On the evolution of technical lag in the npm package dependency network

Alexandre Decan, Tom Mens, Eleni Constantinou
Software Engineering Lab, *University of Mons*
Mons, Belgium

Abstract—Software packages developed and distributed through package managers extensively depend on other packages. These dependencies are regularly updated, for example to add new features, resolve bugs or fix security issues. In order to take full advantage of the benefits of this type of reuse, developers should keep their dependencies up to date by relying on the latest releases. In practice, however, this is not always possible, and packages lag behind with respect to the latest version of their dependencies. This phenomenon is described as technical lag in the literature. In this paper, we 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. We explore how technical lag increases over time, taking into account the release type and the use of package dependency constraints. We also discuss how technical lag can be reduced by relying on the semantic versioning policy.

Index Terms—package distribution, dependency network, technical lag, semantic versioning, software evolution, empirical software engineering

I. Introduction

Software developers are continuously confronted with the delicate choice between keeping their software in a stable, working state, and keeping it "as up-to-date as reasonable" w.r.t. external dependencies (e.g., libraries or external systems) in order to benefit from bug fixes, security fixes, and relevant new features. However, updating to more recent versions might lead to an increased risk of backward incompatible changes. Package maintainers advocate to investigate the impact of breaking changes in dependent packages before deciding to update them[1]. Depending on the number of dependencies, this can become an infeasible task. To capture this challenging balance between risks and opportunities of updating dependencies, Gonzalez-Barahona et al. [1] proposed the concept of *technical lag* for reflecting how outdated a software system is with respect to its upstream dependencies. Having a precise way of measuring technical lag allows software developers to make informed decisions on whether and when to update their outdated dependencies.

Technical lag can be particularly important in software package distributions, where packages depend on each other to use third-party functionality and facilitate the development process [2], [3]. Dependencies between packages are defined based on dependency constraints, which specify the version range that is allowed to be installed at any given point in time. Based on such constraints and a set of allowed installable versions, the latest allowed version of the required package is installed and used by a dependent package. Package dependency networks of software package managers have been shown to be brittle because of the large and increasing number of dependencies over time [4]. This is especially the case for the npm package distribution, witnessing an exponential growth in number of packages and dependencies [5]. Zerouali et al. provided preliminary evidence of technical lag in npm by analysing package dependencies on a yearly basis [2].

We go one step further, by analysing the continuous evolution of technical lag at the level of package dependencies and package releases, and relating it to the release type of both dependent and required packages. To this end, we focus on the following research questions:

- RQ_1 How many packages have technical lag?
- RQ_2 How long is the technical lag?
- RQ_3 How frequently are packages updated?
- RQ_4 When does technical lag increase?
- RQ_5 When does technical lag decrease?
- RQ_6 How could technical lag be reduced by proper use of semantic versioning?

By answering these questions, we aim to get a better understanding of technical lag throughout software packages in the npm distributions. These insights can help to bette manage and control technical lag, through tools to monitor and support package dependencies as well as through a more optimal usage of semantic versioning.

The remainder of this paper is structured as follows. Section II motivates the choice of npm as a case study, introduces the necessary terminology and research methodology and describes the dataset used. Sections III-VIII answer the research questions and Section IX discusses our findings. Section X explains the threats to validity of our work and Section XI presents related research. Section XII discusses the future work and Section XIII concludes the paper.

II. Methodology and Terminology

According to the 2018 Stack Overflow Developer Survey[2] to which over 100,000 developers participated, JavaScript is by far the most commonly used programming language

---

[1]"Not all scenarios will require you to update a packages as it could introduce breaking changes to your projects. Do the research first." (https://www.thepolyglotdeveloper.com/2015/03/check-update-outdated-npm-packages/)

[2]https://insights.stackoverflow.com/survey/2018

(accounting for 69.8%). In addition, the npm distribution was observed to have a higher distribution of package dependencies than other package distributions [5]. This increases the risk of packages suffering from technical lag due to outdated dependencies. For these reasons we selected npm for our empirical study. The metadata of each npm package release (such as the name, version, and list of dependencies) is stored in a .json manifest file. Dependencies are used to specify other packages that are explicitly required by the release. The range of allowed versions can be restricted using dependency constraints. By default, when a package is installed using the npm package manager, the latest release of each required package satisfying the dependency constraint is installed.

Let us present the terminology and notations that will be used in the remainder of this paper.

Release. Let E be a package distribution, i.e., a set of packages. Given a package $p \in E$, **releases**(p) denotes the set of releases of p. Every release $r \in$ **releases**(p) has a release date r_{date} and a version $r_{version} = (major, minor, patch)$. The triple $r_{version}$ reflects the *semantic versioning* mechanism suggested by npm. A simple set of rules dictate how version components should be incremented when a package is updated. Package updates corresponding to bug fixes that do not affect the API should only increment the *major* version component, backward compatible updates should increment the *minor* component, and backward incompatible updates have to increment the *major* component.

Release order. For any package p we assume two total orders $<_t$ and $<_v$ on its releases, respectively based on their *date* and their *version*. In the latter case, the versions are compared first based on their *major* component, then on their *minor* component, and then on their *patch* component. For any release r we write $prev(r)$ and $next(r)$ to refer to the previous and next release w.r.t. $<_t$, if it exists. Similarly, we write $prev_v(r)$ and $next_v(r)$ to refer to the previous and next release w.r.t. $<_v$, if it exists.

Release type. For each $r \in$ **releases**(p) such that $prev_v(r)$ exists, we define its release type r_{type} as MAJOR if $r_{version}$ and $prev_v(r)_{version}$ have distinct *major* components, as MINOR if they have similar *major* but distinct *minor* components, and as PATCH if they agree on all components except *patch*.

Dependency. A release r has a (potentially empty) set r_{deps} of dependencies. A dependency $d \in r_{deps}$ is defined as a pair $(d_{target}, d_{constraint})$ composed of a target package $d_{target} \in E$ and a dependency constraint $d_{constraint}$ over the releases in **releases**(d_{target}). If $r' \in$ **releases**(d_{target}), we note $r' \vDash d$ if $r'_{version}$ satisfies $d_{constraint}$. Dependency constraints allow package maintainers to explicitly select the desirable or allowed versions of a dependency, and exclude the undesirable ones, e.g., those that can contain backward incompatible changes. Dependency constraints are typically used to specify a minimal (e.g., $>= 1.2.3$), maximal (e.g., $< 1.3.0$) or exact version (e.g., $= 1.4.2$) of a dependency. The

npm package manager relies on the semver[3] tool to identify the version numbers satisfying a dependency constraint. It provides specific version constraint notations, such as tilde $\sim$ and caret $\wedge$ to allow releases to benefit from backward compatible dependency updates. $\sim$ allows for automatic updates of patches only, while $\wedge$ allows for automatic updates of both patches and minor releases. By default, the npm package manager selects for installation the highest available version that satisfies the dependency constraint.

Available and installable releases. Let package $p \in E$ and t a point in time. The set **available**$(p, t) = \{r \mid r \in$ **releases**$(p) \wedge r_{date} \leq t\}$ contains all releases of p that were available for installation at time t. Given a dependency d, the set **installable**$(d, t) = \{r \mid r \in$ **available**$(d_{target}, t) \wedge r \vDash d\}$ contains all available releases of the target package d_{target} that satisfy the dependency constraint.

Technical lag. Let d be a dependency and t a point in time. We define **missed**$(d, t) = \{r \mid r \in$ **available**$(d_{target}, t) \wedge r >_v \max_{<_v}$ **installable**$(d, t)\}$. This set captures the highest releases (w.r.t. $<_v$) of the target package that cannot be installed at time t because of the dependency constraint. Only the releases that are higher than any installable one are comprised in this set. We define the *technical lag* $\Delta_t(d, t)$ induced by d at time t as follows:

$$\Delta_t(d, t) = \begin{cases} t - \min\{r_{date} \mid r \in \mathbf{missed}(d, t)\} \\ 0, \text{if } \mathbf{missed}(d, t) \text{ is empty} \end{cases}$$

Technical lag represents the time during which d prevents the use of a newer version of its target package. By abuse of notation, we also use $\Delta_t(r, t)$ to refer to the technical lag of a release r at time t, which is defined as the maximal technical lag induced by its dependencies:

$$\Delta_t(r, t) = \max\{\Delta_t(d, t) \mid d \in r_{deps}\}$$

Example. The following example illustrates our definitions. Consider two fictitious packages p_1 and p_2. Let $r_1 \in$ **releases**(p_1), such that r_1 has a dependency $d = (p_2, \sim 1.0.0)$. Constraint $\sim 1.0.0$ only allows patch updates over release 1.0.0. Because of this dependency, the installation of r_1 requires a release of p_2 to be installed. By default, npm selects the highest available version that satisfies the dependency constraint. As a consequence, the selected releases of p_2 can be different depending on the installation time of r_1. Figure 1 shows, by means of green arrows, which release of p_2 will be installed at different points in time T_2, T_4, T_6 and T_9.

TABLE I
INSTALLABLE AND MISSED RELEASES FOR DEPENDENCY
$d = (p_2, \sim 1.0.0)$, AND ASSOCIATED TECHNICAL LAG.

t	$\max_{<_v}$ **installable**(d, t)	**missed**(d, t)	$\Delta_t(d, t)$
T_2	1.0.0	$\emptyset$	0
T_4	1.0.1	$\emptyset$	0
T_6	1.0.1	$\{1.1.0\}$	$T_6 - T_5$
T_9	1.0.2	$\{1.1.0, 2.0.0\}$	$T_9 - T_5$

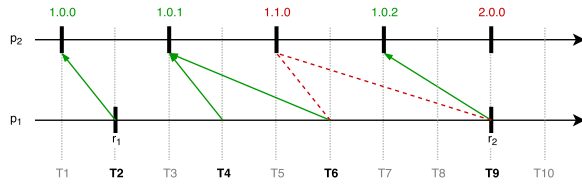[3]https://docs.npmjs.com/misc/semver

Fig. 1. Selected release at T_2, T_4, T_6 and T_9 for dependency $(p_2, \sim 1.0.0)$.

Table I shows, for each considered time point (first column), the release of p_2 that is selected during r_1 installation (second column), the set of releases of p_2 that are missed due to the dependency constraint (third column), and the resulting technical lag (fourth column). For example, even though version 1.1.0 of p_2 is available at T_6, it will not be selected (indicated by the red dotted line in Figure 1) because it does not satisfy the dependency constraint $\sim 1.0.0$. Consequently, r_1 does not rely on the highest available version of p_2 at T_6 and has a technical lag induced by its dependency d. This technical lag is computed as $\Delta_t(d, T_6) = T_6 - T_5$, the difference between the considered time point T_6 and the release date T_5 of the missed version 1.1.0. At T_9, even if the latest version 1.0.2 of p_2 satisfies the dependency constraint $\sim 1.0.0$, release r_1 is still lagging behind because the higher version 1.1.0 is missed. The technical lag of d at T_9 is then $\Delta_t(d, T_9) = T_9 - T_5$.

Assume that a new version r_2 of p_1 is released at T_9 with a dependency constraint $d' = (p_2, {}^{\wedge}1.0.0)$ allowing both minor and patch updates starting from version 1.0.0 of p_2. At T_9, the highest installable version of p_2 that satisfies this constraint is 1.1.0. Even if version 2.0.0 is already "missed", $\Delta_t(d', T_9) = 0$ because it corresponds to the difference between the considered time T_9, and the release time of the "missed" version 2.0.0, i.e., $T_9 - T_9 = 0$. At T_{10}, however, we have $\Delta_t(d', T_{10}) = T_{10} - T_9$ that reflects that version 2.0.0 is missed for a certain amount of time.

Dataset. Our analysis relies on the 2017-11-02 dump of the open source discovery service libraries.io [6]. Since we focus on npm packages, we only consider the metadata from the manifest of each package provided by the official npm registry. For each release of each npm package, we consider its list of dependencies. We restrict ourselves to runtime dependencies only, since we are only interested in those dependencies that are required to install and execute the package. We also exclude dependencies that target packages that were not available through the package distribution (e.g., packages that are hosted directly on the web or on Git repositories). This pre-filtering step resulted in 610,096 npm packages, 4,202,099 releases of these packages and 20,240,402 runtime dependencies between them.

We applied additional filters to our dataset for the technical lag analyses. First, we excluded 322,840 pre-releases of packages, based on their version number, e.g., 1.0.0-alpha, 1.2.3-beta.0, 2.0.1-rc. According to the npm semantic versioning tool, such releases "are meant to be unstable and are expected to have breaking changes" and thus, npm does not install a pre-release unless explicitly stated in the dependency constraint. Next, we filtered out 427,568 packages (and their corresponding releases) that were either never updated (i.e., packages that have only one release), not updated recently (i.e., no update since January 2017), or isolated in the sense that they had no direct nor reverse dependencies and are thus of no interest in our analysis.

Our filtered dataset consists of 120,084 npm packages, 1,447,709 releases of these packages and 8,044,034 runtime dependencies between them. This represents 20% of all packages, 35% of all releases and 40% of all dependencies of the original dataset. The earliest package release date of the filtered dataset was registered on 2010-11-09, and the latest on 2017-11-02. A replication package of our analysis is available on https://doi.org/10.5281/zenodo.1283203.

III. RQ_1: HOW MANY PACKAGES HAVE TECHNICAL LAG?

With the first research question we aim to gain an initial understanding of the omnipresence of technical lag in the npm ecosystem. To this extent, we quantify the technical lag of each release and each dependency over time.

For each package p in npm we gathered each release $r \in releases(p)$. At release date r_{date} we computed the technical lag $\Delta_t(r, r_{date})$ and $\Delta_t(d, r_{date})$ for each dependency $d \in r_{deps}$. Similarly, at the date of the next release (if any) we computed $\Delta_t(r, next(r)_{date})$ and $\Delta_t(d, next(r)_{date})$. The difference between the two Δ_t values represents the number of dependencies or releases for which a technical lag got introduced somewhere during the life of the release.
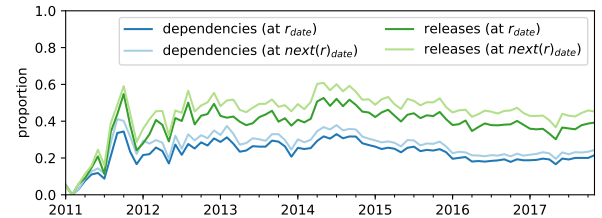

Fig. 2. Monthly proportion of releases r and dependencies $d \in r_{deps}$ having a technical lag, i.e., having $\Delta_t(r, t) > 0$ and $\Delta_t(d, t) > 0$ for $t \in \{r_{date}, next(r)_{date}\}$.

Figure 2 shows the monthly proportion of dependencies and releases for which technical lag is strictly positive. We observe that the proportion of dependencies undergoing a technical lag oscillates, since 2012, between 17% and 33%, with a median of 24% of all dependencies in npm lagging behind. The proportion of releases affected by a technical lag is higher, between 28% and 53% (median is 40%), suggesting that the dependencies inducing a technical lag are spread over many different releases. Starting from September 2014, we start to observe a decrease in technical lag. A possible reason for this might be the increasing adoption of dependency management tools such as David DM, Gemnasium and Greenkeeper. Trockman et al. [7] studied the adoption of such tools by npm packages, by analysing badges in their corresponding GitHub repositories. Among other findings,

they report that dependency-manager badges signal practices that lead to fresher dependencies.

While Figure 2 suggests that technical lag affects a large proportion of releases, this proportion must be nuanced: not all releases are always in a situation where a technical lag is possible. Indeed, in order for a technical lag to affect a release, it is necessary that the latter depends on a package for which a new release is available, and that this new release does not satisfy the dependency constraint, and therefore cannot be installed automatically.

In order to distinguish between releases that are not in a potential lag situation, those that are in a potential lag situation, and those that have effectively a technical lag, we compared, for each release r and for each dependency $d \in r_{deps}$, the sets **available**(d_{target}, r_{date}) and **available**$(d_{target}, next(r)_{date})$. This allowed us to identify which are the dependency targets that were updated during the period from r_{date} to $next(r)_{date}$. Similarly, we compared **installable**(d, r_{date}) and **installable**$(d, next(r)_{date})$ to identify which updates are not automatically accepted because of the dependency constraint and, therefore, induce a new technical lag. Figure 3 shows the monthly proportion of releases r for which (1) a new version of a dependency target is available before $next(r)_{date}$, and (2) a new version of a dependency target is missed before $next(r)_{date}$.
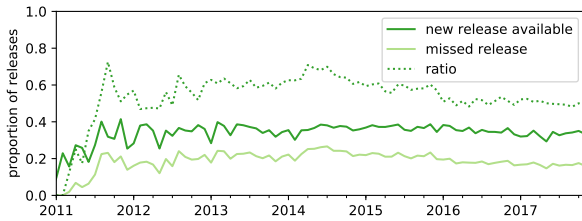

Fig. 3. Monthly proportion of releases r for which (1) a new version of a dependency target is available before $next(r)_{date}$, (2) a new version of a dependency target is missed before $next(r)_{date}$. The dotted line corresponds to the ratio of (2) over (1).

We observe that the proportion of releases that could have a technical lag is relatively stable over time, around 34%. Similarly, we observe that the proportion of releases having a technical lag is stable over time, around 18%. We also observe that the ratio between (1) and (2), i.e., the proportion of releases having a technical lag compared to those that could have a technical lag, fluctuates from 45.7% to 70.9% from 2013 to 2016 (median is 60.8%), before falling to 48.2%-53.6% from 2016 onwards (median is 50.8%).

Findings. One out of four dependencies and two out of five releases suffer from technical lag. One third of all releases have at least one dependency whose target package is updated during its release life, and half of them missed this new version, inducing or increasing the technical lag.

IV. RQ_2: How long is the technical lag?

RQ_1 focused on the extent of the technical lag in terms of releases and dependencies. A low technical lag

can be explained by the time it takes for a release to make modifications to benefit from an update in its dependencies. With RQ_2 we focus on the amplitude (time delay) of that technical lag.
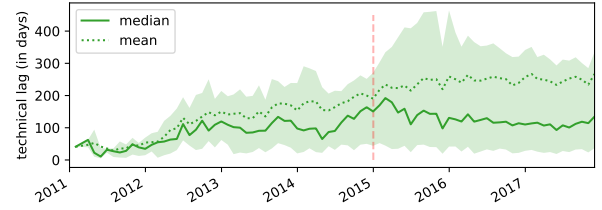

Fig. 4. Monthly distribution of non-zero technical lag $\Delta_t(r, next(r)_{date})$. The shaded area corresponds to the interval between the 25th and 75th percentile.

Figure 4 shows the monthly distribution of technical lag for releases with a strictly positive technical lag $\Delta_t(r, next(r)_{date})$. We observe from Figure 4 that the average technical lag oscillates between 29 and 206 days until 2015. Starting from 2015 (dashed vertical line), the average technical lag oscillates between 215 and 267 days, with a median between 93 and 192 days, witnessing an uneven distribution of the technical lag among the releases. We also observe that since 2015, 25% of the releases have a technical lag greater than 284 days. During the same period, only 25% of the releases had a technical lag lower than 52 days.
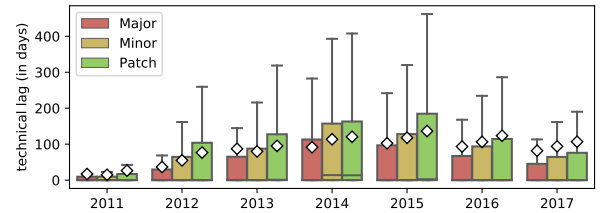

Fig. 5. Yearly distribution of $\Delta_t(r, r_{date})$ by release type.

To determine whether the release type $r_{type} \in \{major, minor, patch\}$ influences the technical lag, Figure 5 shows the technical lag distribution for $\Delta_t(r, r_{date})$ per year by r_{type}. We observe that the technical lag is more important for patch releases, followed by minor releases, then major ones. As new updates in dependency targets can require high maintenance effort in terms of code changes, it is not surprising to observe a higher technical lag for patch and minor releases, which typically have fewer changes and are more "lightweight updates" than major releases.

Findings. From 2015 onwards, the average technical lag for releases with technical lag is of 7 to 9 months. 25% of the releases have a technical lag of more than 9 months, while only 25% have a technical lag less than 52 days.

V. RQ_3: How frequently are packages updated?

Technical lag occurs when a new version of a dependency is not accepted by the dependency constraint of a release

that depends on it. Understanding the dynamics of package updates, and by extension, of dependency targets, makes it possible to better understand and identify which dependency target updates increase or decrease the technical lag of the releases that depend on them, and which releases decrease or increase the technical lag induced by their dependencies.
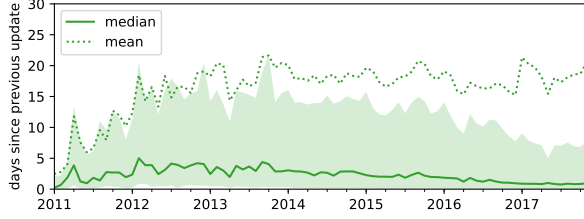

Fig. 6. Monthly distribution of $next(r)_{date} - r_{date}$. Shaded area corresponds to the 25th and 75th percentile.

Figure 6 shows the monthly distribution of a release "lifespan", i.e., the time between the date r_{date} of a release r and the date $next(r)_{date}$ of its next release. We observe that the average time between two consecutive updates ranges from 12 to 22 days starting from 2012. The much lower median value (1 to 5 days) suggests that the time between two consecutive updates is unevenly distributed among packages: some packages are updated very frequently (25% within a day), and other much less frequently (25% after 5 to 22 days).
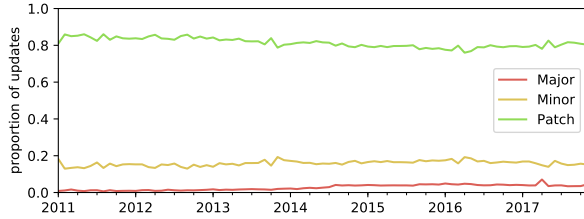

Fig. 7. Monthly proportion of release updates per release type.

To determine whether such a difference in update frequency can be explained by the release type, we distinguished in Figure 7 the monthly proportion of updates per release type (i.e., major, minor or patch). We observe that the vast majority (76% to 86%) of updates are patches, and this has been the case since the beginning of the observation period. Minor and major updates are much less frequent, ranging from 13% to 19% and from 0.6% to 7%, respectively.

This important distinction between the three release types suggests that we need to refine our analysis of the time between consecutive updates to take into account the release type, both for the release that is updated and for its next release (i.e., the one to which a release is updated to). Figure 8 therefore shows the distribution of the time between two consecutive releases, by release type, and by next release type. We observe that the time it takes to update a release essentially depends on the type of the next release. In particular, major releases are released much longer after a previous release than
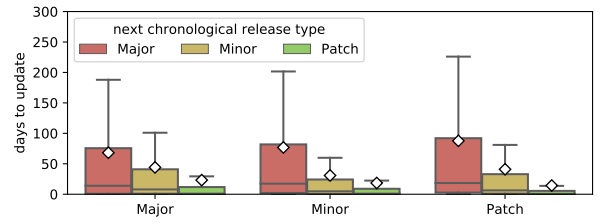

Fig. 8. Distribution of $(next(r)_{date} - r_{date})$ for all releases, grouped by release type of r and $next(r)$.

minor ones. Similarly, minor releases are released much later after a previous release than patch releases. These results are not surprising given the versioning semantics associated with each type of release. However, the results stress the importance of considering the release type when analysing technical lag.

> **Findings.** It takes an average of 12 to 22 days to update a release. The time required to update a release is unevenly distributed, and mainly depends on the type of the next release. Major releases are released much later after a previous release than minor ones. Minor releases are released much later after a previous release than patches.

VI. RQ_4: WHEN DOES TECHNICAL LAG INCREASE?

While RQ_2 revealed that technical lag can be (very) long for many releases, RQ_4 focuses on the conditions that make technical lag increase. A release that already has technical lag at its release date will continue to see its technical lag increase during its lifespan. Therefore, it is interesting to measure the magnitude of this increase as well as its origin.
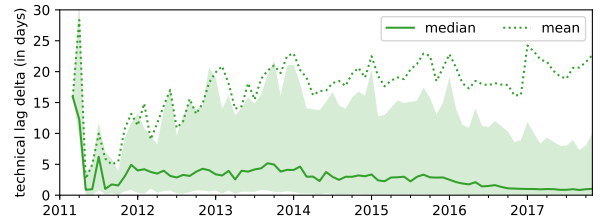

Fig. 9. Monthly distribution of non-zero $\Delta_t(r, next(r)_{date}) - \Delta_t(r, r_{date})$. Shaded area corresponds to the 25th and 75th percentile.

Figure 9 shows the monthly distribution of the non-zero difference in technical lag $\Delta_t(r, next(r)_{date}) - \Delta_t(r, r_{date})$ during the life of each release r. We observe that this distribution is very similar to the one representing the time between two consecutive updates of a release (Figure 6). This is not surprising since, as Figure 2 already indicated, most releases r having a lag at $next(r)_{date}$ already had a lag at r_{date}. Therefore, it is reasonable to expect that the additional lag gained between r_{date} and $next(r)_{date}$ corresponds to the difference between $\Delta_t(r, r_{date})$ and $\Delta_t(r, next(r)_{date})$.

In order to identify which releases of a dependency target are causing this additional technical lag, we computed for each release r the set of "missed dependencies", i.e., all
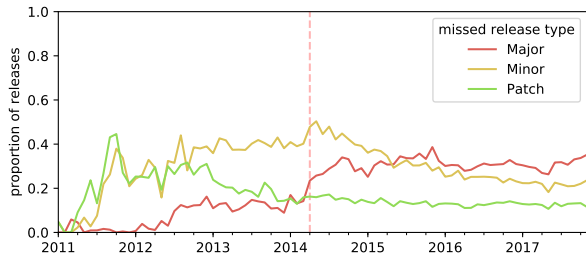
Fig. 10. Monthly proportion of releases r such that there exists $d \in r_{deps}$ with $d_{target} \in \mathbf{missed}(d, next(r)_{date})$. The results are grouped per release type of d_{target}.

dependencies $d \in r_{deps}$ such that d_{target} induces or increases the technical lag of r during its lifespan, i.e., $d_{target} \in \mathbf{missed}(d, next(r)_{date})$. Figure 10 shows the monthly proportion of releases r in such a situation, grouped by the release type of d_{target}. We observe that technical lag is observed mostly due to minor and patch releases of dependency updates until April 2014 (vertical dashed line in Figure 10). From April 2014 onwards, the proportion of major releases exceeds the one of patch releases. From mid 2015 onwards, the proportion of major releases also exceeds the one of minor releases.

These changes are probably the consequence of the introduction of a new semantic constraint *caret*[4] in npm, its use by default in dependency constraints[5], and its specific meaning for versions of the form 0.x.x[6]. As a witness of the confusion induced by these changes, npm changed the default initial version of a package from 0.1.0 to 1.0.0 since *"we cannot ever hope to get everyone to believe what the correct interpretation of 0.x versions are"*[7]. This has lead to an important proportion of releases that went from a 0.x.x version scheme with only minor and patch updates, to a 1.x.x version. We computed that the proportion of releases with a 0.x.x version dropped from 82.6% to 66% in 2014 alone.

The observations for Figure 10 confirm the hypothesis that "higher" release types (major > minor > patch) require more work in their dependent packages and therefore induce additional technical lag. It is nevertheless surprising that minor releases and (especially) patches induce technical lag since these update types are expected to be backward compatible and therefore should require nearly no effort for their adoption.

We hypothesise that the lack of automatic adoption of backward compatible changes is mainly due to the use of too strict dependency constraints, preventing patches or minor releases to be installed automatically. Indeed, Decan et al. observed that, in 2016, around 20% of all npm packages with dependency constraints specified *strict* constraints, preventing new releases of a dependency to be automatically installed [8].

[4]https://github.com/npm/node-semver/pull/41
[5]http://fredkschott.com/post/2014/02/npm-no-longer-defaults-to-tildes/
[6]https://github.com/npm/node-semver/issues/79
[7]https://github.com/npm/init-package-json/commit/363a17bc

Findings. Most of the releases with technical lag already had this lag at release date, and their technical lag continues to increase during their lifespan. Most of the technical lag is due to the minor and patch releases of a dependency target. This is somehow unexpected, as minor and patch releases are supposed to be backward compatible and therefore effortless to adopt.

VII. RQ_5: WHEN DOES TECHNICAL LAG DECREASE?

RQ_4 revealed that technical lag *increases* continuously during the lifespan of a release. RQ_5 aims to study when and how technical lag is *reduced*. To assess this, we compare for each release r the technical lag at its release date r_{date} with the technical lag of its previous release $prev(r)$ at the same time, i.e., $\Delta_t(r, r_{date}) - \Delta_t(prev(r), r_{date})$. A smaller technical lag implies that "effort" has been made to decrease the lag (e.g., the dependency constraint has been adapted to accept newer releases of a dependency target). An identical technical lag means that the dependency constraint has not been modified. A higher technical lag means that the dependency constraint has been modified on purpose to exclude the most recently used version of a dependency target. The latter case is expected to be extremely rare, as it corresponds to very specific situations such as the presence of a vulnerability or a bug that must be avoided as much as possible by dependent releases.
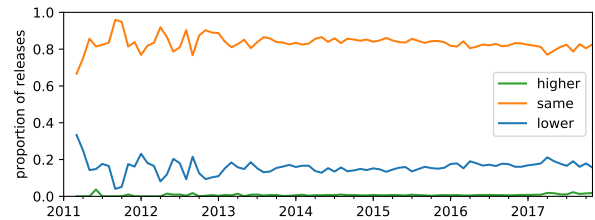

Fig. 11. Monthly proportion of releases with a higher, similar or lower technical lag than their previous release.

Figure 11 shows the monthly proportion of releases r for which technical lag has increased, decreased or remained the same. We observe that these proportions remain relatively stable from 2013 onwards. In most cases (77% to 89%) technical lag does not change from one release to the next. As expected, the proportion of releases in which technical lag increases is negligible. We observe a decrease of technical lag in only 11% to 21% of the cases.
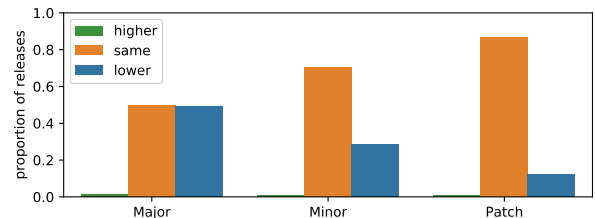

Fig. 12. Proportion of releases (grouped per release type) with a higher, identical or lower technical lag than their previous release.

We hypothesise that this decrease is strongly related to the release type of either r or $prev(r)$. Indeed, it seems likely that the decision to rely on a more recent release of a dependency target should be carried out as part of a larger update (e.g., a major update). Figure 12 shows the proportion of releases of each type with a higher, identical or lower technical lag than the previous release. It shows that a larger proportion of major releases (49%) decrease technical lag, compared to minor (29%) or patches (12%).

Figure 13 shows the proportion of releases, grouped by update type, which adopted a previously missed major, minor or patch release of a dependency target. We observe that between 40% and 50% of the major updates adopt a previously missed release of a dependency target. For comparison, only around 25% of the missed releases are adopted during minor updates, and only 13% during patch updates.
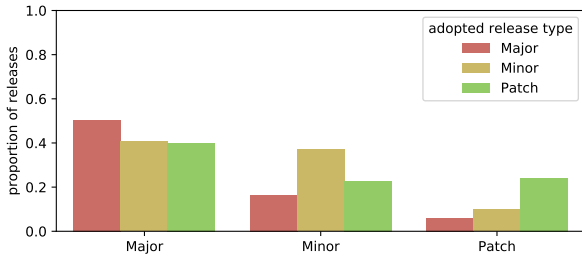

Fig. 13. Proportion of releases (grouped by release type) adopting at least one previously missed major, minor or patch update of a dependency.

We expected major releases that were missed to be adopted during major updates, as they usually contain backward incompatible changes and require additional efforts to be adopted. However, we did not expect such low proportions of adopted minor and patch releases. Only 23% (resp. 24%) of previously missed patch releases are adopted during a minor (resp. patch) update, versus 40% during major updates. This is surprising, as they are expected to be backward compatible and thus effortless to adopt, especially patches that should be adopted quickly as they are supposed to fix bugs and security issues.

> **Findings.** Most of the technical lag is reduced during major updates (and, to a lesser extent, during minor updates). Major releases of a dependency are typically adopted during a major update, and minor releases during a major or a minor update. Less than one third of all backward compatible releases are adopted during minor or patch updates.

VIII. RQ_6: HOW COULD TECHNICAL LAG BE REDUCED BY PROPER USE OF SEMANTIC VERSIONING?

Let us reconsider Figure 13 in the light of semantic versioning. It revealed that a relatively low proportion of releases adopt minor or patch updates of their dependencies.

Since patch releases are typically used for fixing bugs or security issues, it is not surprising that they do not represent an "ideal time" to update dependencies and to adopt new releases, even if these are expected to be backward compatible. Minor releases, however, could be reasonably expected to reduce

their technical lag by making use of more recent minor or patch updates of their dependencies. In fact, using tilde $\sim$ or caret $\hat{}$ in the dependency constraint would enable this. For example, "$\sim 1.2.3$" permits releases up to the next *minor* release (excluded). Similarly, "$\hat{}1.2.3$" permits releases up to the next *major* release (excluded). If, in contrast, strict or maximal constraints (e.g., "$1.2.3$" or "$<= 1.2.3$") would be used, no recent updates of the dependency will be accepted.

In order to assess to which extent a proper use of semantic versioning could help to reduce the effect of technical lag, we carried out a "what if" analysis. Figure 14 shows what would happen if dependency constraints would be "loosened" to allow for either patches, or both patches and minor releases to be accepted automatically. While we observe that the automatic adoption of patches would not change much, allowing (backward compatible) minor releases and patches would reduce the proportion of releases suffering from technical lag. Indeed, from April 2014 onwards (vertical dashed line in Figure 14), allowing backward compatible releases lowers the proportion of releases suffering from technical lag by an average of 17.6%.
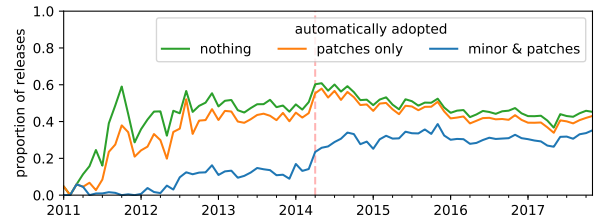

Fig. 14. Monthly proportion of releases r having a technical lag $\Delta_t(r, next(r)_{date}) > 0$ under the "what if" analysis.

> **Findings.** If dependency constraints would rely on semantic versioning rules that enable automatic updates of backward compatible changes, the proportion of releases suffering from technical lag could be reduced by 17.7%.

IX. DISCUSSION

Our empirical results revealed that many package releases exhibit technical lag. Package maintainers may impose too strict dependency constraints that refrain from updating their package dependencies to more recent versions, possibly because they are concerned with the extra effort or risk it would entail. However, assuming that package developers respect the semantic versioning policy, there is nothing that should prevent them from benefiting from backward compatible updates provided through minor or patch updates. In fact, enabling such a more flexible update policy has shown to be advantageous. For example, Cox et al. [9] observed that up-to-date systems are four times less likely to suffer from security issues and backward incompatibilities than systems that are up-to-date.

In earlier work we studied the evolution of vulnerabilities in npm [10]. Based on these results, we assessed to which extent automatic updates to patches or minor releases would allow dependent releases to benefit from fixes to vulnerabilities. The
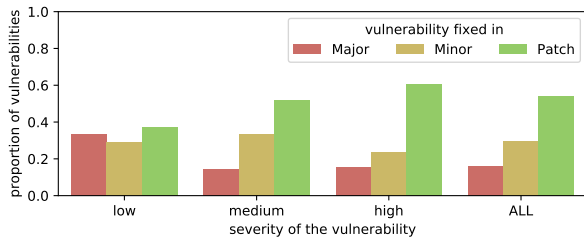
Fig. 15. Proportion of vulnerabilities fixed in patch, minor or major releases.

results are shown in Figure 15. Most vulnerabilities are indeed fixed during a patch (54%) or a minor release (30%), while only 16% are fixed during a major release.

Actionable result. Package maintainers should use semantic versioning to benefit from automatic backward compatible updates of patches and minor releases. This would reduce the risk of suffering from vulnerabilities as most of them are fixed during minor and patch updates.

In response to RQ_1, we observed a decrease in the monthly proportion of package releases and dependencies suffering from technical lag. 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. By analysing badges in GitHub repositories, Trockman et al. [7] studied the adoption by npm packages of dependency management tools. Based on interviews with developers they concluded that dependency-management badges signal practices that lead to fresher dependencies, indicating attention to updates and security patches.

Actionable result. Dependency management tools help package maintainers to reduce the technical lag of packages.

While it is useful for package maintainers to know if their dependencies are up to date, the npm community as a whole can also benefit from having an ecosystem-wide view of the impact of technical lag. Indeed, our analyses have revealed that some changes in the npm policy may lead to important changes in technical lag over time. This was for example the case when npm introduced changes to the semantic constraint caret, causing an important shift of packages from the 0.x.x to the 1.x.x version scheme (see Section VI).

Another example is the presence of unmaintained packages (e.g., because its maintainers are no longer available). Ecosystem managers should reduce such cases to a minimum. Indeed, if unmaintained packages depend on other packages, their technical lag will continue to increase over time, since there is nobody that will update the package if new releases of dependent packages appear. As a consequence, these packages will incur an increased risk of bugs and security issues, as there is nobody that will monitor and fix these problems.

Actionable result. Ecosystem managers should adopt an ecosystem-wide view of technical lag, in order to monitor its temporal evolution, as well as the impact of changes in the package distribution policy or tooling.

The npms service (https://www.npms.io) continuously computes a *popularity*, *quality* and *maintenance* score for each npm package, using information from a variety of sources, such as GitHub, the David dependency manager, and the Node Security Platform. The number of outdated dependencies is used as one of the factors to compute these scores, but this is not sufficient to provide a historic view of the dependency freshness of each package. It would therefore be useful to include the technical lag of package releases as a factor of package maintainability, as it indicates the ability of the package to remain up to date with respect to its dependencies.

Actionable result. Dependency monitoring tools and even the npm package manager should incorporate information about the technical lag of installed packages.

As observed in Figure 10, since mid 2015 major releases are the most important cause of technical lag. This is not surprising, since backward incompatible changes are only expected to take place in major releases (assuming semantic versioning policy is respected). Since dependent packages are less likely to upgrade to major releases of their dependencies, package maintainers should take this into account. For example, if an important bug or security vulnerability is detected that is also relevant for earlier releases, the maintainer should strive to backport the fixes to those previous releases that are known to be required by other packages. This will reduce the risk and technical lag of those dependent packages that did not upgrade yet to the most recent major release.

Actionable result. When providing a new major release, maintainers should strive to support dependent packages to update to such a release as easily as possible. They should also strive to backport important fixes to earlier releases.

X. Threats to validity

The accuracy of our results relies on the correctness of the package dependency metadata extracted from libraries.io. However, this metadata was manually checked for correctness in previous work [5].

It may also be the case that some packages (and their history) have been removed from npm before the extraction date, in which case they are not considered during our analysis. This is now prohibited by npm since April 2016.[8] This threat is unlikely to affect our results because of the huge number of packages we considered, and because we filtered out all dependencies pointing to non-existing package releases.

We restricted the dependencies of the npm package dependency network to those required to install and execute the package (i.e., runtime dependencies). Dependencies that are only required to develop or test a package were excluded

---

[8]http://blog.npmjs.org/post/141905368000/changes-to-npms-unpublish

from our analyses because not every package declares a complete and reliable list of development or test dependencies, and because these dependencies are unlikely to affect the production environment where technical lag could matter.

Another threat relates to the dates we considered when computing the technical lag of a release. For each package we measured the technical lag at the date of each release (i.e., the initial technical lag) and at the date of the next release, assuming that this represents the expected "end-of-life" of the release. In practice, a user can install or depend explicitly on a release that is not the latest available one. In that case, the technical lag could be higher than the one we computed for that release. Since this potentially higher technical lag is a consequence of an explicit choice made by the user (and not by the maintainer of the package), we consider that it should not be taken into account in our analyses. The values we reported are therefore more representative of the inherent technical lag of a package, and can be considered as a lower bound of the technical lag that can be observed in practice.

A final threat relates to the generalisability of our findings. The approach could be replicated for dependency networks of other package distributions, but the findings may be quite different from those obtained for npm, due to the fact that each package distribution and community has different policies, practices and culture [4], [8].

XI. RELATED WORK

Many researchers have studied the evolution characteristics of package dependency networks. Wittern et al. [11] analysed different evolution characteristics of npm packages, such as their dependencies, update frequency, popularity and versioning policy. They observed that maintainers use different versioning conventions for their packages, that are not always compatible with the recommended semantic versioning policy. This practice resulted in different version adoption ratios. Decan et al. [8] compared the topology of npm with the one of the CRAN and RubyGems package dependency networks. They studied the use of dependency constraints and found that, while strict dependency constraints increase the risk of missing important updates, they also prevent backwards incompatibility issues. Raemaekers et al. [12] investigated the use of semantic versioning in 22k Java libraries in Maven over a seven-year time period. They found that breaking changes appear in one third of all releases, including minor releases and patches. Somehow surprisingly, they observed that breaking changes have little influence on the actual delay between the availability of a release and its adoption by dependent packages. Bogart et al. [4] conducted a qualitative comparison of npm, CRAN and Eclipse, to understand the impact of community values, tools and policies on breaking changes. By interviewing developers, they found that there are two main types of mitigation strategies to reduce the exposure to changes in dependencies: limiting the number of dependencies, or depending only on "trusted" packages.

Many researchers have studied the phenomenon of outdated dependencies. McDonnell et al. [13] studied the evolution of the Android API, and the adoption of API updates by client applications. Among other findings, they observed that about 28% of API references in client applications are outdated, with a median time lag of 16 months. 22% of these outdated API usages eventually upgrade to newer API versions, at a much slower rate than the average API release interval. Kula et al. [14] analyzed over 6k Java libraries in Maven to investigate the latency in adopting the latest release of dependency targets. They showed that maintainers are more likely adopt the latest release for newly introduced dependencies, but less likely to adopt them at the beginning of their projects. In a follow-up study [3], they investigated 4.6k GitHub projects with 2.7k library dependencies and found that more than 80% of the studied systems have outdated dependencies.

Cox et al. [9] proposed different metrics to quantify a software system's *dependency freshness*. They measured the version sequence number, release date and number delta to capture four criteria, namely technology independence, ease of implementation, simplicity to understand, and enable root-cause analysis. They assessed the usefulness of these metrics through interviews with five technical consultants at SIG and showed that their objective metrics are in agreement with the subjective perception of dependency freshness.

Gonzalez-Barahona et al. [1] introduced the concept of technical lag to measure how outdated a system is with respect to its dependencies. They defined technical lag in terms of a lag function and lag aggregation function for packages. Zerouali et al. [2] measured such technical lag for npm package dependencies in terms of time and of number of missed versions. They observed that a large number of dependencies in npm have a technical lag of several months. However, our analysis differs from theirs in many ways. First of all, we explicitly excluded development dependencies as they tend to lead to an overapproximation of technical lag. We also computed technical lag not only for dependencies, but also for releases, as an aggregation of the technical lag of its dependencies. The way in which they computed technical lag, by comparing the latest installable release with the latest available one, is not consistent with how the npm package manager works, leading to inaccurate results when multiple "branches" of a package are maintained in parallel, a common phenomenon for popular npm packages. Instead, one should consider not the *latest* release, but the release corresponding to the *highest version*. Another difference is that we computed technical lag at the release dates of both the current *and* the next release of each package, allowing us to provide insights about the change in technical lag during a release lifespan, to assess the variation in technical lag along subsequent releases, and to identify when technical lag is reduced by a new release.

The phenomenon of technical lag and outdated dependencies has been shown to increase the risk of security vulnerabilities. Cox et al. [9] analyzed 75 Java systems in Maven, and split them into four risk categories based on their *dependency freshness*. They compared the systems in each category w.r.t. reported vulnerabilities and found that systems with a low dependency freshness are more than four times

as likely to contain security issues in these dependencies. Derr et al. [15] conducted a survey with more than 200 app developers in the Android ecosystem to investigate the use of outdated libraries, and reported that almost 98% of 17k actively used library versions with a known security vulnerability could be easily fixed by updating the library to a fixed version. Decan et al. [10] empirically studied nearly 400 security reports for 269 npm packages. They found that these vulnerabilities affect more than 72k other packages due to dependencies. They also reported that more than 40% of the releases depending on a vulnerable package do not automatically benefit from the security fixes because of too restrictive dependency constraints.

A large body of research focused on the evolution of library APIs. Wu et al. [16] studied the API evolution in Apache and Eclipse. Raemaekers et al. [12] analysed breaking changes and deprecated methods in Maven packages in presence of semantic versioning. Robbes et al. [17] empirically studied the ripple effect of deprecated APIs in the Smalltalk ecosystem. Hora et al. [18] explored the evolution of the Pharo ecosystem. This line of research differs from our own by its level of granularity: by analysing fine-grained API changes (e.g., at method level), it becomes possible to assess the effort and impact required to upgrade client applications to newer API versions. Carrying out such fine-grained static analyses is not feasible at the level of the entire npm ecosystem, because of the highly dynamic nature of the JavaScript language. While partial solutions and heuristics have been proposed (e.g. through combinations of pointer analysis and use analysis [19]), there are still many remaining open problems and challenges that need to be overcome [20] to make it scale.

XII. FUTURE WORK

We plan to replicate our study on package dependency networks of different package distributions. This will allow us 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. Inspired by [8], who studied the use of dependency constraints in three different package distributions (npm, CRAN and RubyGems), we aim to explore to which extent the specific (semantic) versioning policy and the use of dependency constraints influences the presence of technical lag.

Inspired by [9], [10], [15], we aim to empirically study the negative aspects of technical lag, such as the increased risk of suffering from security vulnerabilities and bugs. We 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 (e.g., packages that have a high technical lag, that induce technical lag on their dependents, that quickly reacting to dependency updates, etc.). With the results of this survey, we aim to identify and understand the reasons that lead developers to manage dependency updates in a specific way (e.g., why do some package maintainters update dependencies mainly during major updates, why don't they use dependency constraints that automatically allow for backward compatible releases, etc.). Such a study would complement the interviews carried out by Cox et al. [9], who assessed dependency freshness awareness by developers and validated the relevance and utility of several dependency freshness related metrics through interviews with five technical consultants.

XIII. CONCLUSION

Package distributions, like npm for JavaScript, are composed of huge interdependent collections of reusable software packages. These packages can suffer from technical lag if they depend on outdated packages, for example because the imposed dependency constraints prevent the package from installing a more recent version of its dependencies. Technical lag reflects the duration of time during which a package remains out of date with respect to a dependent package.

Based on the libraries.io dataset, we carried a longitudinal empirical study of such technical lag for 120k packages in the npm package dependency network, over an eight-year time period. We 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). We 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.

We observed that package releases suffer from technical lag if they do not benefit from the latest updates of a dependency target. It takes on average less than three weeks for a package to be updated, but this time is unevenly distributed and also depends on the type of the update (major, minor or patch). Large proportions of dependencies and releases of npm packages suffer from technical lag because of these updates in dependency targets. Since 2015, the technical lag of package releases ranges between 7 to 9 months. Technical lag is mainly reduced during major updates, even if it is mainly due to minor and patch releases that are supposed to be backward compatible and thus effortless to adopt. A proper use of semantic versioning would clearly help to further decrease the effect of technical lag. A "what if" analysis revealed that the proportion of releases suffering from technical lag could be reduced by nearly 18% if backward compatible updates of a dependency target would be automatically adopted.

XIV. ACKNOWLEDGEMENTS

REFERENCES

[1] J. M. Gonzalez-Barahona, P. Sherwood, G. Robles, and D. Izquierdo, "Technical lag in software compilations: Measuring how outdated a software deployment is," in *IFIP International Conf. on Open Source Systems*, 2017, pp. 182–192.

[2] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. Gonzalez-Barahona, "An Empirical Analysis of Technical Lag in npm Package Dependencies," in *Int'l Conf. Software Reuse (ICSR)*, 2018.

[3] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, Feb. 2018.

[4] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an API: Cost negotiation and community values in three software ecosystems," in *Int'l Symp. Foundations of Software Engineering*, 2016.

[5] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, Feb 2018.

[6] A. Nesbitt and B. Nickolls, "Libraries.io open source repository and dependency metadata," Jun. 2017. [Online]. Available: https://doi.org/10.5281/zenodo.808273

[7] A. Trockman, S. Zhou, C. Kästner, and B. Vasilescu, "Adding sparkle to social coding: An empirical study of repository badges in the npm ecosystem," in *International Conference on Software Engineering (ICSE)*, 2018.

[8] A. Decan, T. Mens, and M. Claes, "An empirical comparison of dependency issues in OSS packaging ecosystems," in *Int'l Conf. Software Analysis, Evolution, and Reengineering*, 2017, pp. 2–12.

[9] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *Int'l Conf. Software Engineering*. IEEE Press, 2015, pp. 109–118.

[10] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *International Conference on Mining Software Repositories*, May 2018.

[11] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the JavaScript package ecosystem," in *Int'l Conf. Mining Software Repositories*. ACM, 2016, pp. 351–361.

[12] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning versus breaking changes: A study of the Maven repository," in *Working Conf. Source Code Analysis and Manipulation*, Sept 2014, pp. 215–224.

[13] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the Android ecosystem," in *International Conference on Software Maintenance*, 2013, pp. 70–79.

[14] R. G. Kula, D. M. German, T. Ishio, and K. Inoue, "Trusting a library: A study of the latency to adopt the latest Maven release," in *Int'l Conf. on Software Analysis, Evolution, and Reengineering*, March 2015, pp. 520–524.

[15] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on Android," in *ACM Conf. on Computer and Communications Security*, October 2017.

[16] W. Wu, F. Khomh, B. Adams, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of API changes and usages based on Apache and Eclipse ecosystems," *Empirical Software Engineering*, vol. 21, no. 6, pp. 2366–2412, Dec 2016.

[17] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation? The case of a Smalltalk ecosystem," in *Int'l Symp. Foundations of Software Engineering*. ACM , 2012.

[18] A. Hora, R. Robbes, M. T. Valente, N. Anquetil, A. Etien, and S. Ducasse, "How do developers react to API evolution? A large-scale empirical study," *Software Quality Journal*, vol. 26, no. 1, pp. 161–191, mar 2018.

[19] M. Madsen, B. Livshits, and M. Fanning, "Practical static analysis of javascript applications in the presence of frameworks and libraries," in *Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. ACM, 2013, pp. 499–509.

[20] K. Sun and S. Ryu, "Analysis of javascript programs: Challenges and research trends," *ACM Comput. Surv.*, vol. 50, no. 4, pp. 59:1–59:34, Aug. 2017.