries as some libraries might be incompatible and cannot be installed or used together. Vouillon et al. [49] proposed a theoretical framework to help with the dependency co-installability problem and Jia et al. [64] proposed a practical tool to solve incompatible changes in libraries.

Multiple papers analysed different dependency solving proposals. Abate et al. [60] outlined suggestions for numerous distinct package managers. Wang et al. [50] discussed different methods for solving dependency conflicts in Java and Zhang [29] offered various techniques for dependency management in the Debian community.

Pashchenko et al. [13] explored the security aspects of dependency management and D'mello [56] proposed an innovative solution for dependency management in an effort to make the dependency management more transparent and secure – a blockchain based package control system.

Abate et al. [54] argued that dependency solving for upgrade planning is difficult and reviewed state-of-the-art package managers on that ability while Belguidoum et al. [63] presented a formalization of a deployment system for package dependencies.

Detecting/ Visualizing Dependencies or Libraries in Projects

Solutions for detecting and visualizing libraries and their dependencies help developers with maintenance activities. Ferrarezi et al. [65] aimed to develop a tool for software libraries' visualization and Bauer et al. [67] presented an automatic approach for library usage extraction. Both aimed to understand the detailed use of software libraries in projects. Multiple papers in the third-party library detection and visualization are in the Android domain. Zhan et al. [68] assessed empirically different Android third-party detection tools and Soh et al. [72] proposed an entirely new tool called LibSift for that; Zhang et al. [69] and Li et al. [71] both presented a new technique for identifying Android third-party libraries. Zhan et al. [73] proposed a system that can identify vulnerable third-party library versions in Android applications.

Chen et al. [66] and Isaacs et al. [70] both proposed a visualization of project third-party dependency graphs.

Dependency Versioning

Two papers touched upon the dependency versioning side of dependency management. Decan et al. [74] compared the semantic versioning compliance of four package managers and Dietrich et al. [75] set out to capture the current ways of dependency declarations.

Bloated/imitated dependencies

Bloated dependencies are third-party libraries packaged with the application's compiled code but not necessary for application's build [76]. Two articles investigated bloated

dependencies in Maven: [76] and [77]. Furthermore, to improve maintenance, Sun et al. [78] aimed to detect imitations of library APIs in software projects and Qiong et al. [79] proposed techniques for tracking unused code in JavaScript applications.

Technical lag

Technical lag was another issue some papers explored about third-party dependency management. Zerouali et al. [80, 81] aimed to measure technical lag in the npm ecosystem, Decan et al. [82] investigated the evolution of technical lag in npm as well. Stringer et al. [25] analysed technical lag across multiple major package managers, including npm.

Licensing

Using open source third-parties in software projects can bring upon licensing violations. Qiu et al. [83] set out to detect dependency-related licence violations and Kechagia et al. [85] discussed different aspects of open-source licensing. Bauer et al. [84] performed a case study on licence compliance with an established software vendor.

3.1.2. Security

Another important topic for research has been studying the security implications of using third-party dependencies. Some papers only investigated the impact and propagation of open-source vulnerabilities, while others also offered detection and mitigation techniques.

Library Vulnerabilities

Neil et al. [86] aimed to mine threat intelligence of open-source projects, they then presented and stored this threat intelligence in a security knowledge graph, so that developers can better protect their product development.

A couple of papers carried out large-scale studies of vulnerabilities in different software ecosystems. Decan et al. [87] provided a study on the impact and propagation of security vulnerabilities and their fixes in npm, Zimmermann et al. [5] studied the security threats in the npm ecosystem as well. In [24] a large scale study of project dependencies and their security vulnerabilities was presented for the Maven ecosystem, Alfadel et al. [96] presented an empirical study of the Python ecosystem vulnerability reports. The goal of Prana et al. [88] was to analyse vulnerabilities in three separate open-source communities: Maven, PyPI and RubyGems, by examining the distribution and severity of those vulnerabilities.

Gkortzis et al. [89] focused on further examining the association between software reuse and security vulnerabilities in Maven and in [93] a vulnerability dataset for Maven was presented.

Lauinger et al. [6] completed a comprehensive study on security implications of JavaScript library usage on the Web and in [34] it was explored how npm client projects react to vulnerability library migrations. Pashchenko et al. [90] focused more on the industrial practises rather than academic approaches in reporting the vulnerable dependencies.

Ohm et al. [91] gave an overview of the open-source software supply chain attacks to further facilitate the development of preventive safeguards while Kula et al. [12] studied the impact of security advisories on library migrations.

In [92] the authors investigated whether technical leverage in a software ecosystem can be a source of security vulnerabilities; Cox et al. [39] correlated dependency freshness, i.e. the difference between the used dependency version and the newest dependency version, and security issues. Tellnes [94] showed, that the security of a system is largely determined by the surrounding dependencies and it is beneficial to reduce the reliance on external dependencies.

Pashchenko et al. [13] studied how security concerns affect the overall decision-making for dependency management of developers, Zhang et al. [95] proposed a systematic approach to measuring attack surface brought by package dependencies and Zhan et al. [17] conducted a systematic literature review on Android third-party library research and among other things also studied the security implications.

Detection and Mitigation

In [97] and [99] a method for detecting, assessing and mitigating open-source vulnerabilities was presented, Pashchenko et al. [105] proposed a methodology for counting actually vulnerable dependencies by catering to the need of industrial practice. Staicu et al. [100] proposed a technique to automatically extract vulnerable specifications for JavaScript libraries.

Imtiaz et al. [103] compared different software composition analysis tools on their vulnerability reporting abilities, Dann et al. [104] identified challenges for open-source software vulnerability scanners and Yongming et al [98] designed a completely new tool called CD3T that detects cross-project dependency defects.

Chinthanet et al. [101] focused more on the vulnerability fixes and how they are adopted, the main goal of Alfadel et al. [102] was to understand if developers really adopt the Dependabot security pull requests to fix the dependency vulnerabilities and Hejderup [107] aimed to find out exactly how vulnerable are dependencies in software modules and to better understand why security problems are not dealt with in the npm ecosystem.

In [35] a permission system was designed to protect applications against malicious updates from direct as well as indirect dependencies, Wang et al. [106] presented their experience with building a third-party library supply chain and Pfretzschner et al. [108] addressed the questions how attackers exploit third-party dependencies in the npm ecosystem and how those attacks could be mitigated.

3.1.3. Social

Another theme for third-party dependency research was related to social aspects and explored dependency communities, how they communicate with each other and what kind of decisions developers make regarding those dependencies.

Technology Adoption

Ma et al. [109] aimed to model the uptake of software technologies to better understand if and how they are adopted by developers in the CRAN community. Umm-e-Laila et al. [18] investigated the open-source software adoption factors for critical IT systems and Kula et al. [112] visualized the library adoption with a software universe graph.

Cogo et al. [110] studied the adoption rate of same-day releases in the npm ecosystem while Chinthanet et al. [101] identified the adoption factors of vulnerability fixes for npm. Release adoption was also studied by Kula et al. [43] for latest Maven releases.

The main goal of Alfadel et al. [102] was to understand the degree to which the Dependabot security pull requests are adopted. Xu et al. [111] thoroughly studied why developers decide to use external libraries or re-implement library code themselves.

Recommendation Systems

Some articles have proposed different kind of recommendation systems to help developers choose third-party libraries or suggest suitable release times. Nguyen et al. [1] proposed a CrossRec tool and Katsuragawa et al. [113] proposed a prototype called DSCRec for recommending third-party libraries in Maven projects. Yan et al. [114] took into account

the existing social influence and dependency constraints in software projects for their software recommendation model.

Berhe et al. [36] understood the problem of breaking updates when updating the library dependencies and presented update scheduling recommendations. Yang et al. [115] tackled the problem of untagged open-source software in repositories and proposed an approach for repository tag recommendations.

Interactions

Three papers looked more on the social interactions side of dependency communities, as using third-party libraries can be a collaborative effort. Palyart et al. [116] studied the social interactions in Java and Ruby communities, Yan et al. [114] took into account the social influence for a proposed library recommendation model and Bogart et al. [117] identified the importance of social awareness and networking in ecosystems when it comes to breaking changes.

Developers' Behaviour

Other papers have specifically studied the developers' behaviour in different aspects of dependency management and answer the questions of why and how they do things. Ma et al. [109] studied the uptake of software to understand developers' choices, Bogart et al. [117] explained their actions for breaking changes and Xu et al. [111] investigated peoples' reasons for library reuse and re-implementation.

Salza et al. [14] analysed developers' behaviour regarding updating mobile library dependencies and Kula et al. [12] studied the same for the Maven community. Chen et al. [118] aimed to find out why developers publish trivial packages and if those packages really help the npm ecosystem.

Pashchenko et al. [13] studied the choices of developers in their overall decision-making regarding the management of software dependencies and Dietrich et al. [75] focused specifically on the dependency declaration practices of dependency management.

3.1.4. Ecosystem

A lot of the third-party dependency analysis research aimed to look at the bigger picture and study entire ecosystems, libraries and communications as a whole.

Evolution of Dependencies/Libraries/Communities

Evolution from different standpoints was the focal point of many papers. Decan et al. [23] studied the evolution of package dependencies in the npm, CRAN and RubyGems distributions and Bavota et al. [33] investigated the evolution of dependencies between projects in Java. Java was studied in [123] and [112] as well, where they visualized the evolution of systems and their dependencies in Maven; the latter, in addition to Maven, focusing on CRAN community as well.

Jezek et al. [119] studied the compatibility of API changes as libraries evolve, Raemaekers et al. [120] introduced a way to measure library stability through analysing historical values. Blanthorn et al. [121] explored the evolution and dynamics of different package management communities and Mujahid et al. [122] investigated package evolutions as well, but in a way to discover packages in decline, so that developers choose the right packages that are maintainable.

Ecosystem Evaluation

Some articles evaluated and studied the ecosystems as a whole, for example Soto-Valero et al. [124] analysed the software diversity in Maven and Harrand et al. [125] performed a large-scale empirical study for the client-library relationships in Maven. Santana et al. [128] presented their work to enable the analysis of software ecosystems as a whole from both social and technical perspectives.

Bogart et al. [117] investigated and compared policies and practices regarding breaking changes in 18 different ecosystems, Zheng et al. [126] proposed to analyse the open-source software systems as complex networks and studied the dependencies in Gentoo Linux. Wittern et al. [127] presented an extensive analysis of the whole npm ecosystem and Zhan et al. [17] conducted a large-scale systematic literature review on the Android ecosystem for third-party library related research.

Dependency Networks

Many papers studied the dependency networks of packaging ecosystems or libraries. Han et al. [129] made an empirical study on the dependency networks for deep learning libraries, Decan et al. [87] studied how security vulnerabilities propagate through npm dependency network.

Kikas et al. [22] set out to explore the dependency networks and their evolution of three different state-of-the-art package managers and Decan et al. [21] carried out the same comparison for seven packaging ecosystems. Dynamics of communities were studied through the dependency networks in Blanthorn et al. [121] and Mora-Cantallops et al. [130] showed through building a dependency network that complex network analysis techniques can be applied to evaluate the ecosystem's health. Technical lag was also investigated through the npm dependency network in [82].

Library Evaluation

Some of the research also touched upon the third-party library evaluation in different aspects, as it is especially important to help developers with their maintenance activities and decision-making on which components to choose. Since the client system's quality directly depends on its external dependencies, in [135] the authors proposed a model to estimate the quality of open-source components and Wu et al. [131] mined information about open-source component performance while proposing a model to evaluate the open-source component behaviour. The libraries' impact on its ecosystem was evaluated in [132], where the authors assessed trivial packages to examine their importance and if they can break ecosystems, Korkmaz et al. [133] developed methods to estimate the libraries' impact in R and Python communities and Decan et al. [134] evaluated the early releases and their impact on the whole ecosystems. Miksaa et al. [136] investigated the impact third-party dependencies can have on the replicability of software workflows.

Projects' Connectedness

Sometimes it's relevant to measure and analyse the projects' connectedness or map out different interactions between components. In [137] the authors described a new reference coupling method to detect technical dependencies between projects while Abate et al. [31] studied strong dependencies between software components and German et al. [30] modelled the dependencies among applications, both in the Debian ecosystem.

Parreiras et al. [139] aimed to propose a framework for a marketplace of open source software data to link artefacts within and between software projects, similarly, Ma et al. [138] looked at the whole open-source ecosystem in an attempt to create a collection of data for the entire open source universe to enable cross-referencing.

Harrand et al. [125] studied the client-library relationships in Maven to investigate the spectrum on Maven library usages, Zhou et al. [140] proposed a new approach of aspect-

oriented programming to ensure the interactions between components conform to stated policies and Tellnes [94] showed that software's security relies on its surrounding ecosystem and how it's connected with other projects.

RQ1: Research Objectives summary. Studies done in the third-party library analysis domain aimed to do their research on four main topics: maintenance, security, social and ecosystem. Most work was done on the maintenance of third-party dependencies, where the articles studied library migrations, dependency conflicts, library detection, dependency versioning, bloated dependencies, technical lag and licensing. Security-related papers researched library vulnerabilities and their mitigation. Articles that focused more on the social aspects studied the adoption of third-party libraries, developers' behaviour, interactions and recommendation systems. And lastly, many papers did their research from the ecosystem's side, looking at the evolution of communities, evaluating the ecosystems and libraries, investigating dependency networks and projects' connectedness.

3.2. Research Question 2: Data Sources

Data sources, that were used to gather data for the articles' research, were identified and analyzed. This was done to gain insight into where the base data could be collected from for third-party library analysis related research and allow for research replicability for the readers. Most used data sources are pictured in Figure 4.

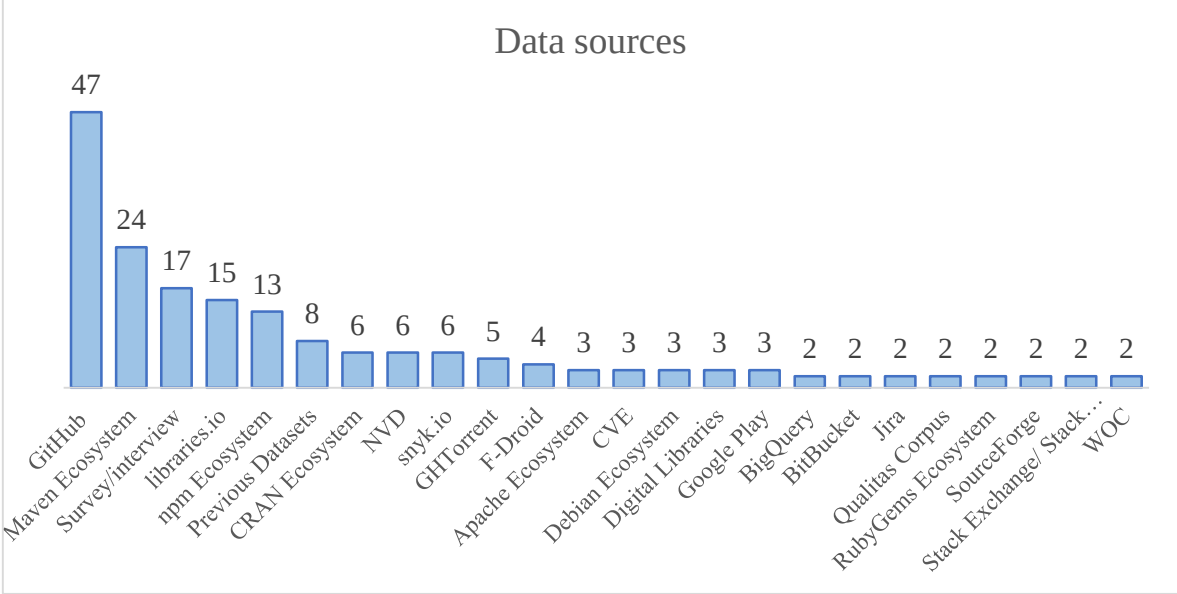

Figure 4. Data sources used in the papers.

3.2.1. GitHub

Almost every third paper used some form of the GitHub repositories to get their research dataset. GitHub[1] can be a good source for data as it is under public domain and contains, in addition to code repositories, other useful development artefacts as well, like development history logs, developer's comments and development networks.

Numerous papers like [80] and [48] used GitHub repositories to collect open-source projects. Many also used GitHub APIs[2] to query the projects, for example Soto-Valero et al. [76] used API to get Java projects according to their number of popularity stars and Ma et al. [109] used API to scrape issues from GitHub related to specific R packages. GitHub Activity Data database[3] was used in [89] to collect projects according to specific criteria.

GHTorrent[4] also utilizes the GitHub REST API and that database was useful when the GitHub timeline and social interactions' info was needed [114].

3.2.2. Package Manager Ecosystems

Other well-used data sources were the package managers' repositories and ecosystem's databases. The Maven/Apache ecosystem was used in the papers which focused on researching the Maven package manager. As Maven was the most researched package manager, it's no surprise so many papers used the Maven Ecosystem as their main data source.

The Maven Central Repository[5] and the Maven 2 Central Repository[6] were useful in many papers as they are public repositories that host Maven artefacts and are a good source of Maven related data, they were used for example in [25] and [46]. The Maven Repository website[7] was used to get category data of Maven libraries [113] and Google's Maven Repository[8] was also used in [68] to crawl third-party library files from. The Maven Dependency Graph [142] was another useful dataset, used in [124], that captured the snapshot of Maven Central as of September 2018, which carried the metadata of Maven artefacts and dependency relationships of more than 223 000 libraries.

[1] https://github.com/
[2] https://api.github.com/
[3] https://console.cloud.google.com/marketplace/details/github/github-repos?filter=solution-type:dataset&id=46ee22ab-2ca4-4750-81a7-3ee0f0150dcb
[4] https://ghtorrent.org/
[5] https://mvnrepository.com/repos/central
[6] https://repo1.maven.org/maven2/
[7] https://mvnrepository.com/
[8] https://maven.google.com/web/index.html

Npm being the second most studied package managers, its registries were used in multitude of papers as well. The npm registry[9] as the main package repository for the npm was used for example in [107] to study its vulnerabilities and in [127] to collect data about the evolution and popularity. The data for vulnerable packages was extracted from the npm advisories registry[10] in [5], and the npm replicate registry[11] was used in [66] to retrieve npm metadata.

Similarly, other package managers' respective repositories were used as data sources, Bogart et al. [117] gathered their R package data from CRAN repositories[12] and Mora-Cantallops et al. [130] used the CRAN metadata database[13]. RubyGems website[14] was used in [23] and RubyGems API[15] in [22]. Debian archives with unspecified locations were used in [29, 60], other smaller ecosystems' repositories were used as well.

3.2.3. Surveys and Interviews

Numerous papers used the data they got from surveys and interviews. Mainly it was used as a way to evaluate their solutions and proposals, carry out a case study to gather more insight into the topic, or complement their empirical research; it was not used much as a way to gather data to base their research on.

As an example, Salza et al. [14] surveyed mobile developers to find out the reasons why they update (or not) their third-party libraries, this was done to complement their software repository mining study. Meanwhile, Bogart et al. [117] used interview case study to first understand how the community deals with breaking changes on library updates, and in the second part they used high-level question survey in many communities to see, where the findings generalise. An evaluation of the usefulness of the proposed "dependency freshness" metric was done in [39] by using interviews.

3.2.4. Libraries.io

In papers that focused on researching multiple package managers, libraries.io Open Source Repository and Dependency Metadata dataset[16] was used as it supported numerous different

[9] https://www.npmjs.com/
[10] https://www.npmjs.com/advisories
[11] http://replicate.npmjs.com/
[12] https://cran.r-project.org/web/packages/available_packages_by_name.html
[13] https://cran.r-project.org/src/contrib/Archive/Metadata/
[14] https://rubygems.org/
[15] https://rubygems.org/pages/data
[16] https://zenodo.org/record/3626071#.Yndvv9pBw2w

package managers and therefore made it easy for the researchers to find comparable data. This was done for example in [60] and [75], which compared a set of different package managers. The libraries.io dataset was also useful for longitudinal analysis, like was done in [82] of technical lag in npm package dependency network, as it carries long historical data.

3.2.5. Previous Datasets

Many papers relied on datasets from previous research. For example, in order to evaluate their proposed new tool CrossRec, Nguyen et al [1] used datasets from the original LibFinder tool paper [143] and LibCUP tool paper [144] to allow for comparison of these three tools. Katsuragawa et al. [113] also relied on a dataset from [77] to create their dataset of projects that use Maven libraries, and Prana et al. [88] used the reaper dataset from Munaiah et al. [145] to find GitHub repositories that contain software projects, as many repositories may not actually contain software projects.

3.2.6. Vulnerability Databases

Papers that carried out security research relied on vulnerability data sources, like snyk.io[17], National Vulnerability Database (NVD)[18] and Common Vulnerabilities and Exposures (CVE)[19] database. Yongming et al. [98] and Zhan et al. [73] for example used both NVD and DVE databases to gather open vulnerabilities and software defect data over the years. NVD was also used in [6, 24, 36, 104] to extract the vulnerability information, CVE database was utilized in [39] as well. The snyk.io dataset was also used to obtain the security vulnerabilities and reports in [34, 87, 91, 92, 96, 101]. Lauinger et al. [6] also used Open Source Vulnerability Database (OSVBD), that is deprecated by today.

3.2.7. Android Databases

In addition to previously mentioned data sources, papers that focused on the mobile domain forged Android applications mainly from F-Droid[20]: [14, 44, 68, 111] and Google Play store[21]: [38, 68, 72]. AndroidTimeMachine dataset [146] was also used in [14] that contains publicly available data on open-source Android applications, JCenter (deprecated) and

[17] https://snyk.io/
[18] https://nvd.nist.gov/
[19] https://cve.mitre.org/index.html
[20] https://f-droid.org
[21] https://play.google.com/store

AppBrain[22] repositories were utilized in [68] and AndroZoo [147] collection in [69]. Li et al. [71] gathered their information from 45 different third-party Android application markets. Nguyen et al. [44] used the library database of LibScout [148] in their work.

3.2.8. Other Data Sources

Multiple other data sources were used in the studies. Papers that carried out literature reviews used different digital libraries to gather research papers [17,18, 68]; BigQuery's GitHub archives were used in [37] and [42]; Jira[23] projects were used in [128, 137], Stack Exchange[24] community was used in [109] and Stack Overflow[25] in [64] to get more qualitative data about the development processes. Qualitas Corpus [149] was another good source of curated software systems code artefacts [59, 102]. In addition to these data sources, there were many others, which can be seen in the data extraction table.

RQ2. Data Sources summary. Many different data sources have been used in the third-party library analysis studies. Most have crawled the metadata from GitHub as it is an open-source platform hosting, in addition to source code, also other valuable information, like development history, comments or communication data. Package management repositories were used as well as they host different third-party library artefacts. Libraries.io dataset gathers info that was useful for articles that carried out research for multiple different package managers. Surveys and interviews were used to gather more qualitative data, vulnerability information was mainly collected from NVD, CVE and snyk.io databases and papers done in Android domain got their data from Google Play, F-Droid and other mobile application repositories.

3.3. Research Question 3: Methods

To understand the research done on third-party dependency analysis more, the methods distinctive in third research field were gathered in Table 3 and analyzed. This includes describing the most used data collection techniques and analysis approaches. The papers were not classified into previously defined study types or methodologies, as the author believes it would not give much information to practitioners who may aim to seek ideas and

[22] https://www.appbrain.com/stats/libraries/
[23] https://www.atlassian.com/software/jira
[24] https://stackexchange.com/
[25] https://stackoverflow.com/

replication guides from this thesis. Rather, an overall descriptive aggregation is given of the research method trends and the exact methods used for each analyzed paper can be found from the data extraction table.

Table 3**.** Research methods.

Research Methods	**Studies**
Mining Repositories and Metadata	[5], [6], [12], [14], [21], [22], [23], [24], [25], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44], [46], [52], [53], [55], [56], [57], [58], [59], [60], [62], [66], [68], [70], [71], [73], [74], [75], [76], [77], [80], [81], [82], [83], [85], [86], [87], [88], [89], [90], [91], [92], [93], [96], [101], [102], [104], [105], [107], [109], [110], [111], [112], [113], [115], [116], [117], [118], [119], [120], [122], [123], [124], [125], [127], [128], [129], [130], [132], [133], [134], [137], [138]
Building Dependency Graphs/ Networks	[5], [22], [23], [24], [30], [33], [41], [53], [66], [69], [71], [73], [78], [86], [90], [91], [112], [119], [122], [130], [132], [133], [137]
Longitudinal Analysis	[5], [14], [21], [22], [23], [33], [39], [57], [58], [74], [77], [80], [82], [87], [122], [132], [134]
Qualitative Methods	[1], [12], [13], [14], [18], [33], [39], [42], [44], [46], [48], [53], [57], [58], [70], [75], [76], [77], [83], [84], [99], [102], [107], [110], [111], [116], [117], [118], [122], [134], [138]
Theoretical Proofs	[31], [49], [54], [63], [95], [126], [131]
Case Studies	[12], [31], [34], [71], [77], [80], [84], [94], [102], [103], [104], [109], [111], [112], [117], [136]
Building a New Tool/ Implementing a New Technique	[1], [29], [32], [35], [45], [48], [51], [59], [61], [64], [65], [67], [69], [72], [73], [78], [79], [90], [98], [100], [107], [108], [136], [139]
Other Methods	[17], [18], [47], [50], [56], [97], [99], [106], [114], [115], [121], [135], [140]

3.3.1. Mining Repositories and Metadata

The most used approach for studying the third-party dependency related aspects has been crawling substantial amount of different metadata, like project repositories, package manager ecosystems, logs etc. These research papers are quantitative and data-heavy, therefore, they are a good way to investigate various phenomena, whole third-party dependency ecosystems, make generalisations and compare and contrast. These papers mainly rely on the datasets of open-source code repositories and package managers and usually they apply specific criteria to the mined data to get a dataset suitable for their particular analysis and study goals. After obtaining their dataset, a study-specific analysis is

carried out, which depends on the goals of the research. This can be anything from statistical analysis, finding correlation or causation, to building dependency networks or visualizing trends. As the analysis of such copious amount of data is not often manually feasible, automated ways have been developed in numerous papers in order to be able to analyse the gathered data. Some approaches have even been released as novel tools in open-source projects, summary of that information can be found in the chapter 3.6.

For example, Prana et al. [88] set out to study, how vulnerable dependencies can affect open-source projects in three different programming languages. They mined their data from GitHub and set restrictions based on programming language and time on the mined projects, as well as the prerequisite, that the project must be scannable by the Veracode SCA tool to allow for subsequent analysis of the collected data. Soto-Valero et al. [76] on the other hand focused on thoroughly researching one aspect of a single package manager, namely the occurrence of bloated dependencies in the Maven ecosystem. They did this by obtaining representative set of 9,770 Maven artefacts from the Maven Dependency Graph database [142]. As they aimed to analyse thousands of artefacts on Maven, they implemented a tool called DepClean that can automatically perform the analysis on Maven projects of their dependencies usage.

In addition to mining software projects, other different data can be used. As an example, Alfadel et al. [102] performed an empirical study using data from 15,242 pull requests that they obtained by using GitHub API, while Dietrich et al. [75] looked at the information on how projects declare dependencies and used a dataset from libraries.io [143] database to study this dependency versioning question.

3.3.2. Building Dependency Graphs

After obtaining the dataset of analysable data, numerous papers built third-party dependency graphs or networks to allow for further analysis. Dependency graphs are useful for depicting dependency relationships in bigger ecosystems and when analysing third-party libraries for a project or in a package manager ecosystem, dependency graphs can be a good way to visualize the relations.

Zimmermann et al. [5] aggregated the metadata of the packages in npm and constructed a dependency graph, without needing to actually download or install every package – in total they considered 676,539 libraries and 4,543,473 dependencies. Mujahid et al. [122] aimed to calculate centrality trends of the packages in the npm ecosystem and built a dependency

graph as well, containing npm packages as nodes and dependency relationships as edges. They updated the graph monthly with new packages and dependencies and computed the centrality metric to rank each package each month and distinguish packages in decline.

A dependency network is created and centrality measures are calculated in [133] also. They used data from Depsy.org[26] to create the dependency network and the structural features of those networks were then used to describe statistical models. A Software Universe Graph (SUG) was presented in [112] to create a structural abstraction of software repositories within an ecosystem, defined as super repositories. This was done to better understand the adoption and popularity of packages within the Maven and CRAN universes.

3.3.3. Longitudinal Analysis

When substantial amount of data collected spanning a time period and the data was analysed according to its changes in time, then a longitudinal study was carried out. This was done a lot when trying to study the evolution of something, like package managers, or a phenomenon related to third-party packages.

A longitudinal study of bloated dependencies in the Maven ecosystem was carried out in [77]. They analysed the usage status of 48,469 dependencies, where the projects' active period ranged from five months to almost three years. Javan Jafari et al. [58] conducted an empirical study where they analysed the prevalence of dependency smells in JavaScript projects and analysed the changes over a five-year period to provide an overview how the smells evolve through time.

Bavota et al. [33] aimed at studying the evolution of dependencies of Maven projects in the Apache ecosystem for a period of 14 years. They analysed the change history logs of 147 Java projects to investigate how the dependencies between the projects evolve with the ecosystem's growth. Similarly, the evolution of dependencies was studied in [21] as well. They compared seven software packaging ecosystems and considered the whole lifetime of each ecosystem up to 2017 while relying on the official registry of each packaging manager.

3.3.4. Qualitative Methods

In addition to quantitative analysis of the mined data, some papers also used qualitative methods, like interviews, surveys with open-ended questions, analysing the feedback from developers and other users or a more thorough manual analysis of some smaller data. This

[26] http://depsy.org/

was done more when the authors wanted to study "how" and "why" some phenomena occur, to complement their empirical research and findings or evaluate their solution proposals. Often also to study developers' behaviour regarding third-party dependency analysis.

Whether or not developers update their dependencies was studied in [12] and the authors interviewed developers on why they do not respond to security advisories. Security related developers' choices were investigated in [12] as well, where they qualitatively researched the developers' decision-making for managing software dependencies through semi-structured interviews.

Bauer et al. [84] carried out a case study in a company that uses open-source libraries in their products to study dependency documentation in terms of licence compliance and tracking, and they used interviews and open surveys in their study. Likewise, an interview case study carried out in [117] aimed to contrast Eclipse, npm and CRAN ecosystems on how their communities handle breaking updates of the libraries.

To identify user goals and gain more insight into needed solutions, Isaacs et al. [70] conducted a study in the form of interviews with Spack maintainers. Qualitative feedback from developers was queried in [76] to analyse how they react to bloated dependencies.

3.3.5. Theoretical Research

Some articles carried out a more theoretical research where they did not gather any data, but presented theoretical proofs, abstractions of an idea or conceptual analysis. As an example Wu et al. [131] explored ways to evaluate open-source library behaviour and performance before integrating it into a project. They proposed the usage and dependency models for that evaluation, as well as the algorithm to mine these models. Zhang et al. [95] defined the project's attack surface exposed through its third-party library dependencies.

Dependency management and formalization has been one of the main research topics that has been covered in the theoretical works. In [46], the notion of strong dependencies was proved through theoretical proofs, the same approach was used in [63] to present a formalization of deployment dependencies and in [49] to develop a theoretical framework for identifying, which libraries can be installed together and which cannot. Theoretical proofs were also used in [54] to show the complexity of the dependency upgrade planning problem is NP-complete and present a Domain Specific Language (DSL) called Common Upgrade Description Format (CUDF).

3.3.6. Case Studies

Case studies were another well-used approach in the papers. As the definition of case study can be slightly different depending on the context, the papers were categorized to use case study approach if it was mentioned so by the papers themselves. Case studies are often used to explore areas that are not that well-researched yet and they have practical relevance [84], which in many cases can be suitable for the third-party dependency analysis related studies.

Bauer et al. [84] aimed to research licence compliance of third-party libraries and for this they conducted a single-case study at one established software vendor. Imtiaz et al. [103] wanted to understand the difference in vulnerability reporting of different software composition analysis (SCA) tools and presented an in-depth case study where they compared the analysis reports of 9 SCA tools on a large web application.

Case studies can also be used to evaluate some proposed solutions or complement previously done research. In [31] a Debian GNU/Linux case study was carried out to verify their proposed notion of strong dependencies as a first step towards modelling semantic, rather than syntactic, inter-component relationships. In [104], a two-fold case study was carried out: first, they investigated the settings regarding the use of third-party libraries in an industrial context; and second, they investigated the prevalence of vulnerable libraries. Their case studies complemented existing work by showing that vulnerable code clones are also introduced during the build processes of client projects.

3.3.7. Building a New Tool

Numerous research also built new tools in their work and usually building a new tool was one part of the article's analysis work or their solution for some problem. Some papers however focused only on building a new tool and aimed at documenting their work and methods for that, these papers are brought out in this section. All new tools presented in the papers can be found in chapter 3.6.

Green et al. [51] documented their work on building the SpackDev tool; Mostero [45] presented a new tool for the Ada language which was built by tagging code releases in public repositories with semantic versioning and can help the Ada community with third-party dependency management. An automated approach was presented in [67] for analysing the dependencies of software projects on third-party libraries and in [108] a static code analysis approach was implemented for T.J. Watson Libraries for Analysis (WALA) to detect the identified attacks and the evaluation of the analysis.

3.3.8. Other Methods

As many different methods have been used in the studies, couple other interesting approaches are brought out and analysed here. Two papers conducted comprehensive systematic literature reviews as their main research to gain insight into their topic of interest. Umm-e-Laila et al. [18] investigated the third-party library adoption factors for critical IT infrastructures and Zhan et al. [17] conducted the first systematic literature review on Android third-party library related research. Systematic literature review was used as a way to complement the other research methods used and investigate existing third-party library detection techniques in [68].

Another innovative research approach has been using machine learning and employing blockchain-based methods. D'mello et al. [56] proposed a system that used blockchain-based smart contracts employing the Ethereum network. Yang et al. [115] trained neural networks to design an algorithm for tag recommendation system for open-source software on GitHub.

RQ3: Research Methods summary. An overwhelming majority of the research articles used the approach of crawling and mining large amounts of metadata from different repositories and then applying data analysis methods specific to their work on that metadata. This allowed for very data-heavy research suitable for studying entire ecosystems and communities, or issues related to the third-party dependencies. Oftentimes dependency graphs and networks were built from the crawled data, and the data also allowed for longitudinal analysis to study trends overtime. Qualitative methods like interviews and questionnaires complemented the data-heavy research providing more in-depth insight into why some aspects of third-party dependency analysis are the way they are.

3.4. Research Question 4 & 5: Problems and Mitigations

Problems discussed in the papers were gathered and categorized to analyse the most pressing issues related to the third-party dependency usage and are presented in Table 4. Proposed solutions are also given to present a general outlook on the ways how to mitigate those problems. Problems are categorized into the same 4 categories, that were used in the research objectives categorization: maintenance, security, social and ecosystem.

Table 4**.** Problems according to studies and package managers.

	Problems	**Studies**	**Package Managers**
Maintenance	Bloated Dependency Management	[76], [77], [78], [79]	Maven
	Outdated Dependencies	[21], [22], [25], [34], [38], [39], [40], [42], [58], [80], [81], [82], [90], [113]	Atom, Cargo, CPAN, CRAN, Dub, Elm, Haxelib, Hex, Maven, npm, NuGet, Packagist, Pub, Puppet, PyPI, RubyGems
	Breaking Changes	[14], [22], [30], [36], [37], [40], [41], [43], [58], [61], [64], [67], [74], [75], [80], [81], [82], [117], [119], [120], [130]	Atom, Bioconductor, Cabal, Cargo, CocoaPods, CPAN, CRAN, Debian, Dub, Elm, Go, Haxelib, Hex, Homebrew, Luarocks, Maven, npm, NuGet, Packagist, Pub, Puppet, PyPI, RubyGems, Stack
	Too Many Dependencies	[32], [41], [47], [51], [56], [59], [61], [65], [70], [74], [111], [118], [123], [126], [132], [136]	Cargo, Debian, Maven, npm, Packagist, Portage, PyPI, RubyGems, Spack
	Insufficient Tools	[32], [45], [56], [80]	Alire, Debian, npm
	Dependency Conflicts and Incompatibilities	[17], [21], [25], [43], [45], [46], [48], [50], [52], [53], [56], [62], [64], [75], [118], [124], [140]	Alire, Atom, Cargo, CPAN, CRAN, Dub, Elm, Haxelib, Hex, Homebrew, Maven, npm, NuGet, Packagist, Pub, Puppet, PyPI, RubyGems
	Hard to Track Dependency Info	[29], [66], [68], [69], [71], [72], [90], [94]	Debian, npm, NuGet, Maven, RubyGems
	Transitive Dependencies	[22], [23], [45], [49], [56], [59], [61], [66], [73], [100], [126], [127], [129], [130]	Alire, Cargo, CRAN, Maven, npm, Portage, RubyGems, PyPI
	Co-Installability	[23], [49], [54], [60], [63], [127]	Bower, Cabal, Cargo, Clojars, CPAN, CRAN, Debian, dnf, Go, Maven, npm, NuGet, Opam, Packagist, Paket, PyPI, RPM, RubyGems
	Library Updates	[25], [33], [36], [39], [54], [97], [110], [130]	Atom, Cargo, Dub, Elm, Haxelib, Hex, Maven, npm, NuGet, Packagist, Pub, Puppet, PyPI, RubyGems
	Legal Issues	[67], [83], [84], [85], [118] [139]	Maven, npm
	Build Problems	[51], [55], [61], [62], [138]	Atom, Bioconductor, Cargo, CPAN, CRAN Luarocks, Maven, npm, Packagist, PyPI RubyGems, Spack
	Unsuitable Approaches	[58], [112]	CRAN, Maven, npm

Security	Using Libraries with Security Vulnerabilities	[6], [12], [13], [17], [24], [35], [38], [44], [67], [73] [86], [87], [88], [91], [92], [93], [95], [96], [97], [98], [99], [100], [102], [103], [104], [105], [106], [107], [108]	Cargo, Bower, Maven, npm, PyPI, RubyGems, Xargo
	Lags in Vulnerability Fixes	[17], [87], [101]	npm
Social	Human Interactions	[116], [125]	Maven, RubyGems
	Finding Suitable Libraries	[1], [45], [111], [113], [114], [115], [118], [121]	Alire, Cargo, CRAN, Elm, Maven, npm, PyPI
	Developers Don't Manage the Dependencies	[12], [13], [14], [17], [18], [21], [34], [38], [39], [44], [52], [89], [94], [112], [133]	Cargo, CPAN, CRAN, Maven, NuGet, npm, Packagist, PyPI, RubyGems
Ecosystem	Packages in Decline	[12], [61], [67], [109], [113], [122]	CRAN, Maven, npm, PyPI
	Third-Party Library Evaluation	[122], [131], [132], [134], [135], [138], [139]	Atom, Bioconductor, Cargo, CPAN, CRAN, Luarocks, Maven, npm, Packagist, PyPI, RubyGems
	Packaging Ecosystem Maintainability	[5], [21], [23], [29], [31], [33], [82], [112], [128], [138]	Atom, Bioconductor, Cargo, CPAN, CRAN, Debian, Ivy, Luarocks, Maven, npm, Packagist, PyPI, RubyGems
	Inter-Dependency Info	[30], [31], [57], [123], [137], [139]	Debian, Homebrew, Maven, Opam, RubyGems

For every problem, the package managers used in its studies are also presented in Table 4 to provide an overview of what has been researched for which package manager. Maven and npm were included in the research of nearly every problem, which was to be expected, when they were the two most researched package managers. A handful of papers [25, 60, 75, 117, 138] included more than 10 different package managers in their work and therefore

widened the coverage of package managers for multiple research problems. Surprisingly, even though the problem of using libraries with security vulnerabilities was researched most in the papers, it did not have a wide coverage of different package managers: most only concentrated on Maven or npm; Cargo, Bower, PyPI, RubyGems and Xargo were represented as well.

3.4.1. Maintenance

Most problems were discussed in the maintenance domain of third-party dependency analysis.

Bloated Dependency Management

Problem. Soto-Valero et al. [76, 77], Sun et al. [78] and Qiong et al. [79] raised one special aspect of the dependency management challenge: bloated dependencies. Bloated dependencies are packages, that are declared as dependencies in the client project, but are not actually necessary for the build or run [76]. This means that the final deployed binary file is unnecessarily large, which becomes an issue when the application is sent over the network or deployed on smaller devices [76]. Overall, they increase the management effort.

Suggested solutions. Understanding the software bloat is crucial to be able to propose debloating tools. As the process of removing unused code from the dependencies is difficult, mainly tool support is suggested. A tool called DepClean was proposed by Soto-Valero et al. [76] that performs automated dependency usage analysis in Maven projects and removed the bloated dependencies. Soto-Valero et al. [77] suggests using debloating tools, especially DepClean, as well. For example, they suggested to do the bloat analysis when preparing for a major release of a project to ensure no bloat is shipped and distributed. This also consequently reduces the number of transitive dependencies for the projects that depend on the new release. They also recommend caution when using the dependency management bots, such as Dependabot, as those can recommend dependency updates without checking if the dependency is actually used. Qiong et al. [79] mentioned, that in practice there are code construction tools such as Browserify and Webpack, as well as Rollup22 tool. However, those methods have downsides and so Qiong et al. [79] proposed a scheme combining static structure analysis and dynamic tracking techniques to identify unused JavaScript code during application execution. Sun et al. [78] investigated the problem of library imitations in the client code, which creates unnecessary waste and proposed a prototype tool to detect those re-implementations.

Outdated Dependencies

Problem. Numerous papers addressed the problem of a system using outdated dependencies. Gonzalez-Barahona et al. [150] coined the term “technical lag” to describe the lag between upstream development and the deployed system. According to Zerouali et al. [80], application can become outdated when it’s depending on outdated components, which can bring upon a higher risk of bugs and security issues. Therefore, it is important to avoid being dependent on outdated dependencies.

Suggested solutions. Zerouali et al. [80] proposed a general formal framework for measuring how outdated a component is. It can be used in practice as it’s able to take into account the dependencies and when developed as a tool in the future, it may help developers evaluate the effects of different component upgrading policies.

Both Cox et al. [39] and Zerouali et al. [81] proposed different metrics to evaluate if dependencies are outdated. Cox et al. [39] presented three metrics to measure single dependency freshness: version sequence number, version release date and version number delta; they then defined a system-level measurement. Zerouali et al. [81] proposed a technical lag metric to assess how outdated a third-party library is compared to its latest available release.

According to Katsuragawa et al. [113] continued usage of outdated technologies is one of the reasons for modern open-source project failures. One way to mitigate this risk is to update and switch to other libraries. They proposed a domain-specific category library recommendation tool called DSCRec to help with efficient library management. Pashchenko et al. [90] said that the simplest solution to mitigate an outdated dependency vulnerability is to update the dependent library so the client uses the fixed version.

Huang et al. [38] argued that automated library updates can solve the outdated components issue. They investigated the feasibility of automatic drop-in replacement library update framework for Android and concluded that this proposal does not currently work in practice, however it provides valuable insight for future library update solutions. Zapata et al. [34] and Mirhosseini et al. [42] suggested using automated approaches as well to help with outdated dependencies, Mirhosseini et al. [42] recommended automated pull requests and project badges, however improvements are needed to these approaches as well to increase the confidence level for developers to perform the update activities.

A more popular solution suggested was to use general package managers or other special tools that aid with the management to automatically update the dependencies [21, 25, 40, 82]. This should decrease the maintenance overhead for developers help keep dependencies up-to-date. Cogo et al. [40] focused more on the library downgrade problem, but downgrades increase the technical lag of client packages.

Another important solution to note was that developers and project maintainers should specifically prioritize the updates of outdated dependencies, however Cox et al. [39] and Zerouali et al. [81] emphasized the balance between being up-to-date and the increasing effort. Kikas et al. [22] mentioned general dependency management practices are needed to keep dependencies up-to-date.

Additionally, semantic versioning was mentioned as a means how some proposed frameworks and automated solutions operate. Semantic versioning helped Zerouali et al. [80] to define the metric for the technical lag; according to Stringer et al. [25], package managers use semantic versioning to automatically update dependencies; Jafari et al. [58] presented semantic versioning as a solution to help manage dependencies and Decan et al. [82] claimed that semantic versioning can be used to reduce technical lag, when dependency constraints would rely on semantic versioning rules that enable automatic dependency updates.

Breaking Changes

Problem. On the one hand, relying on outdated dependencies can bring upon security risks and make the client project more vulnerable, on the other hand updating to a fresh version of a library is not void of problems [80]. Updating a library dependency can introduce breaking changes as the new update can be backward incompatible and cause system runtime failures [119].

Suggested solutions. Jezek et al. [119] suggested developers striving for stability when designing the signatures of library methods, they also recommend adding another layer of constraints to restrict linking and using package naming to distinguish between public and private sections, which helps verification tools to detect the illegal use of private packages. Additionally, programs should not rely on private JRE packages and public constraints should be used only for immutable values. They also suggest some minor changes for standard development tools and the Java language.

Suwa et al. [37] and Bogart et al. [117] suggested regular upkeep of the dependencies to minimize the impact of the changes on the provider package. German et al. [30] mentioned the need for the open-source application developer's input; Bogart et al. [117] also suggested collaborating with upstream projects to minimize the impact of the changes and mentioned the possibility of declining to update to the latest version, replicating the functionality to avoid the dependencies in the first place, and using package managers, that can refrain from doing changes or to announce and label the breaking changes.

The help of package managers and tools is needed as well: [14, 30, 40, 43, 61, 67, 75, 82] mention tool support and [64, 130] mention package management systems. Similarly, already mentioned semantic versioning [58, 74, 75, 80, 82], technical lag framework [80] and dependency management practices [22] have been proposed.

Numerous papers suggested some proactive measures. For example, Raemaekers et al. [120] proposed to design an interface that is both stable and backward compatible enough in subsequent releases but also flexible to adapt to changing requirements. Decan et al. [82] suggested to raise awareness of breaking changes, while Suwa et al. [37] said if we fear library rollbacks, i.e. dependency downgrades, we should avoid updates altogether. This approach however is not sustainable, therefore they emphasise the importance of understanding the migration status of libraries and rollback risks, and propose a method to detect rollbacks. Berhe et al. [36] proposed update scheduling system and said that early and cautious evaluation of the impact of the update is important to avoid breaking updates. Zerouali et al. [81] advised to not adopt newly available packages immediately but wait some time, as those versions can still contain bugs that need new patches. This approach can lessen the risk of library rollbacks. Preventative downgrades can be used in an attempt to avoid issues in future releases, aka the provider version is "locked" [40]. Version locking is suggested by Hejderup et al. [41] as well, in addition to dependency update tooling.

However, when breaking changes have already been introduced to the client, a reactive downgrade can be done [40]. Jia et al. [64] proposed solutions from three different sides: library developers can undo the latest changes, application developers can update their project to adapt to the introduced changes, or the end users can avoid using the incompatible library versions.

Too Many Dependencies

Problem. When a client project depends too heavily on third-party dependencies, it can lead to unusable, corrupted systems, increase the maintenance effort and lower the system's quality [32]. External resources bring risks to the software projects, they impact the decision making during development and increase the maintenance costs [65].

Suggested solutions. Package managers and automatic dependency resolutions are expected to help with the increased maintenance effort [59, 126]. D'mello et al. [56] proposed a blockchain based package control system to decentralise the architecture and provide a more stable way of handling big dependency networks; Green et al. [51] developed a new package manager for the Spack language to help with the management. Chen et al. [118] investigated how trivial packages can affect the overhead of maintaining client's dependencies and concluded the need for using dependency management and search tools; grouping trivial packages can help lessen the dependency load as well, when multiple smaller packages are grouped into one library.

Ferrarezi et al. [65] suggested information visualization techniques and LibViews tool to create visual representations of libraries' metrics and usage to understand the impact of the packages used in a software project. A visualization tool was also presented in [123] as a way to find out about system's past upgrades to help novice maintainers or maintainers of poorly documented systems with many dependencies; experienced maintainers can benefit from knowledge about upgrade decisions made by different systems. Isaacs et al. [70] proposed the importance of visualizations to handle the complex dependencies and developed Graphterm for interactive dependency graph visualization. Too many dependencies can complicate the upgrade actions, Ruscio et al. [32] proposed a model-driven approach to support the upgrades and Uppdatera for performing change impact analysis of third-party dependencies was presented in [41]. Semantic versioning was proposed as a solution in [74] for this so called "dependency hell".

It is important to have the correct and complete provisioning of all external libraries required by the project, however the resolution of some missing libraries is time consuming and error-prone; Atzenhofer et al. [47] developed a tool called LibLoader that automatically resolves missing dependencies in Java projects. If developers want to avoid using too many dependencies, one way for this is to re-implement the libraries in their own code [111]. Chowdhury et al. [132] highlighted that developers should carefully examine third-party

libraries before depending on them and limit the use of trivial packages as to not increase the maintenance costs of the project.

Two papers investigated third-party libraries in the context of software reproducibility in academic research, where experiments need to be replicable. Miksaa et al. [136] investigated the problem of program's workflow reproducibility when third-party libraries are involved and proposed the use of VFramework that can validate and verify the workflow re-executions. Unreproducibility issue was considered in [61] as well, but in regards to builds. They proposed PyDFix that can detect and fix unreproducibility issues in Python builds, which were caused by dependency problems.

Insufficient Tools

Problem. Ruscio et al. [32] and D'mello et al. [56] argued, that at times, the existing solutions and approaches for third-party library analysis and management are not enough and new solutions are needed. Ruscio et al. [32] claimed current tools to be able to predict only a limited set of upgrade faults. D'mello et al. [56] said that existing package managers are cloud-based with an ambiguous ownership, and they are heavily centralised in their architecture, which means they have a single point of failure.

Suggested solutions. A model-driven approach EVOSS to help with the upgrades was proposed in [32]; D'mello et al. [56] developed a blockchain based approach for package management using blockhain smart contracts which decentralised the setup. Some package management tools deploy unwanted updated automatically, therefore Zerouali et al. [80] wanted to address this by providing a general formal framework to measure technical lag. As Ada programming language was missing a package manager, the first third-party library repository manager was developed for Ada language in [45] to help with the package management.

Dependency Conflicts and Incompatibilities

Problems. Dependency conflicts and incompatibilities between library versions complicate the life of developers who have to navigate the system's dependencies [48]. For example, in case of Maven package manager, when a project depends on multiple versions of the same library, JVM loads one version while shadowing the other leading to runtime errors when the shadowed versions are referenced in the project [48].

Suggested solutions. As the intensive use of libraries bring the risk of dependency conflicts, most software management tools, like Maven, support the detection of dependency conflict

issues and give warnings respectively or assist with the automatic dependency conflict resolution [17, 21, 25, 43, 46, 48, 52, 56, 64, 118, 124], a new package manager was proposed for Ada language in [45] to also help with the dependency conflict issues.

Special conflict resolution tools have also been proposed. Wang et al. [48] developed Riddle, an automated approach that generates tests and collects stack traces for projects that are subject to risk of dependency conflicts. Macho et al. [62] proposed BuildMedic to automatically repair dependency related build breakage and Wang et al. [52] presented Decca, an automated detection tool that assesses dependency conflict issues to filter out the benign ones. As dependency conflicts are also related to semantic conflicts, Wang et al. [53] proposed Sensor to produce valid inputs to trigger the semantic conflict issues.

Wang et al. [50] discussed multiple dependency conflict solutions for Maven, including loading the path you actually need in the dependency tree, customizing the class loader and using the Maven shade plugin[27], that has a feature of relocation, which renames classes so that there is no conflict. Both Wang et al. [50] and Jia et al. [64] mentioned a straightforward solution of avoiding using incompatible library versions and upgrading or downgrading the component to find a compatible version. Jia et al. [64] also proposed DepOwl to detect dependency bugs and prevent compatibility failures.

Semantic versioning approach has also been mentioned in multiple papers [25, 56, 75] and the approach of grouping trivial packages to lessen the dependencies can be used here as well [118]. A novel aspect-oriented programming approach was presented in [140] that will help to ensure that the interactions between components conform to stated policies.

Hard to Track Dependency Info

Problem. Knowing the full dependency graph of a project is difficult, dependency relationships between the product and the third-party library are often complex [66]. However, it is important that the software system's developers are aware and keep track of their system's dependencies in order to better maintain the system.

Suggested solutions. In [66] it was suggested to use tools and methods to inspect the health and impact of these dependencies, they themselves proposed the definition of the REM graphs for visualization of dependency graphs that leverage metrics of the health of the dependencies.

[27] https://maven.apache.org/plugins/maven-shade-plugin/

Pashchenko et al. [90] argued that a failure to distinguish own vs third-party dependencies may increase the vulnerability risk and presented a tool, that leveraged the functionality of Maven to extract the library dependencies.

Li et al. [71] extended the library detection research and developed a tool called LibD that would help developers to identify libraries in Android apps. They also proposed two methods for library detection: the first is based on whitelists of known libraries and the other is to directly extract libraries from apps. Zhan et al. [68] gathered five publicly available Android third-party library detection tools (LibD, LibRadar, LibScout, LibPecker, ORLIS) as an online service called LibDetect. Another third-party library detection technique for Android called LibHawkeye was proposed in [69] that used a clustering-based approach. The whitelist method and LibRadar tool were mentioned in [72] as well, and in addition they developed their own package dependency graph (PDF) based approach in a tool called LibSift to detect third-party libraries in Android applications.

One of the more common techniques of third-party library detection is the whitelist approach, which was suggested in [71, 72]. It entails gathering a whitelist of known libraries and matching the names of the packages in a software project to the third-party library package names in the whitelist.

According to Zhang [29] one of the most important issues of software reuse is understanding the dependency relationships between the product and its components. They concluded that dependency analysis should be an effective means to solve this issue and designed a tool called DEx that can retrieve inter-dependency information from different sources.

Keeping the dependencies to a minimum was suggested in [94], as well as decoupling as much as possible. They concluded that empirical analysis and automated static analysis combined with deep knowledge of the system would be the best way to effectively monitor the system's dependencies.

Transitive Dependencies

Problem. Related to hard-to-track dependency info are transitive dependencies: the libraries developers use in their projects often are dependent on other libraries creating a network of transitive dependencies [23]. These dependencies can be an issue in the client project as they oftentimes are not easily noticeable when they are hidden multiple levels below the direct dependency and they can increase the attack-surface [130].

Suggested solutions. General dependency management tools can help with this issue as they monitor the dependencies [23, 56, 59, 61, 73, 87, 130, 126], SCA tools can help maintainers of deep learning libraries to manage transitive dependencies [129]. The importance of package maintainers' awareness is mentioned as it can help to reduce the risk of cascading failures [23, 45, 87, 127]. Limiting the package updates can also help [23], as well as using dependency constraints or policies [23, 87] and semantic versioning [23, 56].

Static program analysis is recommended in [100], general dependency management practices are suggested in [22, 56, 59].

Chen et al. [66] suggested the need for special tools, methods and visualizations to inspect the health of transitive dependencies and their potential impact.

Co-Installability

Problem. Installability constraints are defined for packages regulating which packages can be installed together and which cannot. Belguidoum et al. [63] defined three main types of dependencies: mandatory, optional and negative. Mandatory dependency has a firm requirement of a prerequisite dependency for some package's instalment: for example, a mail server might need a terminal with a specific CPU. An optional dependency may or may not need a prerequisite: a service is provided simply if the prerequisite is fulfilled, but if not, the installation is not denied. A negative dependency denotes a conflict forbidding an installation: for example, a mail transport agent cannot be installed, when another such library is already installed. This is making installation and deinstallation of the components a gamble [63].

Suggested solutions. Decan et al. [23] suggested being aware of the packages the system depends on, including knowing the co-installability and dependency constraints, Vouillon et al. [49] also recommended identifying which components cannot be installed together and checking if those incompatibilities are justified, Abate et al. [60] mentioned containerization and virtual environments gaining traction.

Package managers can be used to do the component installations and removals and dependency solving [23, 54, 60]. A Coinst tool was proposed in [49] to help with co-installabilities and Belguidoum et al. [63] presented a formalization of a static deployment system that ensures the success of installation and deinstallation.

Library Updates

Problem. The need to constantly update libraries can increase the developers' maintenance effort, impact the release schedule of software systems, cause system downtime or introduce new defects [97]. It can also be difficult to keep track of new versions of libraries [110].

Suggested solutions. Abate et al. [54] and Cogo et al. [110] suggested using automatic component management solutions for component upgrades to mitigate the update planning problem and package manager support was mentioned in [25, 33, 97, 130].

Berhe et al. [36] proposed different solutions for the library update problem: an early and cautious evaluation of the impact of the component update, establishing effective upgrade practices, coordinated updates, compiling an overview of all the third-party components and their info to track and schedule the update more reliably, monitoring the updates on a regular basis, analysing previous updates and scheduling updates along the dependency path. An early and cautious evaluation of the update was suggested by Cox et al. [39] as well.

Legal Issues

Problem. Using third-party dependencies can bring upon library licence violations and legal problems when developers overlook the licences of the libraries [83].

Suggested solutions. Qiu et al. [83] suggested developers pay special attention to open source licences to prevent potential legal risks and proposed a method to detect dependency-related licence violations of software projects. Kechagia et al. [85] suggested using legal advice or employing packaging companies as intermediaries between the open-source software community and proprietary software house to undertake the legal responsibilities. Parreiras et al. [139] proposed a framework for marketplace to link software artefacts within projects and across software projects to mitigate the legal problems related to licence decisions.

Bauer et al. [84] described how companies may develop their own tooling to help them manage documented licence information and generate more comprehensive reports of open-source software usage in their products, they also suggested FLOSS compliance processes and automating the licence compliance process as much as possible. Tooling was suggested by Bauer et al. [67] and Chen et al. [118] as well.

Build Problems

Problem. Third-party dependencies affect the program's builds and the managing of a successful build can be difficult, according to Macho et al. [62] there are many reasons for build breakage: testing and build specification issues, especially dependency-related issues like dependency resolution errors or outdated configuration.

Suggested solutions. Green et al. [51] proposed a new tool for Spack language that could help to handle the packages and build management. Mukherjee et al. [61] developed PyDFix to help with reproducability for Python builds. Automated dependency fixing process was presented in [62] as well: MavenLogAnalyze to extract build details from Maven build logs and BuildMedic to automatically repair dependency-related build breakage were proposed. Ma et al. [138] suggested that in addition to tool support, the third-party library developer's input might be needed as well.

Wang et al. [55] developed Watchman tool to continuously monitor dependency conflicts, that can cause build failures and Isaacs et al. [70] recommended the use of package managers and visualizations.

Unsuitable Dependency Management Approaches

Problem. Opting for unsuitable approaches when solving third-party dependency problems can cause additional bugs and vulnerabilities and complicate the project [58, 112]. For example, Grinter [151] presented a case of the Taurus system, that had failed because of poor dependency management in highly distributed environment due to coordination struggles and communication issues – this shows dependency management can be a managerial problem.

Suggested solutions. Javan Jafari et al. [58] suggested semantic versioning for efficient dependency management; DependencySniffer was also introduced for dependency smell detection. They also suggest developers to spend more time and effort on dependency maintenance with proper guidance, for example, including a "dependency maintenance" task to code review checklist whenever a new dependency is added to the project. Kula et al. [112] agreed, that improper dependency management can be fatal to a project and recommended using the "wisdom-of-the-crowd" approach, namely meta-data from different software ecosystems; they also introduced software universe graph as a means to model the popularity and adoption within a software ecosystem.

3.4.2. Security

Other critical problems related to third-party library usage are security vulnerabilities and issues that third-party dependencies can bring to the client project.

Using Libraries with Security Vulnerabilities

Problem. When a project uses third-party libraries with vulnerabilities, libraries that are unsupported or out of date, the project's overall vulnerability and attack-surface increases. Vulnerable components were also mentioned in the *OWASP Top 10 Web Application Security Risks* list in the *A06:2021-Vulnerable and Outdated Components* category [152], which shows the relevance of the issue. When a highly used third-party library contains a security vulnerability, it may directly impact hundreds of dependent software systems, which can lead to significant financial cost and reputation loss [102]. Library dependencies also may bring larger attack surface, however measuring it is non-trivial when evaluating vulnerability severity [95]. Third-party libraries carry the risk of supply-chain attacks through malicious packages [35, 95].

Suggested solutions. Tools help with the third-party library component vulnerability management and vulnerable library detection. Package maintainers should use automated tools to advance the detection and fixing of vulnerabilities and reduce the impact these vulnerabilities have on the dependent packages [6, 13, 17, 38, 87, 96]. The use of Dependabot's security automated pull requests was suggested in [102], Prana et al. [88] recommended to use other open-source tools like OWASP Dependency Check, Bundler-audit, RetireJS or GitHub dependency scan system, that look for known security vulnerabilities in third-party dependencies. Software composition analysis (SCA) tools such as Sonatype, Synopsis, Veracode and WhiteSource can be useful as they identify libraries used in a software project, vulnerabilities and licences associated with those libraries as well as other metrics [88]; in addition to those SCA tools, Imtiaz et al. [103] also suggested OWASP Dependency Check, Snyk, Dependabot, Maven Security Versions (MSV), npm audit and Eclipse Steady. SCA tools were similarly suggested in [104], that also proposed a novel test suite called Achilles, which was created for assessing the vulnerability scanners. Analysis of source code to identify vulnerabilities was proposed in [107] as well.

The idea of using actionable tools, which determine the parts of a software project actually affected by vulnerabilities in a third-party dependency and suggest alternative libraries, was mentioned in [13]. Staicu et al. [100] and Neil et al. [86] suggested using tools that aggregate

known third-party library security vulnerabilities in a repository and report them to the developer, Neil et al. [86] developed such tool in their work and Mitropoulos et al. [93] proposed a vulnerability dataset for the Maven Central Repository.

New specific tools were presented in numerous papers. Ponta et al. [97] proposed Vulas that implemented code-centric and usage-based approach to library vulnerability detection; in [99] Vulas was renamed to Eclipse Steady and released as an open-source tool. Bauer et al. [67] presented an automated approach to analyse the third-party dependencies and implemented it in Java on top of the open-source quality assessment toolkit ConQAT. A new tool called PDGraph was developed in [24] that can assist developers to audit their source code and identify underlying third-party dependency security risks. Third-party dependency update helper was developed in [44] that will aid developers in avoiding vulnerable library versions. ATVHunter was proposed in [73], a tool that detects third-party library vulnerabilities and presents a detailed report on them. A tool called Taser was developed in [100] that automatically extracts vulnerable specifications in JavaScript libraries. Yongming et al. [98] proposed CD3T, a tool that detects and scans public disclosure vulnerabilities in project dependencies. Metrics to determine the effect third-party dependencies have on software projects and that way understand potential source of security vulnerabilities were proposed in [92]. Pashchenko et al. [105] proposed a tool called Vuln4Real for addressing vulnerable dependencies and planning the migration activities.

Ponta et al. [97, 99] highlighted the importance of establishing effective third-party vulnerability management practices, Kula et al. [12] suggested developing strategies to improve developers' personal perception of third-party libraries. Higher awareness of the risks related to security vulnerabilities is needed among package managers on both, the software project as well as the wider ecosystem level [87], and deeper understanding of project dependencies will keep us informed of the project quality [24]. Studying how the vulnerabilities propagate is essential for the health of an ecosystem [96].

Code analysis was suggested in [108] to identify third-party dependency attacks so that the execution environment denies the execution of applications which include dependency-based attacks. Security audit as a part of current defensive programming paradigms could reduce software vulnerabilities before code is released [24] and a code vetting process could help to increase the software quality [96].

To help with the attack surface evaluation, Zhang et al. [95] proposed the use of metrics, namely vulnerability and library level metrics which can assist in prioritizing patching and overall system level metrics which can help developers choose secure and reliable development systems.

The most effective solution for third-party library vulnerabilities is updating the vulnerable dependency [38, 105]. Once a vulnerability is detected, the vulnerable library vendor should release a security patch and the software system developer should update their system to remediate the vulnerability [24]. A more proactive approach would be deprecating packages that suffer continuously from security vulnerabilities [96].

For the third-party library supply-chain attacks, Ferreira et al. [35] and Ohm et al. [91] have suggested multi-factor authentication for the library maintainers, version locking and disablement of install scripts for the library users, the isolation of build processes to reduce potential damage, carefully reviewing all dependencies and their updates and program analysis tools, such as Snyk and GitHub. Constructing and continuously maintaining the third-party library supply-chain is essential [106].

Lags in the Vulnerability Fixes

Problem. When vulnerabilities are introduced in the third-party packages, it may take some time for the fixes to be developed and adopted, so the fixing process is two-sided: on the third-party library side and the client side [101].

Suggested solutions. To mitigate the propagation lags in ecosystem, Chinthanet et al. [101] recommends developers to develop strategies for the most efficient updates via release cycles, better awareness for quicker planning of an update and allocating more time to understand the update's impact before updating. Higher awareness is also suggested in [87] among package maintainers, who should rely on better use of policies and automated tools to detect and fix vulnerabilities faster; automated tool support was mentioned in [17] as well.

3.4.3. Social

Social aspects of third-party library usage can cause another set of issues for the client project.

Human Interactions

Problem. Third-party dependency usage can also be characterized by the need for human interactions, which cannot be automated, increase the effort both from the client as well as the supplier side and introduce further complexities [116]. This can be for example interactions between the software system developer who uses third-party libraries and the developer who develops those third-party libraries. If such interactions are to take significant effort and time, development projects might need to plan for such interactions [116].

Suggested solutions. Palyart et al. [116] investigated the social interactions that occur when open source components are used and that imply the need for increased management effort. They concluded, that developers of the client project may need to become involved in the component project to ensure sufficient quality for use, they also suggest a need for a new theory to guide the social and technical interactions between projects. Harrand et al. [125] described the need for effort from the developers of the third-party packages. One way to decrease this effort is to ignore the minority of clients and focus on the small subset of libraries that are the most used, other way is to automate the library migrations for a limited part of the libraries and that way support a large majority of clients using it.

Finding Suitable Libraries

Problem. In the sea of available third-party libraries, it is difficult for developers to be aware of all the possible components and they can spend a lot of time trying to find exactly the libraries they need [1]. Heterogeneity of the libraries and the dependencies among them can add to the issue as it makes effective mining of the available data difficult [1].

Suggested solutions. Most popular solution for this are different recommendation systems, like LibRec, LibFinder and LibCUP [1]; another similar tool called CrossRec was developed in [1] to recommend third-party libraries that uses a collaborative filtering technique. Yan et al. [114] proposed a session-based social and dependency-aware software recommendation model that takes into account dynamic interests of developers and Katsuragawa et al. [113] presented domain-specific categories in library recommendations while developing a library recommendation prototype DSCRec. Xu et al. [111] mentioned, that one reason behind libraries are re-implemented, is that developers are simply not aware of available packages, therefore library recommendation systems are needed.

Package managers can help with this problem as well as they gather the third-party libraries of a programming language or use a public software repository that hosts the libraries [45]. Chen et al. [118] concluded that better search tools are needed and removing duplicate packages from the ecosystem can help with the abundance of trivial packages so developers can easier find necessary libraries. Blanthorn et al. [121] suggested modelling the software ecosystem to give a clearer picture to the developers and help with choosing the suitable packages.

Yang et al. [115] tackled the problem of untagged open-source software in GitHub, that makes retrieving those packages difficult for developers who want to use them. Some communities have begun tagging open-source software to help user retrieve the components in accordance with their characteristics and Yang et al. [115] further proposed a tag recommendation framework for DepTagRec.

Developers Don't Manage the Dependencies

Problem. The adoption risk of third-party components is often not understood by developers and is probably the most serious mistake developers do when implementing open-source software based solutions. Many of the problems related to the third-party dependency management stem from the fact that developers are often hesitant to update their dependencies, or they don't have the time for it as dependency management is a complex and time-consuming task and that the cost benefit is a demotivating factor [38].

Suggested solutions. Mainly tool support and automated approaches have been suggested to aid developers with this issue [13, 14, 17, 21, 34, 38]. Automated tools can manage the dependency updates, monitor the dependencies and notify developers when action needs to be taken. For example, Nguyen et al. [44] developed a tool called Up2Dep that analyses third-party libraries to provide information about necessary changes when updating a library.

Both Salza et al. [14] and Huang et al. [38] emphasized the importance of developers prioritizing the update effort as the most straightforward countermeasure against vulnerabilities are updates. Similarly, Kula et al. [12] suggested to develop strategies to improve developers' personal perceptions of third-party updates. They also recommended visual aids to provide visual analysis which could provide developers awareness and motivation for updates. Keeping dependencies to a minimum, being aware of them and

combining empirical analysis with automatic methods were suggested in [94] in order to be able to maintain the third-party libraries the system is relying on.

Cox et al. [39] suggested estimating the costs of upgrading the dependencies and using metrics, [112] recommended recording meta-data within ecosystems to provide developers with "wisdom-of-the-crowd" insights, in [13] it was mentioned that to be adopted, dependency updates should be well-indicated, not introduce breaking changes and require significant efforts.

Developers may often also not know, which dependency conflicts are harmful and take no action. Decca was proposed as a solution for this in [52] to aid developers in not overlooking critical issues. Umm-e-Laila et al. [18] suggested the need to explore the adoption barriers in a more systematic way. Gkortzis et al. [89] described different security risks of which developers are often not aware of and recommended different systems and automation tools that notify its users of available security updates. Korkmaz et al. [133] proposed metrics to quantify the impact of third-party libraries.

3.4.4. Ecosystem

Issues related to whole ecosystems of package management are relevant as well.

Packages in Decline

Problem. Next dependency management problem appears when libraries are replaced by some other packages and the support for the technology diminishes, which means the third-party library is in decline and becoming obsolete [109]. When support for the diminishing library is reduced, the project's developer has to create workarounds and the value of their own project might suffer [109].

Suggested solutions. To mitigate this problem, the package developers must evaluate the uptake of their technologies, Ma et al. [109] proposed a model for this uptake evaluation.

Mukherjee et al. [61] and Bauer et al. [67] described that the providers of libraries can discontinue their support and that tool support is needed when this happens, Katsuragawa et al. [113] proposed a library recommendation tool DSCRec to suggest new libraries when a component has to be replaced with another library.

According to Kula et al. [12] visual aids like Library Migration Plots can help with visual analysis for library migrations and Mujahid et al. [122] proposed specific popularity metrics

to determine popular packages and help developers avoid packages in decline, that in the future would need replacing.

Third-Party Library Evaluation

Problem. A problem related to finding suitable packages is evaluating the libraries to ensure proper dependencies are chosen, as the health of a software project depends on the packages it's using [134]. Therefore, it is important for the health and maintainability of a software project to depend on trustworthy and high quality third-party libraries.

Suggested solutions. In addition to limiting the number of dependencies, Decan et al. [134] suggested depending only on trusted packages and avoiding unstable and immature packages that are still in their initial development phase. Pandey et al. [135] mentioned the use of industry standard quality models and, in addition, proposed an ANP based model to prioritize the characteristic of a software components as they argue that not all characteristics are equally important when evaluating the components.

Wu et al. [131] proposed a usage and dependency model to evaluate the open-source component behaviour and three metrics to measure interaction behaviour complexity.

Parreiras et al. [139] mentioned contemporary approaches to increase transparency and provide search over multiple sources of software data, like Krugle, and textbased search for assisting developers with finding meaningful pieces of code, like Snipplr and Koders, however they have their limitations. Therefore, Parreiras et al. [139] proposed a framework for a marketplace for open-source software data to help developers find more meaningful artefacts. Similarly, Ma et al. [138] built an infrastructure to handle the open-source software ecosystem and show how the tens of millions of projects are interconnected.

Chowdhury et al. [132] said that developers should carefully examine packages before depending on them by applying a systematic approach for selecting packages and Mujahid et al. [122] proposed popularity metrics to determine popular packages and packages in decline to help with choosing more sustainable packages.

Packaging Ecosystem Maintainability

Problem. Packaging ecosystems are fragile and prone to maintainability issues as the packages they are hosting are in constant development [23]. A package may get removed, become archived, updated in backward incompatible ways etc. which all complicate the maintainability of the ecosystem [23].

Suggested solutions. Maintainers can be helped by management tools that monitor dependencies and notify when there is an update need [21, 23, 29, 31, 33], in [82] tools like David, Gemnasium and Greenkeeper were mentioned and for npm also an audit tool has been introduced [5].

Decan et al. [23] introduced multiple proactive measures like limiting the package updates, specifying dependency constraints on the versions of the libraries they depend upon, using semantic versioning and being aware of the packages you depend on. Importance of developers' awareness of the packages and the risk entailed was mentioned in [5, 29, 82] as well. In addition, Zimmermann et al. [5] suggested code vetting as a way of defending against vulnerable code and systematically training highly influential maintainers.

In [128] a sociotechnical analysis was proposed as a way to enable the analysis of software ecosystems and the usage of ecosystem visualization tools was suggested. Dependency graphs [31] and software universe graphs [112] can also be used for risk assessment and modelling software ecosystem. Ma et al. [138] built World of Code infrastructure for open-source software ecosystem sustainability through gathering frequently updated collection of version control data to cross-reference information of open-source ecosystems. They proved that World of Code is capable of supporting trend evaluation, ecosystem measurement and package usage determination.

Inter-Dependency Info

Problem. The dependency info from source code to licence information across different dependency artefacts and different projects dependencies is difficult to track [139].

Suggested solutions. Parreiras et al. [139] created a framework for a marketplace for open source software data that could help increase transparency. Blincoe et al. [137] focused on describing Reference Coupling as a new method to detect dependencies between projects, as the awareness of such dependencies is not trivial and previous static dependency approaches did not identify dependencies across projects. Abate et al. [31] also dealt with inter-package relationships and complex maintenance problem and proposed the help of release managers and dependency graphs for risk assessment.

Dozens different applications might be needed to be installed for a complex project in order to build it, also only a real compilation of a project can assess the needed collection of software [30]. Tool support and open source application developer's input were suggested

to help with this; a method to model the inter-dependencies and a method to visually represent them was proposed in [30]. Visualization was also proposed in [123].

Decan et al. [57] argued that as more and more third-party libraries are now being developed and distributed outside of the CRAN ecosystem, inter-repository dependency management will become a problem, and concluded that an automatic package installation and dependency management tool that relies on a central listing of available packages is also needed for R, like exists for JavaScript or Python for example.

3.4.5. Most Investigated Problems Over Time

An area chart of most investigated problems over time has been presented in Figure 5. As the papers published have been growing each year, it is to be expected that most of the problems were from papers from the last few years. Security vulnerability papers have been growing almost exponentially, having been researched in 7 papers in 2020 and in 10 papers in 2021. The problem of breaking changes and using too many dependencies has generally been researched in an upwards trend as well. Packaging ecosystem maintainability and outdated dependencies however have seen a slight decreasing trend in the last few years.

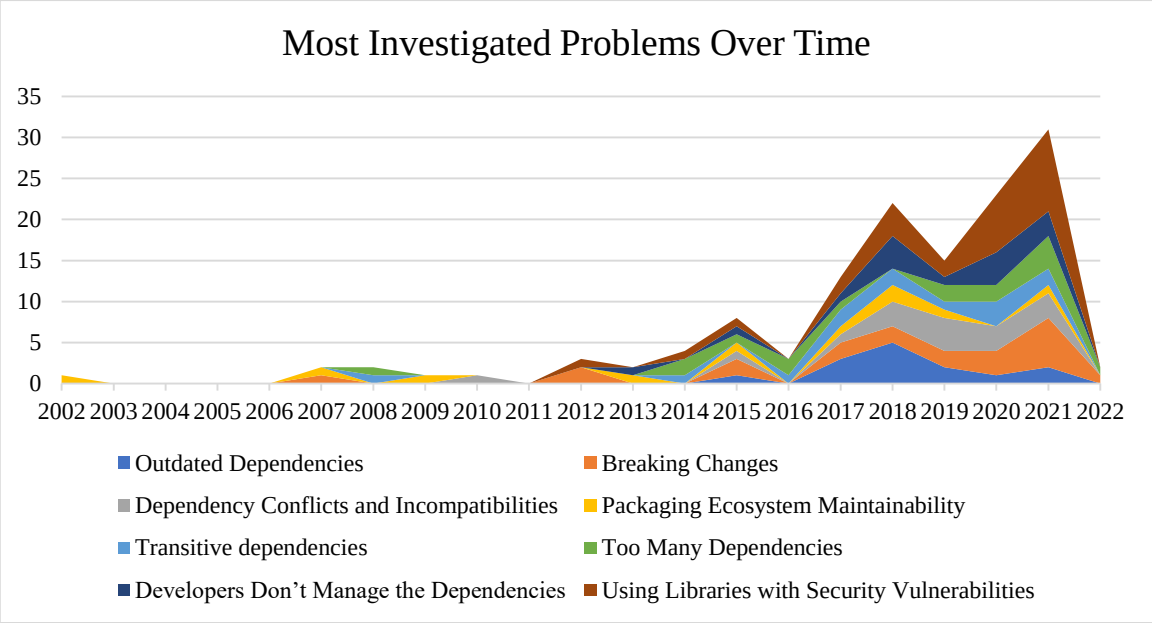

Figure 5. Most investigated problems over time.

RQ4 & RQ5: Problems and Mitigations Summary. Most investigated problems for third-party library analysis have been security related: how vulnerabilities migrate in third-party dependencies, how to detect and solve them and how they evolve in package management ecosystems. Other well-researched problems have been about dependency management: handling outdated dependencies and conflicts, using too many dependencies in a project, transitive dependencies and breaking changes; developers not managing the dependencies in their projects has been an issue as well and package management ecosystem maintainability has been another question seeking an answer. Mitigation techniques proposed for these issues have been mostly automatic approaches: using package managers and other specific automated tools to lessen the manual work. Another important thing to note is the need for developers to prioritize the third-party dependency upkeep.

3.5. Research Question 6: Package Managers

Used package managers were gathered from the reviewed papers to analyse which package managers have been more popular in this research domain and which ecosystems have not gotten that much recognition, this is presented in Table 5. Figure 6 plots the distribution of package managers per year.

Maven was the most investigated package manager in the selection of articles in this thesis with 52 papers including the Maven or Apache ecosystem in their research. Maven was initially released in 2004, the earliest paper mentioning Maven in their research was from 2010 and it has been gradually gaining more interest, with roughly half of the papers published in the last few years in the dependency analysis domain containing Maven. As Java is one of the top programming languages and Maven is its main package manager, there is also incentive for more research.

Npm package manager was another popular research topic with a total of 38 publications using npm. Npm was released in 2010 but started gaining more recognition in scientific articles in the last years, like Maven. As of February 2022, npm repository holds 2.24M JavaScript packages, which is an indication of its significance and therefore it can be expected, that npm's popularity is not decreasing any time soon.

Table 5. Package managers used in the studies.

Package Manager	Studies
Alire	[45]
Atom	[25], [75], [117], [138]
Bioconductor	[117], [138]
Bower	[6], [60]
Cabal	[60], [117]
Cargo	[21], [22], [25], [60], [72], [75], [106], [117], [121], [134], [138]
Clojars	[60]
Cocoapods	[117]
Composer	[115]
CPAN	[21], [60], [75], [117], [138]
CRAN	[21], [23], [57], [60], [75], [109], [112], [117], [121], [130], [133], [138]
Debian	[29], [30], [31], [32], [49], [54], [60], [136]
dnf	[60]
Dub	[25], [75]
Elm	[25], [75], [121]
Go	[60], [117]
Haxelib	[25], [75]
Hex	[25], [75], [117]
Homebrew	[75], [137]
Ivy	[128]
Luarocks	[117], [138]
Maven	[1], [12], [14], [24], [25], [33], [37], [38], [39], [41], [43], [46], [47], [48], [50], [52], [53], [59], [60], [62], [65], [68], [69], [73], [75], [76], [77], [88], [89], [90], [92], [93], [94], [97], [98], [99], [103], [104], [105], [112], [113], [116], [117], [119], [120], [121], [123], [124], [125], [128], [138], [139]
npm	[5], [21], [22], [23], [25], [34], [35], [40], [42], [56], [58], [60], [66], [74], [75], [81], [80], [82], [83], [87], [91], [94], [100], [101], [102], [103], [107], [108], [110], [115], [117], [118], [121], [122], [127], [132], [134], [138]
NuGet	[21], [25], [60], [75], [94], [117]
Opam	[60], [137]
Packagist	[21], [25], [60], [74], [75], [117], [134], [138]
Paket	[60]
Portage	[126]
Pub	[25], [75]
Puppet	[25], [75]
PyPI	[25], [55], [60], [61], [75], [88], [91], [96], [99], [115], [117], [121], [133], [138]
RPM	[54]
RubyGems	[21], [22], [23], [25], [60], [74], [75], [88], [91], [94], [116], [117], [134], [136], [137], [138]
Spack	[51], [70]
Stack	[117]
Xargo	[106]

RubyGems, PyPI and CRAN package managers have been relatively well researched also, with an increased popularity in the recent years. They are however less prominent in the research than for example Java and JavaScript languages' respective package managers Maven and npm. The same can be said for Cargo and Packagist package managers for Rust and PHP programming languages. All these package managers are represented nearly every year since mid-2010s, but they don't have a strong dominance.

Debian package manager as one of the first package managers created was more studied in the earlier years, however it has now lost its popularity. New package managers are released every year and the Debian ecosystem has not managed to maintain its prominence in the dependency management field.

Numerous package managers were mentioned only in one or two publications, which shows either these package managers are relatively new or their users' community is insignificant (Haxelib, Dub, Elm). Some more known package managers from the libraries.io registry were not present at all, for example Meteor for JavaScript.

Most papers focused their research on one specific package manager, more popular choices for that were Maven and npm. Some papers sought to compare two or three packaging ecosystems; for example, Palyart et al. [116] investigated the social interactions in Maven and RubyGems communities, while Decan et al. [23] investigated the package dependency evolutions in npm, CRAN and RubyGems ecosystems. Couple papers aimed to get an even more comprehensive overview of the dependency domain and focused their research on 10+ different package managers. An example of this is Bogart et al. [117], who addressed practices of breaking changes in 18 open source software ecosystems; Stringer et al. [25] compared technical lag of dependencies in 14 major package managers.

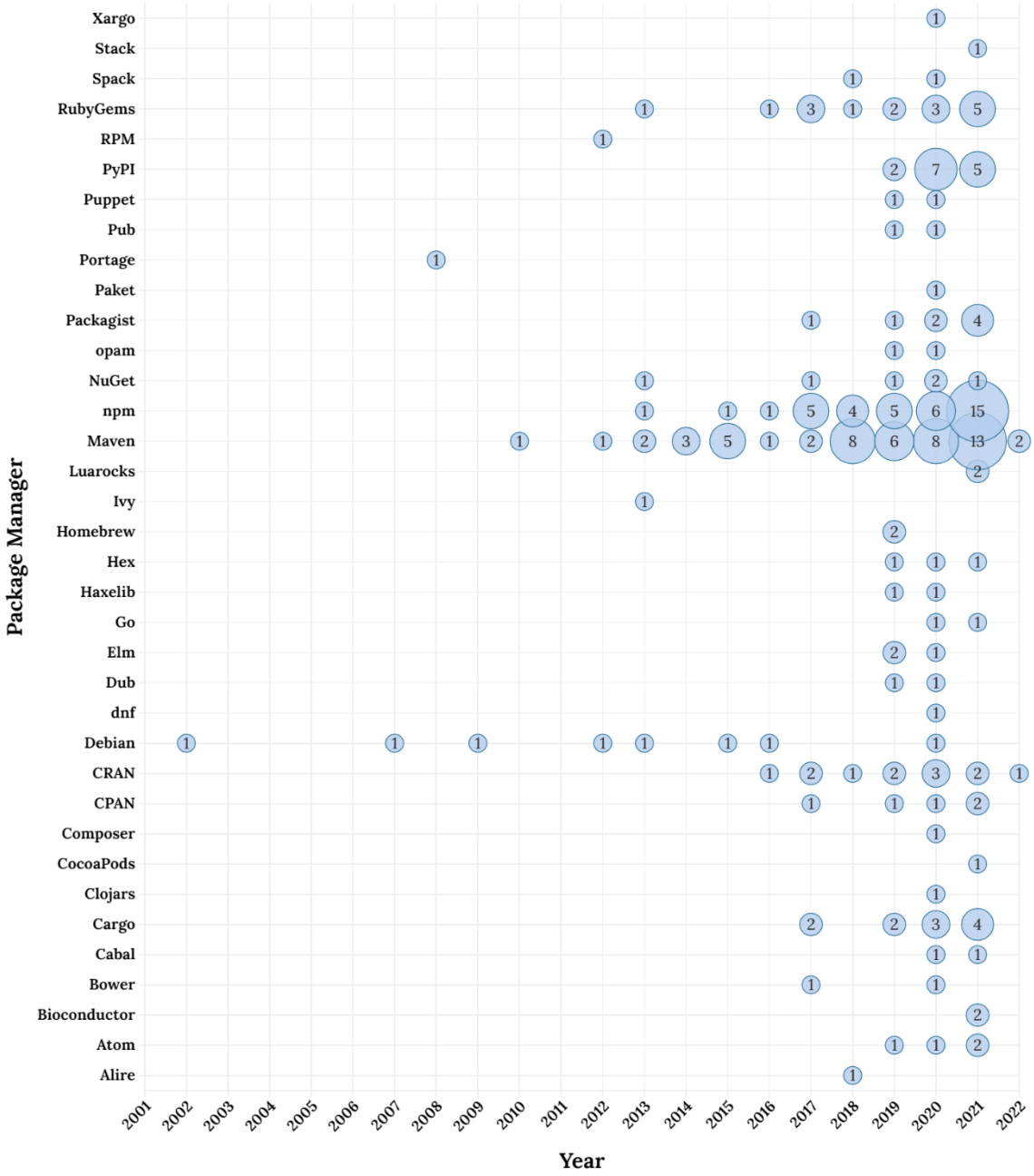

Figure 6. Yearly distribution of package managers used in the studies.

RQ6: Package Managers summary. Most third-party dependency analysis research was focused on Maven and npm package managers, as Java and JavaScript programming languages are currently very popular in the software development community. The representation of Debian package manager was bigger in the starting years of third-party dependency research, however it has been declining as of late. Other package managers, like CRAN, PyPI, RubyGems are being used more and more in the recent years.

3.6. Research Question 7: New Tools

Many of the papers also proposed tools or prototypes as a part of their research to help with different third-party dependency-related issues. An overview of them is given so readers can learn about current methods for dealing with open-source dependency management. New tools are brought in Table 6.

Table 6**.** Tools proposed in the studies and their package managers.

	Tools	**Studies**	**Package Managers**
Maintenance	Dependency Resolvers	[29], [46], [47], [59], [63]	Debian, Maven
	Dependency Analysers	[76], [66], [90], [138], [122], [78], [107], [62], [43]	Maven, npm
	Dependency Conflicts	[32], [41], [44], [49], [52], [55], [58], [64]	Debian, Maven, npm, PyPI
	Tests	[48], [53], [104]	Maven
	Library Detection	[68], [69], [71], [72], [73]	Maven
	Visualizations	[65], [70], [123]	Maven, Spack
	Reproducibility	[61], [136]	PyPI
Security	Vulnerability Detection/Security	[24], [35], [86], [90], [92], [97], [98], [99], [100], [105], [106], [107], [108]	Maven, npm, PyPI
Social	Recommendation Tools	[1], [113], [115]	Maven
Ecosystem	Dependency Managers for a Language	[45], [51]	Alire, Spack

Most new tools were proposed for the Maven package manager, namely dependency resolver and analysers, recommendation tools, tests, security related tools, library detection and visualization techniques. It was to be expected, as most papers did their research on Maven and the Java community is large.

3.6.1. Dependency Resolvers

Ossher et al. [46] tackled the dependency resolving problem by proposing an algorithm for automatically resolving dependencies for open source software. The approach was implemented as a part of Sourcerer, which is an existing infrastructure for large-scale

analysis of open source software and works by cross-referencing the analysed project's missing types with a repository of open-source artefacts. A tool called LibLoader was proposed in [47] to automatically resolve missing dependencies in open source Java projects and help with the correct source code analysis. Belguidoum et al. [63] presented a formalization of installation and deinstallation of components to provide a safe deployment framework, a simple prototype was also developed in OCaml. In [59] API compatibility problems were investigated, that occur when Java systems use automatic dependency resolutions. They proposed an approach based on static type checking that can capture problems at build time and can aid in integration testing. For this, the library of JaCC was extended. Zhang [29] designed DEx, a comprehensive prototype tool that can retrieve interdependency information from different sources so that the user can use one or more dependency resolution techniques.

3.6.2. Dependency Analysers

Some proposed tools didn't necessarily resolve the third-party dependency problems, but were presented for different aspects of third-party dependency analysis. Bauer et al. [67] presented an automated approach to analyse the dependency information and library usage from the source code of a project to support decision making during software maintenance. They implemented this approach on top of a software quality assessment toolkit ConQAT. Pfretzschner et al. [108] used WALA, a program analysis network, to develop their static code analysis tool and Sun et al. [78] proposed a prototype of a tool on top of WALA, that detects imitations of library APIs in client code to achieve better software maintainability. Kula et al. [43] developed a tool PomWalker to extract dependency information from the POM-files of a repository. Soto-Valero et al. [76] presented a tool called DepClean to automatically analyse and remove bloated Java dependencies. Uppdatera tool was proposed in [41] to prepare change impact analysis for library dependencies in Maven and Nguyen et al. [44] presented Up2Dep, that analyses third-party libraries to provide users with information about the fixes or changes they may need to do when updating a library. Ma et al. [138] facilitated open access to the large data collection by developing a tool on top of their World of Code (WoC) infrastructure. It stores growing amount of data in the open-source ecosystem and provides basic capabilities to cross-reference and analyse the data at scale [138]. Mujahid et al. [122] developed a prototype web browser extension called Centrality Checker to detect packages in decline to inform developers of declining packages for better maintainability. MavenLogAnalyzer (MLA) and BuildMedic were proposed in

[62]. MavenLogAnalizer extracts the build results and details from Maven build logs and BuildMedic automatically repairs dependency-related build breakage. REM-Dependabot, an enhancement of the Dependabot tool was proposed in [66] in order to help maintainers of software projects to inspect the health of its dependencies.

3.6.3. Dependency Conflicts

Dependency conflict issues were targeted in numerous papers. Decca, an automated detection tool that assesses dependency conflict issues' severity and filters out benign ones was proposed in [52]. Jia et al. [64] proposed a tool called DepOwl to detect dependency bugs and prevent compatibility failures. DependencySniffer was presented in [58] to analyse projects that use npm and detect recurring dependency management issues, also called dependency smells. A coinst tool was presented in [49], that can be used to identify non-co-installable components in a repository, and Watchman was developed in [55] that performs holistic analysis to continuously monitor dependency conflicts caused by library updates. Di Ruscio et al. [32] implemented their model-driven approach that detects faults before performing updates to third-party dependencies.

3.6.4. Tests

Some papers also developed automated testing approaches and techniques to detect vulnerable dependencies. Wang et al. [48] proposed Riddle, an automated approach that generates tests and collects stack traces to produce and manage dependency conflict issues. Wang et al. [53] proposed a similar technique Sensor, that synthesizes test cases to trigger inconsistent behaviour and produce semantic conflict problems. A novel test suite called Achilles was proposed in [104] to evaluate the performance of third-party dependency vulnerability scanners.

3.6.5. Library Detection

LibDetect framework was proposed in [68] that integrates five publicly available tools as an online service to detect Android third-party libraries. Zhang et al. [69] proposed LibHawkeye, a clustering based technique and Soh et al. [72] presented LibSift tool, both to identify third-party libraries in Android apps. Li et al. [71] implemented their library detection method in a tool LibD.

3.6.6. Visualizations

LibViews, an information visualization tool to create visual representations of libraries' metrics and usage on software projects was presented in [65]. Isaacs et al. [70] proposed Graphterm, a Python tool for interactive dependency graph visualization and Kula et al. [123] proposed a visualization tool for evolution of systems and their library dependencies.

3.6.7. Reproducibility

VFramework to verify and validate workflow re-executions was proposed in [136] to identify whether there exist any differences in software dependencies between two executions of the same workflow. Mukherjee et al. [61] presented PyDFix tool to detect and fix unreproducability in Python builds caused by dependency problems.

3.6.8. Vulnerability Detection/Security

Tools aimed at dependency-related vulnerability detection and security aspects were proposed as well. Ponta et al. [97] proposed a tool called Vulas internally to SAP, that combined static and dynamic analysis to determine the impact of the vulnerable portion of libraries used by an application; in [99] Vulas became available as an open-source tool under the name Eclipse Steady. Yongming et al. [98] proposed CD3T as a command-line tool to detect and scan public disclosure vulnerabilities related to project dependencies. In [100] Taser was developed to automatically extract vulnerable specifications for JavaScript. Neil et al. [86] proposed an alert and query system that warns users of security vulnerabilities. Users can query and receive alerts from the knowledge graph if any threat intelligence is found in their products related to the third-party libraries they use. Pashchenko et al. [90] presented a tool that extracts the library dependencies and identifies the known vulnerabilities affecting them. Pashchenko et al. [105] planned to undergo the SAP procedure to publish the tool behind Vuln4Real for counting actually vulnerable dependencies in industrial context. Rastogi.js was proposed in [107], that is an asynchronous dependency analysis tool capable of identifying vulnerable dependencies, analysing the cascading effect of known vulnerable modules and the version-evolution of modules. Pfretzschner et al. [108] developed static code analysis on top of WALA that detects third-party dependency attacks.

Li et al. [24] proposed PDGraph for generating a directed project dependency graph, that can assist developers to audit their source code when their projects' components have underlying security risks. Massacci et al. [92] provided an online demo for computing

metrics for real-world software libraries and Wang et al. [106] developed a prototype of an automated tool that helps reviewers to identify program locations that likely require a security audit. Ferreira et al. [35] proposed a permission system with a policy and corresponding enforcement mechanism to ensure that malicious updates cannot use security-critical resources for which they do not have permissions. Zhan et al. [73] developed ATVHunter that can report detailed information about vulnerabilities of third-party libraries.

3.6.9. Recommendation Tools

Nguyen et al. [1] and Katsuragawa et al. [113] proposed library recommendation tools to assist developers in choosing the third-party libraries. CrossRec, a novel approach utilizing the collaborative filtering technique to recommend third-party libraries was developed in [1] and a library recommendation prototype DSCRec was presented in [113]. Yang et al. [115] proposed a tag recommendation framework for DepTagRec based on neural network models to help with the untagged third-party libraries issue.

3.6.10. Dependency Managers for a Language

Two papers developed completely new tools for dependency management for programming languages that did not have package managers specifically meant for those languages. Mostero [45] did so for Ada language and developed alr (Alire), that facilitates easy reuse of third-party Ada software and dependency resolution. SpackDev was a system proposed by Green et al. [51] to facilitate simultaneous development of interconnected third-part libraries for Spack.

RQ7: New Tools summary. Different helpful tools have been developed within the third-party library studies. Many new tools handle the security aspects of the third-party dependency management as it is important to detect and mitigate vulnerabilities in third-party dependencies and automatic approaches are the best way to handle them. Numerous tools also help with general third-party dependency management, like detection or conflict resolution. Most tools have been proposed for the Maven package manager.

3.7. Research Question 8: Future Work

Articles' future work suggestions were gathered and summarized in order to find out, which ideas have already been used in the succeeding research, and which suggestions are yet to be investigated, to propose further work objectives for the reader.

3.7.1. Future Work Ideas Already Researched

Some future research suggestions have already been researched in papers succeeding those articles. These ideas are summarized here and remaining future work recommendations are highlighted.

Bloated Dependencies

A large scale analysis of bloated dependencies with a novel tool DepClean development was presented in 2021 [76] and they suggested to further investigate such methodological questions, like when to analyse the bloat, who is responsible for debloating and how to properly manage complex dependency trees in order to avoid dependency conflicts. This research is continued in [77] and, for example, they suggest to analyse the bloat analysis before every release to ensure no bloat is deployed and mainly focus on the importance of developers themselves to manage the bloat.

Yet to be researched: As the idea of bloated dependencies is quite novel and introduced only just in [76], it is one possible avenue for further research. For example, it would be beneficial to investigate the dependency tree management [76] or build additional helper tools for the developers [77]. As the bloatedness of software projects is only investigated in Maven, another idea would be to research this for other package managers.

Technical Lag

Technical lag metric for dependencies in npm was introduced in [81]. They suggested for future research to take into account vulnerabilities and bugs when analysing the technical lag, considering transitive dependencies, carrying out similar analysis for other package managers, studying releases with major version 0 and also accounting for pre-release tags. Inspired by Zerouali et al. [81], this research was continued in [82], where they added the aspect of vulnerabilities to their analysis. They recommended to continue the study for other package managers as well, in addition to exploring the influence of semantic versioning policies on the technical lag, empirically analyse the negative aspects of technical lag and its main causes. Zerouali et al. [80] built on both of these works and also considered

transitive dependencies in their research. Additionally, Zerouali et al. [80] was inspired by Lauinger et al. [6] to include other types of external applications to their research, like deployed websites. Many questions were still left unanswered, therefore they recommended in the future to provide actionable guidelines about the npm dependency updates, to do additional research on reusable software libraries of other kinds, like Linux-based deployments or Docker containers, and continue exploring different ways to measure the technical lag.

Technical lag was studied across 14 package managers in [97], which fulfilled the suggestion to compare technical lag across different package managers. Technical lag was also investigated in the context of mobile applications: Salza et al. [14] explored how technical lag varies in mobile applications over time. In addition, it has been researched how technical lag can be introduced with dependency downgrades [40].

Yet to be researched: No research has considered studying the releases with major version 0 and accounting for pre-release tags to elaborate technical lag research. Studying the influence of semantic versioning policies and dependency constraints on technical lag can be another way for future research, as well as empirically analysing the negative aspects of technical lag and its main causes. Further research should provide actionable guidelines for dependency updates, do additional research on reusable software libraries of other kinds, and continue exploring different ways to measure the technical lag.

Evolution of Package Dependencies

In [23] the dependency issues of three package managers were investigated. In the future, they aim to complement their quantitative analysis with qualitative analysis through surveys. In addition, they want to extend their analysis to other package managers, study mixed source ecosystems (both open and closed software), and carry out socio-technical comparisons of the ecosystems in order to understand the collaboration of the communities. [21] was a follow-up to [23] and extended this analysis by carrying out similar empirical comparison of seven packaging ecosystems. They suggested to further study complex network properties of ecosystem package dependency networks, and study the ecosystem dynamic from socio-technical point of view.

Similar study was done also in [22], which was partially inspired by Wittern et al. [127], however compared to [127], Kikas et al. [22] considered the network analysis in more detail and included applications in their network analysis step. In [22] they suggested to

complement the future research with qualitative work as well. In addition, they recommended to develop a measure to quantify dependency health and the broad level goal of future research would be to support developers with tools in dependency maintenance. This suggestion was fulfilled in [66], as they proposed metrics of health for software dependencies and also developed REM-Dependabot tool to help the maintainers inspect the health of software projects' dependencies. Future work for [606] includes improvements to their current implementation and tool.

Another paper inspired by Decan et al. [21] is [121], that evaluated ecosystem communities and they suggest to further expand their technique to consider co-authorship network, and create models of how software ecosystems evolve to try and replicate their results synthetically and gain insight into how the ecosystems evolve organically.

In [33] a study of the evolution of dependencies in the Maven ecosystem was done. They recommended the future work to build recommendation systems to aid developers in the updating process. Such tools have been proposed for example in [102] and [41]. They also mention that it is worthwhile replicating this study on other ecosystems.

Yet to be researched: Qualitative analysis has been suggested to complement already done quantitative analysis and find out more about developers' behaviour regarding ecosystems' package dependencies. Studying mixed source ecosystems could be novel avenue for future research and carrying out socio-technical comparisons of the ecosystem has been suggested as well. Studying complex network properties of ecosystem package dependency networks, developing a measure to quantify dependency health and supporting developers with tools in dependency maintenance are additional ways to continue this research. Extending the technique used in [121] could also prove to be insightful and Chen et al. [66] planned to improve their implementation of dependency health metrics.

Missing Dependency Resolution

When adding third-party library artefacts to a project, it is often challenging to get these artefacts to build or run, as they may use other dependencies themselves, which creates a long chain of dependencies that need to be resolved. A solution for this in the form of missing dependency resolution algorithm was implemented in [46] as part of Sourcerer and they planned to create an Eclipse plugin so developers could automatically locate artefacts using their system. They also identified version incompatibilities of library versions as a

problem, which was addressed in [47], that developed LibLoader application based on the idea of Ossher et al. [46], and provided a solution for the library version problem.

Improvements to LibLoader tool, that automatically resolves missing Java dependencies, were suggested: extend the tool to also search in other library-containing repositories as it currently only uses the Maven 2 Central Repository, make LibLoader more accessible and use it as part of a micro service architecture, and build a platform for controlling tools for the software quality analysis. One such platform has been developed in [68] that integrates five publicly available tools as online service, however it only provides the functionality of detecting third-party libraries, not performing further quality analysis.

Yet to be researched: Improvements can be done to LibLoader, in order to help developers manage and resolve the dependencies in their Java projects better.

Vulnerability Detection

Open-source vulnerability detection method Vulas was presented in [97]. Improvement ideas were to extend Vulas to support other languages other than Java, and solve the problem of systematically linking open-source vulnerability information to the corresponding source code changes. Vulas was extended in [99], where it was released under the name of Eclipse Steady. The support for other languages was explored, but not formally implemented, therefore this remains as future work, in addition to the vulnerability information linking.

Yet to be researched: Extend Eclipse Steady to support other languages than Java and find a solution for systematically linking open-source vulnerability information to the corresponding source code changes.

Updating Dependencies

Automated dependency upgrades were considered in [42]. They stated as future work identifying possible factors in support of a recommender system for merging pull requests and that future work would need to overcome technical and social challenges of automated third-party migration. Alfadel et al. [102] built on their work and investigated also the social side by researching the degree to which developers adopt the automated pull requests that update dependencies, however they focused more on the security aspect. Improvement recommendations were suggested for the Dependabot tool in [102]: Dependabot should properly handle peer dependencies, be more efficient for projects with a high number of dependencies, and it needs to prioritize security updates. Related to security updates, Pashchenko et al. [13] investigated the choices of developers' overall decision making. They

recommend broadening their study to more countries and correlating the results with different types of industries. They say for the community at large, the most challenging work would be to develop dependency and security analysis tools. Some such tools are summarized in this work in chapter 3.6. and Imtiaz et al. [103] did a study on vulnerability reporting of different software component analysis tools as well. They pointed out two research directions: establishing frameworks to identify false positives for dependency vulnerabilities and building automated approaches for continuous monitoring of vulnerability data.

Similar automated dependency updating analysis was done in [41]. In future work they aim to establish best practices for updating third-party libraries, and for future updating systems they recommend investigating hybrid workflows by combining dynamic and static analysis. The dependency updating study in [39] proposed metrics to evaluate the update cost, and the future work in that area would be to refine the metrics more, estimate the update effort and monitor the metrics' impact on the software quality.

Yet to be researched: Automated third-party migration needs further research, as well as the social side of how well developers are willing to adapt those recommendations. Best practices for third-party updates could be more researched and metrics identified for estimating the update cost could be refined.

Security Vulnerabilities

Security vulnerabilities were researched in [5] and [87] for npm package manager. In [5] it was suggested that potential mitigation techniques should be investigated in the future, numerous mitigation techniques have been proposed for security vulnerabilities related to third-party dependencies and those are summarized in chapter 3.4.2. Decan et al. [87] suggested to replicate the study in other open-source package dependency networks as well. This inspired Alfadel et al. [96] to replicate the study for Python packages. They further recommend investigating additional ecosystems. They also mention an open avenue for further research would be to develop a process that ensures basic security checks before publishing a release of a package, and work on the development of further package security tools should be done to help practitioners with secure development.

Empirical study on dependencies with security vulnerabilities was also done in [24] and they suggested in the future to investigate other project management tools for vulnerable dependencies, such as Conan, NuGet or Gem, to continue their research. They also aim to

leverage the code property graph to determine whether a vulnerability is transitive. [89] was another such article in regards to Maven security vulnerabilities. They recommend future studies to explore clustering similar projects and implementing the toolkit presented in the paper as a workbench for practitioners and researches. A comparative study of Java, Python and Ruby projects' vulnerabilities was done in [88]. In addition to continuing this research with other programming languages, they say the future work lies in investigation into associations between vulnerability types and the factors that promote or mitigate them. They also suggest identifying characteristics of projects that have a good track record of resolving dependency vulnerabilities and how those characteristics could be emulated in other projects. In addition, automated vulnerability detection and update approaches could be investigated, one such work being for example Dependabot [102].

Hejderup et al. [107] proposed in its work to build a potential prototype by using advisories from NSP and data from GitHub to aid developers. They also recommend grading the severeness of a security risk depending on where the dependency code is used and whether this can be exploited with the advisory description to assist developers evaluate the urgency. In addition, they also suggest to investigate how we could resolve dependency security issues with backward compatibility in order to not create breaking changes. Inspired by [107], Pashchenko et al. [90] conducted a similar study in Maven environment and they suggested to extend this study by analysing all existing libraries in Maven Central and identifying a precise model for automatic identification whether a library is vulnerable. Pashchenko et al. [105] built on [90] and both suggested future work to complement existing studies on reasons why developers do not update dependencies – this was researched for example in [12]. In addition to this, Pashchenko et al. [105] also recommended future research to understand how important a list of vulnerabilities is and if the precision of the list would give better results on detecting vulnerable libraries.

Yet to be researched: Further package security tools could be developed to help with secure development, transitive dependencies should be investigated in the context of vulnerability migration and further package managers should be included in the security research. Another idea would be to evaluate the security risks' severeness to decide on the fixing effort and whether gathering a list of known vulnerabilities is a good way of vulnerability detection.

3.7.2. Future Work Related to Specific Tool Improvements

Numerous papers that presented tools or automated approaches for their research proposed further work suggestions related to the improvements of such tools.

Package manager called Alire was proposed for the Ada language in [45] and they aim to solve the open issues related to Alire in the future, like Windows port and cross-platform builds. Staicu et al. [100] proposed a way to automatically extract vulnerable specifications in JavaScript libraries and their next step would be to extend the implementation to support more testing frameworks. LibViews tool for third-party dependency visualization proposed in [65] could be enhanced by including new features, such as visualization of the classes with errors and their code, which methods were removed from each version and what classes use these methods.

In [113] a domain-specific category recommendation system called DSCRec was developed and for future work they proposed to investigate other techniques such as collaborative filtering to improve their results. A recommender system called CrossRec was presented in [1] that used a collaborative filtering technique. They aim to improve CrossRec through equipping it with the ability of recommending different artefacts, like API function calls or code snippets, another idea would be to take into account developers feedback to refine further recommendations, and finally apply the proposed approach to other package managers as it currently is developed for Maven. Another such recommendation system was developed in [114] and they suggest in the future to consider higher order relations in both social and dependency networks.

More work is needed for the integrated model using Spack and SpackDev package manager before it satisfies their requirements [51], however more specific improvement needs are not clarified.

Interactive ASCII dependency graph visualization presented in [70] can be improved in many ways. They intend to improve the graph layout algorithm in the future, improving its readability, plan to account for also the graph attribute data, address the multivariate design issues and experiment with adding interactivity to other Spack analysis commands. Improvements to DepOwl are also needed in the future [64], for example providing an interface for application developers to indicate a fixed version for each library.

Macho et al. [62] proposed BuildMedic to automatically repair dependency-related build breakage. They planned to extend BuildMedic to cover other types of build breakage,

improve its fix time, cover other fixing strategies, like adding dependencies. They also aim to study other projects, specifically projects from industry, and furthermore, they plan to integrate BuildMedic directly into Maven via a plug-in.

Dependency resolution technique proposed in [29] could be improved by indexing more projects in the artefact repository, adding the support of more programming languages and supporting more build systems. In addition, a graphic user interface of DEx developed in the paper may be implemented to enhance user interactions.

An approach called VFramework was proposed in [136] to help with workflow re-executions and future work will focus on identification of the best way of introducing proposed recommendations, as well as applying the framework on further use cases.

Yang et al. [115] proposed a tag recommendation framework for open-source libraries and in the future, they plan to further investigate more useful information of software, like code, design models or git commit messages, and using that to generate tags for open-source libraries.

Bauer et al. [67] presented an automated approach to analyse project's dependence on third-party libraries. They aim to improve the visualization of their approach and evaluate it more thoroughly.

A system that would inform developers of potential vulnerabilities was presented in [86]. Currently, they mined threat intelligence from issues and bugs raised on web-based hosting services for version control, in the future they would like to mine it from other web services. Additionally, other knowledge representation techniques could be used in the future to store threat intelligence.

3.7.3. Future Work Ideas to Be Researched

Most papers have not had follow-up articles done, therefore the future work suggestions presented in those papers are concluded here.

Social Aspects

The adoption of third-party libraries by client software was analysed in [109] and they recommended exploring in the future research, how general the relationship between the actual interdependencies of technologies and their proximity in the technical network may be. In addition, they suggest to investigate the nuances of developer behaviour in greater detail. Palyart et al. [116] studied the social interactions in third-party library use and they

claim, that further research is needed to characterize social interactions and assess their impact within development projects. Another interesting topic would be to predict if a client project will need to interact socially with the library project, and to integrate social network analysis into the investigations of social and technical interactions between projects. Socio-technical relationships of ecosystems are suggested to further investigate in [137] as well.

Licensing

Qiu et al. [83] conducted an empirical study of npm third-party dependency licence related violations and they suggest for future work to extend this study for other open-source ecosystems, carry out new larger-scale developer study and implement tools to help developers with the licensing management. A study of open-source licensing was carried out in [85] as well, and they suggest further research to improve the effectiveness and accuracy of the licence detection method they proposed. They also argue that a more intriguing possibility worth studying would be to investigate protective and permissive licences form closed ecosystems. A case study exploring open-source licence compliance was done in [84] and their study suggests the identified challenges and solutions regarding this compliance can fit into a conceptual model, and in the future it can be investigated, whether this conceptual model can be expanded into a unifying model for open-source dependency management.

Vulnerability Attacks

The analysis done on third-party dependency based attacks in [108] currently has limitations which could be addressed in future work: the size of the applications that could be analysed, dependencies that contain binary code and the use of specific functions. The work done in [91] on software supply chain attacks can also be further expanded by collecting a comprehensive set of existing safeguards and performing a gap analysis, as many safeguards already exist but they aren't used in practice often.

Attack surface is also researched in [95] at the vulnerability level and they propose this metric could be aggregated into higher levels in the future: component level attack surface can be used to evaluate how much risk is brought by each component, package level attack surface can be used to determine which package to use among similar packages, system level attack surface can be used to indicate the overall health of a system. Moreover, presentation tools like attack graph could be used to visualize the risks stemming from third-party dependencies.

Chinthanet et al. [101] investigated the lag between vulnerable release and its fixing release. For future research they suggest to study, how to mitigate these fixing lags and provide strategies for making the most efficient updates via release cycles. Another potential future research avenue is to include a developer survey and developing a tool for managing vulnerability fixing process.

Vulnerable third-party libraries and vulnerability scanners were studied in [104], which highlights the need for further research in this area in order to improve detection algorithms and methods used by existing vulnerability scanners.

Library Updates and Rollbacks

Third-party library updatability was investigated in [38] which inspired also Nguyen et al. [44] in their work. It was suggested to further investigate how library updatability could be improved, such as detecting non-code incompatibilities between library versions, and studying developers' behaviour to best provide them the right support. An approach to detect configuration faults before upgrades was presented in [32] and an idea for future research would be to use this approach for performing checks that are not necessarily faults, for example to understand which services will be deleted or added by each upgrade. It was also suggested to replicate the library update study done in [34] at a larger scale to validate and strengthen its results. Berhe et al. [36] investigated when is it a good time to update a software component and its future work suggestions were to look more at the software component type and extract valuable recommendations and machine learning models to predict optimal release windows.

On the other hand, Suwa et al. [37] focused on library rollbacks. They considered four factors that lead to library rollbacks and in the future they will investigate how to analyse these factors and clarify the influence they have on library selection. Downgrades were studied in [40] as well, they suggest future research to investigate, how the downgrades affect libraries throughout the dependency network. Additionally, further research is needed to investigate why downgrades take a long time to occur, and to which extent the vulnerability advisories influence client developers to downgrade a third-party library.

Trivial Packages

Chen et al. [118] proposed as future work the need for better search tools in order for developers to better find suitable libraries. Experiments are needed to evaluate and improve existing tools and going beyond popularity metrics when ranking packages. Empirical

studies are needed to examine how prevalent duplicated packages really are, evaluate the current npm ecosystem policies and build a tool that automatically combines co-usage trivial packages so that developers could lessen their number of dependencies. Further future research ideas are proposed in [132]. They suggest developing a technique that would detect and evaluate the quality of trivial packages in an ecosystem, as well as when they are used in a project. In addition, as both trivial package articles are written on the npm ecosystem, other package managers could be investigated on this topic.

Library Imitations

Third-party library imitations were investigated in [78] and in the future, they would like to explore the feasibility of path-sensitive approach to improve their proposed solution in terms of precision. Similar study was carried out in [111] and they suggest to further improve library recommendation approaches and investigate state-of-the-art clone detection tools.

A scheme for unused code detection was proposed in [79] and they expect to improve their model with more expansion of merge rules and real user operations will be monitored in order to improve the accuracy of identifying unused functions.

Android Library Detection

An approach to detecting third-party libraries in Android apps was proposed in [72]. They suggested improvements to the proposed LibSift tool: identification of obfuscated third-party libraries and additional information like the type of the library or its maliciousness. As this suggested approach was static analysis, Zhan et al. [68] proposed to also investigate dynamic techniques for third-party library detection. In addition, they suggest investigating how to catch newly emerging third-party libraries and considering also other programming languages other than Java as future work. Zhan et al. [73] added to these future work ideas studying vulnerable third-party libraries used by both free and paid mobile applications, and detecting vulnerable native libraries.

A new library detection method was proposed in [71]. They left as future work to integrate more scalable semantics-based methods into the research, extend the size of their ground truth set and launch a training procedure to determine a reliable threshold that is used in their library detection method. LibHawkeye, a technique for identifying third-party libraries in Android applications developed in [69] left for future work to get rid of the false positives.

As Zhan et al. [17] carried out first systematic literature review on Android third-party library related research, they summarized multiple future research paths. They suggest to

focus on third-party library recommendations for Android, which has already done in traditional research for example in [1] and [113]. Other suggestions include focusing on GUI-related third-party library smell analysis, third-party library updating systems, native library related research, library compatibility analysis, library dynamic features analysis, cross-language library analysis and third-party library related optimizations.

Developers' Perception

The impact of dependency management tools on developers' trusting a third-party library was studied in [43] and they proposed to further investigate the reasons for trust issues and how to certify trust of a recent release, so that in the future, dependency management tools like Maven could ensure the trustworthiness of a third-party library. Kula et al. [12] studied the extent of which developers update their third-party libraries and left for future work to explore the developers' perception on migration effort, specifically investigating the responsibilities regarding the updates. The same kind of study was also done in [14] and they suggest future work to focus on developing techniques for automatic detection of update opportunities. Moreover, they plan to extend the empirical study to include proprietary applications with a focus on the relation between user ratings and third-party library updates.

The library adoption factors for critical IT infrastructures were investigated in [18] and in the future they would like to validate the proposed framework by applying Structure Equation Modelling.

Bogart et al. [117] studied different ecosystems' approaches and practices to breaking changes. They propose that future work should further explore smaller ecosystems with interviews or analysis of artefacts. The practices and values regarding handling breaking changes are context-dependent and thus hard to generalise, therefore a comprehensive theory incorporating such insights is needed. Another interesting future work idea would be to use time series data about developer overlap and historic participation in ecosystems to identify developers who moved to ecosystems with similar or different practices, and see how their behaviour changed.

Library Evaluation

In [120] the authors studied how the stability of a third-party library affects the programs that use it and proposed metrics to measure the stability. For future work they aim at benchmarking those metrics through an analysis of full Maven Central repository. Jezek et

al. [102] conducted a similar experiment, but on a larger set of libraries. More research is needed to explain patterns of how specific changes done to the third-party library are related to their impact on the client program. The same is suggested by Cogo et al. [110] as well, they recommend further research design scalable tools that support the provider libraries in assessing the impact of their code changes on the client system. They also add that more research should be done to define proper ways to determine the release readiness of third-party libraries.

A model to estimate the quality of third-party library components was proposed in [135]. They recommend to investigate whether the impact of human intervention in the proposed model can be lessened as it may lead to fuzziness in the collected data. Korkmaz et al. [133] developed methods to estimate impact measures of third-party libraries and the next step would be to improve the model they proposed by including nodal effects (related to the degree of each node in the network).

In [122] the packages in decline for the npm ecosystem were studied and future work could be carried out to investigate, whether developers find centrality a useful metric when selecting packages. Other ecosystems like PyPI and Maven can also be considered in future work and investigating why packages' centrality is rising or declining is critical. Another interesting idea would be to propose an automated approach to finding future central packages in order to boost their evolution. Finally, when packages in decline are identified, developers could also be assisted in replacing them with alternative package suggestions.

Dependency Solving

Abate et al. [54] summarized the results of a dependency solving competition called Mancoosi International Solver Competition 2010, and their future work proposal was to analyse the structural differences among different problem sets used in that competition to better understand why some solvers performed better than others.

A formalization of deployment dependencies was presented in [63] and they suggested work on two main directions: first, to ensure the guarantee of a deployment, for which the formalization of the properties a deployment system should respect is needed, and second, to extend the proposed system to overcome its current limitations.

Recursive dependency resolution was studied in [59] and as the study was based in static compatibility, this approach could be improved by investigating the use of checked exceptions.

Dependency Visualization

In [123] the evolution of systems was visualized in two ways: in the system-centric perspective and in the library-centric perspective. In the future they aim to enhance their work by including additional interactive and dynamic transitions between those two visualizations. The research in [112] built on Kula et al. [123] and proposed Software Universe Graph to visualize the evolution of software systems. They propose as further work to investigate how applications deal with cross-language interaction; they also aim to improve the accuracy of their model and incorporate more sophisticated techniques and tools. Another interesting avenue for future work would be to trace systems that have abandoned a dependency.

Third-Party Library Data

In [139] the framework for a marketplace of linked third-party library data and artefacts was proposed. In future, they are working on plugins for the implementation to foster awareness and collaboration in software engineering globally. World of Code as a collection of version control data was introduced in [138] and they plan on the platform improvements as well in the future. A vulnerability dataset was collected in [93] of the Maven ecosystem and they propose as future work to observe other package management ecosystems for this kind of data.

Software Ecosystems

A study of software ecosystems was carried out in [128] and they expect to expand their work in two aspects: first, by conducting a systematic mapping of studies to identify and gather metrics used by works on software ecosystems, and second, by proposing a framework for modelling and managing software ecosystems. Software diversity was studied in [124] in the context of Maven and the next step in that research would be to investigate how the natural emergence of software diversity could be amplified.

Harrand et al. [125] performed a study of Maven client-library relationships. They wish to explore novel ways of designing open Java third-party libraries and they want to leverage the existence of a core set of APIs as an instrument to build adapters between APIs that provide similar features.

The CRAN ecosystem was studied in [57] and they would extend their analysis to other, smaller R package sources, such as BioConductor or R-Forge, as currently they mined their data from GitHub and CRAN. They also suggest to investigate, how the use of management

tools is reshaping the R community and affecting the way R packages are maintained. They also would extend their research by taking into account social metadata and other data sources, like mailing lists, issue trackers, statistics, activity on Q&A websites etc. CRAN was studied in [130] as well. They suggest more research to develop specific tools that could support developers to be aware of the best libraries for them, for example identifying libraries with fewer dependencies, more "feature-full" libraries or libraries that are more safe. The awareness of developers could also be studied: what is their approach like regarding dependencies, do they use any quantitative approaches when choosing libraries. Finally, other software ecosystems could be studied from the complex network perspective to compare with CRAN.

Dependencies existing in Gentoo Linux were studied in [126]. For future research, the developer network can be investigated to study the co-evolution between developer and software package networks.

Dependency and Semantic Conflicts

Automated test technique to detect dependency conflict issues for Maven was proposed in [48] and they suggested an alternative mutation strategy for their approach to be investigated in the future. Another dependency conflict work for Java was done in [50] and they proposed the need for a framework to help with resolving dependency conflict issues. Wang et al. [55] was a research focusing on dependency conflicts as well, but they did so in the Python ecosystem as they developed Watchman to monitor dependency conflicts. In the future, they wanted to improve Watchman and generalise their technique to other Python library ecosystems. Python was investigated also in [61] regarding the dependency errors and they highlighted the need to automate the process of extracting error patterns of dependencies. An influential article was written by Wang et al. [52], which was referenced by Wang et al. [48, 55]. It conducted a study on real-world dependency conflict issues and recommended future work to design effective techniques to help developers automatically repair dependency conflict issues. Currently, no specific tools have yet been proposed that would solve the dependency conflict issues, only approaches that will automatically detect dependency conflicts.

In addition to dependency conflicts, semantic conflicts were studied similarly. A test generation technique for triggering semantic conflict issues was presented in [53], which was inspired by Wang et al. [48]. They planned in the future to improve their technique by

combining symbolic execution or fuzzing techniques. Semantics were further studied in [74] and [75]. Decan et al. [74] recommended to carry out a finer-grained analysis to study specific characteristics and details of individual package releases in regards to how they comply to semantic versioning. In addition, more work is needed on how and why breaking changes manifest themselves in individual packages, as well as in entire package dependency graphs, and it may be interesting to expand the analysis beyond the ecosystem boundaries. Dietrich et al. [75] suggested further research to investigate what the technological and social barriers are to wider adaption of semantic versioning.

Library Interactions

Co-installability was researched in [49] and one question left for future work is to investigate, how co-installability evolves along with repository evolution: if a set of co-installable components is still co-installable in a repository that has been updated. Similarly, component interactions were studied in [140] and they proposed an approach to ensure the interactions among components are strictly conformed to stated policies. For future research, they are developing a tool environment that ensures only allowed classes of the components can be accessed during development; this approach will then be applied in large software systems to evaluate its validity.

Other Future Work Ideas

Han et al. [129] analysed third-party libraries related to deep learning and in the future they plan to encompass even more deep learning libraries and incorporate proprietary projects to better generalise their findings. Furthermore, future research could analyse the relationship between direct-transitive dependencies and upgrade-downgrade behaviours.

In order to evaluate third-party library behaviour, Wu et al. [131] proposed usage and dependency model, as well as three metrics based on the model. For future work they left investigating more libraries so that the models and metrics could be validated.

Abate et al. [31] investigated the notion of strong dependencies between software components and they planned future work to identify if the notion of strong dependency can help pinpoint clusters of libraries that should forcibly migrate together. Dependency management using blockchain was described in [56] and they suggest future work to study non-functional characteristics of software properties, such as developer relationships, code styles etc.

The level of technical leverage indicating how much different projects rely on third-party libraries was investigated in [92] for Maven and metrics to quantify it were proposed. They plan to further investigate the impact of those metrics, for example, to see how many dependencies are too many, beyond which maintenance might become unwieldy. Another future work idea would be to investigate the impact of transitive dependencies on technical leverage and also carry out further research on other programming languages and software repositories.

0.y.z. package releases were studied in [134] and they suggest further research in the direction of relying on the development history of a package to assess at a fine level of granularity the 0.y.z releases. The quantitative analysis could also be supported by qualitative analysis to help understand how developers and maintainers perceive the 0.y.z. releases.

RQ8: Future Work summary. Most straightforward suggestion for future work on third-party library analysis would be to carry out already done research on other package management ecosystems in order to facilitate comparison and gain further insight into other programming languages. Quantitative research can be complemented with more qualitative methods by carrying out interviews and surveys to study more of developers' perceptions regarding different aspects of third-party dependency management. Another avenue for future research would be to carry out more studies in the mobile domain as most of the articles focus on traditional software and no paper included analysis specifically on iOS.

4. Discussion

In this chapter, the main findings and already established ways of third-party dependency management studies are discussed. Additionally, conflicting findings and shortcomings are concluded, as well as threats to validity and future work ideas.

4.1. Established Ways of Third-Party Dependency Analysis

Third-party dependencies have been studied from numerous different aspects. Most papers looked at third-party dependency analysis from the client's aka the project developer's point of view: how to maintain the dependencies, handle the security implications and how are the social aspects related to third-party libraries. When incorporating third-party libraries into the software system, the libraries need constant maintenance and upkeep, therefore many papers discussed the aspects related to third-party dependency maintenance, like updating the libraries, solving dependency conflicts and keeping track of the dependencies in software projects. Security vulnerabilities related to third-party libraries was another well-researched side of third-party dependency analysis, as the security of software systems is undoubtedly a critical factor to consider. The social side of third-party library analysis is a more recently emerging topic of research, where the adoption factors, interactions between clients and provider, and developers' behaviour are discussed. This trend could show the progress in the third-party research domain, where researchers do not only study "what" is happening, but also "why" some processes occur. The other, but slightly less-researched side of third-party dependency analysis was the third-party library maintainer's and ecosystem's point of view, where the evolution and evaluation of ecosystems, libraries and dependency networks was investigated. This however should be considered as a valuable research area as well.

The problems the studies handled were in accordance to the research objectives. As it appeared, the most pressing problem as of late has been the security implications of third-party libraries. They can increase the software system's attack surface and insert malicious code into the client, due to which there has been much research into how these vulnerabilities could be detected and mitigated. Automated approaches have mainly been suggested to fix security vulnerabilities, whether in the form of package manager help or a specific vulnerability detection tool. Another impactful problem has been that developers do not manage their third-party dependencies enough and this can be the root problem of many other managerial issues. It has been proved, that developers of the client project often

are not informed enough, do not keep track of the dependencies their system uses and do not prioritize the upkeep effort. This can lead to other already discussed problems, like outdated dependencies, breaking changes or dependency conflicts. This problem could be mitigated with changing the developers' perspective: the maintenance effort should be prioritized, clear guides for the upkeep could be prepared and developers should utilize different automated tools to minimize the manual work.

Most work has been specialized in Maven and npm package managers. This was to be expected, as Java and JavaScript respectively are some of the most used programming languages in the world. These package managers are also seemingly the ones that have been researched first for some new concepts: for example, bloated dependencies in Maven and technical lag in npm. What also makes them popular subject for research, is that there exists plenty of usable data for these package managers in the form of their library repositories and open-source code platforms, like GitHub. A noticeable decline in popularity has been for Debian – this package manager was researched more in the earlier years of third-party dependency analysis research. The reason for this could be that as it was one of the first package managers to be researched, it has already been quite well studied, or the researchers are favouring newer and less-researched systems. It is also possible, however unlikely, that Linux systems are gradually losing their popularity.

As mentioned before, there is an abundance of openly available information and data sources. The most used method for third-party dependency analysis has been quantitative: crawling metadata such as open-source code, third-party library lists, development history or communication information from data sources and repositories, and analysing the data in various ways, such as doing statistical analysis, finding patterns, making generalisations or finding outliers. This has been a good way to study, for example, whole ecosystems as they are data-heavy or create dependency networks. The open-source projects have also been useful in verifying proposed approaches. Therefore, GitHub has been the most used data source for third-party dependency analysis studies. Often, the original source of the third-party libraries is needed and then the library repositories of specific package managers have been useful for researchers. More qualitative approaches have been used to complement existing quantitative research or study topics related to developers' perspective regarding third-party dependency analysis. Then, interviews and surveys have been used as the data sources.

Most new tools that were proposed were for third-party dependency analysis and security. As it is clear developers need more automated support in their third-party dependency maintenance, dependency analysers can help them keep track of existing dependencies, track transitive dependencies, help with updates and conflicts. Security related tools were developed in many cases as well to help developers identify vulnerable libraries and code as well as malicious updates.

4.2. Conflicting Findings

In general, the studies were similar and complemented each other, oftentimes also expanded on previous results. There were no significant conflicting findings or points that would disprove some outcomes. However, some suggestions given in the papers have been brought into question, specifically related to automatic approaches. The use of Dependabot was suggested in [57] to help developers with third-party dependency management. When considering bloated dependencies however, Dependabot often suggests updates for bloated dependencies, which is not actually needed and complicates the update effort [82]. Developers often accept Dependabot update pull requests without considering whether the dependency is used or not. This shows that the proposed tools may not always be reliable or take into account all important aspects of third-party dependency management. However, as the idea of bloated dependencies is a new concept, it is understandable, why the Dependabot approach does not take them into account.

Automatic third-party library updates have been criticised in [29] as well. Even though the automatic approaches are good for developers to lessen the manual work and avoid technical lag, they can also bring upon backward incompatible changes, which need to then be solved manually either way, rendering the automatic approach useless. This shows that even if automatic tools exist, there is no certainty that they will not break anything and therefore knowledgeable developers are still needed.

4.3. Shortcomings

Multiple shortcomings have been identified from the existing papers. Transitive dependencies are a special aspect of third-party dependency management, where it is hard to keep track of them, developers might not know which dependencies the third-party libraries they are using utilize and what exactly might be added to the project through transitive dependencies. There does not however exist a good paper that would concentrate specifically on transitive dependencies. The problem of transitive dependencies has been

touched upon in some papers among other issues, and [129] described problems that can happen when transitive dependencies are resolved automatically, but no study focused on the security implications of transitive dependencies or how developers could better identify such dependencies. In author's opinion, the topic is gaining more and more relevance as software ecosystems are growing bigger and becoming more interconnected, as are the software projects using third-party libraries. Thus it would be good to better understand how transitive dependencies occur, change over time and what exactly are their vulnerabilities. Another interesting avenue would be to identify for different software ecosystems, which libraries are the main libraries the others reference and appear most as the transitive dependencies.

Another shortcoming spotted was that third-party library detection approaches have mainly only been suggested for mobile domain, specifically Android. One paper that was not specifically for Android was [50] that presented a visualization tool for Maven dependencies, however that was mostly meant as an information visualization application. Traditional software development could also benefit from similar approaches that have been presented for Android [94, 114], or even developing an online tool, like was done in [93] for Android.

Security vulnerability problem has been investigated for surprisingly few package managers: Cargo, Bower, Maven, npm, PyPI, RubyGems, Xargo. Considering security aspect of third-party dependency management is crucial to take into account, many other bigger package manager communities, like CRAN, NuGet, Packagist, or also smaller, like Elm, Get, Hex, could benefit from having a vulnerability overview for ecosystems.

Even though the usage of package managers is not as relevant in the C/C++ community, there do exist package managers, such as Conan, vcpkg, Spack, Hunter and Buckaroo. Only Spack was featured in two papers that did not consider the C/C++ ecosystem specifically, and other package managers have not been investigated. The C/C++ ecosystem currently does not have a good paper investigating the third-party usage in their community.

Similarly, iOS has not been researched from the third-party dependency aspect. All mobile domain papers in this thesis considered only Android domain. CocoaPods package manager that exists for the iOS systems was used in [72], however it was one of many package managers used in the study and it was on the breaking changes topic, there does not therefore exist a good study that gives an overview of the iOS third-party library usage specifically.

4.4. Future Work

Future work ideas proposed by the papers were concluded in section 5.7., ideas for future work can also be found from section 6.3., where shortcomings were discussed. The most straightforward approach would be to replicate existing studies for new package managers and programming languages, also continue the research in the mobile domain, especially iOS. A more challenging avenue for future research would be to study the whole periphery of the entire third-party library ecosystem. Currently much is known for specific dependency ecosystems, like for example Maven or npm, but less research has been done to study the third-party library ecosystem as a whole, not depending on a specific programming language. This needs to cross-reference abundant data and up until now this has understandably been considered too complex. However, [74] created a collection of such data called World of Code, which could be utilized in the future in a comprehensive study of third-party libraries and dependencies. Additionally, this literature review could be continued by concentrating on a specific aspect of the third-party dependency analysis topic as a more general overview of the whole area is given in this thesis. For example, studying the security-related papers more thoroughly or focusing more on the social aspects could be interesting.

4.5. Threats to Validity

Most threats to validity come from the methodology used in the study. Some papers might have been excluded from the analysis as they might not have come up with the chosen search strings, or they did not exist in the databases. To mitigate this threat, the search strings were tested to see whether they found papers from the relevant starting set. In addition, two databases were used in the search: EBSCO Discovery Service and Google Scholar, which both aggregated results from numerous well-known academic databases.

Inaccuracies and biases could have occurred during data extraction and data analysis as this was done manually by only the author of the thesis – there were no cross-checks done by a secondary person. This threat was mitigated to some extent by clearly describing and documenting the extraction approach and data synthesis, data extraction table has been provided as well with this thesis to allow referencing.

The exact results of the data search might not be fully replicable due to constant updates to the databases. The thesis includes only a few papers published in 2022 because the search was conducted at the end of 2021, therefore the results might already be slightly outdated,

due to the topic of third-party dependency analysis growing in popularity every year as has been concluded in this paper.

5. Conclusion

The aim of this thesis was to provide an aggregate view of the relevant studies done in the field of third-party dependency analysis. This paper presented a systematic literature review in this domain and created an overview of the contributions of the empirical studies. 125 research articles were gathered and analyzed to find answers to 8 research questions.

The first research question aimed to find out third-party dependency analysis related studies' research objectives. Most studies were done in the third-party dependency maintenance field, many also looked into the security implications and social aspects, while others did their research from ecosystem's point of view.

The second research question identified the data sources used in these studies. Data was gathered from different data repositories, GitHub being the most used open-source platform and package manager's repositories being used often as well.

The third research question identified methods used in the studies. As most of the papers were data-heavy, they mined metadata from previously mentioned data sources to study whole ecosystems and find trends. Qualitative methods like interviews and surveys were used to complement quantitative studies.

The next research questions aimed to find out problems presented in third-party dependency research and their mitigations. Issues related to third-party dependency maintenance and security implications were highlighted the most. To mitigate those vulnerabilities, automated approaches and tool support was suggested, as well as the need to prioritize the upkeep of third-party libraries used in the software systems.

The next research question identified package managers used in the studies. Maven and npm have been the most investigated package managers while Debian is seemingly losing its popularity as a research subject.

The next research question investigated, which new tools have been developed in the scope of these papers. Most novel tools aid with third-party dependency maintenance and security vulnerability detection.

The last research question aimed to find out the future work avenues. Existing research could be continued by carrying it out for other package managers to facilitate comparison. Third-party dependency analysis in the mobile domain is another interesting avenue for

future research, and this thesis could be complemented by another more in-depth literature review in the field of security-related third-party dependency analysis papers.

References

[1] Nguyen, P. T., Rocco, J. D., Ruscio, D. D., Penta, M. D., "CrossRec: Supporting software developers by recommending third-party libraries," *Journal of Systems and Software*, Volume 161, March 2020. DOI: https://doi.org/10.1016/j.jss.2019.110460.

[2] *Package management basics*, MDN Web Docs web page. Available from: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Understanding_client-side_tools/Package_management [Accessed 17th October 2021].

[3] *RPM timeline*, RPM web page. Available from: https://rpm.org/timeline.html [Accessed 17th October 2021].

[4] *Chapter 4. A Detailed History*, Debian web page. Available from: https://www.debian.org/doc/manuals/project-history/detailed.en.html [Accessed 17th October 2021].

[5] Zimmermann, M., Staicu, C.-A., Tenny, C., Pradel, M., "Small World with High Risks: A Study of Security Threats in the npm Ecosystem," *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19)*, August 2019, pp. 995-1010.

[6] Lauinger, T., Chaabane, A., Arshad, S., Robertson, W., Wilson, C., Kirda, E., "Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web," *The Network and Distributed System Security Symposium*, February 2017. DOI: https://doi.org/10.14722/ndss.2017.23414.

[7] *Third-party software component*, Academic Dictionaries and Encyclopedias web page. Available from: https://en-academic.com/dic.nsf/enwiki/204648 [Accessed 17th October 2021]

[8] Szyperski, C. A., *Component Software: Beyond Object-Oriented Programming.* 2nd edition, Addison-Wesley Longman Publishing Co., Inc., 2002.

[9] Li, M., Wang, W., Wang, P., Wang, S., Wu, D., Liu, J., Xue, R., Huo, W., "LibD: Scalable and Precise Third-Party Library Detection in Android Markets," *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, pp. 335-346. DOI: https://doi.org/10.1109/ICSE.2017.38.

[10] Robillard, M., Walker, R., Zimmermann, T., "Recommendation systems for software engineering," *IEEE Software*, Volume 27, Issue 4, July-August 2009, pp. 80–86. DOI: https://doi.org/10.1109/MS.2009.161.

[11] Papadopoulo, A., *Should developers use third-party libraries?* Scalable Path web page. Available from: https://www.scalablepath.com/blog/third-party-libraries/ [Accessed 17th October 2021]

[12] Kula, R. G., Germán, D. M., Ouni, A., Ishio, T., Inoue, K., "Do developers update their library dependencies?" *Empirical Software Engineering*, Volume 23, May 2017, pp. 384-417. DOI: https://doi.org/10.1007/s10664-017-9521-5.

[13] Pashchenko, I., Vu, D.-L., Massacci, F., "A Qualitative Study of Dependency Management and Its Security Implications," *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*, October 2020, pp. 1513–1531. DOI: https://doi.org/10.1145/3372297.3417232.

[14] Salza, P., Palomba, F., Di Nucci, D., Lucia, A., Ferrucci, F., "Third-party libraries in mobile apps: When, how, and why developers update them," *Empirical Software* Engineering, Volume 25, May 2020, pp. 2341–2377. DOI: https://doi.org/10.1007/s10664-019-09754-1.

[15] Abdalkareem, R., Oda, V., Mujahid, S., Shibab, E., "On the impact of using trivial packages: an empirical case study on npm and PyPI," *Empirical Software Engineering*, Volume 25, January 2020, pp. 1168-1204. DOI: https://doi.org/10.1007/s10664-019-09792-9.

[16] Libraries.io web page. Available from: https://libraries.io/ [Accessed 17th October 2021]

[17] Zhan, X., Liu, T., Fan, L., Li, L., Chen, S., Luo, X., Liu, Y., "Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review," *IEEE Transactions on Software Engineering*, 2021. DOI: https://doi.org/10.1109/TSE.2021.3114381.

[18] Umm-e-Laila, F., Najeed Ahmed Khan, S., Asad Arfeen, T., "Framework for Identification of Critical Factors for Open Source Software Adoption Decision in Mission-Critical IT Infrastructure Services," *IETE Journal of Research*, October 2021. DOI: https://doi.org/10.1080/03772063.2021.1994036.

[19] Pashchenko, I., Vu, D. L., Massacci, F., "Preliminary Findings on FOSS Dependencies and Security," *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion),* June 2020, pp. 284-285. DOI: https://doi.org/10.1145/3377812.3390903.

[20] Kitchenham B., Brereton P., "A systematic review of systematic review process research in software engineering," *Information and Software* Technology, Volume 55, Issue 12, December 2013, pp. 2049–2075. DOI: https://doi.org/10.1016/j.infsof.2013.07.010.

[21] Decan, A., Mens, T., Grosjean, P. "An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems," *Empirical Software Engineering,* Volume 24, October 2017, pp. 381-416. https://doi.org/10.1007/s10664-017-9589-y.

[22] Kikas, R., Gousios, G., Dumas, M., Pfahl, D., "Structure and evolution of package dependency networks," *In Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17),* May 2017, pp. 102–112. DOI: https://doi.org/10.1109/MSR.2017.55.

[23] Decan, A., Mens, T., Claes, M., "An Empirical Comparison of Dependency Issues in OSS Packaging Ecosystems," *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER),* February 2017, pp. 2-12. DOI: https://doi.org/10.1109/SANER.2017.7884604.

[24] Li, Q., Song, J., Tan, D., Wang, H., Liu, J., "PDGraph: A Large-Scale Empirical Study on Project Dependency of Security Vulnerabilities," *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN),* June 2021, pp. 161-173. DOI: https://doi.org/10.1109/DSN48987.2021.00031.

[25] Stringer, J., Tahir, A., Blincoe, K., Dietrich, J., "Technical Lag of Dependencies in Major Package Managers," *2020 27th Asia-Pacific Software Engineering Conference (APSEC),* December 2020, pp. 228-237. DOI: https://doi.org/10.1109/APSEC51365.2020.00031.

[26] EBSCO Discovery Service web page. Available from: https://www.ebsco.com/products/ebsco-discovery-service [Accessed 17th October 2021]

[27] Google Scholar web page. Available from: https://scholar.google.com/ [Accessed 17th October 2021]

[28] Saldana, J. M., *The Coding Manual for Qualitative Researchers*. 3rd edition. London: SAGE Publications Ltd, 2015.

[29] Zhang, H., *A comprehensive approach for software dependency resolution*, University of Victoria, Department of Computer Science, Master's Thesis, 2002.

[30] German, D. M., Gonzalez-Barahona, J. M., Robles, G., "A Model to Understand the Building and Running Inter-Dependencies of Software," *14th Working Conference on Reverse Engineering (WCRE 2007)*, October 2007, pp. 140-149. DOI: https://doi.org/10.1109/WCRE.2007.5.

[31] Abate, P., Di Cosmo, R., Boender, J., Zacchiroli, S., "Strong dependencies between software components," *In Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09)*, October 2009, pp. 89–99. DOI: https://doi.org/10.1109/ESEM.2009.5316017.

[32] Di Ruscio, D., Pelliccione, P., "A model-driven approach to detect faults in FOSS systems," *Journal of Software Evolution and Process*, Volume 27, April 2015, pp. 294–318. DOI: https://doi.org/10.1002/smr.1716.

[33] Bavota, G., Canfora, G., Di Penta, M., Oliveto, R., Panichella, S., "How the Apache community upgrades dependencies: an evolutionary study," *Empirical Software Engineering*, Volume 20, October 2015, pp. 1275–1317. DOI: https://doi.org/10.1007/s10664-014-9325-9.

[34] Zapata, R. E., Kula, R. G., Chinthanet, B., Ishio, T., Matsumoto, K., Ihara, A., "Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages," *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, September 2018, pp. 559-563. DOI: https://doi.org/10.1109/ICSME.2018.00067.

[35] Ferreira, G., Jia, L., Sunshine J., Kästner, C., "Containing Malicious Package Updates in npm with a Lightweight Permission System," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, May 2021, pp. 1334-1346. DOI: https://doi.org/10.1109/ICSE43902.2021.00121.

[36] Berhe, S., Maynard, M., Khomh, F., “Software Release Patterns When is it a good time to update a software component?” *Procedia Computer Science*, Volume 170, 2020, pp. 618-625. DOI: https://doi.org/10.1016/j.procs.2020.03.142.

[37] Suwa, H., Ihara, A., Kula, R. G., Fujibayashi, D., Matsumoto, K., “An Analysis of Library Rollbacks: A Case Study of Java Libraries,” *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, December 2017, pp. 63-70. DOI: https://doi.org/10.1109/APSECW.2017.25.

[38] Huang, J., Borges, N., Bugiel, S., Backes, M., “Up-To-Crash: Evaluating Third-Party Library Updatability on Android,” *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, June 2019, pp. 15-30. https://doi.org/10.1109/EuroSP.2019.00012.

[39] Cox, J., Bouwers, E., van Eekelen, M., Visser, J., “Measuring Dependency Freshness in Software Systems,” *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, May 2015, pp. 109-118. DOI: https://doi.org/10.1109/ICSE.2015.140.

[40] Cogo, F. R., Oliva, G. A., Hassan, A. E., “An Empirical Study of Dependency Downgrades in the npm Ecosystem,” *in IEEE Transactions on Software Engineering*, Volume 47, no. 11, November 2021, pp. 2457-2470. DOI: https://doi.org/10.1109/TSE.2019.2952130.

[41] Hejderup, J., Gousios, G., “Can we trust tests to automate dependency updates? A case study of Java Projects,” *Journal of Systems and Software*, Volume 183, January 2022, pp. 1-13. DOI: https://doi.org/10.1016/j.jss.2021.111097.

[42] Mirhosseini, S., Parnin, C., “Can automated pull requests encourage software developers to upgrade out-of-date dependencies?” *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, October 2017, pp. 84-94. DOI: https://doi.org/10.1109/ASE.2017.8115621.

[43] Kula, R. G., German, D. M., Ishio, T., Inoue, K., “Trusting a library: A study of the latency to adopt the latest Maven release,” *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, pp. 520-524. https://doi.org/10.1109/SANER.2015.7081869.

[44] Nguyen, D. C., Derr, E., Backes, M., Bugiel, S., "Up2Dep: Android Tool Support to Fix Insecure Code Dependencies," *Annual Computer Security Applications Conference (ACSAC '20)*, December 2020, pp. 263–276. DOI: https://doi.org/10.1145/3427228.3427658.

[45] Mosteo, A. R., "Alire: a library repository manager for the open source Ada ecosystem," *Ada User Journal*, Volume 39, Number 3, September 2018, pp. 189-196.

[46] Ossher, J., Bajracharya, S., Lopes, C., "Automated dependency resolution for open source software," *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, May 2010, pp. 130-140. DOI: https://doi.org/10.1109/MSR.2010.5463346.

[47] Atzenhofer, T., Plösch, R., "Automatically Adding Missing Libraries to Java Projects to Foster Better Results from Static Analysis," *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, September 2017, pp. 141-146. DOI: https://doi.org/10.1109/SCAM.2017.10.

[48] Wang, Y., Wen, M., Wu, R., Liu, Z., Tan, S. H., Zhu, Z., Yu, H., Cheung, S.-C., "Could I Have a Stack Trace to Examine the Dependency Conflict Issue?" *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019, pp. 572-583. DOI: https://doi.org/10.1109/ICSE.2019.00068.

[49] Vouillon, V., Di Cosmo, R., "On software component co-installability," *ACM Transactions on Software Engineering and Methodology*, Volume 22, October 2013, pp. 1-35. DOI: https://doi.org/10.1145/2522920.2522927.

[50] Wang, Y., Xing, C., Sun, J., Zhang, S., Xuanyuan, S., Zhang, L., "Solving the Dependency Conflict of Java Components: A Comparative Empirical Analysis," *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, May 2020, pp. 109-114. DOI: https://doi.org/10.1109/BigDataSecurity-HPSC-IDS49724.2020.00029.

[51] Green, C., Amundson, J., Garren, L., Gartung, P., Sexton-Kennedy, E., "SpackDev: Multi-Package Development with Spack," *EPJ Web of Conferences*, Volume 245, November 2020. DOI: https://doi.org/10.1051/epjconf/202024505035.

[52] Wang, Y., Wen, M., Liu, Z., Wu, R., Wang, R., Yang, B., Yu, H., Zhu, Z., Cheung, S.-C., "Do the dependency conflicts in my project matter?" *In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*, October 2018, pp. 319–330. DOI: https://doi.org/10.1145/3236024.3236056.

[53] Wang, Y., Wu, R., Wang, C., Wen, M., Liu, Y., Cheung, S.-C., Yu, H., Xu, C., Zhu, Z., "Will Dependency Conflicts Affect My Program's Semantics?," *IEEE Transactions on Software Engineering*, February 2021, DOI: https://doi.org/10.1109/TSE.2021.3057767.

[54] Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S., "Dependency solving: A separate concern in component evolution management," *Journal of Systems and Software*, Volume 85, October 2012, pp. 2228-2240. DOI: https://doi.org/10.1016/j.jss.2012.02.018.

[55] Wang, Y., Wen, M., Liu, Y., Wang, Y., Li, Z., Wang, C., Yu, H., Cheung, S.-C., Xu, C., Zhu, Z., "Watchman: Monitoring Dependency Conflicts for Python Library Ecosystem," *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, May 2020, pp. 125-135. DOI: https://doi.org/10.1145/3377811.3380426.

[56] D'mello, G., González-Vélez, H., "Distributed Software Dependency Management Using Blockchain," *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, February 2019, pp. 132-139. DOI: https://doi.org/10.1109/EMPDP.2019.8671614.

[57] Decan, A., Mens, T., Claes, M., Grosjean, P., "When GitHub Meets CRAN: An Analysis of Inter-Repository Package Dependency Problems," *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2016, pp. 493-504. DOI: https://doi.org/10.1109/SANER.2016.12.

[58] Javan Jafari, A., Costa, D. E., Abdalkareem, R., Shihab, E., Tsantalis N., "Dependency Smells in JavaScript Projects," *in IEEE Transactions on Software Engineering*, August 2021. DOI: https://doi.org/10.1109/TSE.2021.3106247.

[59] Jezek, K., Dietrich, J., "On the Use of Static Analysis to Safeguard Recursive Dependency Resolution," *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, August 2014, pp. 166-173, DOI: https://doi.org/10.1109/SEAA.2014.35.

[60] Abate, P., Di Cosmo, R., Gousios, G., Zacchiroli, S., "Dependency Solving Is Still Hard, but We Are Getting Better at It," *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 547-551. DOI: https://doi.org/10.1109/SANER48275.2020.9054837.

[61] Mukherjee, S., Almanza, A., Rubio-González, C., "Fixing dependency errors for Python build reproducibility," *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, July 2021, pp. 439–451. DOI: https://doi.org/10.1145/3460319.3464797.

[62] Macho, C., McIntosh, S., Pinzger, M., "Automatically repairing dependency-related build breakage," *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 106-117. DOI: https://doi.org/10.1109/SANER.2018.8330201.

[63] Belguidoum, M., Dagnat, F., "Dependency Management in Software Component Deployment," *Electronic Notes Theoretical Computer Science*, Volume 182, June 2007, pp. 17-32. DOI: https://doi.org/10.1016/j.entcs.2006.09.029.

[64] Jia, Z., Li, S., Yu, T., Zeng, C., Xu, E., Liu, X., Wang, J., Liao, X., "DepOwl: Detecting Dependency Bugs to Prevent Compatibility Failures," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, February 2021, pp. 86-98. DOI: https://doi.org/10.1109/ICSE43902.2021.00021.

[65] Ferrarezi, J. C., Neto, M., Dias, D. R. C., Pilastri, A. L., De Paiva Guimarães, M., Ferreira Brega, J. R., "LibViews - An Information Visualization Application for Third-Party Libraries on Software Projects," *2016 20th International Conference Information Visualisation (IV)*, July 2016, pp. 136-140. DOI: https://doi.org/10.1109/IV.2016.43.

[66] Chen, Z., German, D. M., “REM: Visualizing the Ripple Effect on Dependencies Using Metrics of Health,” *2020 Working Conference on Software Visualization (VISSOFT)*, October 2020, pp. 61-71. DOI: https://doi.org/10.1109/VISSOFT51673.2020.00011.

[67] Bauer, V., Heinemann, L., “Understanding API Usage to Support Informed Decision Making in Software Maintenance,” *2012 16th European Conference on Software Maintenance and Reengineering*, March 2012, pp. 435-440. DOI: https://doi.org/10.1109/CSMR.2012.55.

[68] Zhan, X., Tianming, L., Yepang, L., Yang, L., Li, L., Haoyu, W., Xiapu, L., “A Systematic Assessment on Android Third-party Library Detection Tools,” *in IEEE Transactions on Software Engineering*, September 2021. DOI: https://doi.org/10.1109/TSE.2021.3115506.

[69] Zhang, Y., Wang, J., Huang, H., Zhang, Y., Liu, P., “Understanding and Conquering the Difficulties in Identifying Third-party Libraries from Millions of Android Apps,” *in IEEE Transactions on Big Data*, June 2021. DOI: https://doi.org/10.1109/TBDATA.2021.3093244.

[70] Isaacs, K. E., Gamblin, T., “Preserving Command Line Workflow for a Package Management System Using ASCII DAG Visualization,” *in IEEE Transactions on Visualization and Computer Graphics*, Volume 25, no. 9, September 2019, pp. 2804-2820. DOI: https://doi.org/10.1109/TVCG.2018.2859974.

[71] Li, M., Wang, P., Wang, W., Wang, S., Wu, D., Liu, J., Xue, R., Huo, W., Zou, W., “Large-Scale Third-Party Library Detection in Android Markets,” *in IEEE Transactions on Software Engineering*, Volume 46, no. 9, September 2020, pp. 981-1003. DOI: https://doi.org/10.1109/TSE.2018.2872958.

[72] Soh, C., Kuan Tan, H. B., Arnatovich, Y. L., Narayanan, A., Wang, L., “LibSift: Automated Detection of Third-Party Libraries in Android Applications,” *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, December 2016, pp. 41-48. DOI: https://doi.org/10.1109/APSEC.2016.017.

[73] Zhan, X., Fan, L., Chen, S., Wu, F., Liu, T., Luo, X., Liu, Y., "ATVHunter: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, February 2021, pp. 1695-1707. DOI: https://doi.org/10.1109/ICSE43902.2021.00150.

[74] Decan, A., Mens, T., "What Do Package Dependencies Tell Us About Semantic Versioning?" *in IEEE Transactions on Software Engineering*, Volume 47, no. 6, June 2021, pp. 1226-1240. DOI: https://doi.org/10.1109/TSE.2019.2918315.

[75] Dietrich, J., Pearce, D., Stringer, J., Tahir, A., Blincoe, K., "Dependency Versioning in the Wild," *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, March 2019, pp. 349-359. DOI: https://doi.org/10.1109/MSR.2019.00061.

[76] Soto-Valero, C., Harrand, N., Monperrus, M., Baudry, B, "A comprehensive study of bloated dependencies in the Maven ecosystem," *Empirical Software Engineering*, Volume 26, no. 3, May 2021. DOI: https://doi.org/10.1007/s10664-020-09914-8.

[77] Soto-Valero, C., Durieux, T., Baudry, B., "A longitudinal analysis of bloated Java dependencies," *In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*, August 2021, pp. 1021–1031. DOI: https://doi.org/10.1145/3468264.3468589.

[78] Sun, C., Khoo, S., Zhang, S. J., "Graph-based detection of library API imitations," *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, September 2011, pp. 183-192. DOI: https://doi.org/10.1109/ICSM.2011.6080785.

[79] Qiong, G., Li, W., "An Optimization Method of Javascript Redundant Code Elimination based On Hybrid Analysis Technique," *2020 17th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*, December 2020, pp. 300-305. DOI: https://doi.org/10.1109/ICCWAMTIP51612.2020.9317462.

[80] Zerouali, A., Mens, T., Gonzalez-Barahona, J., Decan, A., Constantinou, E., Robles, G., "A formal framework for measuring technical lag in component repositories – and its application to npm," *Journal of Software Evolution and Process*, Volume 31, August 2019. DOI: https://doi.org/10.1002/smr.2157.

[81] Zerouali, A., Constantinou, E., Mens, T., Robles, G., Gonzalez-Barahona, J. M., "An Empirical Analysis of Technical Lag in npm Package Dependencies," *International Conference on Software Reuse (ICSR)*, April 2018. DOI: https://doi.org/10.1007/978-3-319-90421-4_6.

[82] Decan, A., Mens, T., Constantinou, E., "On the Evolution of Technical Lag in the npm Package Dependency Network," *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, June 2018, pp. 404-414. DOI: https://doi.org/10.1109/ICSME.2018.00050.

[83] Qiu, S., German, D. M., Inoue, K., "Empirical Study on Dependency-related License Violation in the JavaScript Package Ecosystem," *Journal of Information Processing*, Volume 29, April 2021, pp. 296-304. DOI: https://doi.org/10.2197/ipsjjip.29.296.

[84] Bauer, A., Harutyunyan, N., Riehle, D., Schwarz, G. D., "Challenges of Tracking and Documenting Open Source Dependencies in Products: A Case Study," *Open Source Systems: 16th IFIP WG 2.13 International Conference, OSS*, May 2020, pp. 25-35. DOI: https://doi.org/10.1007/978-3-030-47240-5_3.

[85] Kechagia, M., Spinellis, D., Androutsellis-Theotokis, S., "Open Source Licensing Across Package Dependencies," *2010 14th Panhellenic Conference on Informatics*, September 2010, pp. 27-32. DOI: https://doi.org/10.1109/PCI.2010.28.

[86] Neil, L., Mittal, S., Joshi, A., "Mining Threat Intelligence about Open-Source Projects and Libraries from Code Repository Issues and Bug Reports," *2018 IEEE International Conference on Intelligence and Security Informatics (ISI)*, November 2018, pp. 7-12. DOI: https://doi.org/10.1109/ISI.2018.8587375.

[87] Decan, A., Mens, T., Constantinou, E., "On the Impact of Security Vulnerabilities in the npm Package Dependency Network," *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, May 2018, pp. 181-191. DOI: https://doi.org/10.1145/3196398.3196401.

[88] Prana, G. A. A., Sharma, A., Shar, L. K., Foo, D., Santosa, A. E., Sharma, A., Lo, D., "Out of sight, out of mind? How vulnerable dependencies affect open-source projects," *Empirical Software Engineering*, Volume 26, July 2021. DOI: https://doi.org/10.1007/s10664-021-09959-3.

[89] Gkortzis, A., Feitosa, D., Spinellis, D., "Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities," *Journal of Systems and Software*, Volume 172, February 2021. DOI: https://doi.org/10.1016/j.jss.2020.110653.

[90] Pashchenko, I., Plate, H., Ponta, S. E., Sabetta, A., Massacci, F., "Vulnerable open source dependencies: counting those that matter," *In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '18)*, October 2018, pp. 1–10. DOI: https://doi.org/10.1145/3239235.3268920.

[91] Ohm, M., Plate, H., Sykosch, A., Meier, M., "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks," *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, June 2020, pp. 23-43. DOI: https://doi.org/10.1007/978-3-030-52683-2_2.

[92] Massacci, F., Pashchenko, I., "Technical Leverage in a Software Ecosystem: Development Opportunities and Security Risks," *Proceedings of the 43rd International Conference on Software Engineering*, May 2021, pp. 1386–1397. DOI: https://doi.org/10.1109/ICSE43902.2021.00125.

[93] Mitropoulos, D., Gousios, G., Papadopoulos, P., Karakoidas, V., Louridas, P., Spinellis, D., "The Vulnerability Dataset of a Large Software Ecosystem," *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, September 2014, pp. 69-74, DOI: https://doi.org/10.1109/BADGERS.2014.8.

[94] Tellnes, J., *Dependencies: No Software is an Island*, The University of Bergen, Department of Informatics, Master's Thesis, 2013.

[95] Zhang, S., Zhang, X., Ou, X., Chen, L., Edwards, N., Jin, J., "Assessing Attack Surface with Component-Based Package Dependency," *Network and System Security*, 2015, pp. 405-417. DOI: https://doi.org/10.1007/978-3-319-25645-0_29.

[96] Alfadel, M., Costa, D. E., Shihab, E., "Empirical Analysis of Security Vulnerabilities in Python Packages," *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2021, pp. 446-457. DOI: https://doi.org/10.1109/SANER50967.2021.00048.

[97] Ponta, S. E., Plate, H., Sabetta, A., "Beyond Metadata: Code-Centric and Usage-Based Analysis of Known Vulnerabilities in Open-Source Software," *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, September 2018, pp. 449-460. DOI: https://doi.org/10.1109/ICSME.2018.00054.

[98] Yongming, Y., Song, H., Cuiyi, F., Chen, L., Chenying, X., "CD3T: Cross-Project Dependency Defect Detection Tool," *International Journal of Performability Engineering*, September 2019, pp. 2329-2337. DOI: https://doi.org/10.23940/ijpe.19.09.p5.23292337.

[99] Ponta, S. E., Plate, H., Sabetta, A., "Detection, assessment and mitigation of vulnerabilities in open source dependencies," *Empirical Software Engineering*, Volume 25, September 2020, pp. 3175–3215. DOI: https://doi.org/10.1007/s10664-020-09830-x.

[100] Staicu, C.-A., Torp, M. T., Schäfer, M., Møller, A., Pradel, M., "Extracting taint specifications for JavaScript libraries," *In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*, June 2020, pp. 198–209. DOI: https://doi.org/10.1145/3377811.3380390.

[101] Chinthanet, B., Kula, R. G., McIntosh, S., Ishio, T., Matsumoto, K., "Lags in the release, adoption, and propagation of npm vulnerability fixes," *Empirical Software Engineering*, Volume 26, March 2021. DOI: https://doi.org/10.1007/s10664-021-09951-x.

[102] Alfadel, M., Costa, D. E., Shihab, E., Mkhallalati, M., "On the Use of Dependabot Security Pull Requests," *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, May 2021, pp. 254-265. DOI: https://doi.org/10.1109/MSR52588.2021.00037.

[103] Imtiaz, N., Thorn, S., Williams, L., "A comparative study of vulnerability reporting by software composition analysis tools," *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Article 5, October 2021, pp. 1–11. DOI: https://doi.org/10.1145/3475716.3475769.

[104] Dann, A., Plate, H., Hermann, B., Ponta, S. E., Bodden, E., "Identifying Challenges for OSS Vulnerability Scanners - A Study & Test Suite," *in IEEE Transactions on Software Engineering*, August 2021, DOI: https://doi.org/10.1109/TSE.2021.3101739.

[105] Pashchenko, I., Plate, H., Ponta, S. E., Sabetta, A., Massacci, F., "Vuln4Real: A Methodology for Counting Actually Vulnerable Dependencies," *in IEEE Transactions on Software Engineering*, September 2020. DOI: https://doi.org/10.1109/TSE.2020.3025443.

[106] Wang, P., Ding, Y., Sun, M., Wang, H., Li, T., Zhou, R., Chen, Z., Jing, Y., "Building and maintaining a third-party library supply chain for productive and secure SGX enclave development," *In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '20)*, June 2020, pp. 100–109. DOI: https://doi.org/10.1145/3377813.3381348.

[107] Hejderup, J. I., *In Dependencies We Trust: How vulnerable are dependencies in software modules?*, Delft University of Technology, Software Engineering Research Group, Department of Software Technology, Faculty EEMCS, Master's Thesis, 2015.

[108] Pfretzschner, B., ben Othmane, L., "Identification of Dependency-based Attacks on Node.js," *In Proceedings of the 12th International Conference on Availability, Reliability and Security (ARES '17)*, August 2017, pp. 1–6. DOI: https://doi.org/10.1145/3098954.3120928.

[109] Ma, Y., Mockus, A., Zaretzki, R., Bradley, R., Bichescu, B., "A Methodology for Analyzing Uptake of Software Technologies Among Developers," *in IEEE Transactions on Software Engineering*, Volume 48, no. 2, February 2022, pp. 485-501. DOI: https://doi.org/10.1109/TSE.2020.2993758.

[110] Cogo, F. R., Oliva, G. A., Bezemer, C.-P., Hassan, A. E., “An empirical study of same-day releases of popular packages in the npm ecosystem,” *Empirical Software Engineering*, Volume 26, September 2021. DOI: https://doi.org/10.1007/s10664-021-09980-6.

[111] Xu, B., An, L., Thung, F., Khomh, F., Lo, D., “Why reinventing the wheels? An empirical study on library reuse and re-implementation,” *Empirical Software Engineering*, Volume 25, January 2020, pp. 755–789. DOI: https://doi.org/10.1007/s10664-019-09771-0.

[112] Kula, R. G., De Roover, C., German, D. M., Ishio, T., Inoue, K., “A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem,” *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 288-299. DOI: https://doi.org/10.1109/SANER.2018.8330217.

[113] Katsuragawa, D., Ihara, A., Kula, R. G., Matsumoto, K., “Maintaining Third-Party Libraries through Domain-Specific Category Recommendations,” *2018 IEEE/ACM 1st International Workshop on Software Health (SoHeal)*, May 2018, pp. 2-9. DOI: https://doi.org/10.1145/3194124.3194129.

[114] Yan, D., Tang, T., Xie, W., Zhang, Y., He, Q., “Session-based Social and Dependency-aware Software Recommendation,” *Applied Soft Computing*, Volume 118, March 2022. DOI: https://doi.org/10.1016/j.asoc.2022.108463.

[115] Yang, L., Wang, L., Hu, Z., Wang, Y., Long, J., “Automatic Tagging for Open Source Software by Utilizing Package Dependency Information,” December *2020 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2020, pp. 137-144. DOI: https://doi.org/10.1109/TASE49443.2020.00027.

[116] Palyart, M., Murphy, G. C., Masrani, V., “A Study of Social Interactions in Open Source Component Use,” *in IEEE Transactions on Software Engineering*, Volume 44, no. 12, December 2018, pp. 1132-1145. DOI: https://doi.org/10.1109/TSE.2017.2756043.

[117] Bogart, C., Kästner, C., Herbsleb, J., Thung, F., “When and How to Make Breaking Changes: Policies and Practices in 18 Open Source Software Ecosystems,” *ACM Transactions on Software Engineering and Methodology*, Volume 30, October 2021, pp. 1-56. DOI: https://doi.org/10.1145/3447245.

[118] Chen, X., Abdalkareem, R., Mujahid, S., Xia, X., “Helping or not helping? Why and how trivial packages impact the npm ecosystem,” *Empirical Software Engineering*, Volume 26, March 2021. DOI: https://doi.org/10.1007/s10664-020-09904-w.

[119] Jezek, K., Dietrich, J., Brada, P., “How Java APIs break - An empirical study,” *Information and Software Technology*, Volume 65, September 2015, pp. 129–146. DOI: https://doi.org/10.1016/j.infsof.2015.02.014.

[120] Raemaekers, S., van Deursen, A., Visser, J., “Measuring software library stability through historical version analysis,” *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, September 2012, pp. 378-387. DOI: https://doi.org/10.1109/ICSM.2012.6405296.

[121] Blanthorn, O. A., Caine, C. M., Navarro-López, E. M., “Evolution of communities of software: using tensor decompositions to compare software ecosystem,” *Applied Network Science 4*, December 2019. DOI: https://doi.org/10.1007/s41109-019-0193-5.

[122] Mujahid, S., Costa, D. E., Abdalkareem, R., Shihab, E., Saied, M. A., Adams, B., “Toward Using Package Centrality Trend to Identify Packages in Decline,” *in IEEE Transactions on Engineering Management*, December 2021, DOI: https://doi.org/10.1109/TEM.2021.3122012.

[123] Kula, R. G., De Roover, C., German, D., Ishio T., Inoue, K., “Visualizing the Evolution of Systems and Their Library Dependencies,” *2014 Second IEEE Working Conference on Software Visualization*, September 2014, pp. 127-136. DOI: https://doi.org/10.1109/VISSOFT.2014.29.

[124] Soto-Valero, C., Benelallam, A., Harrand, N., Barais, O., Baudry, B., “The emergence of software diversity in maven central,” *In Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*, May 2019, pp. 333–343. DOI: https://doi.org/10.1109/MSR.2019.00059.

[125] Harrand, N., Benelallam, A., Soto-Valero, C., Bettega, F., Barais, O., Baudry, B., “API beauty is in the eye of the clients: 2.2 million Maven dependencies reveal the spectrum of client–API usages,” *Journal of Systems and Software*, Volume 184, February 2022. DOI: https://doi.org/10.1016/j.jss.2021.111134.

[126] Zheng, X., Zeng, D. D., Li, H., Wang, F.-Y., "Analyzing open-source software systems as complex networks," *Physica A-statistical Mechanics and Its Applications*, Volume 387, Issue 24 October 2008, pp. 6190-6200. DOI: https://doi.org/10.1016/j.physa.2008.06.050.

[127] Wittern, E., Suter, P., Rajagopalan, S., "A Look at the Dynamics of the JavaScript Package Ecosystem," *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, May 2016, pp. 351-361. DOI: https://doi.org/10.1145/2901739.2901743.

[128] Santana, F. W., Werner, C. M. L., "Towards the Analysis of Software Projects Dependencies: An Exploratory Visual Study of Software Ecosystems," *in Proceedings of the International Workshop on Software Ecosystems*, 2013, pp. 7-18.

[129] Han, J., Deng, S., Lo, D., Zhi, C., Yin, J., Xia, X., "An Empirical Study of the Dependency Networks of Deep Learning Libraries," *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, September 2020, pp. 868-878. DOI: https://doi.org/10.1109/ICSME46990.2020.00116.

[130] Mora-Cantallops, M., Sánchez-Alonso, S., García-Barriocanal, E., "A complex network analysis of the Comprehensive R Archive Network (CRAN) package ecosystem," *Journal of Systems and Software*, Volume 170, December 2020. DOI: https://doi.org/10.1016/j.jss.2020.110744.

[131] Wu, J., Liu, Y. P., Jia, X. X., Liu, C., "Mining Open Source Component Behavior and Performance for Reuse Evaluation," *2008 The 9th International Conference for Young Computer Scientists*, November 2008, pp. 1241-1247. DOI: https://doi.org/10.1109/ICYCS.2008.261.

[132] Chowdhury, M. A. R., Abdalkareem, R., Shihab, E., Adams, B., "On the Untriviality of Trivial Packages: An Empirical Study of npm JavaScript Packages," *in IEEE Transactions on Software Engineering*, March 2021. DOI: https://doi.org/10.1109/TSE.2021.3068901.

[133] Korkmaz, G., Kelling, C., Robbins, C., Keller, S., "Modeling the impact of Python and R packages using dependency and contributor networks," *Social Network Analysis and Mining*, Volume 10, December 2020. DOI: https://doi.org/10.1007/s13278-019-0619-1.

[134] Decan, A., Mens, T., “Lost in Zero Space - An Empirical Comparison of 0.y.z Releases in Software Package Distributions,” *Science of Computer Programming*, Volume 208, August 2021. DOI: https://doi.org/10.1016/j.scico.2021.102656.

[135] Pandey, A. K., Agrawal, C. P., “Analytical Network Process based model to estimate the quality of software components,” *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, February 2014, pp. 678-682, DOI: https://doi.org/10.1109/ICICICT.2014.6781361.

[136] Miksaa, T., Raubera, A., Mina, E., “Identifying impact of software dependencies on replicability of biomedical workflows”, *Journal of Biomedical Informatics*, Volume 64, December 2016, pp. 232-254. DOI: https://doi.org/10.1016/j.jbi.2016.10.011.

[137] Blincoe, K., Harrison, F., Kaur, N., Damian. D., “Reference Coupling: An exploration of inter-project technical dependencies and their characteristics within large software ecosystems,” *Information and Software Technology*, Volume 110, June 2019, pp. 174–189. DOI: https://doi.org/10.1016/j.infsof.2019.03.005.

[138] Ma, Y., Dey, T., Bogart, C., Amreen, S., Valiev, M., Tutko, A., Kennard, D., Zaretzki, R., Mockus, A., “World of code: enabling a research workflow for mining and analyzing the universe of open source VCS data,” *Empirical Software Engineering*, Volume 26, March 2021. DOI: https://doi.org/10.1007/s10664-020-09905-9.

[139] Parreiras, F. S., Groner, G., Schwabe, D., de Freitas Silva, F., “Towards a Marketplace of Open Source Software Data,” *2015 48th Hawaii International Conference on System Sciences*, January 2015, pp. 3651-3660. DOI: https://doi.org/10.1109/HICSS.2015.439.

[140] Zhou, J., Ji, Y., Zhao, D., Liu, J., “Using AOP to ensure component interactions in component-based software,” *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, April 2010, pp. 518-523. DOI: https://doi.org/10.1109/ICCAE.2010.5452043.

[141] Di Cosmo, R., Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S., “Supporting software evolution in component-based FOSS systems,” *Science of Computer Programming*, Volume 76, December 2011, pp. 1144–1160. DOI: https://doi.org/10.1016/j.scico.2010.11.001.

[142] Benelallam, A., Harrand, N., Soto-Valero, C., Baudry, B., Barais, O., "The Maven Dependency Graph: a Temporal Graph-based Representation of Maven Central," *MSR '19: Proceedings of the 16th International Conference on Mining Software Repositories*, May 2019, pp. 344–348. DOI: https://doi.org/10.1109/MSR.2019.00060.

[143] Ouni, A., Kula, R. G., Kessentini, M., Ishio, T., German, D. M., Inoue, K., "Search-based software library recommendation using multi-objective optimization," *Information and Software Technology*, Volume 83, March 2017, pp. 55-75. DOI: https://doi.org/10.1016/j.infsof.2016.11.007.

[144] Saied, M. A., Ouni, A., Sahraoui, H., Kula, R. G., Inoue, K., Lo, D., "Improving reusability of software libraries through usage pattern mining," *Journal of Systems and Software*, Volume 145, November 2018, pp. 164-179. DOI: https://doi.org/10.1016/j.jss.2018.08.032.

[145] Munaiah, N., Kroh, S., Cabrey, C., Nagappan, M., "Curating github for engineered software projects," *Empirical Software Engineering*, Volume 22, Issue 6, December 2017, pp. 3219-3253. DOI: https://doi.org/10.1007/s10664-017-9512-6.

[146] Geiger, F.-X., Malavolta, I., Pascarella, L., Palomba, F., Di Nucci, D., Bacchelli, A., "A graph-based dataset of commit history of real-world android Apps", *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, May 2018, pp. 30–33. DOI: https://doi.org/10.1145/3196398.3196460.

[147] Allix, K., Bissyande, T. F., Klein, J., Le Traon, Y., "Androzoo: Collecting millions of android apps for the research community," *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, May 2016, pp. 468-471. DOI: https://doi.org/10.1145/2901739.2903508.

[148] Backers, M., Bugiel, S., Derr, E., "Reliable Third-Party Library Detection in Android and its Security Applications," *CCS '16: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, October 2016, pp. 356–367. DOI: https://doi.org/10.1145/2976749.2978333.

[149] Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J., "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies," *2010 Asia Pacific Software Engineering Conference*, November 2010, pp. 336-345. DOI: https://doi.org/10.1109/APSEC.2010.46.

[150] Gonzalez-Barahona, J. M., Sherwood, P., Robles, G., Izquierdo, D., "Technical Lag in Software Compilations: Measuring How Outdated a Software Deployment Is," *13th IFIP International Conference on Open Source Systems (OSS)*, May 2017, pp. 182-192. DOI: https://doi.org/10.1007/978-3-319-57735-7_17.

[151] Grinter, R. E., *Understanding Dependencies: A Study of the Coordination Challenges in Software Development*, PhD. Information and Computer Science. University of California, Irvine, 1996.

[152] *OWASP Top Ten*, OWASP web page. Available from: https://owasp.org/www-project-top-ten/ [Accessed 9th March 2022].

Appendix

I. Content providers for EBSCO Discovery Service

ABC CLIO

Academy of Management

ACM - Association for Computing Machinery

ACP Publishing PTY Limited

ADIS International Limited

Administrative Science Quarterly

Advanstar Communications

Alexander Street Press

Allen Press Publishing Services

American Association for the Advancement of Science

American Bar Association

American Counseling Association

American Economic Association

American Institute of Physics

American Management Association International

American Meteorological Society

American Psychiatric Publishing

American Psychological Association

American Society of Civil Engineers

American Statistical Association

American Theological Library Association

Annual Reviews Inc.

Asian Network for Scientific Information

Associated Press DBA Press Association

Baker & Taylor

BASE

B.C. Decker Inc.

Bentham Science Publishers Ltd.

Berghahn Books

Biblical Archaeology Society

BioOne

Blackwell Publishing

Bloomberg (BusinessWeek)

Books 24x7

Brill Academic Publishers

British Library

British Psychological Society

Business Monitor International

Cambridge University Press / Books

Cambridge University Press / Journals

Canadian Medical Association

Chinese University Press

Columbia University Press

Credo Reference

Datamonitor Plc

ebrary

EBSCOhost Database Subscriptions

Edinburgh University Press

Editions Rodopi BV

EDP Sciences

Elsevier journal metadata

Emerald Group Publishing Limited

Expert Reviews

Forbes Inc.

Future Medicine Ltd

Future Science Ltd.

Georg Thieme Verlag Stuttgart

Guilford Publications Inc. / Journals

Guilford Publications Inc. / Books

H.W. Wilson Company

Harvard Business Publishing

Harvard Law School Journals

Haymarket Media

Henry Stewart Publications LLP

Hindawi Publishing Corporation

Human Kinetics Publishers, Inc.

ICON Group International, Inc.

IEEE

Indiana University Press

Ingenta

Intellect Ltd.

International Reading Association

Internet Scientific Publications LLC

IOS Press

JSTOR

John Benjamins Publishing Co.

John Wiley & Sons Ltd

Johns Hopkins University Press

Karger AG

Lavoisier

LexisNexis

Liverpool University Press

M.E. Sharpe Inc. / Books

M.E. Sharpe Inc. / Journals

Maney Publishing

Martinus Nijhoff

Medknow Publications

Mergent

MIT Press

Modern Language Association

Morningstar, Inc.

Multi-Science Publishing Co Ltd

National Bureau of Economic Research

National Communication Association

National Library of Economics (ECONIS)

National Library of Medicine

National Research Bureau

Nature Publishing Group

NetLibrary

NewsBank, Inc.

Newsweek

Open Science Co. LLC

Organisation for Economic Cooperation & Development

Oxford University Press / Books

Oxford University Press / Journals

Pennsylvania State University Press

Plunkett Research, Limited

PR Newswires Association

Public Library of Science

Purdue University Press

Radcliffe Publishing

Readex

Reed Business Information

Remedica Medical Education & Publishing

Research India Publications

Rittenhouse Books

Rogers Publishing

Royal Society

Sage Publications / Books

Sage Publications / Journals

Science Publications

Scientific Research Publishing

SkillSoft

SLACK Incorporated

Springer Science & Business Media B.V. / Books

Springer Science & Business Media B.V. / Journals

Statistics Canada

Taylor & Francis Informa

Thieme Medical Publishing Inc.

Thomas Telford Ltd

Thomson Reuters

Time Inc.

University of Alabama Press

University of Calgary Press
University of Illinois Press
University of Nebraska Press
University of North Carolina Press
University of Pennsylvania Press
University of Queensland Press
University of Toronto Press
University of Wisconsin Press
US News & World Report
VSP International Science Publishers
Wiley-Blackwell
World Bank Publications
World Book, Inc.
World Scientific Publishing Company

II. Example of Grounded Coding

An example of grounded coding for RQ1: Research Objectives is presented to illustrate the process.

1. First article's RQ1 was considered, that is shown in Figure II-1.

A	B	C	D	E	F	G
Id	**doc ref**	**Article Title**	**Author name(s)**	**Publication year**	**Venue**	**Research Objectives (R1)**
1	28	A comprehensive study of bloated dependencies in the Maven ecosystem.	Soto-Valero, César; Harrand, Nicolas; Monperrus, Martin; Baudry, Benoit	2021	Empirical Software Engineering	They proposed a novel, unique, and large scale analysis of bloated dependencies in the context of the Maven package manager. A qualitative assessment of the opinion of developers regarding bloated dependencies.

Figure II-1. Segment of the data extraction table.

2. In a separate document, the code aka category was marked down. As the first article's study aim was to investigate bloated dependencies, then the category of Bloated dependencies was marked down with its article id as such:

Research objectives.

- Bloated dependencies (1)

3. Then the next article's research objective was taken from the data extraction table shown in Figure III-2 and at first it was checked, whether it would classify under an already existing category Bloated dependencies. It was determined, that article no. 2 would not classify under the existing category, and so a new category of Technical lag was marked down with the next article id:

Research objectives.

- Bloated dependencies (1)
- Technical lag (2)

Id	doc ref	Article Title	Author name(s)	Publication year	Venue	Research Objectives (RI)
1	28	A comprehensive study of bloated dependencies in the Maven ecosystem.	Soto-Valero, César; Harrand, Nicolas; Monperrus, Martin; Baudry, Benoit	2021	Empirical Software Engineering	They proposed a novel, unique, and large scale analysis of bloated dependencies in the context of the Maven package manager. A qualitative assessment of the opinion of developers regarding bloated dependencies.
2	29	A formal framework for measuring technical lag in component repositories — and its application to npm.	Zerouali, Ahmed; Mens, Tom; Gonzalez-Barahona, Jesus; Decan, Alexandre; Constantinou, Eleni; Robles, Gregorio	2019	Journal of Software: Evolution and Process	The main objective of this paper is to propose a formal framework for measuring technical lag

Figure II-2. Segment of the data extraction table.

4. This was done with all the articles in the data extraction table. At first it was checked, if the research objective would fit in the existing categories and if not, then a new category was created.

5. When all articles were analysed in the first round, the second run-through was done to ensure the correctness of the categorizations. Resulting list of codes was this:

- Bloated dependencies, library imitations (1; 55; 92; 98)
- Technical lag (2; 39; 106; 117)
- Developers' behaviour, developers' behaviour analysis (3; 41; 45; 46; 56; 82; 112; 113)
- Technology adoption, which libraries are used in the projects and which releases are adopted (3; 8; 19; 22; 30; 46; 120; 121)
- Updating/downgrading the dependencies, how and when components and their dependencies are updated, problems related to that (4; 21; 41; 42; 54; 56; 59; 63; 64; 101; 105, 120; 122) Rollbacks 62, 70
- Social interactions, how and how much developers communicate with each other (5; 45; 51)
- Dependency resolution/ conflict resolution (6; 10; 11; 14; 28; 35; 36; 48; 52; 58; 72; 74; 75; 78; 102; 110; 111; 112; 114; 116, 118; 124)
- Evolution of dependencies/libraries/ecosystems/communities (7; 20; 21; 25; 79; 80; 108; 121)
- Dependency networks (9; 29; 38; 73; 79; 81; 117)
- Detection ja mitigation solutions (12; 13; 16; 18; 22; 30; 50; 54; 69; 77; 99; 107; 109)

- Recommendation systems, helping developers choose suitable third-party libraries and recommending release times (15; 24; 51; 59, 89)
- Licensing (17; 61; 76)
- Detect/visualize dependencies/libraries in projects (23; 33; 43; 66; 67; 84; 86; 87; 125)
- Third-party library evaluation, evaluating their importance (26; 68; 83; 85; 95)
- Third-party library vulnerabilities and how they affect the projects (27; 29; 31; 34; 42; 44; 53; 56; 57; 60; 64; 90; 97; 100; 103; 112; 115; 119; 123)
- How projects are connected (32; 37; 47; 49; 71; 93; 94 (interactions between components); 103)
- Ecosystem evaluation (40; 45; 71; 88; 91; 104; 119)
- Dependency versioning (65; 113)

6. Then the codes aka categories were in turn categorized themselves. This was done by drawing connections between the codes and grouping them in more abstract categories. This resulted in four major categories of research objectives: maintenance, security, social and ecosystem, which were used in this study:

Maintenance:

- Bloated dependencies, library imitations (1; 55; 92; 98)
- Technical lag (2; 39; 106; 117)
- Updating/downgrading the dependencies, how and when components and their dependencies are updated, problems related to that (4; 21; 41; 42; 54; 56; 59; 63; 64; 101; 105, 120; 122) Rollbacks 62, 70
- Dependency resolution/ conflict resolution (6; 10; 11; 14; 28; 35; 36; 48; 52; 58; 72; 74; 75; 78; 102; 110; 111; 112; 114; 116, 118; 124)
- Licensing (17; 61; 76)
- Dependency versioning (65; 113)
- Detect/visualize dependencies/libraries in projects (23; 33; 43; 66; 67; 84; 86; 87; 125)

Security aspects:

- Third-party library vulnerabilities and how they affect the projects (27; 29; 31; 34; 42; 44; 53; 56; 57; 60; 64; 90; 97; 100; 103; 112; 115; 119; 123)
- Detection and mitigation solutions (12; 13; 16; 18; 22; 30; 50; 54; 69; 77; 99; 107; 109)

Social aspects:

- Developers' behaviour, developers' behaviour analysis (3; 41; 45; 46; 56; 82; 112; 113)
- Technology adoption, which libraries are used in the projects and which releases are adopted (3; 8; 19; 22; 30; 46; 120; 121)
- Social interactions, how and how much developers communicate with each other (5; 45; 51)
- Recommendation systems, helping developers choose suitable third-party libraries and recommending release times (15; 24; 51; 59, 89)

Ecosystem:

- Evolution of dependencies/libraries/ecosystems/communities (7; 20; 21; 25; 79; 80; 108; 121)
- Dependency networks (9; 29; 38; 73; 79; 81; 117)
- Third-party library evaluation, evaluating their importance (26; 68; 83; 85; 95)
- How projects are connected (32; 37; 47; 49; 71; 93; 94 (interactions between components); 103)
- Ecosystem evaluation (40; 45; 71; 88; 91; 104; 119)

III. Terminology

The most used terminology and topic-specific concepts are defined and explained to help the reader understand this paper better.

Bloated dependency – libraries that are declared as dependencies for an application, but that are actually not necessary for the application's build or to run it [28]

Breaking change/update – any change in a third-party library, that could cause a fault in a dependent project [72]

Co-installability – refers to whether a set of third-party libraries can or cannot be installed together [55]

Dependency – third-party software that the application depends on [2]

Dependency conflict – this term generally refers to incompatibilities between library versions [41], wrong loaded version [75], version constraint conflicts [99] and other similar issues, that can vary depending on the software development ecosystem

Dependency downgrade – reverting to any older version of the dependency [97]

Dependency freshness – the difference between the used version of a dependency and latest version of a dependency [91]

Dependency rollback – reverting to the previous version of the dependency [89]

Dependency upgrade/update – adopting a newer version of the dependency

Package manager – a system that manages project dependencies [2]

Third-party software library – reusable components developed by someone other than the original vendor and that are not the system's own elements [7]

Third-party library reuse – replacing self-implemented code with a third-party library [73]

Third-party library re-implementation – replacing third-party library with self-implemented code [73]

Transitive dependency – dependency that a third-party library is using, indirect dependency [34]

IV. Licence

Non-exclusive licence to reproduce the thesis and make the thesis public

I, **Liisa Sakerman**,

1. grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, my thesis

   Analysis of Third-Party Dependencies – A Systematic Literature Review,

   supervised by Kristiina Rahkema.

2. I grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in points 1 and 2.

4. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Liisa Sakerman
17/05/2022