An Overview and Catalogue of Dependency Challenges in Open Source Software Package Registries

Tom Mens[1], Alexandre Decan[1,2]

[1]Software Engineering Lab, University of Mons, Belgium
[2]FRS-FNRS Research Associate

Abstract

While open-source software has enabled significant levels of reuse to speed up software development, it has also given rise to the dreadful dependency hell that all software practitioners face on a regular basis. This article provides a catalogue of dependency-related challenges that come with relying on OSS packages or libraries. The catalogue is based on the scientific literature on empirical research that has been conducted to understand, quantify and overcome these challenges. Our overview of this very active research field of package dependency management can be used as a starting point for junior and senior researchers as well as practitioners that would like to learn more about research advances in dealing with the challenges that come with the dependency networks of large OSS package registries.

Keywords

software ecosystem, package dependency network, component reuse, software library, empirical analysis

1. Introduction

Probably every complex software system today relies, to some extent, on reusable OSS libraries distributed through package managers hosting millions of libraries in their package registries. Such reuse inevitably leads to what is commonly known to developers as the *dependency hell*. Software becomes dysfunctional, outdated, buggy, or insecure due to package interdependencies and updates that lead to conflicts, breaking changes, incompatibilities, security issues, deprecations, and many more. Dealing with such issues requires investing significant time and effort. This is why a lot of empirical research in the last decade has focused on understanding OSS package dependency networks, and on mechanisms to cope with dependency-related challenges.

We provide an overview of the literature on how OSS package reuse practices have evolved in recent years. We propose a catalogue of challenges in OSS package dependency networks and beyond, and present recent empirical research to understand and address each of these challenges. It can be used as a basis for junior and senior researchers as well as practitioners that would like to get a kick-start in the state-of-the-art research challenges and advances in package dependency management.

2. Starting point of the overview

As a starting point for this article we based ourselves on a seminal article reporting on an empirical comparison of the evolution of the dependency networks of seven of the largest package registries for mainstream programming languages [1]. The article provided quantitative insights into the dependency issues that software practitioners face when relying on reusable OSS libraries distributed through large package registries. The article focused on package registries for mainstream programming languages for which complete package metadata was available from the libraries.io monitoring service, including reliable information about package dependencies. Seven ecosystems were studied over a five-year period (2012 – 2016): Cargo for Rust, CPAN for Perl, CRAN for R, npm for JavaScript (JS), NuGet for the

BENEVOL24: The 23rd Belgium-Netherlands Software Evolution Workshop, November 21-22, Namur, Belgium
tom.mens@umons.ac.be (T. Mens); alexandre.decan@umons.ac.be (A. Decan)
0000-0003-3636-5020 (T. Mens); 0000-0002-5824-5823 (A. Decan)

Table 1
Comparing the number of packages in seven package registries between April 2017 and October 2024.

Package manager	Language	# packages 04-2017	# packages 10-2024	Increase factor
npm	JS	462K	4,875K	10.6
NuGet	.NET	84K	539K	6.4
Packagist	PHP	97K	461K	4.8
RubyGems	Ruby	132K	187K	1.4
Cargo	Rust	9K	167K	18.6
CRAN	R	12K	27K	2.3
CPAN	Perl	34K	41K	1.2

.NET platform, Packagist for PHP, and RubyGems for Ruby. The study answered four main research questions pertaining to the evolution of the dependency networks of these package registries:

RQ1: How do package dependency networks grow over time? The dependency networks of all studied package registries were observed to grow over time, though the speed of growth differs, with npm being the largest registry experiencing the fastest growth. The dependency network's complexity in terms of ratio of dependencies over packages was observed to remain stable for CPAN, Packagist and RubyGems, while it tended to increase for Cargo, CRAN, npm and NuGet.

RQ2: How frequently are packages updated? A *Changeability Index* was defined to characterise a registry's propensity to change at time t. The number of package updates was observed to remain stable for Cargo, CPAN and CRAN, while it had a tendency to grow for the other registries. Most package releases were observed to receive updates within a few months. However, the number of package updates was not evenly distributed across packages, with a minority of active packages responsible for most of the package updates. Younger or required packages were found to receive package updates more often. Some of the observed behaviours depended on the age of the package registry.

RQ3: To which extent do packages depend on other packages? A *Reusability Index* was defined to measure the amplitude of reuse (number of required packages) and the extent of reuse (number of dependent packages) at time t. Dependencies were observed to abound in all package registries. Most packages depend on other packages, and the proportion of connected packages increases over time. Dependencies were not evenly spread across packages: $< 30\%$ of the packages were required by other packages, and $< 17\%$ of all required packages concentrated $> 80\%$ of all reverse dependencies. This unequal concentration increased over time.

RQ4: How prevalent are transitive dependencies? The indirect reuse induced by the prevalence of transitive dependencies in a package dependency network causes package failures to propagate. This may impact large parts of the network. The majority of dependent packages were observed to have few direct dependencies but many transitive dependencies. More than half of the top-level packages have a dependency tree of depth 3 or higher. The p*-Impact Index* of a package registry at time t was defined to quantify the number of packages that could have a high potential impact because of their many transitive dependents. A notable increase in this impact over time was observed for Cargo, npm and NuGet, suggesting that these registries are becoming more subject to single points of failure.

Based on the answers to these RQs, the authors observed that three package registries (npm, NuGet and Cargo) faced more difficulties to cope with the rapid growth and increasing complexity of their dependency networks, suggesting that they should make an effort to reduce their complexity and fragility. Table 1 (based on data obtained from libraries.io) shows that these registries have continued their growth, suggesting that this observation still holds today. Comparing the number of packages in the dataset of [1] in April 2017 with the data in October 2024, we observe an 18-fold increase in the number of packages for Cargo, a 10-fold increase for npm, and a 6-fold increase for NuGet.

Due to a lack of complete or reliable dependency data, only seven package registries were included in [1]. Dietrich et al. [2] considered a larger collection of 17 different package managers, investigating

Table 2
Catalogue of dependency-related challenges and associated scientific references since 2018. Additional references are provided in the respective subsections of Section 3.

Dependency-related challenges	References
Outdated dependencies	[14, 15, 16, 17, 18, 19, 20, 21, 22, 23]
Breaking changes and backward incompatibilities	[3, 24, 25, 26, 27, 28, 29]
Versioning policies and update strategies	[2, 5, 30, 31, 32, 33, 34, 35]
Dependency solving	[36, 37, 38, 39]
Bloated and missing dependencies	[4, 9, 40, 41, 42]
Vulnerable dependencies	[8, 17, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52]
Supply chain attacks	[53, 54, 55, 56, 57, 58, 59, 60]
Library deprecation and migration	[61, 62, 63, 64, 65]
Depending on trivial libraries	[66, 67, 68]
Abandoned and unmaintained dependencies	[69, 70, 71, 72]
Incompatible licenses	[73, 74, 75, 76]

over 70 million dependencies, complemented by a survey of 170 developers. Similar in vein, Bogart et al. [3] combined a survey, repository mining, and document analysis to observe the dependency practices across 18 ecosystems and their communities. They observed that all ecosystems share values such as stability and compatibility, but differ in other values, and use different tools, policies and practices to support these values. This implies that findings for one ecosystem may not generalise to another. Researchers have therefore empirically studied dependency issues in specific package registries, such as Maven for Java [4, 5, 6], PyPI for Python [7, 8, 9], Swift PM for Swift [10], the ROS ecosystem [11], and CRAN [12, 13].

3. Dependency-related challenges

The empirical findings of [1] revealed the increasing (transitive) complexity, impact, and growth of OSS package registries. This makes it challenging for developers to maintain (dependencies on) such packages. The remainder of this article presents a literature review on empirical research focusing on these challenges and on strategies and solutions that have been proposed to overcome them. The review is based on the content of scientific articles that appeared since 2018, the publication year of [1]. We considered articles in major software engineering conferences or journals that directly cited this work, and used snowballing to include more recent relevant articles. To avoid missing out on important research advances, we also searched through Google Scholar, Semantic Scholar and ResearchRabbit to identify other relevant recent empirical research in this domain. Table 2 catalogs the dependency-related challenges that we have been able to identify based on our literature review. The remainder of this section discusses the most relevant recent research for each of these challenges.

3.1. Outdated dependencies

Updating one's dependencies is a good strategy to reduce or avoid many dependency issues. By keeping dependencies up to date, one can benefit from the most recent functionalities, bug fixes and vulnerability fixes. Staying up to date also makes it easier to interact with upstream dependency providers, who tend to focus on their most recent package releases. Kula et al. [14] studied 4,600 GitHub software projects and 2,700 library dependencies, revealing that 81.5% of the studied systems have outdated dependencies. To quantify such outdatedness of a package w.r.t. its dependencies, the *technical lag* concept has been introduced [15, 16, 19]. Different from the concept of *technical debt*, which focuses on the internal code quality of a software system, technical lag quantifies how much a package is "lagging behind" w.r.t upstream –often third-party– dependencies. Such lag can be expressed along different dimensions [18]: *time lag* (e.g., the time interval between the current version of a dependency being used and some more

recent version); *version lag* (how many major/minor/patch versions a dependency is behind); *security lag* and *bug lag* (if more recent versions have known fixes for vulnerabilities or bugs that affect the version being used). Which dimension or combination to use ultimately remains the decision of the package maintainer, depending on one's priorities.

Researchers have extended the analysis of outdatedness beyond the boundaries of package registries. Lauinger et al. [17] analysed the reliance of 133K websites on JS libraries, observing that a majority of these websites are at least one patch version behind for one of their included libraries, and that most of them are relying on library versions that are outdated by several years. Zerouali et al. [20] studied outdatedness in Docker, the most popular containerization technology. Considering over 3K container images in Docker Hub, they empirically quantified their outdatedness w.r.t. installed JS, Python and Ruby packages. Zerouali et al. [22] studied the outdatedness of 9,482 Helm charts, configuration files for containerized applications for Kubernetes distributed through the Helm package manager. They observed that around half of the container images used in Helm charts are outdated and nearly nine out of them are exposed to vulnerabilities. Decan et al. [23] studied the reliance of GitHub Actions automation workflows on reusable Actions. They found that these reusable Actions are frequently updated, and that most of the workflows are relying on outdated Action versions, hence lagging behind the latest available version for at least seven months, even though they had the opportunity to be updated during at least nine months.

Tool support. Given the importance of keeping dependencies up to date, many automated tools emerged to help developers in this task, such as Dependabot (now part of GitHub), Gemnasium (now part of GitLab), the independent multi-platform solution Renovate, and Greenkeeper (no longer available). Based on a mixed method empirical analysis, He et al. [77] evaluated the effectiveness of Dependabot in keeping dependencies up to date, observing that projects reduce their technical lag after its adoption. On the downside, Dependabot was found to recommend too many incompatible updates; and the amount of Dependabot notifications was considered too high. Rombaut et al. [78] analysed 93,196 issues opened by Greenkeeper for npm projects hosted on GitHub. Greenkeeper was found to induce a significant amount of overhead and false alarms in reported issues. Hejderup and Gousios [21] recommended improving existing dependency update tools to combine static and dynamic analysis in order to reduce the number of semantically conflicting updates. Dann et al. [79] proposed UPCY, an automated dependency update tool that aims to minimise incompatible dependencies when updating. The tool was validated on 29,698 updates in 380 Maven projects, observing an important improvement compared to the updates recommended by existing tools.

3.2. Breaking changes and backward incompatibilities

Keeping dependencies up to date requires much effort from developers, because of the rapid pace of package updates, but also because these updates can lead to *backward incompatibilities* due to the introduction of *breaking changes*. Bogart et al. [3] investigated the policies and practices of making and facing breaking changes in 18 software ecosystems. They observed that maintainers are frequently exposed to breaking changes, and that ecosystems differ in their approaches to breaking changes. Through a mixed methods empirical study Brito et al. [27] analysed why and how developers introduce breaking changes in libraries. The identified reasons were to support new features, simplify existing APIs, and improve maintainability. They also identified a contrast between library producers and consumers in the perceived effort to overcome breaking changes. On the one side, according to the developers, the effort to adopt these breaking changes is generally limited. On the other side, nearly half of the questions related to breaking changes on StackOverflow are about how to integrate and overcome these breaking changes.

Venturini et al. [29] studied *backward incompatibilities* introduced by breaking changes when upgrading npm dependencies to newer versions. By analysing dependency updates in 384 npm packages they found that 11.7% of them lead to breaking changes, even though 44% of these dependency upgrades were meant to be backward compatible. They observed that more than half of the backward incompatible updates are due to transitive dependencies, and that the usual mitigation strategy is either for the

provider package to release a patch fixing the backward incompatibilities (usually within a week) or for the dependent packages to incorporate these backward incompatible changes in newer version of their packages (taking 4 months on average). In a similar vein, Jayasuriya et al. [80] studied the Maven ecosystem by analysing 142K+ direct dependencies of 18K+ Maven artifacts. 71.6% of the dependencies were outdated, and 11.6% of the dependency upgrades applied to them resulted in breaking changes. Changes in transitive dependencies were a major factor for these breaking changes.

Recently, researchers have started to focus on so-called *semantic* or *behavioural breaking changes*. Jayasuriya et al. [81] conducted an empirical analysis on 30,548 dependencies of 8,086 Maven artifacts to identify the impact of dependency upgrades on behavioral breaking changes in the test suites of client Java projects. Only 2.30% of the dependency upgrades caused client tests to break. Zhang et al. [28] proposed and empirically validated a tool to statically detect semantic breaking changes in third-party libraries used by Java projects by measuring semantic differences.

Tool support. Several language-specific tools help developers to detect breaking changes before an update is released to the clients. Examples include PyCompat [82], DepOwl [83] and AexPy [84] for Python, APIDiff [85], Clirr and Revapi for Java [86], and NoRegrets for JS [24, 25]. These tools analyse whether the changes made to the types used in the public API may break clients. Empirical evidence revealed that such tools are able to catch most breaking changes in practice [87]. Approaches based on type regression testing [24, 26] run the test suites of a library's clients to detect breaking changes. While effective, it can require a lot of storage capacity and execution time. Møller and Torp [25] propose an improved variant of type regression testing for JS libraries, by automatically generating tests from a reusable API model. This approach is shown to run faster and find breaking changes in more libraries.

3.3. Versioning policies and update strategies

Proper versioning policies and update strategies, such as *semantic versioning* [2, 30, 35] can help in mitigating breaking changes. Semantic versioning provides an implicit convention between the library consumers (who specify the range of allowed versions) and library producers (who avoid introducing breaking changes in non-major versions). Decan and Mens [32] investigated semantic versioning compliance in Cargo, npm, Packagist and RubyGems, observing that most packages are compliant with semantic versioning. In a follow-up work [33], they found that semantic versioning is misused in packages not having reached the 1.0.0 version barrier. By analysing 120K library upgrades on Maven, Ochoa et al. [5] found that only a minority of library consumers are affected by breaking changes, and a large majority of the upgrades comply with semantic versioning. A study by Jayasuriya et al. [80] on dependency upgrades of outdated Maven libraries leading to breaking changes revealed that almost half of these breaking changes coincided with violations of the semantic versioning scheme.

While convenient to inform about the presence of breaking changes, semantic versioning does not help developers in overcoming them, and important updates containing bug or security fixes may still be missed due to the breaking changes that come with them. *Backporting* these fixes from more recent releases to less recent ones may be the only solution to benefit from them. Decan et al. [34] empirically studied such backported changes in Cargo, npm, Packagist and RubyGems. They found infrequent use of backporting to maintain previous major versions, even when those versions were still widely used. The lack of backports led thousands of packages exposed to security vulnerabilities even if a fix was available for them. Similar in vein, Cogo et al. [31] investigated dependency *downgrades* in npm. By analysing release notes and commit messages, they found that maintainers downgrade their dependencies either reactively (to avoid defects in a specific version, or to cope with unexpected feature changes and incompatibilities) or pro-actively (to avoid issues in future releases). Moreover, maintainers tend to be more conservative on the dependency versions they use after such downgrades.

3.4. Dependency solving

One of the key responsibilities of package managers is *dependency solving*, in order to ensure that all versions of all installed packages are mutually compatible and non-conflicting, and remain so when

removing and upgrading existing packages or installing new ones [38]. Unfortunately, dependency solving has shown to be an NP-complete problem for most package managers [36, 37]. Many package managers provide ad hoc solutions that are not always complete or lack expressiveness.

Pinckney et al. [39] proposed and evaluated Maxnpm and Pacsolve to overcome the shortcomings of the npm dependency solver. It enables customizable constraints and optimization goals, empowering developers to combine multiple objectives when installing dependencies. Other researchers have proposed more generic formally founded solutions based on constraint solving and optimisation [36].

Functional package managers, such as GNU Guix[1] and Nix[2], provide a robust solution by enforcing a declarative approach to dependency management, requiring all dependencies to be declared upfront to build or run software. This ensures that the software environment is fully defined, with each dependency explicitly specified. Additionally, these managers enable the creation of separate namespaces on-the-fly, allowing multiple versions of the same package to be installed side-by-side without any risk of incompatibility or inconsistencies. This represents a significant improvement over mainstream package managers, which often struggle to achieve similar functionality without complex workarounds or dependency resolution issues.

3.5. Bloated and missing dependencies

Several researchers have studied dependency smells related to *bloated* and *missing* dependencies, for the dependency networks of Maven [4], PyPI [9] and npm [40]. Missing dependencies refer to required packages that are not explicitly declared in the configuration file and hence need to be manually installed to avoid dependency problems. Bloated or unused dependencies are packaged with the application's compiled code but are actually not necessary to build and run the application. Including them can increase the size of the application and possibly affect its performance and security posture. Soto-Valero et al. [41] proposed a novel technique for debloating dependencies in Java projects. The technique relies on bytecode coverage analysis to precisely capture what parts of a project and its dependencies are used when running with a specific workload. 68% of the bytecode of Java libraries and 20% of their total dependencies could be removed through coverage-based debloating. Based on a dataset of 988 client projects, 81% of them successfully compiled and passed their test suite when the original library was replaced by its debloated version. Weeraddana et al. [42] showed that unused dependencies also negatively impact CI resource usage. Based on 20K+ commits in 1,487 projects relying on npm packages, they found that $> 55\%$ of the CI build time was associated with dependency updates triggered by unused dependencies.

3.6. Vulnerable dependencies

A very important challenge for OSS package registries is how to cope with vulnerabilities and security weaknesses in dependencies, either directly or indirectly. They can lead to single points of failure with a huge cascading impact through transitive dependencies. One example of a major incident was discovered in May 2021, with a remote code execution vulnerability in npm's pac-resolver package that received over 3 million weekly downloads [88]. Another example in December 2021 was a vulnerability in Maven's Log4Shell package in the Log4j logging framework for Java that caused widespread damage [89]. These incidents increased public awareness of the need to mitigate OSS supply chain attacks [55].

Because of their importance and impact, security vulnerabilities in package registries are a very active domain of research. Dependency outdatedness is one of the major sources of security vulnerabilities. Starting from a dataset of 133K websites depending on JS libraries, Lauinger et al. [17] observed that 37% of the outdated websites included at least one vulnerable library. Decan et al. [44] studied vulnerabilities in npm packages and observed that while vulnerabilities are quickly fixed after their discovery, it takes a lot of time to adopt the fix for a large fraction of packages that (transitively) depend on the vulnerable package. The main reasons are too restrictive dependency constraints and unmaintained packages.

[1] https://guix.gnu.org
[2] https://nixos.org

Zimmermann et al. [48] aligned with these insights, observing that a lack of maintenance by a small number of maintainers causes many npm packages to depend on vulnerable, unmaintained packages. This confirms that npm suffers from single points of failure. Chinthanet et al. [43] analysed the lag between the vulnerable release and its package-side fixing release. Through an empirical study of the adoption and propagation tendencies of 1,290 package-side fixing releases that impact throughout a network of 1.5M+ releases of npm packages, they found that stale clients require additional migration effort, even if the package-side fixing release was quickly made available. Liu et al. [47] investigated the security threats from vulnerabilities in the npm dependency network. They constructed a dependency-vulnerability knowledge graph capturing 10M+ library versions and 60M+ dependency relations and proposed an algorithm to statically resolve dependency trees and transitive vulnerability propagation paths. Based on it, they empirically studied the evolution of vulnerability propagation in npm. Among many findings, they confirmed outdatedness to be a major source of vulnerable dependencies. To cope with it, they developed DTreme, a vulnerability remediation method that outperforms the official npm audit fix tool. Prana et al. [45] analysed vulnerabilities in libraries used by 450 projects written in Java, Python, and Ruby. They examined the types, distribution, severity and persistence of the vulnerabilities over a one-year period. They found that most vulnerabilities persist throughout the observation period, and the resolved ones take 3-5 months to be fixed.

Several studies have compared npm to other package registries, given that the particularities of specific ecosystems could affect their vulnerability posture. Zerouali et al. [46] compared npm to RubyGems w.r.t. how and when vulnerabilities are disclosed and fixed, how their prevalence changes over time, and how vulnerable packages expose their (transitive) dependents to vulnerabilities. Among many findings, they observed an increase in vulnerabilities in npm, but also a faster disclosure for RubyGems. Moreover, vulnerabilities in npm tend to affect fewer package releases. Alfadel et al. [8] found similar vulnerability characteristics in PyPI and npm, as well as divergences that could be attributed to specific PyPI policies. They empirically studied 1,396 vulnerability reports affecting 698 PyPI packages. Focusing on 2,224 Python projects they observed that more vulnerabilities are discovered over time, and a large portion ($> 40\%$) are only fixed after having been publicly announced. Moreover, more than half of the dependent projects rely on at least one vulnerable package, and it requires seven months to update to a non-vulnerable version.

Tool support. Many tools are available to help developers detect and resolve security weaknesses in vulnerable dependencies. One of them is GitHub's Dependabot, which issues pull requests to automatically update vulnerable dependencies, providing an effective platform for increasing awareness of dependency vulnerabilities and mitigating vulnerability threats. Alfadel et al. [49] studied the degree to which developers adopt Dependabot by investigating 2,904 OSS JS projects. They observed that a majority of security-related pull requests are accepted, often merged within a day. The severity of the dependency vulnerability and the potential risk of breaking changes were not strongly correlated with the time to merge these pull requests. Mohayeji et al. [50] empirically studied receptivity to Dependabot security updates in JS projects, observing that developers tend to delegate the task of fixing vulnerable dependencies and merge the majority of recommended security updates within several days. This is considerably faster than fixing vulnerabilities manually, which often takes up to several months. Similar to earlier research, Alfadel et al. [51] empirically analysed vulnerable dependencies in 6,546 JS applications, observing that 4.63% of them were exposed to dependencies with publicly known vulnerabilities, even if a fix was available in 90.8% of the cases. They proposed DepReveal, a tool to help developers better understand vulnerabilities in their application dependencies and to plan their project maintenance. Wang et al. [52] studied 356,283 active npm packages, observing that, in their latest release, 20% of them still introduce vulnerabilities via transitive dependencies despite the involved vulnerable packages already fixed the vulnerability for over a year. They empirically studied and distilled the remediation strategies to mitigate the fix propagation lag. Based on this, they developed Plumber, a tool to derive customised remediation suggestions for pivotal packages. The tool received positive feedback from many well-known projects.

So-called *Software Composition Analysis (SCA)* tools are being increasingly adopted by practitioners to keep track of potential security risks due to vulnerable dependencies on third-party libraries. Most

SCA tools are based on static and/or dynamic analysis of the project dependency graph. Imtiaz et al. [90] compared nine industry-leading SCA tools on a case study. They observed important variations in the accuracy of vulnerability reporting, suggesting that practitioners should not rely on any single tool, and that SCA tools need to achieve higher precision by avoiding false positives. Dietrich et al. [91] focused on the inverse problem of false negatives (i.e., low recall), when SCA tools miss dependencies on vulnerable components. They found empirical evidence of this for the Maven ecosystem, where somehow obfuscated clones of vulnerable components are deployed in Maven Central, but not marked as vulnerable in vulnerability databases.

3.7. Supply chain attacks

Package dependency networks are part of the wider concept of *software supply chains* [55, 56, 58]. Assuring their security is crucial to avoid *supply chain attacks* such as the malicious update of the SolarWinds Orion monitoring software, shipping a delayed-activation trojan horse that affected thousands of organisations, including the US government [92]. Ohm et al. [54] analysed 174 malicious software packages in npm, PyPI, and RubyGems that were used in real-world supply chain attacks. Nonprofit foundations such as OWASP aim to improve software security in various ways. Related to package dependencies in the software supply chain, they reported in 2022 *Dependency Chain Abuse* to be one of the top 10 CI/CD security risks[3]. It comes with four main attack vectors: (1) *Dependency confusion* aims to to trick clients into downloading a malicious package from a public repository rather than a private internal package with the same name; (2) *Dependency hijacking* aims to compromise clients that upgrade their dependency version to a more recent, malicious version; (3) *Typosquatting* aims to mislead clients into using a malicious package that has a very similar name as the intended package; (4) *Brandjacking* aims to mislead clients in depending on malicious packages that have the characteristics of packages of a trusted brand.

To mitigate supply chain attacks, so-called *software bills of materials* (SBOM) have been proposed as complete, formally structured lists of all software components present in a software product, including their licenses, versions, security vulnerabilities, and vendors.[4] SBOMs are imposed or recommended by US Executive Order 14028 [93] and the EU Cyber Resilience Act [94] to facilitate the transparent management of the software supply chain. Nocera et al. [95] observed a low adoption of SBOM in public GitHub repositories of OSS projects, but with an increasing trend. Although there are significant efforts from academia and industry to facilitate SBOM development, it is still unclear how practitioners perceive SBOMs and what are the challenges of adopting SBOMs in practice. Xia et al. [96] conducted a survey and interviews with SBOM practitioners to understand the current state of SBOM practice, tooling support and concerns for SBOM. They identified several open challenges that need to be further studied, mitigated and addressed. In a similar vein, Stalnaker et al. [97] surveyed 138 practitioners and identified 12 major challenges concerning the creation and use of SBOMs. They propose and discuss four actionable solutions to these challenges, and provide suggestions for future research and development.

Tool support. SBOMs enhances vulnerability detection and facilitates license compliance (see Section 3.11), through the use of SBOM generating tools such as Trivy, Syft, Microsoft's sbom-tool, and GitHub's Dependency Graph. A prerequisite is that such tools achieve full precision and correctness. Unfortunately, Yu et al. [98] observed that the SBOMs generated by these tools were inconsistent and contained dependency omissions, leading to incomplete and perhaps erroneous SBOMs. They consequently proposed best practices for SBOM generation and introduced a benchmark to steer the development of more robust SBOM generators. In a similar vein, [60] observed a high variability in vulnerability reporting by different SBOM generators. Tools such as OWASP Dependency-Track[5] leverage the capabilities of SBOM to identify and reduce risk in the software supply chain.

Solutions. Because secure software supply chains can be hard to attain in practice, the Linux Foun-

[3] https://owasp.org/www-project-top-10-ci-cd-security-risks/CICD-SEC-03-Dependency-Chain-Abuse
[4] The most common SBOM formats are CycloneDX and SPDX.
[5] https://dependencytrack.org

dation proposes *Supply chain Levels for Software Artifacts (SLSA)* [6] as a set of guidelines for supply chain security. Higher levels come with increasing security guarantees, with SLSA L3 providing the highest degree of confidence that the generated SBOM is precise, accurate and has not been tampered with. One way to achieve it is by resorting to the solution of *reproducible builds* [57, 59]. They ensure that, given the same source code, build environment and build instructions, bitwise identical copies of all artifacts are created. The functional package managers mentioned in Section 3.4 are instances of this solution. By requiring all dependencies to be declared upfront, they offer a robust solution to dependency management issues, ultimately supporting more reliable software supply chains.

3.8. Library deprecation and migration

Package registries such as npm or Cargo allow deprecating package releases, e.g., when a specific release is known to be vulnerable, faulty or incompatible. The solution would be to upgrade or downgrade one's dependencies to that package [31, 35]. Package maintainers could also decide to fully deprecate the package if they decide for some reason to cease maintaining it. In that case, the best possible alternative would be to replace one's dependency on that package with alternative packages. Sticking to the deprecated one is likely to lead to security vulnerabilities for which no fixes will be provided. Cogo et al. [61] quantified deprecation in the npm registry, observing that 3.7% of all packages have at least one deprecated release, 31% of those packages do not have any replacement release, and 66% of such packages even deprecated all their releases. They found that 27% of the client packages directly (and 54% transitively) depend on at least one deprecated release.

Given that *library migration* is often the only way to deal with deprecated packages, it has received quite some attention from researchers. Based on an empirical analysis of commits in 19K+ Java projects on GitHub, He et al. [62] identified 14 different migration reasons, of which the most important ones were lack of maintenance, known bugs and vulnerabilities, lack of usability, missing features, poor performance, lack of popularity, complexity, integration problems, and licensing issues. He et al. [63] improved upon existing techniques to recommend the most appropriate library to migrate to, based on filtering approaches that leverage the wisdom of the crowd. Candidate libraries are ranked based on a series of metrics. In a similar vein, Mujahid et al. [65] proposed an approach to automatically identify npm packages requiring replacement, and suggesting alternatives based on wisdom of the crowd. An evaluation showed that 96% of the suggested alternatives were accurate, and 67% of surveyed JS developers responded they consider using these suggestions in the future. Gu et al. [64] opened up the analysis of library migrations by comparing Maven, npm and PyPI. Library migrations were found to be prevalent and similar in nature in all three package registries. For PyPI specifically, an increasing competition was observed between libraries.

3.9. Depending on trivial libraries

Trivial libraries are packages that are either very small or provide little functionality. Depending on such trivial libraries may unnecessarily introduce a high dependency overhead. Some trivial libraries have even led to major incidents, as was the case when the leftpad package was removed from npm in March 2016. This single point of failure caused a breakdown of popular web applications including Facebook and Netflix. It drove researchers to scrutinise the necessity and prevalence of *trivial packages* in package registries.

In an empirical study of trivial packages in the npm and PyPI package registries, Abdalkareem et al. [66] observed that such packages are quite common, making up 16.0% of npm and 10.5% of PyPI. 125 surveyed developers who use trivial packages reported using them because they were *perceived* to be well implemented and tested pieces of code. Contrary to developers' beliefs, only around 28% of npm and 49% of PyPI trivial packages were found to have tests. Surveyed developers were also concerned by the maintenance overhead of depending on trivial packages, which was quantitatively confirmed since 18.4% of the npm and 2.9% of the PyPI trivial packages had more than 20 dependencies. Chen et al. [67]

[6] https://slsa.dev

conducted another survey with 59 JS developers who publish trivial npm packages. The main reported reasons for publishing them were to provide reusable components, testing and documentation, and separation of concerns. On the downside, the surveyed developers reported the challenge of maintaining multiple packages, dependency hell, and the increase of duplicated packages. As a way to cope with these challenges, they suggested grouping trivial packages. This could lead to a reduction in the number of dependencies by approximately 13%. Chowdhury et al. [68] empirically studied the project usage and ecosystem usage of trivial npm packages. They reported that removing a trivial package can impact approximately 29% of the ecosystem. They also revealed that trivial packages are being actively used in central JS files of software projects.

3.10. Abandoned and unmaintained dependencies

Proper dependency management does not suffice to consider technical factors only. Human factors are equally important. Relying on reusable OSS packages all too often ignores the considerable effort required by package maintainers to keep them up to date and fixing reported issues in a timely manner [58]. By depending on reusable packages, one implicitly trusts their associated OSS community. This can be problematic if this community –which is often driven by volunteers– is not sufficiently active or responsive, is too small (e.g., packages maintained by a single developer), or if packages become unmaintained due to maintainer abandonment [69, 71]. Champion and Hill [99] studied the *underproduction* of software projects, where the supply of labour for maintaining them is too small to satisfy the demand of the project users. Analysing 21K+ packages with 461K+ bugs in the Debian distribution they proposed a method to identify underproductive software packages.

Miller et al. [100] quantitatively observed that abandonment is common among widely-used npm libraries, that hundreds of thousands of downstream projects were directly exposed to this abandonment, and that most of these exposed projects never remove or replace an abandoned dependency. Factors that make it more likely to remove abandoned dependencies are more mature project government and dependency management practices, as well as the explicit public announcement of abandoned packages.

Abandoned, and therefore unmaintained, libraries can be a source of highly impactful security threats. An incident in March 2024 was the compromised XZ-Utils software compression package for Linux distributions. Its original well-intentioned maintainer who was no longer able to fully maintain the package. After gaining this maintainer's trust during a period of two years, a malicious attacker took over its maintenance, and introduced a backdoor to authorise remote code execution on affected systems. A similar situation happened in November 2018 for the event-stream npm package, whose maintenance was unknowingly handed over to a malicious developer who subsequently modified the package to include code for stealing crypto-coins [101].

These incidents underscore the importance of rigorous vetting processes to reduce the risk of social engineering that can compromise software integrity. To avoid such incidents, it is of crucial importance to maintain a healthy and sustainable community that is able to attract and retain motivated contributors [102], and to ensure that they have the necessary financial and computing resources to maintain their code. In case packages get abandoned, the ecosystem should rely on community package maintenance organizations (CPMOs), consisting of volunteers that steward and maintain abandoned packages [72].

3.11. Incompatible licenses

A final challenge stems from the incompatibility of OSS licenses that determine the terms and conditions to use or modify reusable libraries within one's own software. A plethora of licenses exist[7], making it increasingly challenging for developers to select an appropriate license for their projects and to ensure that they are complying with the terms of those licenses. Given that the use of incompatible licenses can lead to legal disputes, it is essential to ensure license compatibility when reusing OSS packages, and it is a frequent reason for migrating to an alternative library [62].

[7]The SPDX open standard for SBOM supports 600+ licenses, see https://spdx.org/licenses/

Xu et al. [75] conducted an empirical study of license incompatibilities and their remediation practices in the PyPI ecosystem for Python. They found that 7% of the package releases have license incompatibilities and 61% of them are due to transitive dependencies. They also identified five remediation strategies, including migrating to another library, removing the dependency, pinning versions or changing the license of the dependent package. Inspired by their findings, they proposed Silence, an approach to recommend license incompatibility remediations with minimal costs. Wu et al. [76] conducted an empirical study of license usage, incompatibility and evolution in 33M+ packages across five package registries (Maven, npm, PyPI, RubyGems and Cargo), observing both similarities and differences in license usage across the five registries.

Tool support. In order to detect and resolve incompatible licenses, automated tools are needed. Xu et al. [73] proposed LiDetector, a learning-based tool to detect license incompatibilities. An empirical evaluation of 1,846 projects revealed that $> 72\%$ of the projects suffer from license incompatibility, including popular ones such as the MIT License and the Apache License. In a follow-up work [74], they presented LiResolver, a tool to resolve license incompatibility issues for OSS.

4. Conclusion

The reliance on reusable OSS libraries distributed through package registries continues to increase, and the size of their dependency networks follows suit. Based on recent empirical research we compiled a catalogue of frequently studied dependency-related issues that these registries suffer from. This overview of dependency challenges can be used as a starting point for researchers and practitioners that would like to delve deeper into the empirical research on OSS package dependency networks.

The extent to which they package registries suffer from dependency issues, and the way they deal with them, is highly ecosystem-dependent. There is no single "one fits all" solution given the diversity of targeted programming languages and communities involved in these ecosystems [3]. Versioning policies such as semantic versioning have seen an increase in adoption. The same holds for the indispensable suite of tools to detect and fix dependency issues of various nature. The use of appropriate policies and tools should be promoted further, and tools need to be improved and completed on a continuous basis (e.g., by including dynamic analysis techniques) to cope with the rapidly evolving OSS landscape.

Most of the empirical research focused on npm and Maven, and to a lesser extent on PyPI. This is not surprising, since they come with the largest package registries, providing reusable libraries for JS, Java and Python respectively, the top three of most popular programming languages [103]. It is more surprising that some large package registries and popular programming languages have not received the attention they deserve. This is the case for NuGet, for instance, despite being the fourth-largest package registry (with 539K packages) and despite the popularity of C# and the .NET platform.

With the increasing reuse of OSS packages comes an increasing reliance on software supply chains, hence research on software supply chain security and SBOMs is on the rise. Security issues in package registries can also propagate well beyond the boundaries of the package registry, affecting other ecosystems with an even larger attack surface. As an example, the GitHub Actions workflow ecosystem (created in 2018) has been shown to suffer from vulnerabilities due to its dependence on reusable Actions implemented as JS components depending on npm packages [104].

Acknowledgments

We thank Stefano Zacchiroli, Jens Dietrich and Pol Dell'Aiera for their constructive feedback on this manuscript. This work is supported by the Fonds de la Recherche Scientifique – FNRS under grant numbers J.0147.24, T.0149.22, and F.4515.23, as well as Action de Recherche Concertée ARC-21/25 UMONS3 financed by the Ministère de la Communauté française - Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique.

References

[1] A. Decan, T. Mens, P. Grosjean, An empirical comparison of dependency network evolution in seven software packaging ecosystems, Emp. Softw. Eng. 24 (2019). doi:10.1007/s10664-017-9589-y.

[2] J. Dietrich, D. Pearce, J. Stringer, A. Tahir, K. Blincoe, Dependency versioning in the wild, in: Int'l Conf. Mining Software Repositories (MSR), 2019, pp. 349–359. doi:10.1109/MSR.2019.00061.

[3] C. Bogart, C. Kästner, J. Herbsleb, F. Thung, When and how to make breaking changes: Policies and practices in 18 open source software ecosystems, ACM Trans. Softw. Eng. Methodol. 30 (2021). doi:10.1145/3447245.

[4] C. Soto-Valero, N. Harrand, M. Monperrus, B. Baudry, A comprehensive study of bloated dependencies in the Maven ecosystem, Emp. Softw. Eng. 26 (2021). doi:10.1007/s10664-020-09914-8.

[5] L. Ochoa, T. Degueule, J.-R. Falleri, J. Vinju, Breaking bad? Semantic versioning and impact of breaking changes in Maven Central, Emp. Softw. Eng. 27 (2022). doi:10.1007/s10664-021-10052-y.

[6] F. Reyes, Y. Gamage, G. Skoglund, B. Baudry, M. Monperrus, BUMP: A benchmark of reproducible breaking dependency updates, in: Int'l Conf. Software Analysis, Evolution and Reengineering (SANER), 2024. doi:10.1109/SANER60148.2024.00024.

[7] M. Valiev, B. Vasilescu, J. Herbsleb, Ecosystem-level determinants of sustained activity in open-source projects: a case study of the PyPI ecosystem, in: Joint Meeting on European Softw. Eng. Conf. and Symp. Foundations of Softw. Eng., ACM, 2018, pp. 644–655. doi:10.1145/3236024.3236062.

[8] M. Alfadel, D. E. Costa, E. Shihab, Empirical analysis of security vulnerabilities in Python packages, Emp. Softw. Eng. 28 (2023). doi:10.1007/s10664-022-10278-4.

[9] Y. Cao, L. Chen, W. Ma, Y. Li, Y. Zhou, L. Wang, Towards better dependency management: A first look at dependency smells in Python projects, IEEE Trans. Softw. Eng. 49 (2023). doi:10.1109/TSE.2022.3191353.

[10] K. Rahkema, D. Pfahl, R. Ramler, Analysis of library dependency networks of package managers used in iOS development, in: Int'l Conf. Mobile Software Engineering and Systems (MOBILESoft), 2023. doi:10.1109/MOBILSoft59058.2023.00010.

[11] S. Kolak, A. Afzal, C. Le Goues, M. Hilton, C. S. Timperley, It takes a village to build a robot: An empirical study of the ROS ecosystem, in: Int'l Conf. Software Maintenance and Evolution (ICSME), 2020. doi:10.1109/ICSME46990.2020.00048.

[12] M. Mora-Cantallops, S. Sánchez-Alonso, E. García-Barriocanal, A complex network analysis of the Comprehensive R Archive Network (CRAN) package ecosystem, J. Syst. Softw. 170 (2020). doi:10.1016/j.jss.2020.110744.

[13] M. Mora-Cantallops, M.-A. Sicilia, E. García-Barriocanal, S. Sánchez-Alonso, Evolution and prospects of the Comprehensive R Archive Network (CRAN) package ecosystem, J. Softw.: Evolution and Process 32 (2020). doi:10.1002/smr.2270.

[14] R. G. Kula, D. M. German, A. Ouni, T. Ishio, K. Inoue, Do developers update their library dependencies?, Emp. Softw. Eng. 23 (2018). doi:10.1007/s10664-017-9521-5.

[15] A. Decan, T. Mens, E. Constantinou, On the evolution of technical lag in the npm package dependency network, in: Int'l Conf. Software Maintenance and Evolution (ICSME), 2018. doi:10.1109/ICSME.2018.00050.

[16] A. Zerouali, E. Constantinou, T. Mens, G. Robles, J. González-Barahona, An empirical analysis of technical lag in npm package dependencies, in: Int'l Conf. Software Reuse (ICSR), 2018. doi:10.1007/978-3-319-90421-4_6.

[17] T. Lauinger, A. Chaabane, C. Wilson, Thou shalt not depend on me: A look at JavaScript libraries in the wild, Queue 16 (2018) 62–82. doi:10.1145/3194653.3205288.

[18] A. Zerouali, T. Mens, J. Gonzalez-Barahona, A. Decan, E. Constantinou, G. Robles, A formal framework for measuring technical lag in component repositories and its application to npm, J. Softw.: Evolution and Process 31 (2019). doi:10.1002/smr.2157.

[19] J. Stringer, A. Tahir, K. Blincoe, J. Dietrich, Technical lag of dependencies in major package managers, in: Asia-Pacific Software Engineering Conf. (APSEC), 2020, pp. 228–237. doi:10.1109/APSEC51365.2020.00031.

[20] A. Zerouali, T. Mens, C. De Roover, On the usage of JavaScript, Python and Ruby packages in Docker Hub images, Sci. Comput. Prog. 207 (2021). doi:10.1016/j.scico.2021.102653.

[21] J. Hejderup, G. Gousios, Can we trust tests to automate dependency updates? A case study of Java projects, J. Syst. Softw. 183 (2022). doi:https://doi.org/10.1016/j.jss.2021.111097.

[22] A. Zerouali, R. Opdebeeck, C. De Roover, Helm charts for Kubernetes applications: evolution, outdatedness and security risks, in: Int'l Conf. Mining Software Repositories (MSR), 2023. doi:10.1109/MSR59073.2023.00078.

[23] A. Decan, t. Mens, H. Onsori Delicheh, On the outdatedness of workflows in the GitHub Actions ecosystem, J. Syst. Softw. 206 (2023). doi:10.1016/j.jss.2023.111827.

[24] G. Mezzetti, A. Möller, M. T. Torp, Type regression testing to detect breaking changes in Node.js libraries, in: European Conf. Object-Oriented Programming (ECOOP), 2018. doi:10.4230/LIPIcs.ECOOP.2018.7.

[25] A. Møller, M. T. Torp, Model-based testing of breaking changes in Node.js libraries, in: Joint European Software Engineering Conf. and Symp. Foundations of Software Engineering, 2019. doi:10.1145/3338906.3338940.

[26] S. Mujahid, R. Abdalkareem, E. Shihab, S. McIntosh, Using others' tests to identify breaking updates, in: Int'l Conf. Mining Software Repositories (MSR), 2020. doi:10.1145/3379597.3387476.

[27] A. Brito, M. T. Valente, L. Xavier, A. Hora, You broke my code: understanding the motivations for breaking changes in APIs, Emp. Softw. Eng. 25 (2020). doi:10.1007/s10664-019-09756-z.

[28] L. Zhang, C. Liu, Z. Xu, S. Chen, L. Fan, B. Chen, Y. Liu, Has my release disobeyed semantic versioning? Static detection based on semantic differencing, in: Int'l Conf. Automated Software Engineering (ASE), ACM, 2022. doi:10.1145/3551349.3556956.

[29] D. Venturini, F. R. Cogo, I. Polato, M. A. Gerosa, I. S. Wiese, I depended on you and you broke me: An empirical study of manifesting breaking changes in client packages, ACM Trans. Softw. Eng. Methodol. 32 (2023). doi:10.1145/3576037.

[30] P. Lam, J. Dietrich, D. J. Pearce, Putting the semantics into semantic versioning, in: SIGPLAN Int'l Symp. New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), 2020. doi:10.1145/3426428.3426922.

[31] F. R. Cogo, G. A. Oliva, A. E. Hassan, An empirical study of dependency downgrades in the npm ecosystem, IEEE Trans. Softw. Eng. 47 (2021) 2457–2470. doi:10.1109/TSE.2019.2952130.

[32] A. Decan, T. Mens, What do package dependencies tell us about semantic versioning?, IEEE Trans. Softw. Eng. 47 (2021). doi:10.1109/TSE.2019.2918315.

[33] A. Decan, T. Mens, Lost in zero space: An empirical comparison of 0.y.z releases in software package distributions, Sci. Comput. Prog. 208 (2021). doi:10.1016/j.scico.2021.102656.

[34] A. Decan, T. Mens, A. Zerouali, C. De Roover, Back to the past: Analysing backporting practices in package dependency networks, IEEE Trans. Softw. Eng. 48 (2022). doi:10.1109/TSE.2021.3112204.

[35] A. J. Jafari, D. E. Costa, E. Shihab, R. Abdalkareem, Dependency update strategies and package characteristics, ACM Trans. Softw. Eng. Methodol. 32 (2023). doi:10.1145/3603110.

[36] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, R. Treinen, Managing the complexity of large free and open source package-based software distributions, in: Int'l Conf. Automated Software Engineering (ASE), 2006, pp. 199–208. doi:10.1109/ASE.2006.49.

[37] P. Abate, R. di Cosmo, R. Treinen, S. Zacchiroli, Dependency solving: A separate concern in component evolution management, J. Syst. Softw. 85 (2012). doi:10.1016/j.jss.2012.02.018.

[38] P. Abate, R. Di Cosmo, G. Gousios, S. Zacchiroli, Dependency solving is still hard, but we are getting better at it, in: Int'l Conf. Software Analysis, Evolution and Reengineering (SANER), 2020. doi:10.1109/SANER48275.2020.9054837.

[39] D. Pinckney, F. Cassano, A. Guha, J. Bell, M. Culpo, T. Gamblin, Flexible and optimal dependency management via Max-SMT, in: Int'l Conf. Software Engineering (ICSE), 2023. doi:10.1109/ICSE48619.2023.00124.

[40] A. J. Jafari, D. E. Costa, R. Abdalkareem, E. Shihab, N. Tsantalis, Dependency smells in JavaScript projects, IEEE Trans. Softw. Eng. 48 (2022). doi:10.1109/TSE.2021.3106247.

[41] C. Soto-Valero, T. Durieux, N. Harrand, B. Baudry, Coverage-based debloating for Java bytecode, ACM Trans. Softw. Eng. Methodol. 32 (2023). doi:10.1145/3546948.

[42] N. R. Weeraddana, M. Alfadel, S. McIntosh, Dependency-induced waste in continuous integration: An empirical study of unused dependencies in the npm ecosystem, ACM Softw. Eng. 1 (2024). doi:10.1145/3660823.

[43] B. Chinthanet, R. G. Kula, S. McIntosh, T. Ishio, A. Ihara, K. Matsumoto, Lags in the release, adoption, and propagation of npm vulnerability fixes, Emp. Softw. Eng. 26 (2021). doi:10.1007/s10664-021-09951-x.

[44] A. Decan, T. Mens, E. Constantinou, On the impact of security vulnerabilities in the npm package dependency network, in: Int'l Conf. Mining Software Repositories (MSR), 2018. doi:10.1145/3196398.3196401.

[45] G. A. A. Prana, A. Sharma, L. K. Shar, D. Foo, A. E. Santosa, A. Sharma, D. Lo, Out of sight, out of mind? how vulnerable dependencies affect open-source projects, Emp. Softw. Eng. 26 (2021). doi:10.1007/s10664-021-09959-3.

[46] A. Zerouali, T. Mens, A. Decan, C. De Roover, On the impact of security vulnerabilities in the npm and RubyGems dependency networks, Emp. Softw. Eng. 27 (2022). doi:10.1007/s10664-022-10154-1.

[47] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, X. Peng, Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem, in: Int'l Conf. Software Engineering (ICSE), 2022. doi:10.1145/3510003.3510142.

[48] M. Zimmermann, C.-A. Staicu, C. Tenny, M. Pradel, Small world with high risks: A study of security threats in the npm ecosystem, in: USENIX Security Symp., 2019.

[49] M. Alfadel, D. E. Costa, E. Shihab, M. Mkhallalati, On the use of Dependabot security pull requests, in: Int'l Conf. Mining Software Repositories (MSR), 2021. doi:10.1109/MSR52588.2021.00037.

[50] H. Mohayeji, A. Agaronian, E. Constantinou, N. Zannone, A. Serebrenik, Investigating the resolution of vulnerable dependencies with Dependabot security updates, in: Int'l Conf. Mining Software Repositories (MSR), 2023. doi:10.1109/MSR59073.2023.00042.

[51] M. Alfadel, D. E. Costa, E. Shihab, B. Adams, On the discoverability of npm vulnerabilities in Node.js projects, ACM Trans. Softw. Eng. Methodol. 32 (2023). URL: https://doi.org/10.1145/3571848. doi:10.1145/3571848.

[52] Y. Wang, P. Sun, L. Pei, Y. Yu, C. Xu, S.-C. Cheung, H. Yu, Z. Zhu, Plumber: Boosting the propagation of vulnerability fixes in the npm ecosystem, IEEE Trans. Softw. Eng. 49 (2023). doi:10.1109/TSE.2023.3243262.

[53] M. Ohm, A. Sykosch, M. Meier, Towards detection of software supply chain attacks by forensic artifacts, in: Int'l Conf. Availability, Reliability and Security (ARES), ACM, 2020. doi:10.1145/3407023.3409183.

[54] M. Ohm, H. Plate, A. Sykosch, M. Meier, Backstabber's knife collection: A review of open source software supply chain attacks, in: Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), Springer, 2020. doi:10.1007/978-3-030-52683-2_2.

[55] R. Duan, O. Alrawi, R. Pai Kasturi, R. Elder, B. Saltaformaggio, W. Lee, Towards measuring supply chain attacks on package managers for interpreted languages, in: Network and Distributed Systems Security (NDSS) Symp., 2021. doi:10.14722/ndss.2021.23055.

[56] W. Enck, L. Williams, Top five challenges in software supply chain security: Observations from 30 industry and government organizations, IEEE Security & Privacy 20 (2022) 96–100. doi:10.1109/MSEC.2022.3142338.

[57] C. Lamb, S. Zacchiroli, Reproducible builds: Increasing the integrity of software supply chains,

IEEE Software 39 (2022) 62–70. doi:10.1109/MS.2021.3073045.

[58] D. Wermke, J. H. Klemmer, N. Wöhler, J. Schmüser, H. S. Ramulu, Y. Acar, S. Fahl, "Always contribute back": A qualitative study on security challenges of the open source supply chain, in: Symp. Security and Privacy (SP), 2023. doi:10.1109/SP46215.2023.10179378.

[59] M. Fourné, D. Wermke, W. Enck, S. Fahl, Y. Acar, It's like flossing your teeth: On the importance and challenges of reproducible builds for software supply chain security, in: IEEE Symp. Security and Privacy (SP), 2023, pp. 1527–1544. doi:10.1109/SP46215.2023.10179320.

[60] E. O'Donoghue, B. Boles, C. Izurieta, A. M. Reinhold, Impacts of software bill of materials (SBOM) generation on vulnerability detection, in: Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED), ACM, 2024.

[61] F. R. Cogo, G. A. Oliva, A. E. Hassan, Deprecation of packages and releases in software ecosystems: A case study on npm, IEEE Trans. Softw. Eng. 48 (2022) 2208–2223. doi:10.1109/TSE.2021.3055123.

[62] H. He, R. He, H. Gu, M. Zhou, A large-scale empirical study on Java library migrations: prevalence, trends, and rationales, in: Joint European Software Engineering Conf. and Symp. Foundations of Software Engineering (ESEC/FSE), 2021. doi:10.1145/3468264.3468571.

[63] H. He, Y. Xu, Y. Ma, Y. Xu, G. Liang, M. Zhou, A multi-metric ranking approach for library migration recommendations, in: Int'l Conf. Software Analysis, Evolution and Reengineering (SANER), 2021. doi:10.1109/SANER50967.2021.00016.

[64] H. Gu, H. He, M. Zhou, Self-admitted library migrations in Java, JavaScript, and Python packaging ecosystems: A comparative study, in: Int'l Conf. Software Analysis, Evolution and Reengineering (SANER), 2023. doi:10.1109/SANER56733.2023.00064.

[65] S. Mujahid, D. E. Costa, R. Abdalkareem, E. Shihab, Where to go now? Finding alternatives for declining packages in the npm ecosystem, in: Int'l Conf. Automated Software Engineering (ASE), 2023. doi:10.1109/ASE56229.2023.00119.

[66] R. Abdalkareem, V. Oda, S. Mujahid, E. Shihab, On the impact of using trivial packages: an empirical case study on npm and PyPI, Emp. Softw. Eng. 25 (2020) 1168–1204. doi:10.1007/s10664-019-09792-9.

[67] X. Chen, R. Abdalkareem, S. Mujahid, E. Shihab, X. Xia, Helping or not helping? Why and how trivial packages impact the npm ecosystem, Emp. Softw. Eng. 26 (2021). doi:10.1007/s10664-020-09904-w.

[68] M. A. R. Chowdhury, R. Abdalkareem, E. Shihab, B. Adams, On the untriviality of trivial packages: An empirical study of npm JavaScript packages, IEEE Trans. Softw. Eng. 48 (2021). doi:10.1109/TSE.2021.3068901.

[69] G. Avelino, E. Constantinou, M. T. Valente, A. Serebrenik, On the abandonment and survival of open source projects: An empirical investigation, in: Int'l Symp. Empirical Software Engineering and Measurement (ESEM), 2019. doi:10.1109/ESEM.2019.8870181.

[70] R. Kaur, K. K. Chahal, Exploring factors affecting developer abandonment of open source software projects, J. Softw.: Evolution and Process 34 (2022). doi:10.1002/smr.2484.

[71] C. Miller, C. Kästner, B. Vasilescu, "We feel like we're winging it:" A study on navigating open-source dependency abandonment, in: Joint European Software Engineering Conf. and Symp. Foundations of Software Engineering (ESEC/FSE), 2023. doi:10.1145/3611643.3616293.

[72] T. Zimmermann, J.-R. Falleri, A grounded theory of community package maintenance organizations, Emp. Softw. Eng. 28 (2023). doi:10.1007/s10664-023-10337-4.

[73] S. Xu, Y. Gao, L. Fan, Z. Liu, Y. Liu, H. Ji, LiDetector: License incompatibility detection for open source software, ACM Trans. Softw. Eng. Methodol. 32 (2023). doi:10.1145/3518994.

[74] S. Xu, Y. Gao, L. Fan, L. Li, X. Cai, Z. Liu, LiResolver: License incompatibility resolution for open source software, in: SIGSOFT Int'l Symp. Software Testing and Analysis (ISSTA), 2023. doi:10.1145/3597926.3598085.

[75] W. Xu, H. He, K. Gao, M. Zhou, Understanding and remediating open-source license incompatibilities in the PyPI ecosystem, in: Int'l Conf. Automated Software Engineering (ASE), 2023. doi:10.1109/ASE56229.2023.00175.

[76] J. Wu, L. Bao, X. Yang, X. Xia, X. Hu, A large-scale empirical study of open source license usage: Practices and challenges, in: Int'l Conf. Mining Software Repositories (MSR), ACM, 2024, pp. 595–606. doi:10.1145/3643991.3644900.

[77] R. He, H. He, Y. Zhang, M. Zhou, Automating dependency updates in practice: An exploratory study on GitHub Dependabot, IEEE Trans. Softw. Eng. 49 (2023). doi:10.1109/TSE.2023.3278129.

[78] B. Rombaut, F. R. Cogo, B. Adams, A. E. Hassan, There's no such thing as a free lunch: Lessons learned from exploring the overhead introduced by the Greenkeeper dependency bot in npm, ACM Trans. Softw. Eng. Methodol. 32 (2023). doi:10.1145/3522587.

[79] A. Dann, B. Hermann, E. Bodden, UPCY: Safely updating outdated dependencies, in: Int'l Conf. Software Engineering (ICSE), 2023, pp. 233–244. doi:10.1109/ICSE48619.2023.00031.

[80] D. Jayasuriya, V. Terragni, J. Dietrich, S. Ou, K. Blincoe, Understanding breaking changes in the wild, in: Int'l Symp. Software Testing and Analysis (ISSTA), ACM, 2023, pp. 1433–1444. doi:10.1145/3597926.3598147.

[81] D. Jayasuriya, V. Terragni, J. Dietrich, K. Blincoe, Understanding the impact of APIs behavioral breaking changes on client applications, Proc. ACM Softw. Eng. 1 (2024). doi:10.1145/3643782.

[82] Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu, Y. Xiong, How do Python framework APIs evolve? An exploratory study, in: Int'l Conf. Software Analysis, Evolution and Reengineering (SANER), IEEE, 2020. doi:10.1109/SANER48275.2020.9054800.

[83] Z. Jia, S. Li, T. Yu, C. Zeng, E. Xu, X. Liu, J. Wang, X. Liao, DepOwl: Detecting dependency bugs to prevent compatibility failures, in: Int'l Conf. Software Engineering (ICSE), 2021. doi:10.1109/ICSE43902.2021.00021.

[84] X. Du, J. Ma, AexPy: Detecting API breaking changes in Python packages, in: Int'l Symp. Software Reliability Engineering (ISSRE), 2022. doi:10.1109/ISSRE55969.2022.00052.

[85] A. Brito, L. Xavier, A. Hora, M. T. Valente, APIDiff: Detecting API breaking changes, in: Int'l Conf. Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 507–511. doi:10.1109/SANER.2018.8330249.

[86] K. Jezek, J. Dietrich, API evolution and compatibility: A data corpus and tool evaluation, J. Object Technol. 16 (2017). doi:10.5381/jot.2017.16.4.a2.

[87] D. Dig, R. Johnson, How do APIs evolve? A story of refactoring, J. Software Maintenance and Evolution: Research and Practice 18 (2006). doi:10.1002/smr.328.

[88] A. Sharma, Npm package with 3 million weekly downloads had a severe vulnerability, Ars Technica, 2021. URL: https://arstechnica.com/information-technology/2021/09/npm-package-with-3-million-weekly-downloads-had-a-severe-vulnerability/.

[89] R. Hiesgen, M. Nawrocki, T. C. Schmidt, M. Wählisch, The race to the vulnerable: Measuring the Log4j shell incident, in: Network Traffic Measurement and Analysis Conf. (TMA), 2022. doi:10.48550/arXiv.2205.02544.

[90] N. Imtiaz, S. Thorn, L. Williams, A comparative study of vulnerability reporting by software composition analysis tools, in: Int'l Symp. Empirical Software Engineering and Measurement (ESEM), ACM, 2021. doi:10.1145/3475716.3475769.

[91] J. Dietrich, S. Rasheed, A. Jordan, T. White, On the security blind spots of software composition analysis, in: ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED), 2024. doi:10.48550/arXiv.2306.05534.

[92] R. Alkhadra, J. Abuzaid, M. AlShammari, N. Mohammad, Solar Winds hack: In-depth analysis and countermeasures, in: Int'l Conf. Computing Communication and Networking Technologies (ICCCNT), 2021, pp. 1–7. doi:10.1109/ICCCNT51525.2021.9579611.

[93] J. Biden, Executive order 14028: Improving the nation's cybersecurity, 2021. URL: https://www.federalregister.gov/d/2021-10460.

[94] European Union, The cyber resilience act, 2022. URL: https://www.cyberresilienceact.eu/.

[95] S. Nocera, S. Romano, M. Di Penta, R. Francese, G. Scanniello, Software bill of materials adoption: A mining study from GitHub, in: Int'l Conf. Software Maintenance and Evolution (ICSME), 2023. doi:10.1109/ICSME58846.2023.00016.

[96] B. Xia, T. Bi, Z. Xing, Q. Lu, L. Zhu, An empirical study on software bill of materials: Where we stand and the road ahead, in: Int'l Conf. Software Engineering (ICSE), 2023. doi:10.1109/ICSE48619.2023.00219.

[97] T. Stalnaker, N. Wintersgill, O. Chaparro, M. Di Penta, D. M. German, D. Poshyvanyk, BOMs away! Inside the minds of stakeholders: A comprehensive study of bills of materials for software systems, in: Int'l Conf. Software Engineering (ICSE), 2024. doi:10.1145/3597503.3623347.

[98] S. Yu, W. Song, X. Hu, H. Yin, On the correctness of metadata-based SBOM generation: A differential analysis approach, in: Int'l Conf. Dependable Systems and Networks (DSN), 2024. doi:10.1109/DSN58291.2024.00018.

[99] K. Champion, B. M. Hill, Underproduction: An approach for measuring risk in open source software, in: Int'l Conf. Software Analysis, Evolution and Reengineering (SANER), 2021. doi:10.1109/SANER50967.2021.00043.

[100] C. Miller, M. Jahanshahi, A. Mockus, B. Vasilescu, C. Kästner, Understanding the response to open-source dependency abandonment in the npm ecosystem, in: Int'l Conf. Software Engineering (ICSE), IEEE/ACM, 2025.

[101] D. Goodin, Widely used open source software contained bitcoin-stealing backdoor, Ars Technica, 2018. URL: https://arstechnica.com/information-technology/2018/11/hacker-backdoors-widely-used-open-source-software-to-steal-bitcoin/.

[102] E. Constantinou, T. Mens, An empirical comparison of developer retention in the RubyGems and npm software ecosystems, Innovations in Systems and Software Engineering 13 (2017). doi:10.1007/s11334-017-0303-4.

[103] S. Cass, The top programming languages 2024, 2024. URL: https://spectrum.ieee.org/top-programming-languages-2024.

[104] H. Onsori Delicheh, A. Decan, T. Mens, Quantifying security issues in reusable JavaScript Actions in GitHub workflows, in: Int'l Conf. Mining Software Repositories (MSR), 2024. doi:10.1145/3643991.3644899.