

	Article Title	Author name(s)	Publication year	Venue	Research Objectives (R1)	Data Sources (RQ2)	Methods (RQ3)	Problems (RQ4)	Mitigation Techniques (RQ5)	Package Managers (RQ6)	New Tools (RQ7)	Future Works (RQ8)	
1	A comprehensive study of bloated dependencies in the Maven ecosystem.	Soto-Valero, Ccarr, Harrand, Nicolas; Monperus, Martin; Baudy, 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.	Maven Central repository, GitHub api, Maven Dependency Graph	They automatically analysed 9,639 artifacts and their 723,444 dependencies. Second, they performed a user study involving 30 artifacts, for which the code is available as open-source on GitHub and which are actively maintained. For each project, they used DEPCLEAN to generate a POM file without bloated dependencies and submitted the changes as a pull request to the project.	Challenge of dependency management: the existence of bloated dependencies. Bloated dependencies needlessly increase the difficulty of managing and evolving software applications. The major issue with bloated dependencies is that the final deployed binary file includes more code than necessary: an artificially large binary is an issue when the application is sent over the network (e.g., web applications) or it is deployed on small devices (e.g., embedded systems). Bloated dependencies could also embed vulnerable code that can be exploited, while being actually useless for the application. Overall, bloated dependencies needlessly increase the difficulty of managing and evolving software applications.	To overcome this problem, they have developed DEPCLEAN, a tool that performs an automatic analysis of dependency usage in Maven projects.	Maven	A tool called DEPCLEAN to automatically analyze and remove bloated dependencies in Java applications packaged with Maven.	When to analyze bloat? (in every build, in every release, after every POM change), who is responsible for debloat of direct or transitive dependencies? (the lead developers, any external contributor), how to properly manage complex dependency trees to avoid dependency conflicts? These methodological questions are part of the future work to further consolidate DEPCLEAN.	
2	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	libraries.io, GitHub	They empirically analyzed the history of package update practices and technical lag for more than 500K packages with about 4M package releases over a seven-year period. They considered both development and runtime dependencies, and studied both direct and transitive dependencies. They also analyzed the technical lag of external GitHub applications depending on npm packages.	A recent version of an application may be outdated due to depending on components that were not updated to their latest versions. On the other hand, updating to more recent releases of reusable components is not for free, since it might lead to a risk of facing backward incompatible changes. Developers rely on package management tools to automate deployments, specifying which package releases satisfy the needs of their applications. However, these specifications may lead to deploying package releases that are outdated or otherwise undesirable because they do not include bug fixes, security fixes, or new functionality. In contrast, automatically updating to a more recent release may introduce incompatibility issues.	They address this issue by providing a general formal framework for measuring how outdated (i.e., how far away from a certain “ideal” state) is a given component or its deployment. Its goal is to enable developers to decide on a more informed basis whether or not to seize the opportunity of relying on more recent component releases, while taking into account the increased risks that may result from keeping one’s dependencies outdated. Technical lag can be measured in many ways. The mechanism of semantic versioning, that allows component maintainers to attach some semantics to their component releases (e.g., whether it is a major release, minor release or patch), helps in letting them define a metric for that lag, based on the version of the releases	npm	-	The proposed technical lag framework can be made more useful by building specific tools that can be included in continuous integration systems. Findings can be turned into actionable guidelines about npm dependency updates, and with time, be made applicable to other package managers. The results also open the door for more research on technical lag and its measurement in reusable software libraries of other kinds, from Linux-based deployments, to Docker containers, or embedded systems built with reusable components. From a research point of view, their technical lag framework can be used to continue exploring different ways of measuring technical lag, taking into account security vulnerabilities, closed issues, pull requests, etc. More precise lag measurements could involve a dynamic analysis of the actual source code to uncover the presence of breaking changes.	
3	A Methodology for Analyzing Uptake of Software Technologies Among Developers	Yixing Ma; Audris Mockus; Russel Zaretzki; Randy Bradley; Bogdan Bichescu	2022	IEEE Transactions on Software Engineering	They model software technology adoption by developers and provide insights on specific technology attributes that are associated with better visibility among alternative technologies. Thus, their findings have practical value for developers seeking to increase the adoption rate of their products.	WOC-DATA, WOC, METCRAN API, StackExchange, GitHub api	They leverage social contagion theory and statistical modeling to identify, define, and test empirically measures that are likely to affect software adoption. They leverage a large collection of open source version control repositories (containing over 4 billion unique versions) to construct a software dependency chain for a specific set of R language source-code files. They formulate logistic regression models, where developers’ software library choices are modeled, to investigate the combination of technological attributes that drive adoption among competing data frame implementations in the R language: tidy and data.table. To describe each technology, they quantify key project attributes that might affect adoption (e.g., response times to raised issues, overall deployments, number of open defects, knowledge base) and also characteristics of developers making the selection (performance needs, scale, and their social network).	If a particular technology chosen by a developer is later supplanted by another, incompatible technology, the support for the supplanted technology is likely to diminish. Reductions in support for the supplanted technology result in increased effort on the part of the developer to either provide fixes upstream or to create workarounds in their software. The value of the developer’s creation to new downstream projects may diminish in favor of the now more popular alternative technology. As a consequence, both the importance of a developer’s product and their reputation may suffer.	To remedy these two risks, developers must understand how attributes of their software products may be perceived among potential and actual downstream adopters (consumers of the technology), especially in relation to alternatives; competing technologies these adopters may have. It is natural to adopt the position that open source software development should be investigated from a supply chain perspective, which also pertains to distributed decision and supply networks among different stakeholders: model the uptake	CRAN	-	It is important to note that the proximity in the technical network may not always fully reflect the actual interdependencies between the technologies and future work is needed to explore how general this relationship may be. The framework they provided allows future researchers to investigate the nuances of developer behaviour in much greater detail and apply it to other contexts.	
4	A model-driven approach to detect faults in FOSS systems	Davide Di Ruscio; Patrizio Pelliccione	2015	Journal of Software: Evolution and Process	They propose a model-driven approach and supporting tools to prevent specific classes of system configuration faults before performing the real upgrade. They present in detail the fault detector component of Evos, which aims to offer a fully automatic, cost-effective, and pragmatic way of improving the quality of the system.	-	The Evos approach and its failure detector component have been proposed to enhance the predictive ability of meta-installers. The failure detector checks the system configuration by executing a set of queries. Therefore, the failure detector ability to discover errors strongly depends on the list of queries that have been identified.	Current tools are able to predict a very limited set of upgrade faults before deployment, and this leaves a wide range of faults unpredicted. The evolution management of component-based systems is intrinsically difficult and requires techniques, algorithms, and methods, which are both expressive and computationally convenient in order to be used in practice. The numerous implicit or explicit relationships among components cannot be neglected because they can easily lead to unusable and corrupted systems. Current upgrade management tools are only aware of static dependencies that are declared in package metadata. Unfortunately, the upgrade of Linux distributions involves also configuration scripts that are executed during upgrade deployment. These scripts, called maintainer scripts, can seriously influence upgrades, thus, it is not surprising that an apparently innocuous package upgrade can end up with a broken system state	They propose a model-driven approach and supporting tools to prevent specific classes of system configuration faults before performing the real upgrade. Evos (Evolution of free and Open Source Software) 6 is a model-based approach to support the upgrade of FOSS systems.	Debian	Fault detector	The idea and the infrastructure of the fault detector can be reused for performing checks that are not necessarily faults. For instance, they could reuse the approach for understanding which services will be deleted or added by each upgrade.	
5	A Study of Social Interactions in Open Source Component Use	Marc Palyart; Gail C. Murphy; Vaden Masram	2018	IEEE Transactions on Software Engineering	They investigate the social interactions that occur for 5,133 pairs of projects, from two different communities (Java and Ruby) representing user projects that depend on a component project. They consider such questions as how often are there social interactions when a component is used? When do the social interactions occur? And, why do social interactions occur?	GitHub archive	They analyzed 5,133 pairs of projects from GitHub where the first member of the pair is a project using a component(s) developed in the project represented by the second member of the pair.	The use of open source components may also introduce new complexities, such as a need to interact with the developers of the project supplying the component. Developers of the component may also be affected, potentially having to expend effort responding to and supporting the projects that rely upon the component. These human actions are unlikely to be easily automatable; if these interactions are significant in number or effort involved, development projects may need to plan for the interactions.	Developers of the project using the component may need to become involved in the component project and expend their own manual effort to ensure the component has sufficient features or is of sufficient quality for use. Socio-technical congruence theory promotes alignment between development artifacts and communication channels within a project. The results of the study they report in the paper suggest a different theory is needed to guide social and technical interactions between projects. Between projects, a lack of social interaction when a technical dependency is made may indicate that the component is of high-quality (i.e., no defects occurring) and has sufficient features	Maven, RubyGems	-	Future research is needed to characterize these social interactions and assess their impact within software development projects. An interesting avenue for future research is to be able to predict whether a user project will need to interact socially with a component project and the likely extent of those interactions. Also an idea for future research is to integrate social network analysis into the investigations of social and technical interactions between projects.	
6	Alire: a library repository manager for the open source Ada ecosystem	Alejandro R. Mosteo	2018	Ada User Journal	This work presents a working prototype tailored to the Ada compiler available to open source enthusiasts, GNAT.	-	The new tool was built by indexing and tagging code releases in public repositories with a semantic version, which in turn enabled the possibility of dependency resolution and easy upgrades.	The difficulty to be aware of available libraries, obsolescence of code that becomes unmaintained and incompatibilities between versions of a same library, or among different libraries being used simultaneously. The dependency or D.L. hell and one of the most dreaded experiences is ending in a broken configuration during an upgrade. Programmers, however, do not all use the same distribution, nor even the same operating system, since today they can resort to about half a dozen generalist operating systems.	To address those problems, one of the most notable efforts in the open source world are the different Linux distributions. Either based on distribution of source code, like Gentoo, or of binaries, like Debian, these communities have since long dealt with the problem of packaging consistent systems for different architectures. Given the polarizing nature of programming languages it is thus unsurprising that many languages have seen efforts aimed at providing an easy way of distributing libraries for those languages	Alire	alr: that facilitates easy dissemination and reuse of third-party Ada projects	Solving open issues related to alr, like windows port and cross-platform builds.	
7	An empirical comparison of dependency issues in OSS packaging ecosystems	Alexandre Decan; Tom Mens; Maëlck Claes	2017	IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)	This article studies issues related to the evolution of package dependencies in three seemingly similar packaging ecosystems, namely the npm, CRAN and RubyGems package distributions for the JavaScript, R and Ruby programming language, respectively.	CommonJS Package Registry, CRAN, GitHub, RubyGems website	They used different approaches for extracting the historical package metadata for each packaging ecosystem, due to the different tools used by these ecosystems. Package releases were collected for each ecosystem. For each package release, they extracted from its metadata the release date, the release version number and the list of packages on which it depends (from field dependencies for npm, Depends and Imports for CRAN and runtime for RubyGems). Based on all the data, they computed the monthly package dependency graph of each ecosystem from 2007 to 2016.	Dependencies increase the complexity of the packaging ecosystem as a whole because the need to manage these dependencies in presence of multiple package releases. This complexity can be the cause of many maintainability issues and failures (a package may get removed entirely from the ecosystem, a package may become archived because it no longer passes the quality checks or because its developer is no longer available, a package may be updated in backward incompatible ways, and so on). While package tend to have few direct dependencies, they have a much higher number of transitive ones. Dependency constraints can lead to co-installability issues, and can prevent packages to benefit from potentially important dependency updates.	To prevent or reduce the risk of such cascading failures, it is important for package maintainers to be aware of the packages they depend upon. Maintainers are asked to somewhat limit frequent package updates. Maintainers can be helped in the management of dependency updates by tools that monitor dependencies and notify the maintainers when a new release of a package dependency is available, or when an important update needs to be deployed. In order to avoid packages becoming broken due to a dependency update, most ecosystems allow package maintainers to specify dependency constraints on the versions of the packages they depend upon. Ideally, the combination of dependency constraints with semantic versioning should make it easier for package maintainers to manage dependency updates.	npm, CRAN, RubyGems	-	In future work, following the spirit of mixed method research, they aim to complement their current quantitative analysis and comparison of packaging ecosystems with a qualitative analysis. This will be achieved by carrying out surveys with ecosystem developers, but specifically focused on software dependencies issues. Among others, this will increase their understanding of why package are updated (e.g., bug or security fixes, API changes, etc.), and how developers of dependent packages are affected and react to updates. They aim to extend their empirical analysis to other packaging ecosystems, by including ecosystems for other programming languages, as well as by considering package distributions for other domains. They also aim to study mixed source ecosystems (containing a combination of both open and closed software, potentially using a wide variety of software licenses) in order to explore how this affects the problems, challenges and solutions related to dependency issues. Finally, they aim to carry out socio-technical comparisons of the considered ecosystems, in order to understand how the collaboration structure of the community of ecosystem contributors affects the technical package dependency structure and conversely.	
8	An empirical study of same-day releases of popular packages in the npm ecosystem	Filipe R. Cogio; Gustavo A. Oliva; Cor-Paul Bezemer; Ahmed E. Hassan	2021	Empirical Software Engineering	Their objective is to study the same-day releases that are published by popular packages in the npm ecosystem. They design an exploratory study to characterize the type of changes that are introduced in same-day releases, the prevalence of same-day releases in the npm ecosystem, and the adoption of same-day releases by client packages.	npm registry	They mined several pieces of data from the npm registry. Four steps were employed in their data collection. The first step was performed to collect the package json files from npm packages. The second step identified packages for their study. In the third step, they selected the patch releases, identified same-day and prior-to-same-day releases, as well as identified dependencies in which the provider was updated. Finally, the fourth step was to analyze how the deployed files change between adjacent npm releases.	They used different approaches for extracting the historical package metadata for each packaging ecosystem, due to the different tools used by these ecosystems. Package releases were collected for each ecosystem. For each package release, they extracted from its metadata the release date, the release version number and the list of packages on which it depends (from field dependencies for npm, Depends and Imports for CRAN and runtime for RubyGems). Based on all the data, they computed the monthly package dependency graph of each ecosystem from 2007 to 2016.	They mined several pieces of data from the npm registry. Four steps were employed in their data collection. The first step was performed to collect the package json files from npm packages. The second step identified packages for their study. In the third step, they selected the patch releases, identified same-day and prior-to-same-day releases, as well as identified dependencies in which the provider was updated. Finally, the fourth step was to analyze how the deployed files change between adjacent npm releases.	Tools such as Dependabot, which automates dependency management. Developers of tools such as Dependabot should consider explicitly signaling same-day releases to client packages. They encourage popular npm packages to have, in addition to their regular release pipeline, an optimized release pipeline for same-day releases. Provider packages will benefit to timely understand how the implemented changes impact their client packages.	npm	-	Future research should design scalable, ecosystem-ready tools that support provider packages in assessing the impact of their code changes on client packages. Future research to define proper ways to determine release readiness in software ecosystems.
9	An Empirical Study of the Dependency Networks of Deep Learning Libraries	Junxiao Han; Shaiguang Deng; David Lo; Chen Zhi; Jianwei Yin; Xin Xia	2020	IEEE International Conference on Software Maintenance and Evolution (ICSM)	They analyzed the project purposes, application domains, dependency degrees, update behaviors, and update reasons as well as the version distributions of deep learning projects that depend on different deep learning libraries.	GitHub api	They extracted open source projects in GitHub that belong to the “used by” list of TensorFlow, PyTorch, and Theano via the dependency graph provided by GitHub API. They used the GitHub API to extract more information about the deep learning projects: full name, description, readme content, main programming language, number of stars, number of contributors, etc.	Inter-project dependence may increase the risk of maintainability issues and failures. Users cannot directly manage the dependencies of deep learning libraries, which may result in security vulnerabilities or code crashes due to the issues existed in deep learning libraries of dependent projects.	Deep learning users may need to utilize tools such as Software Composition Analysis (SCA) to deal with transitive dependencies of vulnerable versions of deep learning libraries. Developers and researchers can also provide thorough tutorials to help users effectively understand the update changes of their libraries. Users and researchers should pay more attention to the version management and form a good habit, including creating requirement files and managing the library versions. Besides, developers and researchers can also provide automatic tools to help users generate requirement files and record versions automatically.	-	-	In the future, they plan to encompass more deep learning libraries and incorporate proprietary projects to expand the generalization of their results. Moreover, they also encourage further studies to analyze more additional questions and extend their work, e.g., to analyze the relationship between direct/transitive dependencies and upgrade/downgrade behaviors.	

	Article Title	Author name(s)	Publication year	Venue	Research Objectives (R1)	Data Sources (RQ2)	Methods (RQ3)	Problems (RQ4)	Mitigation Techniques (RQ5)	Package Managers (RQ6)	New Tools (RQ7)	Future Works (RQ8)
10	Automated Dependency Resolution for Open Source Software	Joel Ossher; Sushil Bajracharya; Cristina Lopes	2010	IEEE/ACM International Conference on Mining Software Repositories (MSR)	They present a method for automatically resolving dependencies for open source software.	Maven 2 Central Repository	Their approach to automated dependency resolution works by cross-referencing a project's missing type information with a repository of candidate artifacts. There are three main components to this approach. First, a collection of candidate artifacts must be created. Second, there needs to be a method for determining the missing types for a project. Third, there must be an algorithm for selecting artifacts from the candidate repository based on the missing type information.	Simply getting a newly downloaded artifact to build or run in the first place. This is in a large part due to the artifacts themselves using other artifacts, potentially creating a long, and sometimes poorly documented, chain of dependencies. It greatly complicates those wishing to study open source software, as many analyses require declaratively complete programs. While manually resolving dependencies for a few projects is merely tedious, it is completely unfeasible for thousands.	There are a variety of things artifact developers can do to simplify this dependency resolution. For researchers, one approach is automated resolution of missing dependencies in open source software. One approach is to pick a few projects to study and to manually ensure that they are declaratively complete. While effective, this approach is totally non-scalable, and fails to take advantage of the sheer breadth of the open source community. Another possibility is to modify the analyses being performed so as not to require declaratively complete programs. This can be quite successful, but is only possible on a case-by-case basis and necessarily introduces some degree of fuzziness. A related approach is to perform fuzzy type inference on the projects in order to fake declarative completeness. While this eliminates any need to modify the analyses themselves, it still introduces uncertainty.	Maven	They implemented the dependency resolution algorithm as part of Sourcerer, an infrastructure for the large-scale indexing and analysis of open source code	In the near future they plan on creating an Eclipse plugin that developers can use to automatically locate artifacts using their system.
11	Automatically Adding Missing Libraries to Java Projects to Foster Better Results from Static Analysis	Thomas Atenhofer; Reinhold Plösch	2017	IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)	In this paper they presented a tool infrastructure for automatically finding missing libraries for open source projects. The LibLoader application was developed based on the idea of Ossher et al.	Maven 2 Central Repository	LibLoader uses Understand to detect missing artefacts by analysing the source code of a project. This information on missing artefacts is used to identify missing external libraries by searching via REST-API of the Maven 2 Central Repository.	It is important to strive for a correct (proper version) and complete provisioning of all external libraries required by a project. The resolution of these missing libraries is a time-intensive and non-trivial task.	This resolution needs tool support.	Maven	LibLoader, that automatically resolves missing dependencies in open source Java projects	LibLoader will be extended to also search in other library-containing repositories. To make LibLoader more accessible, it will be used as part of a micro service architecture. They will build a platform for controlling and managing tools for the software quality analysis.
12	Beyond Metadata: Code-centric and Usage-based Analysis of Known Vulnerabilities in Open-source Software	Serena Elisa Ponta; Henrik Plate; Antonino Sabetta	2018	IEEE International Conference on Software Maintenance and Evolution (ICSME)	In this paper they present a novel method to detect, assess and mitigate OSS vulnerabilities that improves on state-of-the-art approaches, which commonly depend on metadata to identify vulnerable OSS dependencies.	-	They generalize the vulnerability detection approach by considering fixes independently of the vulnerable libraries; they use static analysis to determine whether vulnerable code is reachable and through which call paths; they combine static and dynamic analysis to overcome their mutual limitations; they define metrics which support the choice of alternative library versions that are not vulnerable, highlighting which options are more likely to minimize the update effort and the risk of incompatibility.	Using OSS components with known vulnerabilities. Merely detecting the inclusion of vulnerable libraries does not cater for the needs of the entire software development life-cycle. Library updates can impact the release schedule, require additional effort, cause system downtime, or introduce new defects.	Establishing effective OSS vulnerability management practices, supported by adequate tools, and tools helping to detect known vulnerable libraries are available nowadays, either as OSS or as commercial products	Maven	Vulas, tool implementing code-centric and usage-based approach that scans Java software.	Extend Vulas to support languages other than Java, they are considering the use of information extracted from the construct bodies. Finally, the problem of systematically linking open-source vulnerability information to the corresponding source code changes (the fix) remains open.
13	CD3T: Cross-Project Dependency Defect Detection Tool	Yongming Yao; Song Huang; Cuiyi Feng; Chen Liu; Chenying Xu	2019	International Journal of Performance Engineering	In this paper, they design a cross-project dependency defect detection system based on Java, called CD3T.	NVD and CVE databases	Software dependency package security checking system with four functions: task control scheduling, file enumeration, dependency analysis and result generation	Third-party code introduced by libraries, dependent libraries, third-party components, references and dependencies will have code defects and vulnerabilities in upstream libraries that will inevitably affect downstream software. Even a software development team with strong security awareness often fails to ensure that there are no security vulnerabilities in the and third-party components they refer to. Most software development teams do not have a strong sense of security or complete security capabilities. So often times, once an unsafe third-party dependency library or component containing defects is used by a software project, the software project is used again. This cross-reference or layer-by-layer reuse is basically doomed to inevitably form a "snowball effect" of security issues.	SDL technology, cross-project software dependency defects as well as corresponding risk management and threat modeling techniques.	Maven	CD3T is a command-line tool that uses a Dependency-Inspect-Core to detect and scan public disclosure vulnerabilities related to project dependencies.	-
14	Could I Have a Stack Trace to Examine the Dependency Conflict Issue?	Ying Wang; Ming Wen; Rongxin Wu; Zhouwei Liu; Shin Hwei Tan; Zhiliang Zhu; Hai Yu; Shing-Chi Cheung	2019	IEEE/ACM International Conference on Software Engineering (ICSE)	They proposed the first automated test generation technique to produce the failure introducing conditions and inputs of DC issues.	GitHub, developers' feedback	RIDDLE, built on top of ASM and EVOSUITE, combines condition mutation, search strategies and condition restoration.	Intensive use of libraries in Java projects brings potential risk of dependency conflicts	Most software management tools such as Maven support the detection of potential DC issues and give warnings accordingly. RIDDLE, an automated approach that generates tests and collects crashing stack traces for projects subject to risk of dependency conflicts.	Maven	RIDDLE, an automated approach that generates tests and collects crashing stack traces for projects subject to risk of dependency conflicts.	An alternative mutation strategy is to perform selective mutation of branch conditions that cannot be solved by EVOSUITE to reduce the cost of condition restoration. As identifying the unsolvable branch conditions reaching to target branches may incur more overhead, they choose to mutate all branch conditions to drive the program executions towards the target branches in one step. They leave the investigation of the efficiency of this alternative as future work.
15	CrossRec: Supporting software developers by recommending third-party libraries	Phuong T. Nguyen; Juri Di Rocco; Davide Di Russo; Massimiliano Di Penta	2020	Journal of Systems and Software	In this paper, they present CrossRec, a recommender system to assist open source software developers in selecting suitable thirdparty libraries.	GitHub, Maven, Dataset from previous papers	CrossRec is a recommender system that encodes the relationships among various OSS artifacts by means of a semantic graph and utilizes a collaborative filtering technique to recommend third-party libraries.	Due to the heterogeneity of resources and their corresponding dependencies, developers need to spend a lot of effort to search for relevant items.	LibRec, LibFinder, and LibCUP are the most advanced techniques for library recommendation to support OSS developers. CrossRec, a novel approach utilizing a collaborative filtering technique to recommend third-party libraries.	Maven	CrossRec, a novel approach utilizing a collaborative filtering technique to recommend third-party libraries.	For future work, they will investigate in detail the importance of Novelty and Sales Diversity in the recommendation outcomes. They are working to equip CrossRec with the ability to recommend different artifacts, such as API function calls and code snippets, as well as to extract background data from various sources, such as Eclipse. A different improvement direction would be to account for developers' feedback, or capture the extent to which a suggestion has been actually integrated into the project, to refine further recommendations. Finally, they plan to apply the proposed approach to other ecosystems based on different languages such as C++ and C#.
16	Detection, assessment and mitigation of vulnerabilities in open source dependencies Empirical Study on Dependency-related License Violation in the JavaScript Package Ecosystem	Serena Elisa Ponta; Henrik Plate; Antonino Sabetta	2020	Empirical Software Engineering	They provide an updated presentation of their approach, including an extended description of the method they use to establish whether an OSS library contains vulnerable code. They propose five criteria to automatically establish whether a given OSS library contains the vulnerable or the fixed code with respect to a given vulnerability.	GitHub	The code-centric and usage-based approach 1) Constructing the license dictionary; 2) Constructing the license compatibility network 3) Constructing the historical meta-data dataset; 4) Constructing the dependency network; 5) Dependency-related license violation detection. Preliminary questionnaire targets the authors of packages detected as having dependency-related license violations.	Vulnerabilities discovered in OSS libraries may affect the dependent applications.	Establishing effective OSS vulnerability management practices, supported by adequate tools, tools helping to detect known vulnerable libraries are available nowadays, either as OSS or as commercial products	Maven, PyPi	Eclipse Steady, (vulas) the open source implementation of their code-centric and usage-based approach is the tool recommended to scan Java software products at SAP.	Finally, the problem of systematically linking open source vulnerability information to the corresponding source code changes (the fix) remains open
17		Shi Qiu; Daniel M. Germar; Katsuro Inoue	2021	Journal of Information Processing	To address the dependency-related license violation issue, in this paper they propose an approach to detect dependency-related license violations of software projects in such OSS ecosystems.	npm registry		Dependency-related license violation if the developers overlook the license of the dependency	When developers reuse OSS, they should pay special attention to open source licenses to prevent potential legal risks.	npm	-	They plan to extend their study on dependency-related license violations to other OSS ecosystems. They also consider a new large-scale developer survey as their future work. They also plan to implement some tools to help developers in maintaining licenses and managing license violations.
18	Extracting Taint Specifications for JavaScript Libraries	Staleu, Cristian-Alexandru; Torp, Martin Toldam; Schafer, Max; Möller, Anders; Pradel, Michael	2020	IEEE/ACM International Conference on Software Engineering (ICSE)	In this work, they propose a technique for automatically extracting taint specifications for JavaScript libraries, based on a dynamic analysis that leverages the existing test suites of the libraries and their available clients in the npm repository. Due to the dynamic nature of JavaScript, mapping observations from dynamic analysis to taint specifications that fit into a static analysis is non-trivial. Their main insight is that this challenge can be addressed by a combination of an access path mechanism that identifies entry and exit points, and the use of membranes around the libraries of interest.	Top-1000 most depended upon packages (GitHub)		Vulnerabilities can have severe consequences to the security of applications, resulting in, e.g., cross-site scripting and command injection attacks. Packages are often not self-contained, but they in turn depend on other npm packages.	There are tools that aggregate known security vulnerabilities in specific versions of individual libraries and report them to the developer directly. Static program analysis.	npm	Taser	Their next step is to extend the implementation with support for more testing frameworks, and then apply the approach in production.
19	Framework for Identification of Critical Factors for Open Source Software Adoption Decision in Mission-Critical IT Infrastructure Services	Umm-e-Laila, F.; Najed Ahmed Khan, S.; Asad Arifeen, T.	2021	IETE Journal of Research	This paper investigates and analyzes OSS adoption factors for "critical IT infrastructure" by conducting a comprehensive review of the relevant literature. Furthermore, this paper proposes a framework that can help the critical IT industry to have increased confidence in OSS.	Springer Link, Science Direct, ACM, Scopus, IEEE Xplore, Survey	Literature review, online survey	OSS adoption risk that IT companies face when integrating OSS components into their solutions is not well understood and probably the most serious mistake to avoid when implementing OSS-based solutions	There is a need to explore OSS adoption barriers in a more systematic way	-	-	In the future, this framework model will also be validated by applying Structure Equation Modelling (SEM)
20	How Java APIs break – An empirical study	Jezek, Kamil; Dietrich, Jens; Brada, Premek	2015	Information and Software Technology	They have studied the extent of the problem in real world programs. They were interested in two aspects: the compatibility of API changes as libraries evolve, and the impact this has on programs using these libraries	Qualitas corpus	This study is based on the qualitas corpus version 20120401. A data set consisting of 109 Java open-source programs and 564 program versions was used from this corpus. They have investigated two types of library dependencies: explicit dependencies to embedded libraries, and dependencies defined by symbolic references in Maven build files that are resolved at build time. They have used JaCC for API analysis, this tool is based on the popular ASM byte code analysis library	System runtime failures due to API changes in libraries that evolve independently. Challenges for maintaining application consistency,	A commonly used solution is to add another layer of constraints to restrict linking. Developers should use package naming to distinguish between public and private parts (e.g., internal packages in many OSGi projects). This allows verification tools to detect the illegal use of private packages. Developers should strive for stability when designing the signatures of API methods because changes are almost always binary incompatible. Unfortunately, there is evidence that most developers are only familiar with rules of source compatibility, and not aware of the subtle differences between binary and source compatibility. Programs should not rely on private JRE packages (sun.* and similar) because these packages might not be available on alternative Java platforms such as Android. Finally, public constants should be used only for values that are never to be changed, otherwise inlining can cause applications to rely on obsolete values. By "never" they do not only refer to "during the execution of the program", but rather "during the entire lifecycle of the program". Only the first part is enforced by the compiler and the JVM. Some relatively minor changes to standard development tools and the Java language could be very effective	Maven	-	It is interesting to see that while most changes that might cause problems are in M2 and MOD, while the category that does cause most problems on actual clients is M1 (and also C2, but this category includes M1). Moreover, problems in MOD had no impact on clients. Further research is needed to explain this pattern.

Id	Article Title	Author name(s)	Publication year	Venue	Research Objectives (R1)	Data Sources (RQ2)	Methods (RQ3)	Problems (RQ4)	Mitigation Techniques (RQ5)	Package Managers (RQ6)	New Tools (RQ7)	Future Works (RQ8)
21	How the Apache Community Upgrades Dependencies: An Evolutionary Study	Bavota, Gabriele; Cunfro, Gerardo; Di Penta, Massimiliano; Oliveto, Rocco; Panchella, Sebastiano	2015	Empirical Software Engineering	In this paper they studied the evolution of dependencies between projects in the Java subset of the Apache ecosystem, consisting of 147 projects, for a period of 14 years, resulting in 1,964 releases. Specifically, they investigated how dependencies between projects evolve over time when the ecosystem grows, what are the product and process factors that can likely trigger dependency upgrades, how developers discuss the needs and risks of such upgrades, and what is the likely impact of upgrades on client projects.	DOAP files, Apache release archives	They analyzed the change history of the 147 Java software systems, in the period of time going from June 1999 to April 2013 resulting in 1,964 releases. They used a crawler and a code analyzer developed in the context of the MARKOS European project. The crawler is able to identify for a given project of interest the list of available releases with their release date as well as its SVN address. This information is extracted by crawling DOAP (Description Of A Project) files available on the Internet. Each DOAP file provides information about a specific open source project and it is released by the project itself. This work has mainly an observational nature, i.e., it aimed at empirically investigating a phenomenon - dependency upgrades in a software ecosystem - from both quantitative and qualitative point-of-view. (exploratory, longitudinal, crawling)	When one project undergoes changes and issues a new release, this may or may not lead other projects to upgrade their dependencies. Some APIs may have changed their interface, or might even be deprecated, which requires the adaptation of its client. A library/component might have changed its license making it legally incompatible with the program using it. All these scenarios suggest that updating a library/component in large ecosystems is a complex and daunting task, which requires to ponder several factors.	The problem can be dealt with update management tools available in many operating systems—e.g., Windows, Linux, MacOS	Maven	-	Future work could start from the observations made in this paper to build recommenders aimed at supporting developers in the complex dependency upgrade decision making activity. Clearly, such recommenders should combine a careful analysis of the aforementioned factors with an analysis of release notes to understand the pros and cons of upgrades. Also, their findings on developers' discussions can represent the first step toward the building of an ontology of concepts that can be used to analyze and categorize discussions. Finally, it is also worthwhile to replicate this study on other ecosystems, possibly also considering factors that were not investigated in this work.
22	Lags in the release, adoption, and propagation of npm vulnerability fixes	Bodin Chinthanet; Raula Gaikovina Kula; Shane McIntosh; Takashi Ishio; Akinori Ihara; Kenichi Matsumoto	2021	Empirical Software Engineering	They conduct an empirical investigation to identify lags that may occur between the vulnerable release and its fixing release from a case study of npm JavaScript ecosystem	GitHub, snyk.io	First, they perform a preliminary study of 231 package-side fixing releases of npm packages on GitHub. Second, they conduct an empirical study of 1,290 package-side fixing release s to analyze their adoption and propagation tendencies throughout a network of 1,553,325 releases of npm packages.	Vulnerabilities in third-party dependencies and whole ecosystems.	To mitigate propagation lags in an ecosystem, they recommend developers and researchers to develop strategies for making the most efficient update via the release cycle, developing better awareness mechanisms for quicker planning of an update, and allocate additional time before updating dependencies	npm	-	For future research on how to mitigate these propagation lags in an ecosystem they suggest that researchers should provide strategies for making the most efficient update via the release cycle. Practitioners also need more awareness of the release cycle to determine the update. Potential future avenues for researchers include a developer survey to a better understanding of the reason for releasing and adopting fixes, a performance improvement plan for highlighting the fixing release tool, a tool for managing and prioritizing vulnerability fixing process.
23	LibViews - An Information Visualization Application for Third-Party Libraries on Software Projects	Juliana Cassiano Ferrarazi; Mario Neto; Diego Roberto Colombo Diaz; André Luiz Pilastri; Marcelo De Paiva Guimarães; José Remo Ferreira Brega	2016	International Conference on Information Visualisation (IV)	The main contribution of this paper is LibViews, an information visualization application that helps developers to understand software libraries and their detailed use in a software project.	Maven	LibViews libraries analysis is based on these metrics, providing library stability measurements on demand, through an interactive and intuitive representation to assist developers on making decisions in the software development and maintenance	These external resources bring some risks to the software project, as impact on decision making during development and maintenance, since software libraries are independent projects being developed at the same time by different developers. Risk of runtime failures due to changes in libraries that evolve independently	Information Visualization (IV) techniques	Maven	LibViews, an information visualization application to create visual representations over libraries metrics and usage on software projects	As future works, LibViews will be enhanced including some features, such as visualization of the classes with errors and their entire code, and which methods were removed from each version and what classes use these methods.
24	Maintaining Third-Party Libraries through Domain-Specific Category Recommendations	Katsuragawa, Daiki; Ihara, Akinori; Raula Gaikovina; Matsumoto, Kenichi	2018	IEEE/ACM International Workshop on Software Health (SoHealth)	In this paper, they propose domain-specific categories (i.e., grouping of libraries that perform similar functionality) in library recommendations that adds in library maintenance.	Maven Repository website, GitHub, dataset from previous papers	They created a major that captures the DSC of library dependencies. They performed two datasets activities of (1) extraction of systems and libraries and (2) mapping the categories to the libraries	Two cited reasons for modern Open Source project failures are the risk of becoming obsolete (i.e., no longer useful) and continued usage of outdated technologies (i.e., to outdated, deprecated or suboptimal technologies, including programming languages, APIs, libraries, frameworks, and so on). Searching for useful libraries is difficult.	To cope with the search and maintenance of libraries, existing research leverages software library recommendation systems	Maven	A library recommendation prototype DSCRec	For future work, they would to investigate other techniques and combining existing techniques such as collaborative filtering to improve their results.
25	Measuring software library stability through historical version analysis	Steven Raemackers; Arie van Deursen; Joost Visser	2012	IEEE International Conference on Software Maintenance (ICSM)	The goal of this paper is to introduce a way to measure interface and implementation stability. They evaluate how stable a set of frequently used third-party libraries is in terms of method removals, implementation change, the ratio of change in old methods to change in new ones and the percentage of new methods in each snapshot.	Apache Common libraries	They analyze historical values of metrics, weighted by the times methods, classes or packages are being used. To collect data on dependencies used in industrial systems they create an infrastructure to extract third-party library dependencies for Java as defined in Maven build files. They apply their metrics to the most frequently used Apache Commons libraries and they give an example of the application of their metrics to a library	If a new version of a library introduces breaking changes, then system developers are either forced to update their system to work with the new version or they must keep using the old version of the library	Designing the correct interface that is stable and backward compatible enough in subsequent releases but also flexible enough to adapt to changing requirements	Maven	-	They aim at benchmarking the proposed metrics, through an analysis of the full Maven Central repository
26	Mining Open Source Component Behavior and Performance for Reuse Evaluation	Ji Wu, Yong Po Liu, Xiao Xia Jia, Chao Liu	2008	International Conference on Young Computer Scientists (ICYCS)	This paper proposes the usage and dependency model to evaluate the OSC behavior from both static and dynamic views; and three metrics based on the model to measure interaction behavior complexity between component and its context.	-	The usage and dependency model is built, the invocation matrix and the mining algorithm are proposed	Because of the lack of effective documentation, it is difficult to get the necessary information to evaluate OSC.	There is several evaluation criteria (framework) proposed toward selecting proper COTS components, the usage and dependency model to evaluate the DSC behavior from both static and dynamic views, and three metrics based on the model to measure interaction behavior complexity between component and its context.	-	-	How to use the models and metrics is not well introduced, and needs more components to validate. In the next step, they plan to select more components to evaluate and compare the measurements.
27	Mining Threat Intelligence about Open-Source Projects and Libraries from Code Repository Issues and Bug Reports	Lorenzo Neil, Sudip Mittal, and Anupam Joshi	2018	IEEE International Conference on Software Intelligence and Security Informatics (ISI)	In this paper, they described their method for mining threat intelligence about open-source projects and libraries from host repository issues and bug reports	GitHub	They mine threat intelligence from issues and bugs raised on web-based hosting service for version control like, GitHub, GitLab, Bitbucket etc. They extract vulnerabilities raised on these platforms and represent them in a security knowledge graph. The knowledge graph then becomes a store for various vulnerabilities and exposures present in various open source projects, products and libraries	The developers link these products and code libraries with little consideration of threats, vulnerabilities, and exposures present in them. The open source libraries linked by a developer can themselves link to other vulnerable libraries and so on	In order to better protect the product in development, it is necessary to create a repository of known vulnerabilities in these open source libraries and projects	-	A system that will inform a developer about potential threats and vulnerabilities	For future work, they would like to mine threat intelligence from other web services such as GitHub, Code Trige, etc. They would also like to investigate other knowledge representation techniques that can be used to store threat intelligence.
28	On Software Component Co-Installability	Vouillon, Jerome Di Cosmo, Roberto	2013	ACM Transactions on Software Engineering and Methodology	This article develops a novel theoretical framework, based on formally certified semantic preserving graph-theoretic transformations, that allows them to associate to each concrete component repository a much smaller one with a simpler structure, that they call strongly flat, with equivalent co-installability properties.	-	They use formally certified semantic preserving graph-theoretic transformations	A fundamental challenge for the maintainability and the scalability of such software systems is the ability to quickly identify the components that can or cannot be installed together	From a maintenance point of view, one needs to identify which components cannot be installed together, in order to check whether these incompatibilities are justified or due to erroneous dependencies.	Debian	coinst tool, that can be used to extract and visualize the co-installability kernel for any GNU/Linux distribution.	For future work, they would like to mine threat intelligence from other web services such as GitHub, Code Trige, etc. They would also like to investigate other knowledge representation techniques that can be used to store threat intelligence.
29	On the Impact of Security Vulnerabilities in the npm Package Dependency Network	Decan, Alexandre; Mens, Tom; Constantinou, Eleni	2018	IEEE/ACM International Conference on Mining Software Repositories (MSR)	This paper provides an empirical study of the propagation of security vulnerabilities and their fixes in the package release history of the npm distribution of Node.js packages.	snyk.io, libraries.io	Their approach aimed at identifying packages releases affected by a vulnerability in a precise way by considering the version constraints specifying the security vulnerability, the version constraints specified for package dependencies as well as the exact release that is selected for installation depending on when the installation is performed	Security vulnerabilities are among the most pressing problems in open source software package libraries. It may take a long time to discover and fix vulnerabilities in packages. In addition, vulnerabilities may propagate to dependent packages, making them vulnerable too	Higher awareness among npm package maintainers of the risks incurred by security vulnerabilities, not only at the level of individual packages, but also at a wider ecosystem level by considering package dependencies. Package maintainers should also rely on better use of policies and automated tools to detect and fix vulnerabilities faster, and to reduce the impact of vulnerabilities on dependent packages.	npm	-	They aim to replicate their analysis on other open source package dependency networks. An alternative way to study vulnerabilities and their impact would be to rely on their CWE type
30	On the Use of Dependabot Security Pull Requests	Mahmoud Alfallad, Diego Elias Costa, Emad Shihab, Mouaffak Mikhailali	2021	IEEE/ACM International Conference on Mining Software Repositories (MSR)	Their main goal is to understand the degree to which developers adopt Dependabot security PRs that tackle dependency vulnerabilities in open source projects	GitHub	They perform an empirical study involving data from 15,243 Dependabot security PRs that belong to 2,904 active open-source JavaScript projects from GitHub. They examine how often Dependabot security PRs are accepted (merged) and how long it takes to merge them, in order to determine to what extent developers of open-source projects adopt and respond to Dependabot security PRs.	The use of dependencies increases the impact of security vulnerabilities. A security vulnerability in a highly used dependency may directly impact hundreds of applications, leading to significant financial costs and reputation loss	The open source community has taken active measures to deal with security vulnerabilities in dependencies. For example, Dependabot.	npm	-	Dependabot needs to properly handle peer dependencies. Dependabot needs to be more efficient for projects with a high number of dependencies. Dependabot needs to prioritize security updates by more fine-grained analysis of dependency vulnerabilities.
31	Out of sight, out of mind? How vulnerable dependencies affect open-source projects	Gede Artha Azriadi Prana; Abhishek Sharma; Lewin Khin Shar; Darius Foo; Andrew E. Santosa; Emad Shihab; Abhishek Sharma; David Lo	2021	Empirical Software Engineering	In this work, they analyze vulnerabilities in open-source libraries used by 450 software projects written in Java, Python, and Ruby. Their goal is to examine types, distribution, severity, and persistence of the vulnerabilities, along with relationships between their prevalence and project as well as commit attributes.	GitHub, Veracode Software Composition Analysis, dataset from previous papers	In this work, they used Veracode Software Composition Analysis (SCA) tool to perform an empirical study on a sample of projects and their associated commits on GitHub. Their data was obtained by scanning versions of the sample projects after each commit made between November 1, 2017 and October 31, 2018 using an industrial software composition analysis tool, which provides information such as library names and versions, dependency types (direct or transitive), and known vulnerabilities.	Third-party libraries may contain varying amount of security vulnerabilities. Developers often do not review the security of third-party libraries, analysis on a software project's entire dependency tree can become very complex. Unchecked dependencies and other potential issues such as outdated dependencies and license issues.	Open-source tools such as OWASP Dependency Check, Bundler-audit, and RetireJS that can check for publicly-known security vulnerabilities in open-source dependencies. GitHub provides a service that scans dependencies of a given project in several supported languages for publicly known vulnerabilities. Sonatype, Synopsys, Veracode, and WhiteSource offer software composition analysis (SCA) tools that can identify open-source libraries used in a given software project, and vulnerabilities associated with these libraries (including those not yet in public vulnerability databases), associated licenses, and other metrics. SCA tools enable development teams to identify vulnerable dependencies and other potential issues such as outdated dependencies and license issues.	Maven, pypi, RubyGems	-	A potential direction of future work is expansion of the scale of the study to cover projects written in other programming languages supported by Veracode SCA, as well as investigation of commits from longer time period. Beyond this, future work lies in investigation into associations between dependency vulnerability types as well as the factors that promote or mitigate them. Another element of potential future work related to their study is the identification of characteristics of projects with track record of resolving dependency vulnerabilities quickly and how the characteristics can be emulated in other projects. In addition, they are also interested in investigating the techniques to automatically identify and update vulnerable dependencies in project codebase.
32	Reference Coupling: An exploration of inter-project technical dependencies and their characteristics within large software ecosystems.	Kelly Blincoe; Francis Harrison; Naypreet Kaur; Daniela Damian	2019	Information and Software Technology	In this paper, they described Reference Coupling, a new method that uses solely the information in developers online interactions to detect technical dependencies between projects.	GitHub, GitLab, BitBucket, SourceForge, Jira, IBM Rational solution for Collaborative Lifecycle Management (CLM)	They validate their method on two datasets — one from open-source projects hosted on GitHub and one commercial dataset of IBM projects. They manually analyze the identified dependencies, categorize them, and compare them to dependencies specified by the development team. They examine the types of projects involved in the identified ecosystems, the structure of the identified ecosystems, and how the ecosystems structure compares with the social behavior of project contributors and owners.	Projects depend on one another, and yet awareness of such dependencies is not trivial. Existing static dependency analysis approaches do not identify dependencies across projects.	Identifying technical dependencies to external projects within the ecosystem is important. Methods for extracting external dependencies from a project's source code or configuration files have been proposed.	RubyGems, Homebrew, npm	-	Their Reference Coupling method opens the door for future research in software ecosystems including studying the socio-technical relationships, evolution, health and success of ecosystems.
33	REM: Visualizing the Ripple Effect on Dependencies Using Metrics of Health	Zhe Chen, Daniel M. German	2020	IEEE Working Conference on Software Visualization (VISSOFT)	In this work, they propose the Ripple Effect of Metrics (REM) dependency graphs, a visualization of dependency graphs that leverages metrics of the health of dependencies.	The npm replicate registry, GitHub	Their work has two variants of the visualization: a full dependency graph and a filtered dependency graph. The visualization of REM is implemented using a thirdparty Python graphing library, Plotly	Dependency graphs are growing in size and complexity. One of the current challenges in software development is that it is difficult to understand the dependencies of a given application. Transitive dependencies can break any software application	Project developers need tools, methods and visualizations to inspect the health of these transitive dependencies and their potential impact	npm	REM-Dependabot	For future work, one significant change they plan to add to their current implementation is to include the consideration of dependency version constraint. The second future work is 1) to develop a mechanism that interactively compares different metrics of health on the same REM dependency graph, 2) to have an interactive way to allow user view ripple effect of different REM metrics, and 3) to explore the possibility of complementing the REM graph with real-time data instead of the collected data that they used in this work which can quickly age over time.
34	Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities.	Antonios Gkourtzis; Daniel Feitoso; Dionidis Spinellis	2021	Journal of Systems and Software	They further examine the association between software reuse and security threats	GitHub Activity Data database, GitHub	They looked into the most popular Java projects in the GitHub Activity Data database and constructed a dataset with 1244 projects, containing information regarding the size of both native and reused code, as well as vulnerability information obtained from the static analyzer SpotBugs and the owasp Dependency-Check tool	Security risks, developers not aware of the risks.	Integration service.	Maven	-	It is desirable to investigate other programming languages, automated build systems and package managers. Future studies could explore more indepth research questions related to, for example, features that could cluster similar projects in terms of size, also including a qualitative analysis to explain each cluster. On the other hand, the toolkit reported in paper could be implemented as a workbench that could benefit practitioners and researchers alike by fostering in-house analyses or future studies.

Id	Article Title	Author name(s)	Publication year	Venue	Research Objectives (R1)	Data Sources (RQ2)	Methods (RQ3)	Problems (RQ4)	Mitigation Techniques (RQ5)	Package Managers (RQ6)	New Tools (RQ7)	Future Works (RQ8)
35	Solving the Dependency Conflict of Java Components: A Comparative Empirical Analysis	Wang, Yongzhi; Xing, Chengli; Sun, Jinan; Zhang, Shikun; Xuanyuan, Sisi; Zhang	2020	IEEE Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing (HPSC), IEEE Intl Conference on Intelligent Data and Security (IDS)	This article discusses several common methods for resolving dependency conflicts, explains their principles, summarizes their advantages and disadvantages, and puts forward comments for empirical solution	-	Discussion with working examples	Dependency conflicts	Load the one you really need in the dependency tree. Upgrade or downgrade component to find a compatible version. Shading the conflicting versions. Customize class loader.	Maven	-	Framework that is friendly to novices and does not require human changes.
36	SpackDev: Multi-Package Development with Spack	Chris Green; James Amandson; Lynn Garren; Patrick Gartung; Elizabeth Sexton Kennedy	2020	EPI Web of Conferences	They describe SpackDev's features and development over the last two years, initial experience using SpackDev in the context of the LArSoft liquid argon detector toolkit, and work remaining before it can be considered a fully-functional multi-package build system for HEP experiments utilizing Spack.	Spack	Building a new tool	Managing builds is difficult with external dependencies. An application consists of many parts of different origins and organizational responsibilities.	A new tool to help with the managing.	Spack	SpackDev is a system to facilitate the simultaneous development of interconnected sets of packages. Intended to handle packages without restriction to one internal build system, SpackDev is integrated with Spack as a command-extension in order to leverage features such as dependencies calculations and build system configuration, and is generally applicable outside HEP	More work is required before an integrated model using Spack and SpackDev satisfies all their requirements for development, packaging, and distribution of experimental software.
37	Strong dependencies between software components	Pietro Abate; Jaap Boender; Roberto Di Cosmo; Stefano Zacchicholi	2009	ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)	An empirical study of strong dependencies and sensitivity is presented, in the context of one of the largest, freely available, component-based system.	Debian GNU/Linux	Debian GNU/Linux case study, theoretical proofs, longitudinal study	Inter-package relationships can get quite complex, maintenance of large repository ecosystems is difficult	Release managers, use dependency graphs for risk assessment	Debian	-	Future works is planned in various directions. First of all the notion of installation impact set needs to be refined. They believe that the notion of strong dependency and the clusters entailed by strong dominance can help in identifying clusters of packages which should forcibly migrate together. The dependency management process should be studied also qualitatively to understand issues developers are facing. Second, based on the vulnerability measures and network aspects, a measure quantifying dependency health should be developed. Combining network information with data about testing efforts, code analysis, the number of maintainers etc, into an aggregated dependency health measure. The broad level goal of the future research is to support developers with tools in dependency management and maintenance and provide analytics for package maintainers about their packages and the overall ecosystem trends. Their next goal is to turn the code used in this paper into a set of reusable tools to analyze any package ecosystem based on GitHub and repository data.
38	Structure and Evolution of Package Dependency Networks	Kikas, R., Gousios, G., Dumas, M., Pfahli, D.	2017	IEEE/ACM International Conference on Mining Software Repositories (MSR)	The goal of this work is to study the state of current dependency networks, to understand their characteristics, and to reason about their future evolution.	npm api, rubygems api GitHub, GHTorrent	They compose a network of projects based on dependency relations to understand how the dependency network evolves and how susceptible it is to a removal of a random project.	Introducing third-party libraries makes a project dependent on them. Dependencies need to be kept up-to-date to prevent exposure to vulnerabilities and bugs. Bugs can also originate through transitive dependencies. Developers might not have an overview of all the transitive dependencies as they did not include them themselves. Updating dependencies also entails risks, as new versions may break existing functionality or API correctness.	Dependency management practices	npm, RubyGems, Cargo	-	
39	Technical Lag of Dependencies in Major Package Managers	Stringer, Jacob; Tahiri, Amjad; Blincoe, Kelly; Dietrich, Jens.	2020	Asia-Pacific Software Engineering Conference (APSEC)	This study provides a large scale analysis of technical lag across the major package managers currently in use.	libraries.io	They conducted a mixed-methods study using open-source project data obtained from 14 package managers using the libraries.io dataset.	Dependencies usually evolve simultaneously with a project, so they can become outdated if the version requested by the program does not get updated regularly to the latest available version. Keeping dependencies up to date induces a significant overhead on a developer's time, multiple versions of a dependency can be used by different submodules of a project. When manually updating dependencies, they are often not updated immediately, resulting in technical lag	It is possible for package managers to automatically update dependencies, based on the industry-led Semantic Versioning initiative	Atom, Cargo, Dub, Elm, Hazelib, Hex, Maven, npm, NuGet, Packagist, Pub, Puppet, PyPI, RubyGems	-	
40	The Emergence of Software Diversity in Maven Central	César Soto-Valero, Amine Benellam, Nicolas Harnad, Olivier Barais, and Benoit Baudry	2019	IEEE/ACM International Conference on Mining Software Repositories (MSR)	In this work, they analyze software artifacts from the perspective of one essential characteristic enforced by Maven Central: immutability	Maven Dependency Graph	They measure activity, popularity and timeliness of a subset of 73, 653 libraries by multiple versions, which represents 61.81% of the total number of artifacts in Maven Central. They empirically investigate whether the diversity of library versions is a valuable design choice.	The redundancy in multiple versions can introduce conflicts among dependencies. The projects that depend on a library need to explicitly update their dependency descriptions in order to benefit from the update	The Maven dependency management system	Maven	-	<p>Their next step is to investigate how they can amplify this natural emergence of software diversity through dependency transformations at the source code level.</p> <p>These findings and implications represent the main input for their future research agenda, mainly focused on designing and developing new techniques and tools able to automatically identify opportunities of version change, and apply them flawlessly. Moreover, they plan to extend the empirical study to proprietary and larger applications, with a particular focus on the relationship between user ratings and third-party library updates. Finally, they plan to investigate the impact of the developers' behavior looking in particular at security vulnerabilities, as already done in the traditional context</p>
41	Third-party libraries in mobile apps. When, how, and why developers update them.	Pasquale Salza; Fabio Palomba; Dario Di Nucci; Andrea De Lucia; Filomena Ferrucci	2020	Empirical Software Engineering	The bridge the gap by investigating when, how, and why mobile developers update third-party libraries.	F-Droid repository, ANDROIDTIMEMAC HINE, Github, the Maven repository, survey	They mined the evolution history of 2752 open-source applications to study the problem. They surveyed 73 mobile developers in order to collect their opinions on third-party libraries updates, and particularly on the motivations behind their decisions to update or not.	The aspects related to how the changes made to libraries, developers don't update the dependencies.	Automatic support for the updates, Prioritizing update effort	Maven	-	
42	Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages	Rodrigo Elizalde Zapata; Raula Gaikovina Kula; Bodin Chhinhanet; Takashi Ishio; Kenichi Matsumoto; Akimori Ihara	2018	IEEE International Conference on Software Maintenance and Evolution (ICSMSE)	To analyze the impact of safe clients, they explore how clean and used client projects react to vulnerability library updates.	snky.io, GitHub	They performed an exploratory study at the function level and access of the library Application Programming Interface (i.e., API). They manually identified and validated vulnerability fixes and how they affected the client code. In an empirical study of npm projects and their dependencies, they manually examined a total of 60 projects investigating three cases of high priority vulnerabilities to understand how safe and unsafe projects handle migration to safer dependency versions	Client developers are not practicing library migrations highlighting the perils and effect such as technical lag on the project, with rippling effect to the ecosystem. Client developers struggle keeping up with updates, stating that either they were unaware of the opportunity, or that the cost benefit to update was a demotivating factor. Relying on third-party APIs entails losing control over a part of a software system's functionality. APIs keep evolving, introducing new functionality or providing bug fixes. Depending on the amount of changes, e.g., in case of a major new release of an API, backward-compatibility might not be guaranteed. An external API might not yet be completely mature. Thus, it could introduce bugs into the current software project, which might be difficult to find and hard to fix. The provider of an API might decide to discontinue its support, such that maintainers can no longer rely on them for new functionality and bug fixes. The license of a library or a project might change, making it impossible to continue the use of a particular API.	Automatic approaches	npm	-	<p>Immedate future work is a more rigorous replication of this study at a larger scale to validate and strengthen results.</p>
43	Understanding API Usage to Support Informed Decision Making in Software Maintenance	Veronika Bauer; Lars Heinemann	2012	European Conference on Software Maintenance and Reengineering	They present an approach that automatically extracts information about library usage from the source code of a project and visualizes it to support decision making during software maintenance.	-	Their approach automatically analyzes software systems with the goal of determining the degree of dependence to its included libraries. They statically analyze the code to determine the dependencies and use the extracted information to produce a visualization to gain a quick overview of the library dependencies.	Client developers are not practicing library migrations highlighting the perils and effect such as technical lag on the project, with rippling effect to the ecosystem. Client developers struggle keeping up with updates, stating that either they were unaware of the opportunity, or that the cost benefit to update was a demotivating factor. Relying on third-party APIs entails losing control over a part of a software system's functionality. APIs keep evolving, introducing new functionality or providing bug fixes. Depending on the amount of changes, e.g., in case of a major new release of an API, backward-compatibility might not be guaranteed. An external API might not yet be completely mature. Thus, it could introduce bugs into the current software project, which might be difficult to find and hard to fix. The provider of an API might decide to discontinue its support, such that maintainers can no longer rely on them for new functionality and bug fixes. The license of a library or a project might change, making it impossible to continue the use of a particular API.	Tool support is needed	-	They present an automated approach to analyze the dependencies of software projects on external APIs. The approach is implemented in Java on top of the open source software quality assessment toolkit ConQAT	<p>To complete the drill-down on the visualization side, the next steps are to integrate the source code with the results in a way that dependencies are directly highlighted where they occur. Furthermore, they are planning to evaluate their approach more thoroughly. They envision extending the presented approach to provide further support for API evolution scenarios. They are planning to employ their approach in the context of a wider spectrum of quality analyses and risk assessments.</p>
44	Vulnerable open source dependencies: Counting those that matter	Ivan Paschenko; Henrik Plate; Serena Elisa Ponta; Serena Elisa Ponta; Fabio Massacci	2018	ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)	Their paper addresses the over-inflation problem of academic and industrial approaches for reporting vulnerable dependencies in OSS software, and therefore, caters to the needs of industrial practice for correct allocation of development and audit resources	Maven Central repository	Failure to distinguish own vs third-party dependencies may incorrectly present as an insecure ecosystem with several vulnerable dependencies ("dependency hell") what in reality is just a project that has broken its components into several libraries and did not fix its own vulnerable code. Some transitive dependencies may actually be controlled directly from the project under analysis.	If an outdated direct dependency is affected by a known vulnerability, the simplest solution to mitigate this vulnerability is to update the dependent library to use the fixed version of the dependency	Package users may invest in regular review to keep up with changes, collaborate with upstream projects to minimize the impact of those changes, decline to update to the latest versions (at the risk of missing bug fixes or security updates), or replicate functionality to avoid dependencies in the first place. Package maintainers can refrain from performing changes, announce and clearly label breaking changes, or help their users to migrate from old to new versions.	Maven	A tool that leverages the functionality of Apache Maven to extract the library dependencies and applies code-level matching approach to identify the known vulnerabilities affecting them	<p>As future directions of their research they take the following steps: • to investigate the situation outside of the Maven ecosystem, for example targeting npm or pip. • to extend this study to analyze all the existing libraries in Maven Central. • to identify a precise model for automatic identification of whether a certain library is halted. • to complement the existing studies on the reasons why developers do not update dependencies with an investigation of developers' behavior with regard to security-related updates.</p>
45	When and How to Make Breaking Changes: Policies and Practices in 18 Open Source Software Ecosystems	Bogart, Chris; Kastner, Christian; Herbsleb, James; Thang, Ferdian	2021	ACM Transactions on Software Engineering and Methodology	Ecosystems differ in their approaches to breaking changes, and there is no general theory to explain the relationships between features, behavioral norms, ecosystem outcomes, and motivating values. They address this through two empirical studies.	npm registry, CRAN repositories, git repositories of Eclipse, Github repositories, libraries.io, WOC, interview, survey	In an interview case study, they contrast Eclipse, NPM, and CRAN, demonstrating that these different norms for coordination of breaking changes shift the costs of using and maintaining the software among stakeholders, appropriate to each ecosystem's mission. In a second study, they combine a survey, repository mining, and document analysis to broaden and systematize these observations across 18 ecosystems.	Improvements that a maintainer makes to a shared package may affect many users of that package, these actions may require review. From developers whose software depends on that package		Atom, CRAN, CPAN, RubyGems, Cargo, PyPI, NuGet, Maven, Packagist, NPM, Hex, CocoaPods, Go, Cabal, Stack, Luarocks, Bioconductor	-	<p>Further exploration of small ecosystems, for example with interviews or analysis of artifacts, should be a priority for future work. The relationships between practices and values are context-dependent and thus hard to generalize. A comprehensive theory incorporating such insights is a task for future work. Future work using time series data about developer overlap and historic participation in ecosystems would allow researchers to identify specific developers that moved to ecosystems with different or similar practices and values (according to their survey data) and use interviews, surveys, or data mining to see if and how their behavior changed.</p>
46	Why reinventing the wheels? An empirical study on library reuse and re-implementation	Xu, Bowen; An, Le; Thang, Ferdian; Khomh, Fouate; Lo, David	2020	Empirical Software Engineering	In this work, they investigated the reasons behind library reuse and re-implementation	F-Droid, Github, open survey, previous paper	They first crawled data from two popular sources, F-Droid and GitHub. Then, potential instances of library reuse and re-implementation were found automatically based on certain heuristics. Next, for each instance, they further manually identified whether it is valid or not. They then conducted two types of surveys (i.e., individual survey to corresponding developers of the validated instances and another open survey for library reuse and re-implementation. Finally, they perform qualitative and quantitative analysis on the survey responses and commit logs of the validated instances.	Increased dependencies, libraries are badly documented, devs are not aware of the	Library recommendation systems	-	-	<p>First, the state-of-the-art clone detection tool can potentially be applied to identify more instances of library reuse and re-implementation. Second, more diverse data sources can be considered, such as GitHub, BitBucket, and SourceForge. Third, more types of program languages can be analyzed, such as JavaScript which is the most commonly used programming language at the time of writing.</p>

Id	Article Title	Author name(s)	Publication year	Venue	Research Objectives (R1)	Data Sources (RQ2)	Methods (RQ3)	Problems (RQ4)	Mitigation Techniques (RQ5)	Package Managers (RQ6)	New Tools (RQ7)	Future Works (RQ8)
47	World of code: enabling a research workflow for mining and analyzing the universe of open source VCS data	Ma, Yuxing; Dey, Tapajit; Bogart, Chris; Amreen, Sadika; Valiev, Marat; Tufko, Adam; Kennard, David; Zaretski, Russell; Mockus, Audris	2021	Empirical Software Engineering	They study how are the tens of millions of projects in the OSS ecosystem periphery interconnected through technical dependencies, code sharing, or knowledge flow	GitHub	They create a very large and frequently updated collection of version control data in the entire FLOSS ecosystems named World of Code (WoC), that can completely cross-reference authors, projects, commits, blobs, dependencies, and history of the FLOSS ecosystems and provide capabilities to efficiently correct, augment, query, and analyze that data.	OSS ecosystem sustainability. Developer data is missing.	Building an infrastructure to handle the ecosystems	Maven, npm, Laa rocks, Bioconductor, Packagist, RubyGems, CPAN, Atom, Cran, PyPI, Cargo	Tool for WoC infrastructure	WoC platform improvements
48	Do the Dependency Conflicts in My Project Matter?	Wang, Y.; Liu, Z.; Wang, R.; Yang, B.; Yu, H.; Zhu, Z.; Wen, M.; Wu, R.; Cheung, S.-C.	2018	ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering	In this paper, they conducted an empirical study on real-world DC issues collected from large open source projects.	GitHub, Apache ecosystem	They collected 135 real-world dependency conflict issues from the Java projects hosted on the Apache ecosystem across 16 categories (e.g., big-data, FTP, library and testing)	Dependency conflicts, existing tools do not differentiate benign from harmful warnings. Developers may overlook the harmful ones and take no resolution actions, which might lead to serious consequences.	Most buildings tools provide their own dependency management strategies to help developers select one version of the duplicate classes during the packaging process	Maven	Decca, an automated detection tool that assesses DC issues' severity and filters out the benign ones	In future, they plan to design effective techniques to help developers automatically repair DC issues. Another plan is to extend their approach to other build frameworks such as Gradle.
49	A Model to Understand the Building and Running Inter-Dependencies of Software	German, D.M.; Gonzalez-Barahona, J.M.; Robles, G.	2007	Working Conference on Reverse Engineering (WCRE)	The goal of this paper is to create a framework to model, extract, study and visualize the dependencies among applications.	Debian	Inter-dependency graph	In order to build and install a complex application it might be necessary to download, build, install and configure dozens of different applications. Run-time dependencies are also tricky, since it is easy miss a dependency which is only needed when certain path of execution is followed. Only a real compilation in a controlled environment can assess the exact collection of software needed for building the software. Security and reliability implications are also important since running any application implies potentially running any part of the full tree of dependencies it needs.	Tool support, open source application developer's input	Debian	-	-
50	A Comparative Study of Vulnerability Reporting by Software Composition Analysis Tools	Imtiaz, Nasif; Thorne, Seaver; Williams, Laurie	2021	ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)	The goal of this study is to understand the difference in vulnerability reporting by various SCA tools. Understanding if and how existing SCA tools differ in their analysis may help security practitioners to choose the right tooling and identify future research needs.	The analysis reports of 9 industry-leading SCA tools	They present an in-depth case study by comparing the analysis reports of 9 industry-leading SCA tools on a large web application, OpenMRS, composed of Maven (Java) and npm (JavaScript) projects.	Known vulnerabilities in dependencies are one of the top ten security risks	Software composition analysis (SCA) tools are used to report known vulnerabilities in the open source dependencies of a software.	npm, Maven	-	They point out two research directions in the SCA space: i) establishing frameworks and metrics to identify false positives for dependency vulnerabilities; and ii) building automation technologies for continuous monitoring of vulnerability data from open source package ecosystems.
51	Session-based Social and Dependency-aware Software Recommendation	Yan, Dengcheng; Tang, Tianyi; Xie, Wenxin; Zhang, Yiyen; He, Qiang	2022	Applied Soft Computing	In this paper, they aim to model the dynamic interests of developers with both social influence and dependency constraints, and propose the Session-based Social and Dependency-aware software Recommendation (SSDRec) model.	GitHub, Libraries.io	Session-based Social and Dependency-aware software Recommendation model SSDRec is composed of four components: (1) Dependency constraint: they use a graph attention network to capture the dependency relations among software packages and obtain the embedding of each software package. (2) Dynamic interest modeling: they use a recurrent neural network to model the sequence of software packages in a session and obtain the final embedding of developer's dynamic interests in this session. (3) Social influence: they use another graph attention network to capture the social influence from neighbor developers and obtain the final embedding of each developer. (4) Recommendation: they use softmax to estimate the probability a developer will choose a given software package.	Information overload, developers often need to spend much time searching software packages they are interested in. Complicated dependency relations	Recommendation systems	-	-	In future, they will consider higher order relations in both social and dependency networks.
52	Will Dependency Conflicts Affect My Program's Semantics?	Wang, Ying; Wu, Rongxin; Wang, Chao; Wen, Ming; Liu, Yeyang; Cheung, Shing-Chi; Yu, Hai; Xu, Chang; Zhu, Zhiqiang	2020	IEEE Transactions on Software Engineering	In this paper, they presented an effective and automated test generation technique SENSOR, which are capable of producing valid inputs to trigger the SC issues.	GitHub	They first simulate a series of dependency conflicts for a given project by altering the actually-loaded versions of its referenced libraries. Then they execute the project's associated tests to see if it can capture the inconsistent behaviors introduced by the version substitution.	Dependency conflict issues, SC issues are difficult to diagnose	An automated technique to detect SC issues is highly desirable	Maven	SENSOR, which is capable of producing valid inputs to trigger the SC issues.	In future, they plan to combine symbolic execution or fuzzing techniques with their technique to improve its test input exploration capability.
53	Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks	Oltm, Marc; Plate, Henrik; Sykoseh, Arnold; Meier, Michael	2020	Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)	This work is meant to facilitate the future development of preventive and detective safeguards by open source and research communities.	snky.io, libraries.io, deprecated mirrors, and public research repositories	This paper presents a dataset of 174 malicious software packages that were used in real-world attacks on open source software supply chains, and which were distributed via the popular package repositories npm, PyPI, and RubyGems. Those packages, dating from November 2015 to November 2019, were manually collected and analyzed. The paper also presents two general attack trees to provide a structured overview about techniques to inject malicious code into the dependency tree of downstream users, and to execute such code at different times and under different conditions.	Open source software supply chain attacks	Multi-factor authentication for open source maintainers, version pinning and disablement of install scripts for open source users, or the isolation of build processes and hardening of build servers, raising general awareness among stakeholders	npm, PyPI, RubyGems	-	It will be important to collect a comprehensive set of existing safeguards, and perform a gap analysis with regard to the attack vectors. For example, they expect new techniques and tools to scan entire package repositories for suspicious packages, e.g. on the basis of the observation that malicious code is reused across packages of the same campaign, and even across languages. In this context, the manually curated and labeled dataset allows for supervised learning approaches that support the automated and repository-wide search for malicious packages. Moreover, with regard to existing and new mitigation strategies, the presented dataset may pose as a benchmark. Last, acknowledging the importance of a comprehensive and up-to-date dataset, it will be necessary to continue its curation
54	Containing Malicious Package Updates in npm with a Lightweight Permission System	Gabriel Ferreira, Limin Jia, Joshua Sunshine, Christian Kattner	2021	IEEE/ACM International Conference on Software Engineering (ICSE)	In this paper, they design a lightweight permission system and a corresponding enforcement mechanism that protects applications against malicious updates from a large number of packages in direct and indirect package dependencies.	npm repository, GitHub	They propose a permission system that sandboxes packages and enforces per-package permissions in Node.js applications, i.e., they enforce a least-privilege design at the package level. Their approach is not the first to sandbox individual npm packages, and there is a large design space for possible solutions. However, their approach identifies a novel design that provides protections for a large subset of packages without requiring changes to package implementations and with negligible overhead. They align their design with the requirements and values of the Node.js/npm community and propose it as one useful building block in a security strategy.	Risk of supply-chain attacks through malicious packages updates	Defenses include carefully reviewing all dependencies and dependency updates, hardening the package infrastructure (e.g., transport security, two-factor authentication), and various forms of program analysis and anomaly detection (e.g., Snky.io, npm, GitHub). Rather than using automatic updates with version ranges, developers may lock package versions or use bots to only update dependencies after executing tests. Some Node.js applications are deployed within a sandbox (e.g., containers), reducing potential damage	npm	Permission system with a policy and corresponding enforcement mechanism that sandboxes individual packages, rather than entire applications, ensuring that malicious updates cannot use security-critical resources for which they do not have permissions.	-
55	A Longitudinal Analysis of Bloated Java Dependencies	César Soto-Valero, Thomas Durieux, Benoit Baudry	2021	ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering	This work proposes the first longitudinal analysis of software bloat. They focus on bloat among software dependencies in the Java/Maven ecosystem.	GitHub, dataset from previous papers	They conduct a large scale empirical study about the evolution of these dependencies in Java projects. They analyze the emergence of bloat, the evolution of the dependencies statuses, and the impact of bloat on maintenance. They have collected a unique dataset of 31,515 versions of dependency trees from 435 open-source Java projects. Each version of a tree is a snapshot of one project's dependencies, for which they determine a status, i.e. bloated or not. They rely on DepClean, the state-of-the-art tool to detect bloated dependencies in Maven projects. They analyze the evolution of 48,469 distinct dependencies per project.	Bloated dependencies	Debloating tools	Maven	-	Their dataset and case studies on the origin of bloat provide valuable references for the rapid identification of practices that result in dependency bloat. Those references can be used to build dedicated bots that ask for additional checks, e.g. when a new dependency appears in the dependency tree, or to establish guidelines for developers when they maintain pom.xml files
56	Do Developers Update Their Library Dependencies? An Empirical Study on the Impact of Security Advisories on Library Migration	Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, Katsuro Inoue	2018	Empirical Software Engineering	In this paper, they investigate the extent of which developers update their library dependencies. Their goals are to investigate (1) whether or not library dependencies are being updated and (2) the level of developer awareness to library migration opportunities.	GitHub, interview	They performed a large-scale empirical study to track library migrations between an application client (defined as a system) and their dependent library provider (defined as a library). The study encompasses 4,659 projects, 8 case studies and a developer survey.	Library migrations, vulnerable dependencies, dependency hell	Develop strategies to improve a developer personal perception of third-party updates, especially in cases when effort must be allocated to mitigate a severe vulnerability risk. Visual aids such as the Library Migration Plots (LMP) provide a rich visual analysis, which could prove useful awareness and motivation for developers quickly update.	Maven	-	For future work, they plan to further explore the developers perception on migration effort. Specifically, they would like to better understand the responsibilities of updating when using a third-party library dependency
57	Small World with High Risks: A Study of Security Threats in the npm Ecosystem	Markus Zimmermann; Cristian-Alexandru Staicu; Cam Tenn; Michael Pradel	2019	USENIX Conference on Security Symposium	This paper studies security risks for users of npm by systematically analyzing dependencies between packages, the maintainers responsible for these packages, and publicly reported security issues.	npm advisories registry	They study the potential of individual packages and maintainers to impact the security of large parts of the ecosystem, as well as the ability of the ecosystem to handle security issues. Their analysis is based on a set of metrics defined on the package dependency graph and its evolution over time. Overall, their study involves 5,386,239 versions of packages, 199,327 maintainers, and 609 publicly known security issues.	Ecosystem security risks	Make developers who use third-party packages more aware of the risks entailed by depending on a particular package. To warn developers about unpatched vulnerabilities in their dependencies, the npm audit tool has been introduced. A proactive way of defending against both vulnerable and malicious code is code vetting. Another line of proactive defense could be to systematically train and vet highly influential maintainers	npm	-	Potential mitigation techniques should be investigated in the future.
58	Dependency solving: A separate concern in component evolution management	Pietro Abate; Roberto Di Cosmo; Raffi Treinen; Stefano Zacchiroli	2012	Journal of Systems Software	This work provides substantial coverage of concepts and problems that are common in component managers equipped with automatic dependency solving abilities, for any non-trivial component model. They review state-of-the-art package managers and their ability to keep up with evolution at the current growth rate of popular component-based platforms	-	They start by studying the complexity of the upgrade problem that package managers for component-intensive software platforms have to face. Theorem proofs. They present a DSL called CUDF (for Common Upgrade Description Format), whose documents describe instances of the component upgrade problem.	Components have expectations on the deployment context: they may need other components to function properly and may be incompatible with some other components. The upgrade planning problem: in any non-trivial component model dependency solving is NP-complete	To maintain component assemblies, (semi-)automatic component manager applications are used to perform component installation, removal, and upgrades on target machines. Package managers incorporate numerous functionalities: trusted retrieval of components from remote repositories; planning of upgrade paths in fulfillment of deployment expectations (also known as dependency solving); user interaction to allow for interactive tuning of upgrade plans; and the actual deployment of upgrades by removing and adding components in the right order, aborting the operation if problems are encountered at deploy-time	Debian, RPM	-	It is important to notice that the solvers perform differently on different problem sets: for example, p2cuuf shows better results than the others in the category debian-duf, and it will be surely interesting to analyse, in future work, the structural differences among the different problem sets.

Id	Article Title	Author name(s)	Publication year	Venue	Research Objectives (R1)	Data Sources (RQ2)	Methods (RQ3)	Problems (RQ4)	Mitigation Techniques (RQ5)	Package Managers (RQ6)	New Tools (RQ7)	Future Works (RQ8)
59	Software Release Patterns When is it a good time to update a software component?	Solomon Berber, Marc Maynard, Foutse Khomh	2020	Procedia Computer Science	The objective of this paper is to reduce the risk of breaking updates by reviewing software release patterns and proposing update scheduling recommendations. This paper describes an approach that aims to help development teams reduce the risk of breaking updates.	GitHub, National Vulnerability Database (NVD)	The data setup in this section contains three parts, including 1) the data sources; 2) the data scope; and 3) the resulting data set. They extract as many patterns as possible that are potentially valuable to improve a software component update time. Update time recommendations are given.	Third party software components can be updated independently of the main mobile web applications. This complicates maintaining a stable software product. While third party software components often allow for faster software development, their maintenance costs are continually rising. A rising number of dependent software components paired with faster release cycles lead to more security updates, version upgrades, version downgrades, or version deprecation. This increased version release volatility automatically increases the risk of a single version release update breaking an entire software product	An early and cautious evaluation of the impact of the update on its dependent components based on the evaluation result, a coordinated update of all affected software components. Before an update can be scheduled, compiling an overview of all third party components and their current versions and other details, allows an agile team to more reliably track and schedule update. Monitoring updates on a regular basis. Analyse dependent third party software component updates. Schedule updates along dependency path	-	-	The future work includes a more granular look at software component type, topic patterns and extracting more valuable recommendations, and machine learning models to predict optimal release windows.
60	Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web	Tobias Lauringer, Abdelheri Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, Engin Kirda	2017	Network and Distributed System Security Symposium	In this paper, they conduct the first comprehensive study of client-side JavaScript library usage and the resulting security implications across the Web.	Bower, Wappulyzer, public CDNs, GitHub, libraries websites, Open Source Vulnerability Database (OSVDB), the National Vulnerability Database (NVD), public bug trackers, GitHub comments, blog posts, vulnerabilities detected by Retire.js	First, they need to collect metadata about popular JavaScript libraries, including a list of available versions, the corresponding release dates, code samples, and known vulnerabilities. Second, they must be able to determine if JavaScript code found in the wild is a known library. Third, they need to crawl websites while keeping track of causal resource inclusion relationships and match them with detected libraries.	Modern websites often include popular third-party JavaScript libraries, and thus are at risk of inheriting vulnerabilities contained in these libraries. Vulnerabilities in JavaScript Libraries	Corresponding tools, but they may not be as well-established as in more mature ecosystems	Bower	-	The need for more thorough approaches to dependency management, code maintenance and third-party code inclusion on the Web.
61	Challenges of Tracking and Documenting Open Source Dependencies in Products: A Case Study	Andreas Bauer, Nikolay Harutyunyan, Dirk Riehle, Georg-Daniel Schwarz	2020	IFIP International Conference on Open Source Systems	They performed exploratory single-case study research at one large established software vendor. They gathered and analyzed the key challenges of tracking and documenting open source dependencies in products. They wanted to understand whether these ad-hoc solutions could be based on a single unified conceptual model for managing dependencies.	Interview, survey	They conducted a single-case case study to study the software dependency documentation in terms of FLOSS license compliance and tracking. They chose a case study company that actively uses open source components in products, while encountering issues in tracking and documenting this use as part of the complete product architecture. Their case study was both descriptive and explanatory. It was descriptive in detailing reports of what the current state of open source governance at the studied company when it came to the software dependency documentation focused on FLOSS license complexities. It was explanatory in presenting some reasons why certain issues in product architecture and open source software dependency documentation arise.	To avoid legal and other risks of using FLOSS in commercial products, such as license noncompliance, software vendors need to manage their FLOSS dependencies. The big amount of FLOSS in their products leads to a time-consuming FLOSS license compliance process.	Many companies develop own tooling to merge documented license information and generate more comprehensive reports about the use of FLOSS in their products. FLOSS compliance processes. To cope with the increasing amount of FLOSS-dependencies, the aim is to automate the license compliance process as much as possible.	-	-	Their study suggests that underlying the various point solutions that they found at this vendor lies a conceptual model that they tentatively call the product (architecture) model. In future cross-vendor work, they will investigate whether this conceptual model can be expanded to become a unifying model for all open source dependency management.
62	An Analysis of Library Rollbacks: A Case Study of Java Libraries	Suwa, Hirohiko; Ihara, Akimori; Kula, Raula; Gaikovina; Fujibayashi, Daiki; Matsumoto, Kenichi	2017	Asia-Pacific Software Engineering Conference Workshops (APSECW)	In this paper, they investigate factors leading to a rollback during library migration. Their goal is to understand the rollback phenomena, including when and how they occur during different library migration situations.	Maven Repository, GitHub, SourceForge, BigQuery	They propose a method to empirically mine and extract historic rollbacks from a project repository. Their method includes the detection and extraction of 50 libraries that were adopted as dependencies in 9,357 GitHub projects (i.e., using Google's BigQuery). To detect rollback and their four factors, they need to collect project and library data. For collecting data, they need a project name, a library name, a library version, a migration date of projects, and the release date of libraries. Except for the release date of libraries, they collected these data using the Project Object Model (POM) file of MAVEN. Rollback is detected based on these data. With MavenRepository as the basic data, they confirmed library release dates by verifying the correctness of the date on the official HP of each library. When that wasn't possible, they confirmed the date with other HPs such as GitHub and SourceForge.	Library rollbacks are additional developer cost, project does not know whether the library poses a rollback risk	If we fear a rollback, we should avoid updates, but it's impossible to indefinitely maintain this strategy. When choosing to update a library, we should check its version type, latency time and release cycle. Understanding the migration status of libraries is meaningful because a rollback is a risk when a project migrates a library. This work proposes a method to detect rollback and collect data	Maven	-	In this study, they only considered four factors of the migration situation. They confirmed a few factors, but not all of them affect rollback. In particular, they believe that the amount of changes in the programs and the reason for rollbacks should be extracted from commit logs. Future work will include how to analyze these factors and clarify what influence they have on library selection.
63	Up-To-Crash: Evaluating Third-Party Library Updatability on Android	Huang, Jie; Borges, Nataniel; Bugiel, Sven; Backes, Michael	2019	IEEE European Symposium on Security and Privacy (EuroS&P)	In this work, they extend the updatability analysis to the runtime of apps. They implement a solution to update third-party libraries with drop-in replacements by their newer versions.	Maven repository, Google Play	They describe a two-stage experiment to test apps' runtime behaviors before and after a library update. In the first stage, they apply an automated library updating framework that they developed according to the proposal of prior work. This framework allows replacing an outdated library inside an app with a newer version without additional host code adoption. During the second stage, two automated user interface (UI) tests are performed to evaluate the behavioral correctness of target apps after drop-in replacements of library updates	App developers know little to nothing about the libraries' internals, the attack surface of the host app is unavoidably increased if an included library contains vulnerabilities. Most of the developers do not deem library updates as a reason for app version increment. Developers tend to preserve the outdated library versions to avoid additional efforts for resolving incompatibility with the newer library versions	The most straightforward countermeasure against such vulnerabilities are updates: a third-party library vendor would release a fixed version, an automated updating mechanism could be a way out of this dilemma. API-compatibility based library update framework	Maven	-	Future work could investigate certain problem classes in a focused way.
64	Measuring Dependency Freshness in Software Systems	Joel Cox; Eric Bouwers; Marko van Eekelen; Joost Visser	2015	IEEE/ACM International Conference on Software Engineering (ICSE)	In this paper they aim to make updating third party dependencies more transparent by introducing metrics to quantify the use of recent versions of dependencies, i.e. the system's "dependency freshness".	Maven.org repository, dataset from Cadairu, Common Vulnerabilities and Exposures (CVE)	Before defining a system-level measurement, they first define a measurement to quantify the "freshness" of a single dependency. In this paper, the term "freshness" is used to denote the difference between the used version of a dependency and desired version of a dependency. The freshness values of all dependencies, are aggregated to the system-level using a benchmark based approach. This system level metric is validated along three axis. First, they investigate the assumption that up-to-date versions of a dependency are desired by investigating the relationship between known security vulnerabilities and the system-level dependency freshness. Secondly, the usefulness of the metric is evaluated by interviews with practitioners. Lastly, the variability of the metric is assessed to make sure the changes in the value of the metrics can be placed into context.	Developers need to update to new releases of these dependencies to avoid security and compatibility risks. In practice, prioritizing these updates is difficult because the use of outdated dependencies is often opaque. Older versions of a dependency can contain security issues which can put the system using that version of a dependency at risk. More recent versions of dependencies often include fixes related to the stability of the dependency. Using up-to-date versions normally makes it easier to upgrade the dependencies. Updating dependencies can come at a high cost because manual testing is required to check for regressions after a dependency update. The team working on the software system may have little influence on the development process of the dependency, yet it relies on the developers of the dependency to provide nontrivial security and bug fixes. Bug fixing and adding features to a system is often prioritized over preventive maintenance. Dependency updates may contain trivial changes not directly affecting the system, which makes it hard to justify the update effort at the time of the release of the dependency	Estimate the costs of upgrading dependencies, metrics	Maven	-	To improve the usefulness and applicability of the proposed metrics they envision three area's of future work. Metric refinements: The current metrics do not take attributes of a dependency into account, such as its size, functionality or popularity. Neither does it look at how the dependency is used by the system, how tightly the dependency is coupled or how often it is called. These attributes could provide an additional layer of granularity and possibly reduce the outlier sensitivity of the dependency-level metrics. Update effort estimation: While the metric gives a good indication of how a system is performing with regards to dependency freshness, it is still hard to put this number into context. Impact on software quality: Metrics are often used to monitor and improve the quality of software systems. Now that a metric is defined to monitor the dependency freshness of a system it would be interesting to see whether applying this metric has a positive impact on the dependency management of a system.
65	What do package dependencies tell us about semantic versioning?	Alexandre Decan; Tom Mens	2021	IEEE Transactions on Software Engineering	This article empirically compares semver compliance of four software packaging ecosystems (Cargo, npm, Packagist and Rubygems), and studies how this compliance evolves over time.	libraries.io	They empirically studied the degree of semver-compliance in four large packaging ecosystems (Cargo, npm, Packagist and Rubygems), by analysing the dependency constraints in their package dependency network over a five-year time period, considering runtime dependencies only. Dependency constraints were classified in three categories: those that are compliant with the semver specification, those that are more permissive and those that are more restrictive.	Problems related to software bugs, security vulnerabilities, outdated and unmaintained packages, and source code that is of low quality, untested or duplicated. Such technical problems can become quite difficult to manage because of the sheer size and complexity of package dependency networks and because of the speed of growth and change of such networks. Devs need to keep their package dependencies up to date to be able to benefit from bug and security fixes and new functionalities, but it may require significant effort to upgrade the package dependencies, especially if changes are backward incompatible. Maintainers of reusable packages need to regularly update their packages with new functionalities and bug or security fixes to keep their "clients" satisfied; but they should avoid introducing breaking changes too frequently as this imposes an extra burden on the maintainers of dependent packages.	Semantic versioning has been proposed as a solution to the so-called "dependency hell".	Cargo, npm, Packagist, Rubygems	-	Such a study could be complemented by finergrained in-depth analyses, that study the specific characteristics and internal details of individual package releases, and how they relate to semver-compliance. A finer-grained analysis is also required to understand why and how breaking changes manifest themselves, in order to estimate the effort required to address them, for individual packages as well as for entire package dependency graphs. It may be interesting to expand the analysis beyond the ecosystem boundaries
66	A Systematic Assessment on Android Third-party Library Detection Tools	Xian Zhan, Tianming Liu, Yeqang Liu, Hao Yu, Li Li, Haoyi Wang, Xupu Lao	2021	IEEE Transactions on Software Engineering	A comprehensive understanding of these tools will help make better use of them. To this end, they conduct a comprehensive empirical study to fill the gap by evaluating and comparing all publicly available TPL detection tools based on six criteria: accuracy of TPL construction, effectiveness, efficiency, accuracy of version identification, resiliency to code obfuscation, and ease of use. Besides, they enhance these open-source tools by fixing their limitations, to improve their detection ability. Finally, they build an extensible framework that integrates all existing available TPL detection tools, providing an online service for the research community	Google Play, F-Droid, Maven Central, Jcenter, Google's maven repository, AppBrain, digital libraries	To evaluate the accuracy of different tools in module decoupling, they collect a set of apps including the TPLs, and these datasets cover various situations in TPL construction. To evaluate the effectiveness and efficiency, they collect 221 Android apps from Google Play and their corresponding 59 unique TPLs with 2,115 version files. To verify the version identification accuracy, they collect two different datasets that include the two sets of open-source apps from F-Droid with their corresponding in-app TPLs and version files. To evaluate the resilience to code obfuscation, they also build a dataset that includes obfuscated apps with various obfuscation techniques by different obfuscators.	TPLs may carry malicious or vulnerable code, TPLs as noises could affect the detection results of repackaging detection, malware detection etc.	Various tools have been developed to identify TPLs	Maven	LibDetect framework which integrates the five publicly available tools as an online service	Future research on vulnerable TPL detection and understandings is necessary and meaningful. Catch emerging TPLs, identify TPLs by using dynamic techniques, Consider TPLs developed in other languages.
67	Understanding and Conquering the Difficulties in Identifying Third-party Libraries from Millions of Android Apps	Yanghua Zhang; Jice Wang; Heting Huang; Yujing Zhang; Peng Liu	2021	IEEE Transactions on Big Data	In this paper, they present LibHawkeye, a new clusteringbased technique to identify third-party libraries in millions of Android apps.	Androzo	First, they improve the accuracy of library identification through constructing and using intra-app dependency graphs. They discard the package homogeneity when building intra-app dependency graphs to avoid identifying multiple libraries with the identical package name as the same library. Then, each weakly connected component in the pruned and merged dependency graph is considered as a potential library instance. They calculate the feature of each instance and identify libraries by clustering instances with identical features. Last, they propose a novel refinement for clustering results to reduce false positives	Emerging third-party libraries may introduce a lot of privacy risks and other security threats. Nevertheless, current approaches to libraries identification are far away from the demand for accuracy and efficiency. It is observed that many libraries aggressively collect personal data	Tools that identify third party libraries	Maven	LibHawkeye, a new clusteringbased technique to identify third-party libraries in millions of Android apps	In this paper, they try to refine the clustering result by package names or other attributes, which may eliminate normal library instances. They leave it as further work to get rid of false positives more sufficiently.

Id	Article Title	Author name(s)	Publication year	Venue	Research Objectives (R1)	Data Sources (RQ2)	Methods (RQ3)	Problems (RQ4)	Mitigation Techniques (RQ5)	Package Managers (RQ6)	New Tools (RQ7)	Future Works (RQ8)
68	On the Triviality of Trivial Packages: An Empirical Study of npm JavaScript Packages	Md Atique Reza Chowdhury, Rahe Abdulkareem, Enad Shihab, Braam Adams	2021	IEEE Transactions on Software Engineering	In this paper, they empirically examine trivial packages relative importance and their use cases from two point of views, from the projects usage and ecosystem usage.	GHTorrent	They perform an empirical study by analyzing more than 15,000 JavaScript projects, of which 3,965 depend on trivial packages. To examine project usage, they use static analysis to determine the centrality of the files that use trivial packages and analyze how widely the trivial packages are used in these files. To examine ecosystem usage, they leverage network analysis to examine the role of trivial packages in the ecosystem's dependency network. They conducted a two-fold case study on 7,024 Java projects developed at SAP. First, they scanned the 7,024 Java projects with Eclipse Steady, OWASP Dependency Check, and a commercial vulnerability scanner to gain an in-depth understanding of the use of open-source dependencies at SAP. They applied three different vulnerability scanners that use different vulnerability databases to avoid being subject to the shortcomings of a particular scanner or database.	Increased maintenance costs, an increased risk of exposure to vulnerabilities and even legal issues. Trivial packages can break whole ecosystems, they also introduce some negative side effects (e.g., they may lack proper tests and may introduce significant dependency overhead)	Trivial package developers should put more effort to keep their trivial packages well-maintained and up-to-date, developers should be careful when publishing these trivial packages. Proper updating practices should be followed, even for these trivial packages. Developers should carefully examine such packages before depending on them. Developers should apply a systematic approach when selecting an external package that they want to add and make sure to consider whether the package is trivial. Developers should consider some code enhancement techniques to limit the use of trivial packages. Developers should give as much attention to trivial packages as they do to larger and more complex packages. Ecosystem maintainers should provide a mechanism to warn developers about these trivial packages	npm	-	First, they would like to develop an advanced technique to detect and evaluate the quality of trivial package in an ecosystem since their results reveal that trivial packages play a key role in the ecosystem. Second, they want to devise an automatic approach to identify trivial package so developers can be aware that the packages that they use are trivial. Finally, since their study examines only the importance of JavaScript packages, they would like to investigate the notion of triviality in other/more software ecosystems.
69	Identifying Challenges for OSS Vulnerability Scanners - A Study & Test Suite	Andreas Dann, Henrik Plate, Ben Hermann, Serena Elisa Ponta, Eric Bodden	2021	IEEE Transactions on Software Engineering	This paper studies (i) types of modifications that may affect vulnerable open-source dependencies and (ii) their impact on the performance of vulnerability scanners.	SAP, NVD	They collected data from npm, 1 the largest ecosystem supporting the Javascript programming language, containing more than 600 K reusable packages. To detect downgrades, they sorted the releases of the client packages according to a branch-based ordering algorithm. They parsed the versioning statement of all dependencies in the package.json files according to the adopted grammar by npm. Subsequently, they determined the resolved version of each provider according to the versioning statement used by the client in that release. They also collected the list of commits, issues, and release notes that are associated with a statistically representative sample of the releases that contain at least one downgrade, from which they obtained information that was used to answer RQ1.	The use of vulnerable open-source dependencies. Developers and distributors frequently fork, patch, re-compile, re-builds, or re-package existing OSS. As a result, the same vulnerable code may occur in different, modified dependencies, thereby posing a challenge for the detection of known-vulnerable OSS	To detect vulnerable OSS, research and industry have developed several open-source vulnerability scanners, e.g., the open-source tools OWASP Dependency-Check (OWASP) and Eclipse Steady, the free tool GitHub Security Alerts, and commercial tools such as Snyk, Black Duck, or WhiteSource.	Maven	a novel test suite for assessing vulnerability scanners, Achilles	Their study shows that the identified modifications are a major challenge for the detection of vulnerable OSS as none of the scanners considered is able to handle all types of modification. Thereby, their work highlights the need for further research in the area to improve the detection algorithms and methods implemented by vulnerability scanners.
70	An Empirical Study of Dependency Downgrades in the npm Ecosystem	Filipe Roseiro Cogo, Gustavo A. Oliva, Ahmed E. Hassan	2021	IEEE Transactions on Software Engineering	In this paper, they study why and how downgrades occur in the npm ecosystem	npm registry, GitHub	They perform a systematic empirical analysis of 2,169,273 dependencies that are declared by 829,410 client artifacts towards the 94 most popular libraries available in Maven Central. The Maven Dependency Graph provides a snapshot of Maven Central as of September 6, 2018. This forms the dataset for their study: 5225 libraries (union of all versions of the 94 most popular libraries), 901,876 clients, summing up to 2,169,273 dependencies. First, they analyze the client-side, to determine to what extent each declared dependency is actually used at least once in their code. Second, they analyze the API side of dependencies to determine how different API types are used by the clients. They split this analysis into two steps: they start by investigating the usages of an API, cumulating all its versions; later, they analyze the most popular version of each API. Finally, they propose a new actionable way to explore the continuum of dependencies and assess the impact of focusing on a small subset of APIs. For tasks where costs and effort increase with the size of APIs such as API migration, the tradeoff can be made between cost and number of clients supported.	Updating providers makes a client package more susceptible to potential problems in the new version of the provider. In the latter case, client packages might end up downgrading a provider, i.e., reverting it to an older version. Downgrades increase the technical lag of client packages.	Tools that aid the management. A preventive downgrade is performed to avoid potential issues from future releases of the provider	npm	-	Further research must be carried out to understand how downgrades affect packages throughout the dependency network. Further research is necessary to understand why downgrades tend to take long to occur. Further research should be performed to understand the extent to which vulnerability and security advisories might be influencing client developers to downgrade a provider.
71	API beauty is in the eye of the clients: 2.2 million Maven dependencies reveal the spectrum of client-API usages	Nicolas Harrand; Amine Benellalam; César Soto-Valero; François Bettega; Olivier Barais; Benoit Baudry	2022	Journal of Systems and Software	In this work, they perform a large scale empirical study of client-API relationships in the Maven ecosystem, in order to investigate this seeming paradox between the observations in industry and the research literature	Maven Central	They perform a systematic empirical analysis of 2,169,273 dependencies that are declared by 829,410 client artifacts towards the 94 most popular libraries available in Maven Central. The Maven Dependency Graph provides a snapshot of Maven Central as of September 6, 2018. This forms the dataset for their study: 5225 libraries (union of all versions of the 94 most popular libraries), 901,876 clients, summing up to 2,169,273 dependencies. First, they analyze the client-side, to determine to what extent each declared dependency is actually used at least once in their code. Second, they analyze the API side of dependencies to determine how different API types are used by the clients. They split this analysis into two steps: they start by investigating the usages of an API, cumulating all its versions; later, they analyze the most popular version of each API. Finally, they propose a new actionable way to explore the continuum of dependencies and assess the impact of focusing on a small subset of APIs. For tasks where costs and effort increase with the size of APIs such as API migration, the tradeoff can be made between cost and number of clients supported.	To support all clients, developers need to invest effort that is proportional to the total size of APIs. Yet, accepting to support only the majority of clients, which use the core API types, allows for significant effort savings	If API developers are willing to ignore a minority of clients, they can focus their maintenance, documentation and development effort on the small subset of the API that is the most used. It also opens opportunities to automate library migration, only supporting a limited part of API targeted while supporting a large majority of clients using it.	Maven	-	They wish to explore novel ways of designing public Java APIs in order to reduce the number of types exposed to clients. This may be addressed by the nature of modules introduced in Java 9. Second, they wish to leverage the existence of a core set of API types as an instrument to build adapters between APIs that provide similar features, focused on the subset of most used API elements. This is motivated by the growing challenges of dependency management and the need to abstract dependencies from their concrete implementation in order to address these challenges
72	Watchman: Monitoring Dependency Conflicts for Python Library Ecosystem An empirical comparison of dependency network evolution in seven software packaging ecosystems	Wang, Ying, Wen, Ming; Liu, Yeping; Wang, Yihe; Li, Zheming; Wang, Chao; Yu, Hai Cheung; Shing Chi; Xu, Chang Zhu, Zhiliang	2020	IEEE/ACM International Conference on Software Engineering (ICSE)	In this paper, they conducted an empirical study on 235 real-world dependency conflict issues.	GitHub, PyPI	They empirically studied 235 DC issues in 124 popular Python projects, which were reported on GitHub in the last five years. To evaluate Watchman, they played back the evolution history of all libraries on PyPI, from 1 Jan 2017 to 30 Jan 2019 and deployed Watchman to detect DC issues.	Dependency issues that can cause build failures when the installed version of a library violates certain version constraints on the library.	Specific fixing strategies in Python. Watchman-a new technique to continuously monitor dependency conflicts for the PyPI ecosystem.	PyPI	Watchman, which performs a holistic analysis from the perspective of the entire PyPI ecosystem, to continuously monitor dependency conflicts caused by library updates	In future, they plan to further improve the detection capability of Watchman and generalize their technique to other Python library ecosystems such as Anaconda to make it accessible to more developer communities.
73		Alexandre Decan; Tom Mens; Philippe Grosjean	2017	Empirical Software Engineering	They carried out an empirical comparison of the package dependency networks of seven packaging ecosystems, each associated to a different programming language, and available online, namely Cargo, CPAN, CRAN, npm, NuGet, Packagist and RubyGems.	libraries.io, dataset from previous work, CRAN, GitHub (extractor)	Statistical analysis techniques: survival analysis (a.k.a. event history analysis), Lorenz curve and the Gini index. Most considered statistical analyses considered the whole lifetime of each ecosystem up to 1 January 2017. libraries.io extracted all the metadata from the CRAN, GitHub and the other packages provided by the official registry of the packaging manager.	Complicated and changing dependencies often referred to as the 'dependency hell'. If not properly maintained, the presence of such dependencies may become detrimental to the ecosystem quality. Developers are reluctant to upgrade their dependencies, while outdated dependencies have been shown to be more vulnerable to security issues.	Package maintainers can be assisted in managing their dependency updates by automated tools that monitor dependencies and notify the maintainers when a new release of a package dependency is available, or when an important update needs to be deployed.	Cargo, CPAN, CRAN, npm, NuGet, Packagist, RubyGems	-	Constitute the automated analysis and extraction of dynamic dependencies. Determining external factors and how they influence the ecosystem growth. It would be worthwhile to study, compare and exploit the complex network properties of ecosystem package dependency networks as part of future work. Study the ecosystem dynamics from a socio-technical point of view, combining information from the package dependency network with information from the social network of ecosystem contributors.
74	Distributed Software Dependency Management Using Blockchain	Gavin D'mello; Horacio Gonzalez-Velaz	2019	Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)	This work describes the development of a blockchainbased package control system which is decentralised, reliable, and transparent.	npm repository	The proposed system employs blockchain-based smart contracts to decentralize the package management. They have employed the blockchain network Ethereum, which uses Solidity for the creation of smart contracts. The Geth binary was used to run the Ethereum node. The web3 package was used for the client side implementation. Solidity was used to write the smart contract. The web3 client gives a nice interface to interact with Ethereum based smart contract. A CentOS instance was created on OpenStack to run the Geth binary and also the IPFS server. Implementation was evaluated with npm	Software packages tend to have direct and transitive dependencies on other packages, which make them vulnerable and/or prone to failure if any dependency is unpublished or compromised. Dependencies are not necessarily straightforward and can have multiple nesting levels. While different software package managers offer mirrors and streaming, most package managers are heavily centralised in their architectures, which can present a single point of failure and a source of inconsistencies when package components or versions change. Managing dynamic versions and software dependencies is complicated, as the program dependency graph for large programs is difficult to handle. Microservice-based cloud architecture increases the challenge, since the search space is significantly large to completely understand conflicts between dependencies. The term 'DLL hell' has been coined. Programs using the different versions of the same library tend to break in case there are major changes in the package	Package managers are expected to be 'intelligent enough' to handle different versions of the same package. A CUDF (Common Upgrade Description Format) document was proposed to keep a track of the package definition and its dependencies. Some package managers use semantic versioning or 'semver' principles. These principles should be clearly understood by the authors and users. A security-oriented management framework, CHAINIAC has been used to verify integrity and authenticity for software release processes based on decentralised nodes	npm	-	Further work should study dependencies at multiple levels as their current work has only taken into account single-level. Ideally, it may be useful to study in detail cliques which can uncover not only functional software properties, but also nonfunctional characteristics such as developer relationships, code styles, and other useful traits. One distinct possibility is to build upon their previous work on software migration (patternbased and document to graph NoSQL databases) to enable multi-level, multi-dependency component migration.
75	When GitHub Meets CRAN: An Analysis of Inter-Repository Package Dependency Problems	Alexandre Decan; Tom Mens; Maëlick Claes; Philippe Grosjean	2016	IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)	They explore how the use of GitHub influences the R ecosystem, both for the distribution of R packages and for interrepository package dependency management. They also discuss how these problems could be addressed.	CRAN, GitHub (extractor), interview	The proposed system employs blockchain-based smart contracts to decentralize the package management. They have employed the blockchain network Ethereum, which uses Solidity for the creation of smart contracts. The Geth binary was used to run the Ethereum node. The web3 package was used for the client side implementation. Solidity was used to write the smart contract. The web3 client gives a nice interface to interact with Ethereum based smart contract. A CentOS instance was created on OpenStack to run the Geth binary and also the IPFS server. Implementation was evaluated with npm	The research is based on an April 2nd 2010 snapshot of the FreeBSD ports. This contains 21,458 ports organized into 64 categories, starting from accessibility applications and ending in X11 window managers. They downloaded the ports' archives (R8GB) and unpacked them into a 173GB fully searchable source code tree. By executing a special Makefile target for each port they gathered a set of 71,872 port dependencies. They were then able to use the GraphViz gvt tool to find the transitive closure of all dependencies leading to a given port and thereby obtain a measure of its reuse. They used two methods to find each port's software license. The first method involved finding the most popular licenses, according to the ranking of the sourceforge.net portal. Their second method involved using the ohloh project's account tool to determine the license associated with each port's file. Under both methods they sorted the results by the license identified and assumed that the port's license was the one associated with the largest number of files.	The R package-based software ecosystem would strongly benefit from an automatic package installation and dependency management tool, like the ones that are currently available for other package-based software ecosystems.	CRAN	-	They could extend their analysis to other, smaller R packages sources, like BioConductor or R-Forge. They are planning a more extensive survey to get better insight in the use of GitHub as a package distribution platform and its impact on the R ecosystem. It would be useful to investigate how the use of tools for the R community is reshaping the R community and affecting the way in which R packages are being managed. They also aim to extend their package analysis by taking into account social metadata. Similarly, they could take into account other data sources pertaining to R package development, such as mailing lists, issue trackers, activity on Q&A websites such as Stack Overflow, download statistics, and many more.
76	Open Source Licensing Across Package Dependencies	Maria Kechagia; Diomidis Spinellis; Stephanos Andreouellis-Theotakis	2010	Pushellenn Conference on Informatics	In this paper, they discuss various aspects of oss licensing, and present an empirical study on FreeBSD ports collections concerning their licensing dependencies, in an attempt to identify specific patterns.	FreeBSD	Their research is based on an April 2nd 2010 snapshot of the FreeBSD ports. This contains 21,458 ports organized into 64 categories, starting from accessibility applications and ending in X11 window managers. They downloaded the ports' archives (R8GB) and unpacked them into a 173GB fully searchable source code tree. By executing a special Makefile target for each port they gathered a set of 71,872 port dependencies. They were then able to use the GraphViz gvt tool to find the transitive closure of all dependencies leading to a given port and thereby obtain a measure of its reuse. They used two methods to find each port's software license. The first method involved finding the most popular licenses, according to the ranking of the sourceforge.net portal. Their second method involved using the ohloh project's account tool to determine the license associated with each port's file. Under both methods they sorted the results by the license identified and assumed that the port's license was the one associated with the largest number of files.	A concern regarding the use of licenses that allow the combination of open source and proprietary software is that they effectively undermine the concepts which the OSS movement advocates. It allows a competitor of a noncopyright licensed project to take the source code and build a proprietary, non open source product (e.g. Apple's Mac OS X and FreeBSD). Combining OSS released under different proprietary software house and undertake various technical responsibilities, as well as legal, licensing and intellectual property rights (IPR) issues. Alternatively, permission may be explicitly requested from the OSS project owners to include parts of the code within a proprietary product	Many corporate licenses (e.g. MPL, CPL, EPL, etc.) have been created for companies. Possible strategies around this may include clearly separating, at the architectural level, the pieces of the resulting software product that rely on OSS code from other parts, and license the former as OSS and the latter as proprietary. The use of legal advice would be recommended or alternatively 'packaging companies' could be employed to serve as intermediaries between the OSS community and the proprietary software house and undertake various technical responsibilities, as well as legal, licensing and intellectual property rights (IPR) issues. Alternatively, permission may be explicitly requested from the OSS project owners to include parts of the code within a proprietary product	-	-	A more detailed analysis should factor each license's popularity when looking at the frequency of license pairs across dependencies. Improving the effectiveness and accuracy of their license detection method could allow them to determine the effective license of each application based on the licenses of its components, and also to examine the use of incompatible licenses. A more intriguing possibility worth studying is whether protective licenses, like GPL, and permissive licenses, like BSD, form closed clusters where dependencies among protective and permissive licenses are explicitly avoided.
77	Vuln4Real: A Methodology for Counting Actually Vulnerable Dependencies	Ivan Paschenko; Henrik Platte; Serena Elisa Ponta; Antonino Sabatù; Fabio Mascetti	2020	IEEE Transactions on Software Engineering	Their paper proposes Vuln4Real, the methodology for counting actually vulnerable dependencies, that addresses the over-inflation problem of academic and industrial approaches for reporting vulnerable dependencies in FOSS software, and therefore, caters to the needs of industrial practice for correct allocation of development and audit resources.	Maven Central, MVN repository	To understand the industrial impact of a more precise methodology and to demonstrate Vuln4Real, they considered the 500 most popular FOSS Java libraries used by SAP in its own software. Their analysis included 25767 distinct library instances in Maven. To perform the analysis they have built a tool that leverages the functionality of Apache Maven to extract library dependencies and applies the Vuln4Real postprocessing steps.	Maintaining a secure software supply chain, management for one's software dependencies. This problem is worsened as dependency analysis methodologies are based on assumptions which are suitable for a research analysis but are not valid in an industrial context.	Vuln4Real helps planning the mitigation activities by showing which safe versions of the affected dependencies the developer can adopt directly and for which vulnerable libraries more complex mitigations should be considered. If an outdated direct dependency is affected by a known vulnerability, the simplest solution to mitigate this vulnerability is to update the dependent library to use the fixed version of the dependency	Maven	They are planning to undergo the SAP procedure for publicizing the tool behind Vuln4Real as FOSS	An interesting direction for future research is to understand how important is the list of vulnerabilities. Any improvement in the precision of the list of vulnerabilities will give better results for some libraries. The same consideration might apply to the Developer's view. They also plan to complement this work with a qualitative study on the reasons why developers do not update dependencies with an investigation of developers' behaviour concerning security-related updates.

Id	Article Title	Author name(s)	Publication year	Venue	Research Objectives (R1)	Data Sources (R02)	Methods (R03)	Problems (R04)	Mitigation Techniques (R05)	Package Managers (R06)	New Tools (R07)	Future Works (R08)
78	Dependency Smells in JavaScript Projects	Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, Nikolaos Tsantalis	2021	IEEE Transactions on Engineering	The objective of this paper is to catalog, quantify, and understand recurring periods to project management issues, which they refer to as dependency smells.	Survey, GHTorrent	Through their empirical study, they curated and analyzed a dataset of 1,146 open source JavaScript projects. First, they provide the definition and description of seven identified dependency smells, these definitions are validated with an initial survey of twelve practitioners. They built a tool that detects the aforementioned defined dependency smells in the 1,146 JavaScript projects in their dataset. They then qualitatively investigate their advantages and disadvantages through a survey with 41 JavaScript practitioners. They then conduct an empirical study to examine the prevalence of these dependency smells, and investigate why they are introduced in the studied JavaScript projects. They have analyzed the changes over a 5-year period to provide a more comprehensive view of how the smells evolve through time.	Outdated dependencies and breaking changes, dependency management is difficult, opting for unsuitable approaches can introduce bugs and vulnerabilities into the project, introduce breaking changes, cause extraneous installations, and reduce dependency understandability, making it harder for others to contribute effectively.	Semantic Versioning (SemVer) has been presented as a solution to help effectively manage dependencies	npm	DependencyStuffer that can analyze any JavaScript project that uses npm and detect the presence of dependency smells. The tool can potentially be incorporated into CI pipelines to prevent dependency smells from rippling through software projects	-
79	Evolution of communities of software: using tensor decompositions to compare software ecosystems	Oliver A. Blumenthal, Colin M. Caine, Eva M. Navarro-López	2019	Applied Network Science	In this work, they use tensor factorisation to explore the dynamics of communities of software, and then compare these dynamics between languages on a dataset of approximately 1 million software projects.	libraries.io	They model each ecosystem as tensor decompositions	Choosing the suitable packages is difficult	Modelling the software ecosystems can give a clearer picture to the developers	elm-get, npm, Cargo, PyPi, CRAN, Maven	-	Further work could involve investigating larger components for languages for which they currently have fairly low AMI scores. It would also be fairly straight-forward to extend their technique to consider the co-authorship network that creates the software as another layer in the network. Knowledge of how the ecosystem evolves organically could be used to detect fraudulent packages, especially by unknown authors. The creation of models of software ecosystem evolution from simple sets of rules (for example, 10% of packages are deprecated every 6 months) to try to replicate their results synthetically could also prove insightful. Future work should investigate how to incorporate JavaScript applications in the network and how to attribute their importance in the npm network (e.g., using the number of stars in GitHub). Future work could investigate if developers find centrality a useful metric when selecting packages. Future work needs to investigate if a similar approach can also help identify packages in decline in other ecosystems such as PyPi and Maven. Investigating and understanding why packages' centrality is rising or declining is critical since it helps developers make more informed decisions. Another interesting followup work is to propose an automated approach to finding future central packages so they can receive the attention needed to boost their evolution as early as possible. Finally, identifying packages in decline, the next step should be assisting developers in replacing them. Thus, they plan to develop an approach that suggests better alternative packages for those in decline.
80	Towards Using Package Centrality Trend to Identify Packages in Decline	Suhaib Mujahid, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, Mohamed Aymen Said, and Ibram Adams	2021	IEEE Transactions on Engineering Management	In this paper, they propose a scalable approach that uses the package's centrality in the ecosystem to identify packages in decline.	npm registry, npmjs, the State of JavaScript survey (previous work)	They use the package centrality to identify declining community interest. Specifically, they use the PageRank algorithm to evaluate the shift in their centrality over time. They first build a dependency graph containing all packages in npm as nodes, and their dependency relationships as edges. They update this graph monthly with newly established dependencies and packages and compute the centrality metric for all packages. Each month, they rank the packages based on the value of their centrality metric. To obtain a baseline for their approach, they develop a dataset containing packages in decline and packages not in decline.	New (and often better) packages are continuously being introduced, making other, once popular packages, obsolete, dormant or even deprecated. As such, it is becoming increasingly important for application developers to ensure that they choose the right packages from the ecosystem.	Use popularity metrics to determine popular packages	npm	Prototype web browser extension called Centrality Checker that uses their approach of detecting package in decline.	First of all, the relationship between the CNA metrics of the packages and their quality metrics (e.g. open issues, user downloads, number of developers, binary sizes, among others) should be explored deeper. On one hand, to provide specific tools that developers could use to be aware of the most balanced approach for their packages, whether it is a leaner package with fewer dependencies or a more "feature-full" package directed to a particular audience, and to distinguish between those packages that are safe or those that should be avoided. On the other hand, the awareness of the developers should also be studied: what is their approach in regards to dependencies? Do they use any quantitative approaches when deciding on the dependencies they are going to include in their software? Second, the analysis could be extended with additional measures and network metrics, that could also be further combined with techniques from other disciplines such as NLP. Finally, CRAN is a popular ecosystem, but there are many more that could either be analyzed individually or compared from a complex network perspective to bring additional and potentially valuable findings to the field.
81	A complex network analysis of the Comprehensive R Archive Network (CRAN) package ecosystem	Marçal Mora-Cantallips, Suhaib Mujahid, Salvador Sánchez-Alonso, Elena Garcia-Barriocana	2020	Journal of Systems and Software	Overall, the current analysis aims to demonstrate how complex network analysis techniques can be applied to a OSS package ecosystems (such as CRAN) after building its dependency network, and how the results reflect its scale-free and small-world behavior, the potential vulnerability of some of its packages and the modular structure that is hidden behind the dependency network.	the 'R-hub' search server (see https://r-pkg.org) and the CRAN metadata database	The extraction was executed using R and the "pkgsearch" package which uses the 'R-hub' search server and the CRAN metadata database to provide detailed information about CRAN packages. Data was manually cross-checked to ensure the reliability of the package and the obtained information was complemented with the information directly scraped from the CRAN web repository. A network was constructed. The CRAN package dependency network was analyzed in three dimensions in order to show how complex network analysis can be applied to package ecosystems while benefiting developers, maintainers and contributors.	Third-party libraries introduce both direct dependencies and transitive dependencies that need to be kept updated to prevent vulnerabilities and bug propagation that might endanger the whole ecosystem. Although developers can have a clear vision of the direct dependencies they add to their packages, transitive dependencies might be less clear as they are not included by them, becoming hidden risks or multiple levels below the direct dependency. Even the common action of updating packages entails risks, as changes might break existing functionalities on other packages	A package management system serves the purpose of managing such dependencies, which is important for both functioning and maintenance (e.g., automated updating) of software packages.	CRAN	-	Their study exposes the need for appropriate searching tools. While there exist some tools e.g., npms, future research needs to be conducted experiments to evaluate these tools and make improvements to these existing tools. One example is the need to go beyond popularity metrics such as downloads when ranking packages and considering other semantic- or functionality related attributes when ranking searched packages. Empirical studies that examine how prevalent such duplicated packages really are and how to alleviate the impact of such duplication are needed to support the ecosystem community. Empirically evaluate the current npm ecosystem policies in order to guide new contributions (or other) policies. One future work is to build a tool that automatically combines these co-useage trivial packages and empirically examines its usefulness for both JavaScript developers and the npm ecosystem.
82	Helping or not helping? Why and how trivial packages impact the npm ecosystem	Xiaowei Chen - Rabe Abdalkareem - Suhaib Mujahid - Emad Shihab - Xin Xia	2021	Empirical Software Engineering	They perform a study to understand why developers publish trivial packages and their online survey to them, what they believe to be the drawbacks of publishing them, and how they can alleviate their negative effects if any.	npm registry, survey	Their study is a developer-centered. They conducted an online survey with JavaScript developers who publish trivial packages and qualitatively analyze their responses. To identify the potential survey candidate developers, they first analyzed the npm ecosystem, identified developers that actually publish trivial packages and sent out their online survey to them. They analyzed the npm ecosystems and identify co-usage trivial packages.	There are some downsides of such code sharing that include lower software quality, increased maintenance effort and even legal issues. The overhead of maintaining multiple packages, the increase of dependency hell and difficulty in finding the right packages.	Grouping trivial packages, using dependency management tools, and providing better search tools	npm	-	Their next step is to include nodal effects (related to the degree of each node in the network) into the model which would be a special case of weighted exponential random graph models (ERGM) with nodal effects
83	Modeling the impact of Python and R packages using dependency and contributor networks.	Gizem Korkmaz; Claire Kelling; Carol Robbins; Sally Keller	2020	Social Network Analysis and Mining	In this paper, they develop methods to estimate impact measures, i.e., downloads, of open-source software (OSS), focusing on R and Python packages.	Depsy.org	They use data collected from Depsy.org about the development activities of Python and R packages, and generate the dependency and contributor networks. They develop three Open-Source models for each of the Python and R communities using network characteristics, as well as author and package attributes.	The impact of software packages is difficult to quantify. Even though developers write papers that describe their software, researchers may not know which paper to cite because software packages often have multiple articles associated with them	Different metrics to describe packages, in the context of OSS, other metrics such as downloads should also be evaluated as measures of impact, and this requires an understanding of the features that correlate with these metrics	PyPi, CRAN	-	They intend to further improve the layered ASCII graph layout algorithm in the future. They would like to further explore the effect of the bundling on readability, possibly also considering bundling and re-routing vertical edges to make the graph less wide. Additionally, they plan to explore shifting nodes to there are fewer per line, considering unique marks for the nodes, or adding tooling-like support to address issues with node labeling. They also identified an interest in graph attribute data among Spack developers. While some of this information is available through the indexed trees, no attempt has been made to address this in the existing graph features. They plan to address the multivariate design issues in the ASCII space. They will also experiment with adding interactivity to other Spack analysis commands or implementing support for multiple coordinate views to further support these tasks.
84	Preserving Command Line Workflow for a Package Management System Using ASCII DAG Visualization.	Katherine E. Isaacs; Todd Gamblin	2019	Graphics	To preserve the command line workflow of Spack, they develop an interactive ASCII visualization for its dependency graphs.	Interviews, survey, GitHub	They developed an interactive, terminal-based DAG visualization to support the analysis of package dependency graphs. They seek to improve the dependency graph visualization for Spack. Through a series of interviews with two Spack maintainers they identified their user groups and their tasks with respect to dependency graphs. They focus specifically on graph-related tasks. To support the command-line workflow for Spack community members, they developed graphterm, a Python tool for interactive ASCII dependency graph visualization. They conduct a study to observe the efficiency trade-offs between the visualizations as well as user preferences when visualizing dependency graphs during a command-line workflow.	Building and executing software can be a complicated and frustrating process due to complex requirements in terms of dependencies, their versions and install locations, and how they were compiled. Visualizing is difficult for command line package managers.	Package management systems have been developed to handle these complexities automatically for most users. Package managers provide ease of access to applications by removing the time-consuming and sometimes completely prohibitive barrier of successfully building, installing, and maintaining the software for a system. A package dependency contains dependencies between all packages required to build and run the target software. Package management system developers, package maintainers, and users may consult the dependency graph when a simple listing is insufficient for their analyses. Several graphical visualizations have been proposed for software dependencies. Tool support	Spack	graphterm, a Python tool for interactive ASCII dependency graph visualization.	-
85	Lost in zero space – An empirical comparison of 0.x releases in software package distributions	Alexandre Decan; Tom Mens	2021	Science of Programming	In this paper, they empirically study, for four open source package distributions (npm, Maven, RubyGems, and PyPI), how 0.x package releases and ≥ 1.0 package releases behave differently.	libraries.io	To analyze the considered package distributions, they rely on version 1.6.0 of libraries.io Open Source Repository and Dependency Metadata. For each package distribution, they consider all packages and all their releases, except for the pre-release versions that are known to be unstable and might not satisfy the intended compatibility requirements. They therefore consider only those dependencies that are required to install and execute the package, and hence more accurately reflect what is needed to actually use the package. To reduce noise in the dataset, they manually inspected and removed outlier packages with clearly deviating and undesirable behavior. They carry out deeper analyses to complement the preliminary insights they got, such as studying the activity of 0.x and ≥ 1.0 packages, the time it takes to cross the magic 1.0.0 barrier, the evolution of the release frequency of 0.x and ≥ 1.0 releases and the number of dependent packages for 0.x and ≥ 1.0 packages, and the evolution of dependency constraints in 0.x releases. They also added an entirely novel research question on the characteristics of 0.x and ≥ 1.0 package repositories on GitHub. They complement their analyses with an anecdotal evidence from developers. Finally, an analysis of the version numbers used for initial package releases	The health of a software project can be affected by the maturity of the packages on which it depends.	Limiting the number of dependencies, and depending only on "trusted packages". Devs should avoid extracting libraries from apps without a prior knowledge about the libraries	Cargo, npm, Packagist, RubyGems	-	For example, one could rely on the development history of a package to assess at a fine level of granularity whether 0.x releases actually correspond to rapid development (e.g., based on the number and size of commits and code changes), contain less or less stable features (e.g., based on the number of feature and pull requests), or are more prone to bugs and security vulnerabilities (e.g., based on the number of reported issues). The presented quantitative analysis could be complemented by a full-fledged qualitative analysis based on in-depth interviews with package maintainers and users of these packages. Such interviews can help to understand why package maintainers are reluctant to cross the 1.0.0 barrier, how they perceive 0.x releases, and if they consider them different from ≥ 1.0 releases.
86	Large-Scale Third-Party Library Detection in Android Markets	Menghao Li , Pei Wang , Wei Wang, Shuai Wang, Dinghao Wu , Member, IEEE, Jian Liu , Member, IEEE, Rui Xue , Wei Hui, and Wei Zou	2020	IEEE Transactions on Software Engineering	In this paper, they extend and enrich the library detection research by demonstrating the practical value of their new library detection method.	45 third-party markets	They first decompile the input app and recover the intermediate representation (IR) of the Dalvik bytecode of the app, named smali. Information is then retrieved from the smali code at different levels, i.e., packages, classes, and methods. They also collect the relations among these program elements, including inclusion, inheritance, and invocation relations. They then leverage the retrieved information to build the instances of potential libraries. The next step is to extract the features of each instance as the signature for testing equivalence. In this work, they propose a feature that is sensitive to minor code mutations while resilient to name-based obfuscations. With a predefined threshold of occurrence, third-party libraries are identified by clustering instances with equivalent signatures. That is, if the number of occurrences of a feature in the dataset reaches the threshold, they consider the corresponding code components as a library. For efficient computation, they encode the instance feature into a text sequence and hashes it into a short representation using MD5. They have implemented their library detection method in a tool called LibD and evaluated it with 1,427,395 apps collected from 45 third-party Android app markets.	Widely used third-party libraries leads to new software engineering problems that hurt the security and stability of the apps. For example, with advanced reverse engineering techniques, adversaries are able to tamper with popular advertising libraries and direct the revenues to a station under their control, while preserving the other functionalities of the original apps. The adversaries can then publish the compromised and repackaged apps into an unofficial marketplace. In this way, an attacker can contaminate a large number of apps by just tampering with a few libraries. For another example, when a popular social network library contains a security vulnerability, the threat from this vulnerability would spread to many different apps and influence tons of users.	The security community has longed for reliable techniques to accurately identify libraries in mobile apps at a large scale. The first approach is based on whitelists of known libraries. The other approach is to directly extract libraries from apps without a prior knowledge about the libraries	-	They have implemented their library detection method in a tool called LibD.	They leave it as future work to integrate more scalable semantics-based methods into their research. Determining a footprint threshold regarding real-world Android applications may need further investigation and study. They leave it as future work to extend the size of their ground truth set and launch a rigorous training procedure to decide the threshold.

	Article Title	Author name(s)	Publication year	Venue	Research Objectives (R1)	Data Sources (RQ2)	Methods (RQ3)	Problems (RQ4)	Mitigation Techniques (RQ5)	Package Managers (RQ6)	New Tools (RQ7)	Future Works (RQ8)	
87	LibSift: Automated Detection of Third-Party Libraries in Android Applications	Charlie Soh, Hee Beng Kuan Tan, Yuhuen Leandavich Armatovich, Annamalai Narayanan and Lipo Wang	2016	Asia-Pacific Software Engineering Conference (APSEC)	In this paper, they propose LibSift, an approach to identify TPLs in Android apps based on the package dependency graph (PDG) of the app.	Google Play store	<p>They propose to detect all TPLs used in a given Android app by making use of the natural partitioning of the Android apps and perform module decoupling at the package level. They have implemented a prototype of LibSift in Python code to perform preparations, module decoupling and identification of primary module for the detection of TPLs in Android apps. For the evaluation of LibSift, they first evaluate the accuracy of LibSift on a set of real-world Android apps. Next, they evaluate the performance of LibSift. Finally, they compare LibSift with two state-of-the-art TPLs detection approaches.</p> <p>They present an empirical analysis of the Gentoo network. As some key properties of this network cannot be explained by the existing network evolution models, they have been motivated to develop their own models. They discuss three related known models and then propose two new evolution models, in which a new node is connected to an old node with the probability that depends not only on the degree but also on the age of the old node. They present a computational study based on these models. The results of this study are compared against the findings from the realworld Gentoo network. They have analyzed network properties including degree distribution, sparsity, clustering coefficient, degree growth rate, and node growth. As existing models do not provide satisfactory explanation for the observed characteristics concerning the Gentoo network's degree distribution and clustering coefficient, they have developed and evaluated two new network growth models, DDEA and DAAE, based on the KRL and DM models.</p>	The use of TPLs typically causes the app to include a significant amount of code that is irrelevant and may hinder the process of many program analysis tasks. Apart from hindering many program analysis tasks, the usage of TPLs is also associated with more privacy risks and security threats. For example, popular TPLs may be masqueraded by malicious libraries to mislead users into thinking that it is a legitimate TPL. Furthermore, some TPLs, even the popular ones, are known to be aggressively collecting the users' private data.	Identify and exclude TPLs from different program analysis tasks. One of the most common techniques used to identify TPLs in Android apps is to use a whitelist and match the names of the packages in the app to the TPLs package name listed in the whitelist. LibRadar for TPL detection, they propose LibSift to identify TPLs in Android apps based on the package dependency graph (PDG) of the app	-	LibSift, a tool to automatically detect thirdparty libraries in Android applications	Despite the good results, LibSift can still be improved in multiple ways. For their future work, they plan to extend their current work by improving on the current algorithm and providing additional useful features. For example, it may be useful to recognize and identify the obfuscated TPLs. Furthermore, additional information such as the type of library and the maliciousness of the library are also important for certain program analysis tasks. All these can be achieved by performing static analysis on the code of each of the modules identified by LibSift.	
88	Analyzing open-source software systems as complex networks	Xiaolong Zheng; Daniel Zeng; Huijuan Li; Feiyue Wang	2008	Physica A-statistical Mechanics and Its Applications	In their research, they study the package dependencies existing in Gentoo Linux. By treating packages as nodes and their dependencies as edges, they construct the Gentoo network.	The Portage and the Gentoo Web site	They design an algorithm for extracting the tag features of dependent OSS packages and build dependent feature vectors for OSS. They then combine the vectors with topic of OSS readme file as input to train the neural network and obtain the tag distribution probability of OSS, and subsequently, recommend tags for OSS. Experiments are performed on the OSS dataset that they collected from GitHub, over 43K repositories and evaluate their approach on this dataset.	Software package dependencies often span across many different projects, a package may depend on another package	A package management system serves the purpose of managing such dependencies, which is important for both functioning and maintenance (e.g., automated updating) of software packages.	Portage	-		For future research, additional evaluation of the DDEA and DAAE models needs to be conducted, in both software research and other application domains. As to open-source research, work is being pursued by several groups to analyze the developer network. It will be interesting to study the co-evolution between developer and software package networks. From both engineering (package dependency) and management (developer network) perspectives, investigating intrinsic fitness measures for each node and incorporating such measures in model development for open-source software present many interesting research opportunities. Investigating software systems from a complex network perspective helps one to gain better understanding of relationships among software network structure, object complexities, object interactions, development processes, and system evolution. Studying a number of open-source software systems beyond Gentoo Linux and developing models and measures that could be applicable to open-source software engineering in general could lead to fruitful research contributions
89	Automatic Tagging for Open Source Software by Utilizing Package Dependency Information	Liu Yang, Li Wang, Zhigang Hu, Yanwen Wang and Jun Long	2020	International Symposium on Theoretical Aspects of Software Engineering (TASE)	They propose an approach to automatically generate repository tag based on a neural network and LDA by utilizing package dependencies and readme among OSS in communities.	GitHub	To capture the effect of software dependencies on the dependent projects, they introduce the notion of technical leverage. To facilitate project maintenance and help developers to have more meaningful comparison of changes occurred between two library versions, they propose to use change velocity vectors. They applied the proposed metrics to more than 10K distinct library instances used in the FOSS Java Maven-based ecosystem distinguishing between large libraries (over 100KLoC) and small-medium libraries. They surveyed the usage data of the selected sample from MVNRepository.com and the number of users from BlackDuck OpenHub.	Untagged OSS makes managing and retrieving OSS on GitHub difficult. However, developers sometimes neglect to write tag for repositories	OSS communities have begun tagging OSS to help users retrieve OSS in accordance with their characteristics. Tag recommendation systems are proposed for labeling untagged OSS.	npm, PyPI, Composer.	-	A tag recommendation framework for DepTagRec by using package dependencies based on neural network models	In the future, they plan to further investigate more useful information of software like code, design models and git commit message and using it to generate tag for open-source software, and further expand GitHubDepDataSet.
90	Technical leverage in a software ecosystem: Development opportunities and security risks	Fabio Mascetti; Ivan Paschenko	2021	IEEE/ACM International Conference on Software Engineering (ICSE)	They aim to understand the level of technical leverage in the FOSS ecosystem and whether it can be a potential source of security vulnerabilities.	Maven Central Software Repository, Snyk.io, BlackDuck OpenHub, MVNRepository.com	They collected three data sets that provide indicators about the evolution and popularity of packages, from different sources such as the npm registry, GHTorrent and the GitHub project hosting platform.	Developers often decide not to update the third party libraries they are using due to the possibility of introducing incompatible, breaking changes. Using many libraries increases the attack surface, and third-party libraries are known to introduce functionality bugs and security vulnerabilities into the projects that they use them. In some cases, dependent projects keep using outdated components for a decade or more thus increasing also the window of possible exploitations.	Boucharas et al. proposed a standards-setting approach to software product and software supply network modelling. Bonaccorsi and Rossi proposed a simple model to help software developers to decide whether to include FOSS components into their projects. Fur study fills an important gap in the state of the art by providing instruments for evaluation of the impact of technical leverage in the software ecosystems. Metrics.	Maven	They provide an online demo for computing the proposed metrics for real-world software libraries available under the following URL: https://techleverage.eu/		
91	A Look at the Dynamics of the JavaScript Package Ecosystem	Wittem, Erik; Suter, Philippe; Rajagopalan, Shriram	2016	IEEE/ACM International Conference on Mining Software Repositories (MSR)	This paper presents an extensive analysis of the npm ecosystem.	npm registry, GHTorrent, GitHub	Their approach to detecting client code snippets which imitate behaviors of library APIs is based on data dependency graphs. They propose a trace subsumption relation to determine the similarity between data flows in library and client methods. They have implemented a prototype on top of the program analysis framework Wala. Although the prototype currently handles Java programs, its underlying technique is applicable to object-oriented programs in general. They used the compiled code of ten diversified open-source Java projects as experiment subjects ranging from application server, server side infrastructure, object relation mapping framework to code analysis tool.	Packages within npm may depend on each other, and software projects outside of npm, for example applications, may specify dependencies to packages hosted on npm.	Understanding npm provides valuable insights into the rapid growth JavaScript and Node.js experienced within the last years. It also helps to understand how individual packages rose in popularity, prevailed, and sometimes were eventually replaced or disregarded.	npm	-	-	
92	Graph-based detection of library API imitations	Chengnian Sun; Siao-Cheng Khoo; Shao Jie Zhang	2011	IEEE International Conference on Software Maintenance (ICSM)	In this paper, they propose a novel approach based on trace subsumption relation of data dependency graphs to detect imitations of library APIs for achieving better software maintainability.	Ten diversified open-source Java projects	They follow the notion of large-scale distributed data representation defined in the linked open data (LOD) principles [3], where a LOD cloud is a distributed store of multiple data sources with inter and intra links between data items. Thus, a marketplace for open source data consists of a LOD cloud, which is fed with linked software engineering data that are generated by LOD-extractors.	Libraries are not always effectively used by programmers in developing their applications. They often find client code that imitates library API, i.e., client code re-implements the behavior of a library API. Programmers might not be familiar with the library, not aware of all the functionalities, or lost in a huge collection of APIs. Reimplementing the wheel does not only waste unnecessary resources, time and energy, but its failure to abstract away similar code also tends to make software error-prone.	A tool that can detect imitations of library APIs in client code	-	A prototype on top of the program analysis framework Wala	In the future, they would like to explore the feasibility of path-sensitive analysis to further improve their current solution in terms of precision and number of detected valid imitations.	
93	Towards a Marketplace of Open Source Software Data	Fernando Silva Parreira; Gerd Groner; Daniel Schwabe; Fernando de Freitas Silva	2015	Hawaii International Conference on System Sciences	In this paper, they propose a framework for a marketplace enhanced using linked open data (LOD) technology for linking software artifacts within projects as well as across software projects.	Apache mahout library	Their approach is aimed at providing a component dependencies and access control mechanism fit for most of the component-oriented programming scenarios, meanwhile, to keep the access control constrains code separated from the functional code, which means non-intrusive to existing codes. By doing so, they can gain a modular system in an ideal way. They use AspectJ technology to tack the problems occurred in current component based software development in Java and show how it fulfills their goal.	In open source software projects, analysts, developers, architects, project managers and testers produce a large amount of data about software artifacts, from source code to license information. Usually, it is hard to track information crossing multiple artifacts. This lack of transparency results in technical problems, legal problems and market problems. Technical problems comprise poor validation and tests of source code dependencies and low reuse rates of pieces of software. Legal problems include license decisions that are closely related to the business model. Market problems cause consumers to underestimate the value of a specific contribution or the expertise of small and medium enterprises. Although there exist multiple functional (or information sources) stored in various tools or databases, the data contained in the artifacts are not interlinked. Hence, producers and consumers currently do not have transparent access to inter-dependencies.	Contemporary approaches to increasing transparency (e.g., Krugle) provide unified search capabilities over multiple sources of software engineering data. However, these approaches are confined within company's boundaries where a limited and ad-hoc set of repositories is covered. Textbased search is still one of the main aspects (e.g., Snipplr and Koder) for assisting developers with finding meaningful pieces of code. However, none of the current approaches takes advantage of the semantic information in the hidden links between software artifacts, neither do they facilitate external sources to improve search results. In this paper, they present a framework for a marketplace for open source software data.	Maven	-	For the future, they are working on plugins for fostering awareness and collaboration in global software engineering. In the future, the proposed approach for representing and linking open source software data will provide solutions for dealing with decentralized software repositories	
94	Using AOP to ensure component interactions in component-based software	Jingang Zhou; Yong Ji; Dazhe Zhao; Jiren Liu	2010	International Conference on Computer and Automation Engineering (ICCAE)	In this paper, they propose an AOP approach to ensure that the interactions among components are strictly conformed to the sated API usage policies of the components.	-	Literature survey. The present paper will use ANP to prioritize characteristics and sub characteristics of ISO 9126 quality model by assigning weights to them as all quality characteristics and sub characteristics are not of equal importance. The proposed ANP based model will consider that there is inter relation between characteristics of ISO 9126 software quality model. It is also assumed that sub characteristics are also interrelated.	It is still lack of language support to ensure proper interactions among components, i.e. modularity assurance, which usually causes the software hard to maintain and evolve because of the improper dependencies among the components	A new approach to ensure the interactions between components conform to stated policies, AOP	-	-	For future research, they are currently developing the tool environment that ensures the programmers can only access the allowed classes of the components during development and generates the aspects automatically according to the component's description file in a declarative way. They also plan to apply their approach in large software systems to evaluate its validity.	
95	Analytical Network Process based model to estimate the quality of software components	Adesh Kumar Pandey; C. P Agrawal	2014	International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)	The objective of this paper is to propose ANP based model to prioritize the characteristics/subcharacteristics of quality and to 0 estimate the numeric value of software quality.	Literature survey	In this paper they presented the VFramework that enables assessing replicability of workflows. It identifies whether any differences in software dependencies among two executions of the same workflow exist and whether they have impact on the produced results. The VFramework uses the context model which is an extensible ontology allowing for description of workflow and its dependencies. They also conducted a case study in which they investigated the impact of software dependencies on replicability of Taverna workflows used in biomedical research. They tested the impact of software dependencies not only by re-executing workflows in exact environments, but also in similar environments differing in operating system distribution and configuration.	The success of final software applications largely depends upon the selection of appropriate and easy to fit components in software application according to the need of customer. The components which they are going to assembled with their software application may not meet the quality requirements set by the developers and may produce catastrophic results for final product.	Industry standard quality model. Their proposed ANP model	-	-	The proposed model may open the future research direction in other domain of software development as well. Further fuzzy ANP may also be applied to take care of human intervention in the proposed model since it may lead some fuzziness in the collected data.	
96	Identifying impact of software dependencies on replicability of biomedical workflows	Tomasz Miksa; Andreas Rauber; Eleni Mina	2016	Journal of Biomedical Informatics	In this paper they investigate the impact of software dependencies on the replicability of biomedical workflows authored in Taverna. For that purpose they use the VFramework that can verify and validate workflow re-executions. Thus, they can identify whether any differences in software dependencies among two executions of the same workflow exist and whether they have impact on the produced results	Three workflows	To analyze the Maven repository they used FindBugs, a static analysis tool that was also used for research purposes. FindBugs' role is to examine Java bytecode to detect software bugs and separate them into nine categories. Two of them involve security issues.	The context in which the software is run, that is, the infrastructure and the third party dependencies, can have a crucial impact on the final results delivered by a computational experiment	Tools	Debian, RubyGems	VFramework	Future work will focus on analysis of workflow engines architecture and identification of the best way of introducing proposed recommendations. They are also going to apply their framework on further use cases, potentially using different workflow engines.	
97	The Vulnerability Dataset of a Large Software Ecosystem	Dimitris Mitropoulos; Georgios Gousios; Panagiotis Papadopoulos; Vasilios Karakoidas; Panos Louridas; Diomidis Spinellis	2014	Hawaii International Conference on System Sciences	They present a dataset that they produced by applying static analysis to the Maven Central Repository (approximately 265GB of data) in order to detect potential security bugs.	Maven Central		Security bugs	A dataset that they produced by applying static analysis to the Maven Central Repository (approximately 265GB of data) in order to detect potential security bugs.	Maven	-	Thus, future work on their approach could also involve the observation of other ecosystems like the ones mentioned in Section I and projects in different languages like Ruby, Python etc.	

	Article Title	Author name(s)	Publication year	Venue	Research Objectives (R1)	Data Sources (RQ2)	Methods (RQ3)	Problems (RQ4)	Mitigation Techniques (RQ5)	Package Managers (RQ6)	New Tools (RQ7)	Future Works (RQ8)
98	An Optimization Method of Javascript Redundant Code Elimination based On Hybrid Analysis Technique	Gao Qiong; Wenmin Li	2020	International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)	This paper proposes a scheme combining static structure analysis technique and dynamic tracking technique to identify unused code during application execution and optimize the size of application bundle.	GitHub	They use static structure analysis combined with dynamic execution technique to identify and eliminate redundant JS code. In order to evaluate the method of optimizing redundant functions in this paper, the author implements a detection tool based on Nodejs.	Redundant JS code inflates the page weight, which inflates the time it takes to download, parse, and compile JS resources. Therefore, it is very important to optimize the payload of JS resources to improve Web performance.	Code construction tools such as <i>Browserify</i> and <i>Webpack</i> can modularize the application and reduce the source code that is not explicitly required by the program through loading on demand. This paper proposes a scheme combining static structure analysis technique and dynamic tracking technique to identify unused code during application execution and optimize the size of application bundle.	-	A detection tool	In future work, more expansion of merge rules will be added, and real user operations will be monitored to improve the accuracy of redundant functions.
99	Building and Maintaining a Third-Party Library Supply Chain for Productive and Secure SGX Enclave Development	Pei Wang, Yu Ding, Mingshen Sun, Huiho Wang, Tongxin Li, Rongrong Zhou, Zhaofeng Chen, Yiming Jing	2020	ACM/IEEE International Conference on Software Engineering: Software Engineering in Practice	In this paper, they present their experience and achievements with regard to constructing and continuously maintaining a third-party library supply chain for TEE developers.	Rust libraries, GitHub	We port a large collection of Rust third-party libraries into Intel SGX, one of the most mature trusted computing platforms. Their supply chain accepts upstream patches in a timely manner with SGX-specific security auditing. They have been able to maintain the SGX ports of 159 open-source Rust libraries with reasonable operational costs.	The development of TEE software must follow a restrictive programming model to make effective use of strong memory encryption and segregation enforced by hardware. These constraints transitively apply to all third-party dependencies of the software. If these dependencies do not officially support TEE hardware, TEE developers have to spend additional engineering effort in porting them. High development and maintenance cost is one of the major obstacles against adopting TEE-based privacy protection solutions in production.	Constructing and continuously maintaining a third-party library supply chain for TEE developers.	Cargo, Xargo	Meanwhile, they have developed the prototype of an automated tool to help code reviewers identify program locations that likely requires a security audit. This tool analyzes the call graph of the enclave code and warns reviewers about the use of untrusted resources inside SGX.	They expect that at one point in the future, they need to shift their maintenance model.
100	PDGraph: A Large-Scale Empirical Study on Project Dependency of Security Vulnerabilities	Qiang Li; Jinke Song; Dawei Tan; Haining Wang; Jiqiang Liu	2021	IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)	In this paper, they present the first large-scale empirical study of project dependencies with respect to security vulnerabilities.	The vulnerability dataset from NVD, Maven, GitHub	They extracted the vulnerability information from the NVD dataset. They crawled down the full dataset of dependency projects from the Maven Index repository. They utilized the breadth-first search to crawl the partial dataset of dependency projects from the GitHub. They combined the vulnerability information with dependency projects through four matching techniques, including the similarity matching, fallback URL, and third-party sources. Based on the datasets, they propose a new approach, called PDGraph, for generating a directed project dependency graph, where each vertex is a project and each edge represents the project dependency relationship. PDGraph runs common graph algorithms over the dataset, including strongly connected components (SCC) and minimum feedback arc set (MFAS). They extract the relevant information of those projects for detailed analysis, including software versions, usage, popularity, and timestamps.	Security concerns: different software projects may simultaneously suffer a similar flaw or threat caused by mismanagement caused by developers. Adversaries can exploit a security flaw in the reused component to compromise multiple projects or systems	A deep understanding of project dependency will inform of underlying security risks in software development and provide insights for improving project quality. PDGraph can assist developers to audit their source code when their projects' components/libraries have underlying security risks. In current defensive programming paradigms, a security audit might be considered as the best practice to reduce software flaws/vulnerabilities before code is released. Once a vulnerability is detected and disclosed, the professionals and vendor developers should release a security patch and update a new version to remediate the vulnerability.	Maven	PDGraph	In their future work, they will investigate other project management tools for vulnerability dependencies, such as Conan for C/C++, NuGet for C#, and Gem for Ruby. In the future, they will leverage the code property graph to determine whether a vulnerability is transitive and could be exploited among the vulnerable dependency.
101	Can we trust tests to automate dependency updates? A case study of Java Projects	Joseph Hejderup; Georgios Gousios	2022	Journal of Systems and Software	In this paper, they set out to empirically understand how reliable developer tests are in automated dependency updating.	GitHub, Maven Central	First, they select Java repositories with high-quality assurance badges and at-least one test class from GitHub. Then, they build each repository to infer a complete dependency tree of the project along with its source- and test classes. Second, they feed the source classes together with the dependencies of a project to the call extractor and statically extract all its direct and indirect uses of third-party libraries. Third, they use instrumentation to learn all invocations from a project to its dependencies via its test suite. Then, they use the information from the previous step to calculate the dependency coverage of a project. Fourth, they generate mutations of dependencies by inserting simple faults in dependency functions executed by tests. Here, they use dynamic call paths to identify such functions. They can then run both the test suite and the change impact analysis to measure the detection score. Finally, they harvest Dependabot pull requests in a real-time fashion and then manually evaluate how both test suites and change impact analysis perform in practice.	Client programs may unexpectedly discover regression-inducing changes, such as bugs or semantic changes that break code contracts. Discovering, debugging and resolving such issues remains a challenging task for development teams. Poor test coverage of dependency usage in client code can lead to missing update-induced regressions.	Developers can mitigate the risk of integration errors by either using restrictive strategies, such as version locking, or permissive strategies involving dependency update tooling. A more popular option among developers is the use of services providing automated dependency updating, such as <i>greenkeeper</i> , <i>Dependabot</i> , and <i>renovate</i> , that use project test suites to detect regression changes on every new update.	Maven	Updatera, a tooling for performing change impact analysis of library dependencies in Maven	In future work, they aim to establish best practices for updating third-party libraries. As a first step, they aim to understand whether developers direct testing efforts towards dependencies and uncover the strategies they use. Moreover, they also intend to explore hybrid workflows through data-driven methods for efficient update checking by combining dynamic and static analysis. In future updating systems, tool creators should investigate hybrid workflows to complement gaps in regression testing with static analysis and help developers with prioritizing testing efforts.
102	On the Use of Static Analysis to Safeguard Recursive Dependency Resolution	Kamil Jezek; Jens Dietrich	2014	EUROMICRO Conference on Software Engineering and Advanced Applications	In this paper, they describe the problems that can occur when transitive dependencies are resolved automatically using examples from real-world programs. They then present an empirical study to assess the extent of the problem when the popular Maven tool is used, and propose an approach based on static type checking that can capture many of the problems described at build time.	Qualitis corpus version 20120401	They propose a static analyser that verifies all component interfaces. They processed corpus programs as follows: they gathered all corpus programs that used Maven, i.e., projects with a pom.xml build file, the Maven Enforcer plugin was then invoked on these projects in order to detect projects with dependencies on multiple versions of the same component, JaCC was invoked on these projects to detect type incompatibilities, the results from steps 2 and 3 were used to find out whether the version multiplication causes actual incompatibilities.	Web of entities a system depends on may be larger than what is immediately obvious, and a large number of the entities may be hard to discover even with rigorous analysis. Problems occur when the system is tightly coupled to a dependency, or when there is no control over what dependencies the system has. This situation can quickly land the system in "Dependency hell". Especially since the dependencies of a system often have a set of dependencies as much as possible, and at the very least being aware of them is essential to the security and reliability of a system. Empirical analysis and automated static analysis of source code- combining these techniques with a deep knowledge of the system and its surroundings. Package managers can help resolve and download dependencies.	There are well known solutions to this problem, based on the idea to run separate, potentially conflicting versions of the same library in parallel. For Java, custom classloaders can be used for this purpose. OSGi framework is based on this idea. Build tools with automatic dependency resolution	Maven	The performed study is based on static compatibility, i.e. signature matching, using the same rules used by the JVM when linking classes. There is some room to improve this, for instance by investigating the use of checked exceptions. Extending the tool to include certain aspects of behavioural compatibility is another interesting area for future research.	
103	Dependencies: No Software is an Island	Jørgen Tellnes	2013	Master's thesis	In this thesis, they will show that the security and availability of a system are largely determined by the surrounding "ecosystem" of dependencies, and that techniques to reduce the reliance on a system's dependencies—software libraries, services and infrastructures—are hugely beneficial.	npm datastore	They will first evaluate different tools for modelling and mapping dependencies, and then perform a case study of a real-world system called Dynamic Presentation Generator. Further, they will look closer at package management and build systems, study npm, and explain the concept of dependency explosion.	Keeping these dependencies to a minimum, decoupling as much as possible, and at the very least being aware of them is essential to the security and reliability of a system. Empirical analysis and automated static analysis of source code- combining these techniques with a deep knowledge of the system and its surroundings. Package managers can help resolve and download dependencies.	npm, Maven, RubyGems, NuGet	-	Can complex adaptive systems learn to become increasingly robust over time?	
104	Towards the Analysis of Software Projects Dependencies: An Exploratory Visual Study of Software Ecosystems	Francisco W. Santana, Claudia M. L. Werner	2013	International Workshop on Software Ecosystems	In this paper they present an ongoing work that aims to enable the analysis of software ecosystems from both technical and sociotechnical perspectives.	Apache Sling mailing lists, its modules' trunk directories, Sling's JIRA	Proof of concept, their approach consists on the usage of information visualization techniques to depict a SECO, applied over software projects repositories' data extracted using mining software repositories (MSR) methods. Their proposal features three main steps: data extracting, data processing, and SECO visualization. To demonstrate the feasibility of their approach, they conducted an exploration of the relationships of a SECO in the form of a proof of concept.	A poor choice of components can undermine the quality of the developed product, a software ecosystem can also impose obstacles to the development of a software project when the project is made upon components with low technical quality	Sociotechnical analysis, tools for ecosystem visualization	Maven, Ivy	-	Expand this proposal under the following two major aspects: SECO Metrics: given that software ecosystems form a recent subject for Software Engineering studies, there is a lack of papers that list formal metrics and indicators applicable for SECOs. They intend to conduct a systematic mapping of studies to identify metrics used by recent works on SECOs context, gathering a list of metrics that will enable them to further investigate SECOs. Expand Research Perspective: their initiative is part of a greater context of SECOs exploration, with other researchers interested in the investigation of IT and Governance aspects and the proposal of a framework for modeling and managing SECOs. In the future they expect to integrate their solutions and provide more complete studies on SECOs.
105	Can Automated Pull Requests Encourage Software Developers to Upgrade Out-of-Date Dependencies?	Samim Mirhoseini; Chris Parnin	2017	IEEE/ACM International Conference on Automated Software Engineering (ASE)	In this study, they evaluated two mechanisms, (1) automated pull requests, and (2) badges, in order to see if they can encourage developers to upgrade software dependencies.	GitHub's public dataset on Google BigQuery, GitHub Archive dataset on Google BigQuery, Travis-CI API, GitHub API, Git history of public projects on GitHub, survey	They extracted data from five sources. As a result, they obtained 7,470 projects. From these projects, 3619 were using badge notification, 2,578 were using pull request notification, and 1,273 did not use any tool and belong to the control group. After collecting the data, the primary analysis involved determining whether a package was updated or not. They constructed a survey with questions related to developer's experiences with dependency management and their perspectives on tool design.	Developers neglect to update legacy software dependencies, resulting in buggy and insecure software. One explanation for this neglect is the difficulty of constantly checking for the availability of new software updates, verifying their safety, and addressing any migration efforts needed when upgrading a dependency, updating dependencies can be a time consuming task. An important challenge is to convince developers that they should upgrade despite all the associated difficulties and risks.	Emerging tools attempt to address this problem by introducing automated pull requests and project badges to inform the developer of stale dependencies.	npm	-	Future work could identify possible factors, such as rollbacks or number of signature changes, in support of a recommender system for merging pull requests. Future work will need to overcome both the technical and social challenges associated with automated API migration. They believe that by comparing two different tools with similar activity levels, they can provide a more fair baseline for comparing automated pull requests. While this somewhat mitigates the threats, this does not eliminate them. Instead, they believe this offers preliminary evidence that can be further validated in future studies.
106	An Empirical Analysis of Technical Lag in npm Package Dependencies	Ahmed Zerouali; Eleni Constantinou; Tom Mens; Gregorio Robles; Jesus Gonzalez-Barahona	2018	International Conference on Software Reuse (ICSR)	They introduce a technical lag metric for dependencies in package networks, in order to assess how outdated a software package is compared to the latest available releases of its dependencies.	libraries.io	They measured technical lag based on dependencies between package releases in the npm registry.	Software becoming out of date with respect to more recent package releases. This implies that bug fixes and new functionality are not utilized by applications using such library packages - technical lag	Npm packages could benefit of better procedures for updating. However, it is essential to mention that one should always balance between being up-to-date and the increasing effort, cost and risk of updating. Developers should not start using newly available packages immediately because they may still contain bugs that need new patches. technical lag metric	npm	-	In future work, they aim to extend their analysis of technical lag by taking into account other issues such as vulnerabilities and bugs. They also aim to carry out similar analyses for other package managers, while considering transitive dependencies, and carry out cross-ecosystem comparisons. Studying the different behaviour of releases with major version 0 is part of future work. In this paper they ignored such pre-release tags. Taking them into account when computing technical lag is part of future work.

Id	Article Title	Author name(s)	Publication year	Venue	Research Objectives (R1)	Data Sources (RQ2)	Methods (RQ3)	Problems (RQ4)	Mitigation Techniques (RQ5)	Package Managers (RQ6)	New Tools (RQ7)	Future Works (RQ8)
107	In Dependencies We Trust: How vulnerable are dependencies in software modules?	Joseph Hejderup	2015	Master's thesis	The goal of this thesis is to establish the presence of modules depending on security vulnerabilities in the npm registry and security practices in dependency management to better understand why library maintainers are not patching or dealing with security problems.	Node Security Project (NSP), npm registry, GitHub and npm RESTful APIs	<p>Their research approach consists of three phases: Preliminary phase: 1. Explore and identify security advisories compatible with npm modules. 2. Devise a crawling strategy to extract dependency information modules in the registry. 3. Create a snapshot of the npm registry in order to achieve reproducibility of the study. Quantitative phase: 1. implement a vulnerability scanner with the advisory data as the search criteria and the npm registry as the study object. 2. collect the results and perform early statistical analysis. 3. implement an extension of the vulnerability scanner to assess the depth and width of the "dependencies of dependencies" chain for the cascading effect. 4. perform similar data analysis operation on the results and measure the depth of the chain and width. 5. implement historical-version analysis features in order to inspect modules that use or still use a dependency that was flagged by a security advisory. 6. Measure the time latency of updating from a vulnerable version to a non-vulnerable version. Qualitative phase: 1. Modules and attributes that express popularity such as high-downloads, most dependent, GitHub stars are selected for further inspection. 2. Inspect source code, communication, and discussion for each module in an explorative fashion to discover potential security malpractices; awareness of the vulnerable dependency and related actions are particularly inspected. 3. compile the discoveries and report on the findings with supporting data from the qualitative and quantitative phases.</p>	<p>Considering the high usage of third-party source code, the lack of policies in the publication phase to the npm repository and dependency management in applications, this could potentially open up lucrative weaknesses for attackers to exploit. The vast majority of approaches for program analysis in client-side JavaScript leverages a combination of JavaScript and Node.js. The client-side approaches require execution of JavaScript code in the browser and a crawling strategy to explore all states in the web application. This is challenging in server-side JavaScript where dynamic analysis requires semi-manual intervention to explore all code execution paths, and pure static analysis has low accuracy on call-graph construction. Another critical concern is the rate of false positives and false negatives in source code analysis.</p>	<p>One possible approach to perform a risk assessment of third-party components is by analyzing third-party artifacts from a source code perspective. Analysis of the source code can identify potential security vulnerabilities in code sections of the software.</p>	npm	Rastogi.js, an asynchronous dependency analysis tool.	<p>Intelligent Dependency Checker: They proposed a dependency checker that leverages information from the security advisories and library maintainers. A potential prototype can be built by using advisories from NSP and data from GitHub to aid developers making informed decisions about the security situation. Grading the severeness of a security advisory from the context of a module: Classifying modules based on the actual code usage of the dependency in the software system and the vulnerability description of the advisory. This requires tant analysts to assess where the dependency code is leveraged and whether this can be exploited with the advisory description. An automated process of this type of assessment could assist library maintainers evaluate the urgency and time required for performing a patchwork (in case of backward compatibility issues). Security Advisories and Breaking Changes: How do breaking changes impact the work of updating security vulnerabilities and how much of an advantage is this for an attacker? A study of the software architectures and the amount work required for patching a vulnerability is of interest. How can we develop best practices that allows us to resolve security issues with backward compatibility in a manageable small timespan?</p>
108	Visualizing the evolution of systems and their library dependencies	Raula Gaikovina Kula, Coen De Roover, Daniel German, Takashi Ishio, Katsuro Inoue	2014	IEEE Working Conference on Software Visualization (VISSOFT)	In this paper, they visualize the evolution of systems and their library dependencies from two perspectives.	Maven 2 Central Repository	<p>Their visualizations are implemented through two separate, but linked perspectives: a) system-centric (System Dependency Plots) and b) library-centric (Library Diffusion Plots). The dependency relations are derived from the dynamic linking of libraries (pom.xml) provided from the Maven 2 Central Repository ecosystem. To illustrate the use of their visualizations, they designed a cognitive walk-through of how a developer would use their tool with four case scenarios.</p>	<p>System maintainers face several challenges stemming from a system and its library dependencies evolving separately. Novice maintainers may lack the historical knowledge required to efficiently manage an inherited system. While some libraries are regularly updated, some systems keep a dependency on older versions. On the other hand, maintainers may be unaware that other systems have settled on a different version of a library</p>	<p>Knowledge about a system's past upgrade decisions with respect to a library can help maintainers. Such historical information is particularly useful for novice maintainers and maintainers of poorly documented systems with many dependencies. More seasoned maintainers, on the other hand, can benefit from knowledge about upgrade decisions made by different systems. Visualization.</p>	Maven	Visualization tool	<p>Future enhancements include additional interactive and dynamic transitions between the two visualizations.</p>
109	Identification of Dependency-based Attacks on Node.js	Brian Pfretzschner, Lotfi ben Othmane	2017	International Conference on Availability, Reliability and Security	This paper addresses two questions: How attackers can exploit third-party dependencies in Node.js environment? and How to mitigate such attacks? This paper focuses on malicious code located in third-party dependencies and actively tries to attack the dependent applications by exploiting weaknesses of JavaScript and Node.js.	-	<p>They implemented code analysis that detects the attacks using the code analysis tool T.J. Watson Libraries for Analysis (WALA). WALA is a fast, efficient, and extensible analysis framework. The framework transforms the source code to an Intermediate Representation (IR) (a kind of a simple language) form that they use in the analysis. In WALA, 4 attacks were addressed: Global leakage, Global manipulation, Local manipulation, Dependency-tree manipulation. The analyses are integrated into open-source serverless platform OpenWhisk. The integration is designed such that, when a code is provided for execution, OpenWhisk creates a Docker container, analyzes the code, and other processes with executing the code or declines to execute it if it contains a dependency-attack. They developed 20 test cases to verify the analysis.</p>	<p>An attacker could use the dependencies to stage an attack on target Node.js applications: use of third-party dependencies allows for potential attacks because dependencies could be controlled by attackers</p>	<p>Writing code that avoids these attacks. Modify Node.js environment to fix the underlying weaknesses. Unfortunately, these weaknesses are features of JavaScript and Node.js and are design choices. The second approach is to use code analysis to identify such attacks. The analysis needs to be used such that the execution environment denies execution of applications that include dependency-based attacks.</p>	npm	They developed static code analyses using T.J. Watson Libraries for Analysis (WALA)	<p>The analysis has currently several limitations including the limitation of the size of applications that could be analyzed (Max. 9 MB), dependencies that contain binary code, and use of specific functions such as the eval. Future work will address these problems.</p>
110	Fixing Dependency Errors for Python Build Reproducibility	Suchita Mukherjee, Abigail Almanza, Cindy Rubio-Gonzalez	2021	ACM SIGSOFT International Symposium on Software Testing and Analysis	In this paper, they investigate how open-source Python projects specify dependency versions, and how their reproducibility is impacted by dependency packages.	BugSwarm, BugSnPy, GitHub	<p>To further motivate their work, they investigate the frequency in different dependency version specifications used in 1,292 BugSwarm artifacts, i.e., 2,584 builds. Their approach consists of parsing the original log of each build. They adopt this method instead of directly analyzing the source code because of two reasons. Firstly, they only identify the dependencies that are installed and used, thus avoiding dependencies that may have been declared in unused sections of the source code. Secondly, this approach allows them to identify all transitive dependencies being installed, which are not declared in the source code but are required by other dependencies. The two main components of PyDfFix are LogErrorAnalyzer for identifying dependency-related errors causing unreproducibility, and IterativeDependencySolver for fixing unreproducible builds due to dependencies. PyDfFix takes as input the current build log and the original build log. PyDfFix first identifies dependency errors and possible dependency packages causing these errors using LogErrorAnalyzer. This is followed by iteratively building a patch that makes the build reproducible again by IterativeDependencySolver. The iterative algorithm for building the patch re-runs the build with intermediate patches and analyzes the new build logs produced to further identify errors and problematic dependency version specifications. This process continues until the build becomes reproducible, or all patch options have been tested and deemed not useful. To evaluate their approach to fix unreproducible builds by fixing dependency errors, they use the software artifacts from two bug datasets built from real-world open-source projects.</p>	<p>Dependence on other software packages has also led to more build breakage and spread of bugs in dependency networks. The growth of dependency packages in each programming language's ecosystem has created transitive dependencies between these packages. Such dependencies can have the effect of propagating bugs and vulnerabilities, and the removal of a package central to such dependency networks can affect up to 30% of the existing applications. Backward compatibility may change with updated packages, potentially leading to unwanted changing behavior that affects reproducibility. As older versions of dependency packages become deprecated, or the package index URLs become stale, reproducing artifacts that require those older versions becomes difficult.</p>	<p>While it is possible to manually fix dependency issues when re-using an application, it requires expensive developer hours and domain knowledge. Tools</p>	PyPi	PyDfFix to detect and fix unreproducibility in Python builds caused by dependency errors.	<p>In the future, they would like to automate the process of extracting error patterns for dependencies, which would improve their identification of unreproducible builds due to dependency errors, and the precision of PyDfFix at identifying candidate dependencies.</p>
111	A comprehensive approach for software dependency resolution	Zhang Hanyu	2002	Master's thesis	The objective of their research is to offer various techniques to retrieve software dependencies so that when one technique is not available, a user may still have other options to retrieve dependency information.	Debian 6.0.0. main archive	<p>They present four dependency resolution techniques: the source code technique, the build technique, the binary technique, and the spec technique. The binary and spec techniques are developed based on previous research. Each technique retrieves dependency information by analyzing a specific source and can be used alone for dependency resolution. The main principle is to apply various techniques to the same target project so that multiple sets of interdependency data can be retrieved (each set corresponds to a particular technique). The retrieved sets will be cross-validated and merged to achieve the super set of dependencies. The presented techniques to extract inter-dependency by analyzing build scripts, source code, spec, and binary files are realized by a prototype tool DEX. Their evaluation is composed of two studies: Study I, manually verifying the dependencies found by DEX on a collection of sampling projects. Study II, programmatically cross-validating dependencies retrieved by each technique on all the Debian projects whose dependencies can be retrieved by all four dependency resolution techniques.</p>	<p>FOSS is not supported by hardware manufacturers as well as proprietary software, many commercial software companies are hostile to FOSS which leads to FOSS' compatibility problems; the source code of many FOSS products are not well documented which makes them hard to understand; there is still a long way to improve usability for FOSS. The quantity, quality, and organization of reusable libraries make them problematic to use or even to find. FOSS may not be planned and designed as well and could have complex code and less modular architecture, which increases the complexity of the applications that reuse FOSS elements, limits the extensibility and reusability of such applications, and increases the cost of maintenance as well. Software reuse may increase the chance of the so-called orphan component. Incompatible licenses may be another undesirable consequence of software reuse. One of the most important issues of software reusing is to understand the dependency relationship between a software product and its reused components</p>	<p>Dependency analysis may be a constructive and effective means to solve many software reuse related problems. Automate the dependency resolution process</p>	Debian	DEX is designed as a comprehensive tool that can retrieve interdependency information from different sources.	<p>Future research work involves improving the dependency resolution techniques presented in this thesis. They pointed out that the low coverage rate problem suffered by the source code technique can be approved by indexing more projects in the artifact candidate repository. The source code technique may also be improved by adding the support of programming languages such as Java, Perl, etc. The build technique can be improved by supporting more build systems. It should also be updated when new library testing macros are supported in CMake and Autotconf. In addition, a graphic user interface of DEX may also be implemented for better user interactions.</p>
112	A Qualitative Study of Dependency Management and Its Security Implications	Ivan Paschenko; Due-Ly Vu; Fabio Massimo	2020	ACM SIGSAC Conference on Computer and Communications Security	The purpose of this study is to qualitatively investigate the choices and the interplay of functional and security concerns on the developers' overall decision-making strategies for selecting, managing, and updating software dependencies.	Interview	<p>To understand the state-of-the-art they looked in Elsevier Scopus for papers published between 2006 and 2019 that report findings on one of the code groups and that mention surveys, interviews, case or qualitative studies, etc. Their qualitative study is based on semi-structured interviews with 25 enterprise developers, who are involved in development of web, embedded, mobile, or desktop applications. Each interview (lasting 30' on average) was recorded and transcribed. The transcripts were anonymized and sent back to the interviewees for confirmation. Each conversation was then coded along the lines of applied thematic analysis to provide a quantitative assessment of the collected qualitative data.</p>	<p>Vulnerable dependencies are a known problem in the software ecosystems, because free and open-source software (FOSS) libraries are highly interconnected, and developers often do not update their project dependencies, even if they are affected by known security vulnerabilities. Optimal selection of vulnerabilities.</p>	<p>Static analysis tools that allow developers to identify both functionality and security issues in the own code of their software projects. Optimal selection of (FOSS) libraries could be facilitated with high-level metrics that show that a library is well-supported, mature, and not affected by security vulnerabilities. Dependency updates break dependent projects, so if ain't broken, don't touch it rules the world. To be adopted, security fixes should be well-indicated, not introduce breaking changes, and not require significant efforts. To maximize utility, dependency analysis tools should generate alerts only relevant to the library fragments used by dependent projects and suggest possible mitigation strategies along with estimation if they introduce breaking changes. Actionable tools should determine which part of the dependent project is actually affected by the vulnerability in a dependency and suggest alternative libraries that provide similar functionality along with the estimation of the cost of switching to that library.</p>	Atom, Cargo, CPAN, CRAN, Dub, Elm, HaxeLib, Hex, Homebrew, Maven, npm, NuGet, Packagist, Pub, Puppet, PyPi, Rubygems	-	<p>Several nuances are still unaddressed by their study, starting from broadening our studies to more countries to correlating results with different types of industries (e.g., financial companies, critical infrastructures, or social media - as they cover all of them but with too few samples each). The most challenging future work for them and the community at large is to develop the dependencies and security analysis tools required by our developers</p>
113	Dependency Versioning in the Wild	Jens Dietrich; David Pearce; Jacob Stringer; Amjed Tahir; Kelly Blincoe	2019	IEEE/ACM International Conference on Mining Software Repositories (MSR)	They set out to capture the current practice regarding how developers declare dependencies, and investigate whether and how developers change their approach as projects mature.	libraries.io, survey	<p>They used a data set from libraries.io for this study. The data set contains dependency data in CSV format that they imported into a PostgreSQL database for further analysis and processing. In order to conduct a comparative study across different package managers, they developed a taxonomy to represent various ways to declare dependencies. They developed a set of classifications by iteratively reviewing the dependency declarations across all package managers. To enable the analysis of the large dataset, this analysis was automated by developing a set of rules that map regular expressions to the appropriate classifications. They decided to use a purely syntactic classification. Rules were developed for each package manager by reviewing its version constraint specification. They verified the accuracy of the classification. To supplement the analysis of the libraries.io data and gain some insights into the intention of programmers using a particular strategy, a survey was created that asked developers about their declaration habits for dependency management</p>	<p>One challenge now faced by software developers is deciding, for a given package, on which version to depend. Version ranges have the disadvantage that builds can now fail if changes between versions are not backwards compatible. Developers also have to understand how incompatible changes between versions arise. Semantic contract violations present an even bigger challenge for detection.</p>	<p>Semantic versioning has arisen as a popular approach for managing package evolution which uses structured versions of the form "major.minor.micro". Tool support</p>	-	-	<p>Interesting topics for future research include more detailed analysis of what the technological and social barriers to the wider adaptation of semantic versioning are, and how particular communities deal with this.</p>

	Article Title	Author name(s)	Publication year	Venue	Research Objectives (R1)	Data Sources (RQ2)	Methods (RQ3)	Problems (RQ4)	Mitigation Techniques (RQ5)	Package Managers (RQ6)	New Tools (RQ7)	Future Works (RQ8)	
114	Dependency Solving Is Still Hard, but We Are Getting Better at It	Pietro Abate; Roberto Di Cosmo; Georgios Gousios; Stefano Zacchiroli	2020	IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)	In this paper they look back at proposals from dependency solving research dating back a few years.	libraries.io, Debian, RedHat, Eclipse	They have conducted a census of the dependency solving capabilities of current package managers. They have included in the census major language-specific package managers from libraries.io as well as package managers from notable Free/Open Source Software (FOSS) distributions and platforms, such as Debian, Redhat and Eclipse	Dependency solving is a hard (NP-complete) problem in all non-trivial component models due to either mutually incompatible versions of the same packages or explicitly declared package conflicts.	One of the key responsibilities of package managers is dependency solving. Novel dependency management approaches have emerged. On the one hand, containerization and virtual environments have gained significant traction; functional package managers have become more popular, due to analogies with container technology and a surge in the interest for scientific and build reproducibility.	Go, npm, Packagist, opam, PyPI, NuGet, Paket, Maven, RubyGems, Cargo, CPAN, Bower, Clojars, CRAN, cabal, Debian, dnf	-	-	
115	Assessing Attack Surface with Component-Based Package Dependency	Su Zhang, Xinwen Zhang, Xinming Ou, Liqun Chen, Nigel Edwards, Jing Jin	2015	Network and System Security	They propose a systematic approach of measuring attack surface exposed by individual vulnerabilities through component level dependency analysis.	-	They propose an attack surface at vulnerability level. In a nutshell, they process a weighted (component-based) dependency graph through breadth first search, they calculate an impact factor for each component (within the dependency graph from the vulnerable component) from the given vulnerability, they then add up all of these impact factors into one number, indicating the attack surface exposed by the target vulnerability	Packages depended by a number of applications are usually more exposable than "ground" software (with no dependent), attackers have more incentive to intrude a system through each of these dependents (or their dependents).	The attack surface brought by package dependency should not be ignored, and accurately measuring the attack surface is non-trivial when evaluating vulnerability severity. Vulnerability and component level metrics can assist system administrators in prioritizing patching or hardening plans towards the entire system, while the overall package and system level metrics can help developers to choose secure and reliable development images, platforms, and specific systems.	-	-	They propose an attack surface at vulnerability level. The metric could also be aggregated into higher levels. Component level attack surface will let stake holders to know how much risk is brought by each individual component and plan hardening accordingly. Package level attack surface can be used to determine which package to depend upon among similar packages. System level attack surface can be used to indicate the health level of individual systems/images. This will help potential users to decide which image to use. Experiments can also be conducted under different environments along with other approaches. Moreover, presentation tools like attack graph can be used to visualize risks from software dependencies.	
116	Automatically Repairing Dependency-Related Build Breakage	Christian Macho; Shane McIntosh; Martin Pinzger	2018	IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)	In this paper, they propose BUILDMDIC, an approach to automatically repair Maven builds that break due to dependency-related issues.	GitHub, the Maven Central repository, JBoss, The Spring repository, The Atlassian repository	They use BUILDHIF to mine the changes in the Maven build specifications of 23 Java open source projects and introduce MAVENLOGANALYZER (MLA), a tool to extract details from the execution log of a Maven build. They then study these changes to find out which types of changes have been performed by developers in the past to repair dependency-related build breakage. They introduce BUILDMDIC, an approach to automatically repair dependency-related build breakage. BUILDMDIC implements three repair strategies that they derive from the most frequent developer fixes. They evaluate BUILDMDIC using an additional 84 randomly selected revision pairs that exhibit a build breakage	The main reasons for build breakage are dependency-related issues, such as dependency resolution errors or outdated dependency configuration.	Automated dependency fixing process	Maven	MAVENLOGANALYZER (MLA), a tool to extract the build result and build details from a Maven build log. BUILDMDIC, an approach to automatically repair dependency-related build breakage	-	They plan to extend BUILDMDIC with further strategies to cover other types of build breakage, such as COMPILATION_FAILED, and to cover breakage that BUILDMDIC currently cannot repair. Furthermore, they will improve the fix time by investigating models to predict the best next strategy. Concerning dependency issues, they plan to extend this work by implementing other fixing strategies, such as adding dependencies. They also aim at extending the study to other projects, in particular to projects from industry. Furthermore, they plan to integrate BUILDMDIC directly into Maven (via a plug-in) to immediately invoke the repair, once an issue is detected.
117	On the Evolution of Technical Lag in the npm Package Dependency Network	Alexandre Decan, Tom Mens, Eleni Constantinou	2018	IEEE International Conference on Software Maintenance and Evolution (ICSME)	In this paper,they perform an empirical study of technical lag in the npm dependency network by investigating its evolution for over 1.4M releases of 120K packages and 8M dependencies between these releases.	libraries.io	Based on the libraries.io dataset, they carried a longitudinal empirical study of such technical lag for 120k packages in the npm package dependency network, over an eight-year time period. They analysed how many packages exhibit technical lag over time, how widespread this lag is over the entire npm package distribution, and which types of package releases are more subject to technical lag depending on their "semantic" version (major, minor or patch). They also explored when, and for which types of releases (major, minor or patch), technical lag tends to increase over time, and which types of release updates (major, minor or patch) tend to reduce technical lag.	Updating to more recent versions might lead to an increased risk of backward incompatible changes. Package dependency networks of software package managers have been shown to be brittle because of the large and increasing number of dependencies over time packages lag behind with respect to the latest version of their dependencies	Package maintainers advocate to investigate the impact of breaking changes in dependent packages before deciding to update them. Dependency management tools such as David, Gemnasium and Greenkeeper help maintainers to keep their dependencies up-to-date, by suggesting updates when new releases of dependent packages become available	npm	-	They plan to replicate their study on package dependency networks of different package distributions. This will allow them to compare the extent of technical lag across different ecosystems, in order to gain a better understanding of how the specific policies, culture and tools affect the presence of technical lag and its evolution over time. They aim to explore to which extent the specific (semantic) versioning policy and the use of dependency constraints influences the presence of technical lag. They aim to empirically study the negative aspects of technical lag, such as the increased risk of suffering from security vulnerabilities and bugs. They also aim to understand the main causes of technical lag in packages, by quantitatively studying the relation between technical lag and a wide range of socio-technical package characteristics such as their age, code size, developer community, usage popularity, and update frequency. This quantitative analysis will be complemented by a qualitative survey, targeting maintainers of relevant packages. With the results of this survey, they aim to identify and understand the reasons that lead developers to manage dependency updates in a specific way.	
118	Dependency Management in Software Component Deployment	Meriem Belgacem; Fabien Dagnat	2007	Electronic Notes in Theoretical Computer Science	The goal of this paper is to present a formalization of deployment dependencies. In this paper, they present the formalization of a static deployment system that ensures the success and safety of installation and deinstallation.	-	A formalization of installation and deinstallation of components has been presented.	Installing (or deinstalling) a component is often a gamble since all the dependencies are difficult to find. Component-based distributed systems are hard to deploy for two main reasons: the complexity of their structure and the complexity of the deployment tasks. Current tools do not manage these complexities properly because the descriptions that they allow lack of expressiveness. The absence of proper descriptions of system and component requirements makes it impossible to ensure safe installation and deinstallation.	Deployment tools and deployment specifications. To face the evolution towards component based systems, their aim is to build a tool with formal foundations ensuring the success and safety of deployment	-	-	We are working on two main directions. First, their objective is to ensure the guarantee of the deployment. For this, a formalization of the properties a deployment system should respect (success, safety, ...) is needed. The goal is then to prove that their system ensures these properties. The second direction is to extend their system to overcome its current limitations.	
119	Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review	Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiaopu Luo, and Yang Liu	2021	IEEE Transactions on Software Engineering	They conduct the first systematic literature review on Android TPL-related research.	ACM Digital Library, IEEE Xplore Digital Library, SpringerLink, and ScienceDirect, Google Scholar, major venues	Following a well-defined systematic literature review protocol, they collected 74 primary research papers closely related to Android third-party library from 2012 to 2020.	TPLs may bring unwanted security risks to mobile users. Malicious TPLs could be introduced into legitimate apps. Adversaries can repackaging an app by adding malicious third-party libraries that can send premium SMS services and steal users' private information. They also often modify ad libraries by changing the revenue destination of ad libraries to redirect the profits. Some TPLs have been reported with behaviors invading users' privacy. The permission model works at the app-level, meaning that TPLs and the host apps share the same privileges, which leads to over-privileged problems. Ad frauds. Vulnerabilities in third-party libraries can pollute the downstream clients. App developers seldom follow the fixed scheme of TPLs and tend to delay the replacement of outdated TPLs in apps, even if these TPLs include severe vulnerabilities. Dependency conflicts. TPLs can be noised that could affect the performance of other app analysis, such as the detection of repackaging and malicious apps.	Tool support	-	-	They suggest future researchers can pay more attention to how to reveal the entire landscape of TPL vulnerabilities. Future research can focus on TPL recommendation, GUI-related TPL smell analysis, TPL updating system design, native libraries related research, library compatibility analysis, TPLs' dynamic features analysis, crosslanguage TPLs analysis. Future researchers can consider handling the challenges, such as the TPL shrink and optimization issues.	
120	Trusting a Library: A Study of the Latency to Adopt the Latest Maven Release	Raula Gaikovina Kula; Daniel M. German; Takashi Ishio; Katsuro Inoue	2015	IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)	In this study, they investigate the impact of dependency management tools such as maven have on trusting a library.	The Maven Super Repository	They implemented their trust dependency algorithms to study the 'latent adoption' phenomena between maven 2 repository libraries. They developed a tool to extract this dependency information from all versions of the POM-files in the repository (PomWalker). They present a set of algorithms to classify initial and introduced library dependency types.	Over time, release updates are sometimes necessary to address bugs or perform other maintenance activities such as refactoring. New features are added to expand functionality. Some updates become mandatory and is crucial patch vulnerabilities. However, sometimes updates break systems, causing incompatibility issues with shared system-wide resources. The tracking and maintenance of libraries has come to be known as dependency management issues ('colloquially termed as 'dependency hell').	To handle dependency management issues, many system maintainers utilize build tools such as Maven and Gradle. These tools promote updates and the trust of latest available library releases in the super repository.	Maven	-	They developed a tool to extract this dependency information from all versions of the POM-files in the repository (PomWalker)	
121	A Generalized Model for Visualizing Library Popularity, Adoption, and Diffusion within a Software Ecosystem	Raula Gaikovina Kula; Coen De Roover; Daniel M. German; Takashi Ishio; Katsuro Inoue	2018	IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)	In this paper, they propose the Software Universe Graph (SUG) Model as a structured abstraction of the evolution of software systems and their library dependencies over time.	Maven central, CRAN	They present the Software Universe Graph (SUG) as a structural abstraction of a super repository. To demonstrate the practicality of the SUG, they construct and apply the SUG model and its operations to a large collection of Maven Central and CRAN super repositories. They then perform model operations on both the Maven and CRAN SUGs to demonstrate the usefulness of their models. By selecting different examples, they show different insights on adoption, diffusion and popularity within each universe.	Meta-data recorded within these ecosystems can provide system maintainers valuable "wisdom-of-the-crowd" insights into these dependency-related questions. They introduce the Software Universe Graph (SUG) as a means to model the realities of popularity, adoption and diffusion within a software ecosystem. metric	-	Maven, CRAN	-	As the rise of cross-language (or platform) systems emerge, potential future research avenues could be the adaptation from a different universe and how system applications deal with cross-universe interaction. For future improvements of accuracy the model can expand beyond the name attribute for lineage classifications. They plan to incorporate more sophisticated techniques and tools used in 'code clone' such as code clone detection and 'origin' analysis to determine common lineage. Another complex but useful operation that was not presented in this paper is the tracing of systems that have abandoned a dependency. They envision that an integration of both the coexistence and DP temporal properties into a single visualization would be beneficial. Immediate future work focuses on evaluating the insights of these queries and visualizations with actual system maintainers.	
122	Up2Dep: Android Tool Support to Fix Insecure Code Dependencies	Duc Cuong Nguyen; Erik Derr; Michael Backes; Sven Bugiel	2020	Annual Computer Security Applications Conference ACASAC	They propose Up2Dep, an Android Studio extension that supports Android developers in keeping project dependencies up-to-date and in avoiding insecure libraries.	Library database of LibScout, F-Droid repository, survey	They developed an Android Studio extension Up2Dep to help Android app developers in upgrading their library dependencies and in avoiding vulnerable library versions. Up2Dep analyzes third-party libraries to provide developers with information about the changes that they may need to perform when updating a library, based on the public API changes between the library versions. Using the collected information about library APIs and their usages on a given project, Up2Dep provides developers feedback on the updatability of outdated library versions. Up2Dep maintains a database of publicly disclosed vulnerabilities and cryptographic API misuse of libraries, and alerts developers if a vulnerable library version is included in their apps. To measure the impact of Up2Dep, they implemented telemetric features inside Up2Dep that gather anonymized information on how developers interact with Up2Dep and that allow developers to provide feedback in-situ on whether the suggested quick-fixes worked as they expected and what they think about such support from Up2Dep.	Third-party libraries, especially outdated versions, can introduce and multiply security & privacy related issues to Android applications. Such libraries could expose user sensitive information to third-party applications, or be a contributing factor for cryptographic API misuse in applications. Even when privacy & security related fixes were available in newer versions of affected libraries, they were not adopted by developers progressed very slowly. The lack of information on whether the current version is secure and on whether the newer version is compatible with the existing code of the app makes developers afraid of migrating their project dependencies to newer versions, and, more importantly, it accustoms developers to stay with the current (although outdated and more likely vulnerable) versions as long as the app is still working.	Ogawa et al. proposed using an external service to split app code and third-party library code from Android application package (APK) files, and then replace the (vulnerable) outdated libraries with their fixed versions. Android Studio itself includes Lint tool to inform developers about updates of third-party dependencies in Android projects. However, Lint only provides developers information on whether there are newer versions of the included libraries, but it does not alert developers about security vulnerabilities of the libraries and it is limited to a list of only two libraries that are classified as privacy risks and without further information about the risks. Tool support.	-	Up2Dep analyzes third-party libraries to provide developers with information about the changes that they may need to perform when updating a library, based on the public API changes between the library versions.	Further investigate how library updatability can be further improved (e.g., detecting non-code, breaking changes between library versions) and study developer's behavior to best provide them the right tool support.	

Id	Article Title	Author name(s)	Publication year	Venue	Research Objectives (R1)	Data Sources (RQ2)	Methods (RQ3)	Problems (RQ4)	Mitigation Techniques (RQ5)	Package Managers (RQ6)	New Tools (RQ7)	Future Works (RQ8)
123	Empirical Analysis of Security Vulnerabilities in Python Packages	Mahmoud Alfaded, Diego Elias Costa, Emad Shihab	2021	IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)	In this paper, they present an empirical study of 550 vulnerability reports affecting 252 Python packages in the Python ecosystem (PyPi).	Libraries.io, Snyk.io	They analyzed 550 vulnerability reports that affect 252 Python packages of which 7,536 package versions are affected.	This level of code reusability supported by software ecosystems also makes the discovery of security vulnerabilities much more difficult, as software systems depend on an increasingly high number of packages.	Studying how vulnerabilities propagate, get discovered and fixed is essential for the health of ecosystems. Code vetting process, a better protocol of publishing package vulnerabilities, should deprecate packages that suffer continuously from vulnerabilities, tools to audit vulnerabilities when installing the packages	PyPi	-	An open avenue for future research is the development of a process that ensures some basic security checks (code vetting) before publishing a release of a package. Future work should focus on broadening their study to other ecosystems and work on the development of package security tools that help practitioners at selecting and using secure software packages.
124	DepOwl: Detecting Dependency Bugs to Prevent Compatibility Failures	Zhouyang Jia, Shaoshan Li, Tingting Yu, Chen Zeng, Eric Xu, Xiaodong Liu, Ji Wang, Xiangke Liao	2021	IEEE/ACM International Conference on Software Engineering (ICSE)	In this paper, they propose DepOwl, a practical tool helping users prevent compatibility failures	StackOverflow, Ubuntu repository	DepOwl approach, evaluated the approach	Existing code is often in the form of libraries, which keep evolving and may introduce incompatible changes (e.g., changing interface signatures). Missues of library versions containing incompatible changes may lead to failures in applications.	When incompatible changes happened, the three roles can prevent CFailures with different solutions: 1) library developers can undo the changes in the latest version, 2) application developers can update the application to adapt the changes, 3) end users can avoid using the incompatible library versions. Dependency management systems	-	DepOwl, to detect DepBugs and prevent CFailures.	The most convenient way to avoid this limitation is to suggest library developers to release binaries with debug symbols when releasing new versions. DepOwl will provide an interface for application developers to indicate a fixed version for each library. This manual effort is the same to most DMSs like pip or Maven. Thus, DepOwl can compile the source code against the fixed library version.
125	Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications	Xian Zhan, Lingling Fan, Sen Chen, Feng Wu, Tianshui Liu, Xupao Lao, Yang Liu	2021	IEEE/ACM International Conference on Software Engineering (ICSE)	They propose a system, named ATVHUNTER, which can pinpoint the precise vulnerable in-app TPL versions and provide detailed information about the vulnerabilities and TPLs.	Maven repository, NVD, GitHub, BitBucket, CVE	At this end, they propose a system, named ATVHUNTER, which can pinpoint the precise vulnerable in-app TPL versions and provide detailed information about the vulnerabilities and TPLs. They propose a two-phase detection approach to identify specific TPL versions. Specifically, they extract the Control Flow Graph as the coarse-grained feature to match potential TPLs in the predefined TPL database, and then extract opcodes in each basic block of CFG as the fine-grained feature to identify the exact TPL versions. They build a comprehensive TPL database (189,545 unique TPLs with 3,006,676 versions) as the reference database. Meanwhile, to identify the vulnerable in-app TPL versions, they also construct a comprehensive and known vulnerable TPL database containing 1,180 CVEs and 224 security bugs.	Detecting TPLs in Android apps is also important for downstream tasks, such as malware and repackaged apps identification. To identify inapp TPLs, we need to solve several challenges, such as TPL dependency, code obfuscation, precise version representation. Unfortunately, existing TPL detection tools have been proved that they have not solved these challenges very well, let alone specify the exact TPL versions. extensive TPL usage attracts attackers to exploit the vulnerabilities or inject backdoors in the popular TPLs, which poses severe security threats to app users. Previous research pointed out that many apps contain vulnerable TPLs, and some of them have been reported with severe vulnerabilities (e.g., Facebook SDK) that can be exploited by adversaries. Even worse, various TPLs are scattered in different apps but the information of TPL components in apps is not transparent. Many developers may not know how many and which TPLs are used in their apps, due to many direct and transitive dependencies. Additionally, about 78% of the vulnerabilities are detected in indirect dependencies, making the potential risks hard to spot	ATVHunter	Maven	ATVHUNTER, which is an obfuscation-resilient TPL detection tool and can report detailed information about vulnerabilities of in-app TPLs	How to find these newly emerging TPLs and dynamically maintain their database will be their future work. They believe that it is meaningful to study the vulnerable TPLs used by both free and paid apps, which is left for future work. Detecting vulnerable native libraries is left for their future work.