Technical Lag of Dependencies in Major Package Managers

Jacob Stringer*, Amjed Tahir*, Kelly Blincoe†, Jens Dietrich‡
*Massey University, New Zealand
†University of Auckland, New Zealand
‡Victoria University of Wellington, New Zealand
jacobstringer@windowslive.com; a.tahir@massey.ac.nz; k.blincoe@auckland.ac.nz; jens.dietrich@vuw.ac.nz

***Abstract*—Background: Third party libraries used by a project (dependencies) can easily become outdated over time, a phenomenon called *technical lag*. Keeping dependencies up to date induces a significant overhead in terms of the resources (e.g. developer time), but necessary to maintain software quality. Aims: This study provides a large scale analysis of technical lag across the major package managers currently in use. Method: We conducted a mixed-methods study using open-source project data obtained from 14 package managers using the *libraries.io* dataset. Results: The majority of fixed version declarations, along with a significant number of flexible declarations, are outdated. Fixed declarations are not regularly updated, except in major updates, so they quickly lag. Despite the prevalence of breaking changes in updates, downgrading declarations to earlier versions are rare. Conclusions: Technical lag is prevalent but preventable across package managers - semantic versioning based declaration ranges would remove the majority of lag. Further tooling uptake is also recommended to minimise technical lag.**

***Index Terms*—semantic versioning, package managers, dependency management, technical lag**

I. Introduction

Most modern software systems are built from existing packages (i.e., modules, components, libraries - henceforth termed *dependencies*) that allow complex functionality to be delivered easily. These libraries are generally created by third-party developers and are linked to a project via a symbolic dependency declaration resolved by modern package managers, such as *Maven* for JVM languages, *Cargo* for Rust, or *npm* for JavaScript. These package managers allow dependencies to be downloaded from a remote repository at build time by declaring constraints which describe the versions of a dependency compatible with the project, giving significant flexibility and agility to the dependency management process.

Dependencies usually evolve simultaneously with a project, so they can become outdated if the version requested by the program does not get updated regularly to the latest available version. Keeping dependencies up to date induces a significant overhead on a developer's time, but is important for the security and overall health of the project [1], [2], [3], [4], and to avoid project bloat, where multiple versions of a dependency are used by different submodules of a project [5]. The Open Web Application Security Project (OWASP) listed "using dependencies with known vulnerabilities" as one of the top 10 most critical security risks to web applications [6].

It is possible for package managers to automatically update dependencies, based on the industry-led *Semantic Versioning* [7] initiative (henceforth termed *semver*). This initiative is based on a structured version scheme that encodes compatibility levels. The intention is to facilitate the automated deployment of patches, while preventing compatibility-breaking changes. However, this is non trivial in practice; research has shown that the versioning scheme is often not used as intended (by semver), and this may lead to changes breaking backwards compatibility [8], [9]. This leads to a situation where a build breaks and an immediate fix is required, or worse, where the project breaks at run-time. This is particularly problematic in dynamically typed languages, where there is not a compilation phase that flags backwards breaking API changes.

While automated dependency updates where the dependency manager has some freedom to choose the best version by some metric have become commonplace in some software ecosystems, a significant number of projects still employ fixed version dependency declarations [10]. This has the advantage of greater control - dependencies only get updated after being (integration-)tested with the rest of the project. The downside of fixed version declarations is that it makes the updating process less agile. It requires developers to manually find updates and change the relevant configuration files by hand. When manually updating dependencies, they are often not updated immediately, resulting in *technical lag* [11], where newer versions - with their bug fixes, security fixes and other improvements - are available but not used. This technical lag has therefore an opportunity cost associated with it [12].

This paper studies this technical lag in detail, and sets out to answer the following research questions:

- **RQ1**: How often do dependencies lag?
- **RQ2**: How much lag is there in dependencies when they are not up-to-date?

We then examine updates to fixed version declarations:

- **RQ3**: How often do developers update their dependencies, what type of updates are most common, and what type of project releases contain dependency updates?
- **RQ4**: How often do developers update when they lag behind, and do the updates bring them up to date?
- **RQ5**: How often do developers make a backwards change to their dependencies, and why?

II. Related Work

Technical lag is a major source of security vulnerabilities. Cox et al. [2] found that projects using outdated dependencies were four times more likely to have security issues than those with up-to-date dependencies, although this analysis is likely to overestimate the risk of vulnerability as Zapata et al. [13] found in an npm based study that 73% of projects did not use the vulnerable functions inherited from their dependencies. Additionally, libraries included transitively are more likely to be vulnerable [1]. Pashchenko et al. [3] noted that many vulnerabilities can be fixed by simply updating dependencies.

Technical lag has anecdotally been linked to brittle code. Raemaekers et al. [14] outlined a case study where a dependency, not updated for several years, was updated to use a new feature. There were numerous cascading changes, taking over a week to resolve, which almost caused the planned new feature to be scrapped. In [8], the same authors studied the version and time lag on 2,984 dependency updates from projects on the Maven repository, finding that major changes to dependency versions were usually included in major updates of a project, and that most projects did not have large amounts of technical lag. More recently, Wang et al. has shown that a large number of Java projects contain technical lag of some kind [5]. Zerouali et al. [15] quantified technical lag in npm, using *libraries.io*, as this study does. They found that the median technical lag was one minor and three micro versions, and that much of the technical lag was due to inheriting lag transitively. Lauinger et al. [1] had similar results in npm using 72 libraries harvested from GitHub, but found that the time lag can often be measured in years. In recent studies, Zerouali et al. [16], [17] showed that JavaScript-based docker images generally included some lag, particularly in micro versions. Our study builds on previous technical lag studies by providing the first large-scale, cross-package manager overview of technical lag.

A number of studies have looked into how dependencies are updated. Wang et al. [5] found that the majority of projects in the Maven ecosystem contained dependencies which had never been updated. Decan et al. [18] found that in npm, CRAN and Rubygems, 80% of dependencies would be updated within 18 months of being declared. Kula et al. [12] investigated latency when adopting library releases and found that developers were more likely to adopt updated versions later into a project's lifecycle. This goes hand in hand with observations from Espinha et al. [19], who notes that early versions of projects are particularly unstable w.r.t. their APIs. Kula et al. [12] also found that the trend is for developers to automatically go to the newest version when introducing new libraries. Decan and Mens [20] found that declarations are updated every 3-7 versions, with more restrictive declarations and pre-`1.0.0` versions being updated more regularly. Decan et al. [21] examined technical lag and updates in the npm ecosystem, showing that dependencies are most likely to be brought up to date during a major version update. They also conducted an experiment finding that adopting semver compliant ranges could reduce technical lag in 18% of dependencies - this test has been mirrored in RQ1 for all package managers included in this study. We build on these studies by considering, across most major package managers, when updates are made in terms of version milestones of a project, classifying the types of updates made, and considering how often updates occur in fixed declarations if there is lag present.

Côgo et al. [22] looked at backwards changes to version declarations in npm, explaining that downgrades were caused by either moving away from buggy versions to a stable release, or as a preventive measure, going from a version range to a fixed version. They found that most downgrades happened to only one dependency at a time, indicating that this is usually done as a fix rather than a policy change. Raemaekers et al. [8] showed that often semver versions were incorrectly updated, with an average 30 backwards compatibility issues even in micro and minor updates - updates that should be fully backwardly compatible. The same study showed that breaking changes in non-major releases has decreased over time from 28.4% in 2006 to 23.7% in 2011, indicating that developers are becoming more aware about semver ideas.

For the declaration update research questions, we focus on fixed declarations only. Dietrich et al. [10] showed that a significant number of dependencies in the *libraries.io* dataset used fixed declarations, particularly in Maven where 98% of declarations were fixed (sub-classified as *soft*, which only differs from *fixed* by how dependency resolution is implemented). Other major package managers relied less on fixed versions, with Pypi, Packagist, npm and Atom having between 10%-20% fixed declarations, and Cargo and Rubygems below 5%. This points to a large disconnect between how each ecosystem manages their dependencies, a finding also noted in Bogart et al. [23]. Bogart et al. notes that upstream and downstream developers have a contract where upstream developers promise to maintain backwards compatibility in a predictable manner, but that attitudes towards these contracts vary significantly by ecosystem. Decan and Mens [20] have found that in Cargo, npm, Packagist and Rubygems, developers are increasingly using semver compliant declarations (i.e., minor or micro ranges) instead of fixed declarations or open ranges.

Previous studies investigated some aspects of technical lag, but were limited to a few specific package managers (e.g., npm or Maven). Our study takes a broader view, providing an empirical viewpoint into technical (version and time) lag in a wide range of package managers. It goes on to consider how semver compliant declarations can decrease technical lag in package managers, look at when dependencies are updated (w.r.t. the dependent project's evolution) to provide insights into challenges still faced by developers in minimising technical lag, and make a case for further tool support that provides safer dependency updates.

III. Methodology

This study uses *v1.4.0* (Dec 2018) of the *libraries.io* dataset [24], containing 2.7M unique open source projects from 37 package managers and 235M dependencies between projects.

TABLE I
DEPENDENCY DECLARATION TYPES

Classification	Explanantion	Example
Fixed	One specific version	`1.3.0`
Micro	Micro versions from min	`[1.3.2, 1.4.0)`
Minor	Micro or minor versions from min	`[1.3.2, 2.0.0)`
At-Most	Any versions up to a limit	`[0.0.1, 1.3.2]`
At-Least	Any versions from min	`[1.3.2, ∞)`
Any	Any versions	`[0.0.1, ∞)`
Range	Any custom range	`[0.3.2, 0.7.1]`

A. Semantic Versioning

Semver [7] is an industry initiative for versioning dependencies. This study focuses solely on projects which adhere to the semver syntax. Semver-compliant versions have three dot separated numbers, with an optional tag, i.e. *major.minor.micro-tag*, for example `v1.2.0-rc1`. Whenever new versions are published, one of the three numbers is incremented, with the numbers to the right of it reset to zero (e.g., a minor update would look like `1.7.2` → `1.8.0`). A dependency has a major change if the major number has changed, and so on for minor and micro changes. As tag updates are considered pre-release by semver principles, we do include them in technical lag calculations. However, as dependencies are sometimes updated within tag releases, they are still included in the analysis as *no-change* updates of a project or dependency.

B. Declaration Classification Process

Table I demonstrates the main types of dependency version declarations. Declarations that only allow one version are termed ***fixed***. Fixed declarations can also be *soft* where a specific version is specified but the package manager may choose another close version in order to satisfy some constraints.

All declarations that allow for more than one version are termed ***flexible***. There are multiple types of flexible declarations. The *micro* and *minor* classifications are based on semver[1]. There are other classifications that are not based on semver standards, including *any*,*at-most*, *at-least*, and *range* classifications. The *at-least* and *any* are the most permissive, and should not suffer from technical lag (this ultimately depends on the package resolution strategy of a package manager), but they risk compatibility issues.

To classify the version declarations into these classifications, the methodology of Dietrich et al. [10] was used (declarations were tested with regular expressions tailored to each package manager). The *FIXED* and *FLEXI* columns in Table II show the number of pairs considered fixed or flexible, respectively.

C. Filtering Process

The following filters were applied to the data:

1) Not Semver-compliant Syntax: Projects with versions that could not be parsed to the semver format specified in Section III-A were discarded, as version ordering would be problematic. Similarly, versions with unusually large numbers (usually related to timestamps) were discarded. The

[1]semver.org does not officially discuss declaration ranges, however package managers (such as npm) consider minor and micro ranges to be based on semver standards. The nomenclature used here is adopted from [20].

TABLE II
PAIRS OF PROJECTS INCLUDED BY PACKAGE MANAGER

PM	FIXED	FLEXI	NOT_SV	SUBCMP	MISSING
Atom	2276	17565	2	642	147
Cargo	1237	81543	90	4030	42
Dub	43	999	0	109	128
Elm	0	3988	0	253	0
Haxelib	192	568	0	266	5
Hex	698	10531	0	592	88
Maven	389044	23860	7358	106631	47015
npm	876534	7449356	2254	335976	19130
NuGet	2010	245725	738	64038	13568
Packagist	19250	474455	19784	70975	19886
Pub	157	16142	0	809	672
Puppet	204	7210	0	1800	242
Pypi	3300	25418	31	1185	569
Rubygems	14940	606994	93	21931	1586

NOT_SV column in Table II shows the number of dependency pairs filtered out due to semver violations. In some cases, they were package manager specific issues, such as only specifying '`@dev`' tags in Packagist (without any accompanying version number), or unresolved variables like '`${project.version}`' in Maven.

2) Missing Project Information: Projects sometimes include dependencies to other projects which were not included in the dataset. This meant that information about their version history was not available, making them unsuitable for our study. The *MISSING* column in Table II shows the number of pairs filtered out due to lack of data (1% of dependencies).

3) Groups of Packages with Coordinated Releases: Often, the developers working on project A also work on its dependency B. This is most regularly associated with the two projects being components of a single overarching project. Often these dependencies are updated as part of the internal release procedures (e.g., coordinated product release, Apache Lucene is an example of such coordinated releases[2]). Including them would lead to under-reporting the level of technical lag in independent projects. The *SUBCMP* column in Table II shows the number of pairs filtered out due to being suspected subcomponents. For the purpose of this study, pairs are considered subcomponents of a wider project if either the first half of their names (minimum 4 characters) are the same, or their entire project names are the same. To ensure that this only captures subcomponent pairs, we conducted a manual analysis with cross-validation on the first 300 pairs per package manager (giving a confidence interval of 95% with a 12% margin of error) – resulted in a $< 1\%$ false positive rate.

D. Package Manager Selection

There were initially 17 package managers with dependencies in the dataset. Three package managers (CPAN, CRAN, Homebrew) were ruled out of scope as their declaration patterns are entirely open range based (the *any* and *at-least* classifications) which means they do not have technical lag due to their dependency resolution strategies. The remaining 14 package managers (shown in Table II) were used to answer the questions about how much technical lag is present in

[2]https://mvnrepository.com/artifact/org.apache.lucene

dependencies (RQ1 and RQ2). To answer RQ3-5 on fixed version update strategies, nine of the package managers with a sufficient number of fixed declaration pairs were used (from the most fixed pairs to the least - npm, Maven, Packagist, Rubygems, Pypi, Atom, NuGet, Cargo and Hex).

E. Quantifying Technical Lag

The approach of quantifying technical lag in this study is similar to previous studies [15], [16], [17], [8], using both version lag and time lag. For each project, the versions are ordered according to semver principles (sorted by major, minor, then micro versions, and finally by tags ordered according to publish dates). For each version of project A, A_i, that has a dependency to project B, B_{dec}, there is technical lag if B_{dec} is not the latest version of project B when A_i is released.

We calculate three different types of lag (major, minor and micro lag) which we formalise below. In this notation, **B** is the set of all versions in project B that were published before A_i. Each version in this set is represented by $B_i \in \mathbf{B}$, and B_{dec} is the version chosen by the package manager based on the declaration. As described in III-A, each version number contains properties *major*, *minor* and *micro*. We denote the major, minor, and micro version numbers of each version, B_i as $B_{i.major}$, $B_{i.minor}$, and $B_{i.micro}$, respectively.

Major lag (L_{major}) is calculated as the distinct major versions later than B_{dec}. More formally:

$$L_{major} = |\{B_{i.major} \mid B_{i.major} > B_{dec.major}\}|$$

To calculate *minor lag* and *micro lag*, we consider narrower version ranges. *Minor lag* (L_{minor}) is calculated as the distinct minor versions later than B_{dec} *within the same major range*. *Micro lag* (L_{micro}) is calculated as the distinct micro versions later than B_{dec} *within the same minor range*. This means that there could be a more newer minor and micro versions than reported, but within other major or minor version ranges. Limiting micro and minor lag to these narrower version ranges allows direct comparisons to be made with semver compliant micro and minor ranges - the most common alternative declarations to *fixed* declarations [10]. More formally:

$$\begin{aligned} L_{minor} = |\{B_{i.minor} \mid\ & B_{i.minor} > B_{dec.minor} \\ & \wedge B_{i.major} = B_{dec.major}\}| \\ L_{micro} = |\{B_{i.micro} \mid\ & B_{i.micro} > B_{dec.micro} \\ & \wedge B_{i.minor} = B_{dec.minor} \\ & \wedge B_{i.major} = B_{dec.major}\}| \end{aligned}$$

The total lag of any release A_i towards project B is defined as the major, minor, and micro lags, following the semver idea that any major lag is more significant than *all* minor lag, and any minor lag is more significant than *all* micro lag.

$$L_{(A_i,B)} = (L_{major}, L_{minor}, L_{micro})$$

Fig. 1 provides a working example of how technical lag is quantified. There, project A `v4.1.0` depends on `v1.0.0` of project B. Since this was not the latest version of project B at the time of release of project A `v4.1.0`, there is technical lag. Both the major and minor lag are 1 since there was one later major release and one later minor release available when

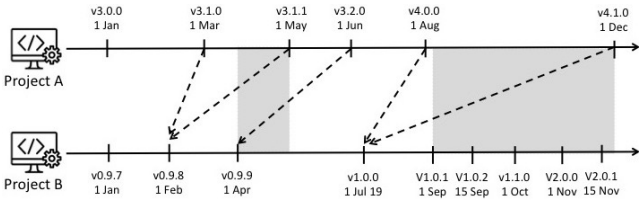

Fig. 1. Quantifying Technical Lag. The arrows indicate the dependencies. Technical lag (grey shaded regions) occurs when Project A depends on a B_{dec} that is not the latest version of B available at the time A_i was released.

`v4.1.0` was released. The micro lag is 2 since there were two later micro releases available within the `v1.0` minor release. Note that there is also a micro release `v2.0.1` available, but since it is not within the `v1.0` release, it is not considered since most flexible versioning strategies would not automatically update to this micro release.

Time lag is calculated as the number of days before the release of A_i that the first version after B_{dec} was available. Time lag is also calculated in three parts: *major*, *minor*, and *micro*. As shown in in Fig. 1, `v4.1.0` has major, minor, and micro lag. Since this release was made on 1 Dec, the time lags are:

- 91 day micro lag since `v1.0.1` was released on 1 Sept.
- 61 day minor lag since `v1.1.0` was released on 1 Oct.
- 30 day major lag since `v2.0.0` was released on 1 Nov.

Note that the release of `v1.0.2` on the 15th of September does not impact the micro time lag since this is not the first micro release after B_{dec}. The dependency began to be out of date the moment `v1.0.1` was released, so the time lag is calculated based on this release date. Lag is counted for every version A_i which has a B_{dec}. Therefore, project pairs with long version histories of dependencies are represented multiple times in the aggregated results.

F. Update Classification

RQs 3-5 focus on the change in the dependency declarations. Due to the complexities in parsing and categorising differences in flexible declarations, this study has focused on fixed declarations. Between any two versions, A_i and A_{i+1}, if there exists a declaration in both, the declarations are compared. This results in classifying the declaration change as no change, a forwards change, or a backwards change. The forwards and backwards changes are further split into major (the major version changed), minor (the minor version changed but not the major version), or micro (only the micro version changed). In the case that one of the two contiguous versions did not have a declaration, no update data was recorded.

To calculate backwards updates, two adjacent versions A_i and A_{i+1} were selected and compared (e.g. `4.0.1` and `4.0.2`). To account for cases of simultaneous development, if A_i was published after A_{i+1}, a new A_i was chosen as the most recent version published before A_{i+1}. This can occur at the boundary between distinct major versions (e.g., `3.5.2` being published after `4.0.0`) in the case where major versions 3 and 4 are both under development together.

G. Validation

There have been two main processes used to transform the data in the *libraries.io* dataset for analysis.

1) Flexible Declarations (RQ1-2): To answer the lag-based questions, all fixed and flexible pairs from III-C were used. This involved parsing raw declaration strings from each of the 14 package managers. To help make this process accurate, the raw strings were classified using the method found in [10], and each common pattern was parsed into (potentially multiple) inclusive ranges. To validate the parsing process (which differed for all 14 package managers), a test suite was created from the example declarations from [10]. These had been chosen after manual validation of the dataset, and therefore are representative of both the specifications from that package manager, as well as common syntactic deviations found in the actual dataset. Each declaration would be tested with three to five boundary versions, resulting in an average of 60-80 tests per package manager.

2) Update Data (RQ3-5): The update information was gathered from pairs that only had fixed declarations. Validation for this section was done in an iterative manner. 10 random pairs from various package managers were manually checked for all data in this section. Where there were issues, failing test cases were created, and the analysis scripts were debugged until they succeeded. 10 more pairs were then reviewed until there were no further issues found. Rare cases such as pairs updating to an older dependency version, or outlier lag values have been considered separately by the same process.

As the backward updates had additional complications due to simultaneous development, noted in Section III-F, to validate these heuristics, we manually analysed a random sample of 95 pairs with backward changes, looking for false positives where our heuristics identified a backward change but one did not occur. This sample size gives us a confidence interval of 95% with a 10% margin of error. The manual validation was done by two of the authors. For each pair of a backward change, we searched the commit history in the related repositories to identify whether it is actually a true backward change. We excluded samples where repository information is not publicly available. Of the 95 manually analysed backward changes identified through our heuristics, 5 out of 95 (5.3%) were false positives (meaning no backward change had occurred). There were two reasons for the false positives. In one case, the dependent project had changed its dependency strategy from a date based versioning system to semver. Thus, the dependency changed from `v2015.03.12` to `v1.4.6`, which was incorrectly identified as a backward change. The other four false positives occurred due to the projects performing parallel release development. In some of these cases, the projects appeared to explicitly maintain a parallel release to continue to offer support for a prior version of a dependency. For example, one project released `v0.7.0` with a dependency on project B `v2.15.0`. Shortly after that release, another release was made called `v0.7.0-projB-2.12` which depended on `v2.12.4` of project B. Our heuristics incorrectly classified these as backward changes since the release of the parallel version depending on the older version of project B occurred after the release which was updated to the new version of project B.

Since our heuristics were able to correctly classify 90 of the 95 (94.7%) backward changes, and the other cases could not be easily identified automatically, we used these heuristics to identify backward changes for the remaining analysis.

H. Identifying reasons for backward changes

To understand the reasons why developers make backwards changes to dependencies, we manually reviewed the random sample of 95 backward changes identified by the heuristics described above. For each backward change, a search was done in the project repositories to look for the related commits, any associated issues or discussions, and release notes to look for mentions of reasons why the backward change was being made. We were able to find information on the reason behind the change for 66 of the backward changes.

We used Thematic Content Analysis [25] to identify the main themes from the reasons found in the project repositories. Three of the authors jointly discussed the reasons to develop the themes. The three authors involved in the process all have prior or current experience in the software development industry as software developers. One is a software engineer and two are academic researchers. The themes were developed through extensive joint discussions between the three authors. In cases of disagreements, discussions continued until a consensus was reached. There were two backward changes that we did not categorize due to a lack of information. When describing the identified themes in the results, we provide detailed examples for each theme, including quotes from developers obtained in our analysis.

Our scripts and the filtered dataset are publicly available for replication purposes [26].

IV. Results and Discussion

The first question, how often do dependencies lag, has been analysed from three separate angles: 1) How does each declaration type affect lag? 2) What types of lag are present? 3) How would increased used of semver ranges affect lag?

RQ1.1: Lag by declaration type.

Table III shows the proportion of declarations that lag in each package manager, based on classifications discussed in III-B. As expected, the more restrictive the declaration is, the more likely it is to lag. Fixed declarations are more likely to lag than micro ranges, which are more likely to lag than minor ranges. *At-most* declarations tend to lag even more than fixed versions. Many smaller ecosystems keep declarations up to date, while the larger ecosystems are more likely to lag, as seen in Table III's *Overall* column, which shows the proportion of lagging declarations once open range declarations are removed. For example, Maven has a significant amount of technical lag within its ecosystem. We also see the effects of including the declarations that cannot lag - NuGet, Pypi and Rubygems, three package managers with high levels of lag,

TABLE III
PERCENTAGE OF DEPENDENCIES THAT LAG*

PM	Fixed	Micro	Minor	At-Most	Range	Overall
Atom	49.9%	40.7%	17.4%	34.0%	17.4%	28.3%
Cargo	36.3%	13.6%	2.8%	51.0%	19.8%	11.7%
Dub	29.9%	13.1%	10.3%	-	12.0%	15.4%
Elm	-	-	-	-	18.1%	18.1%
Haxelib	8.1%	-	-	-	-	8.1%
Hex	34.6%	21.5%	9.8%	50.0%	11.0%	16.2%
Maven	63.2%	29.2%	7.8%	75.0%	14.8%	63.0%
npm	51.6%	34.6%	26.5%	47.2%	38.6%	32.2%
NuGet	39.2%	-	-	69.8%	54.0%	49.4%
Packagist	71.7%	47.7%	23.1%	80.0%	35.9%	31.9%
Pub	57.5%	-	-	11.0%	18.4%	19.0%
Puppet	33.6%	52.7%	8.5%	47.5%	13.0%	15.7%
Pypi	51.5%	-	-	69.8%	23.5%	45.8%
Rubygems	56.6%	39.5%	61.0%	72.0%	22.0%	53.5%

*This excludes declarations that cannot lag, such as *'at-least'* or *'any'*

show much lower levels of lag once their *'any'* and *'at-least'* declarations are included.

A strong reliance on fixed declarations correlates with higher levels of lag. This is especially true for Maven (63% lagging), which primarily uses fixed declarations. Other package managers show similarly high lag in fixed declarations, however, in most other package managers, they are used less frequently, so have a smaller effect on overall levels of lag.

In general, minor range declarations have a low probability of lagging, with npm at a high of 26% to Cargo with a low of 3% lagging minor range declarations. The outlier here is Rubygems in which 61% of its minor ranges are lagging. About 30% of Rubygems declarations are minor ranges, indicating that there is a widespread issue in Rubygems projects where semver compliant declarations are not being kept up to date. In most package managers, custom *ranges*, which do not use micro or minor range shortcuts provided by a package manager, form a small part of the overall number of declarations. They often do not align with semver style micro or minor ranges, but instead form a subset of a micro, minor, or major range. However, in Elm, Packagist and Pub, ranges are the most common way to set up flexible declarations, even where micro or minor ranges are intended.

RQ1.2: Types of lag present.

In addition to considering how dependencies lag based on the type of declaration used, we also considered what type of lag is present in the dependencies. Micro version lag may not mean a significant difference between the dependency used compared to the newest dependency available, but lagging behind by a major (or to some extent, minor) version generally implies that there are more improvements that have been missed out on.

We split how a declaration lags into seven categories- the seven possible combinations of major, minor and micro lag (i.e., major, major & minor, major & micro, major & minor & micro, minor, minor & micro and micro). Fig. 2 shows the average percentages of the type of lag (or no lag) found across all package managers. Due to space constraints, the detailed results from each package manager is provided in our replication package [26].

When a dependency has *major* lag, it is behind the newest

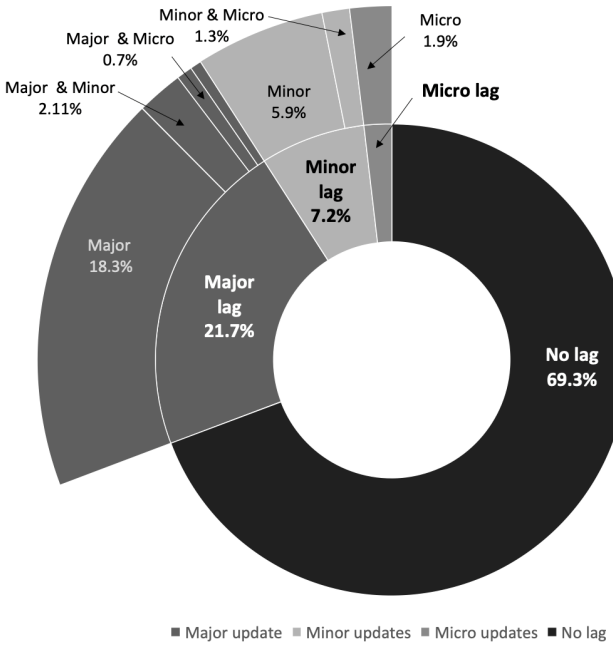

Fig. 2. Type of technical lags

available version by at least one major version, but it is up to date within its declared major version - it has no minor or micro lag. In several package managers, such as Atom, Maven, npm and Packagist, there are a disproportionate number of dependencies that lag only by major versions and therefore are up to date within that specific major range.

In general, there are three distinct groups of package managers separated by the amount of lag present. One group consists of the mature package managers that primarily use fixed or semver compliant ranges. In this group, comprised of Maven, Packagist, npm and Atom [10], between one-third and two-thirds of declarations lag. A second group (NuGet, Pypi and Rubygems) consists of other mature package managers that primarily use open ranges [10] - often relying heavily on *'any'* and *'at-least'* classifications. As these classifications may not lag[3], the package managers overall have low levels of lag (<15%). The third group consists of newer package managers, Cargo, Dub, Elm, Haxelib, Hex, Pub and Puppet. This group has extremely low levels of lag (<12%) with most lag being micro and minor lag.

RQ1.3: How would increased semver adoption affect lag?

An interesting observation from the results of the types of technical lag is that the sum of three categories (minor, minor & micro, and micro) gives the proportion of declarations that semver compliant ranges such as minor ranges would allow a project to have no technical lag. Across all dependencies, this results in 9.2% less dependencies incurring lag, a third of all lag. In Maven, where fixed declarations are the norm, the improvement is even starker - 45% of dependencies (two-thirds of all lagging Maven dependencies) would avoid lag.

[3]NuGet's dependency resolution strategy [27] to take the lowest version possible is at odds with other resolution strategies. Since our analysis assumes the highest version in a range is chosen, the lag in NuGet is under reported.

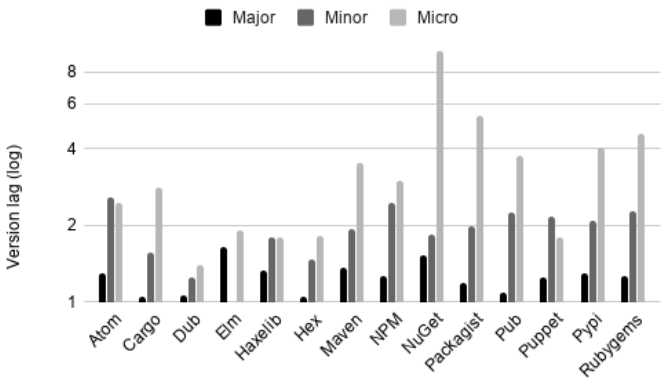

Fig. 3. Technical lag in versions by package manager*

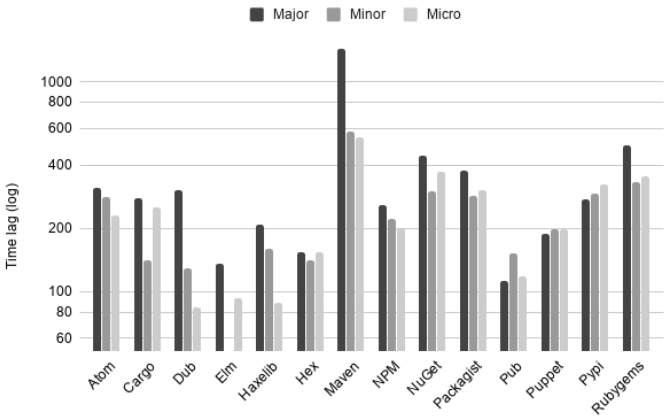

Fig. 4. Technical lag in days by package manager*
* mean values where the relevant type of lag is present in the declaration

RQ1 Summary: Technical lag is common, although the quantity varies widely by package manager. The more permissive open range declarations are much more likely to be current than semver compliant ranges or fixed declarations. Increased use of semver would eliminate a third of all technical lag.

RQ2: How much lag is there in dependencies when they are not current?

Figs. 3 and 4 show the mean value of technical lag (in terms of number of versions) and time lag (in terms of days) by package manager, based on the approach described in Section III-E. Despite dependencies often having technical lag, the median lag for major, minor and micro versions is zero. The data is quite skewed, where a few percent of dependencies are heavily outdated, with the vast majority behind by 0-2 versions. Once dependencies without the respective types of lag have been excluded, the standard deviation of the data is generally similar in size to the mean, indicating a heavily right-skewed distribution.

As shown in Fig. 3, when there is a lag in major releases, all package managers have an average of 1 to 2 major versions lag. The values increase for micro releases, where lag can be close to 10 (NuGet). Micro releases happen much more frequently than minor or major releases, accounting for 67% of the releases in this dataset (2% are major, 16% are minor and 15% are tag updates), so it makes sense that there is more micro lag. NuGet's high micro lag comes from only 0.35% of dependencies with a standard deviation of 3.5 times the mean, indicating that a small handful of outlier projects with large micro lags are responsible for its large result.

For time lag (Fig. 4), the lag in number of days for major releases is higher than minor and micro releases for most package managers. There is a range between Pub, which has an average of 112 days lag, to Rubygems, which has an average of 498 days lag between major versions. In most mature package managers, when lag exists, it is slightly under a year out of date. Maven is an interesting outlier for time lag, as the the value of time lag for major releases is much higher than other package managers (1439 days). This could be due to the Maven ecosystem's long history of simultaneous development in prominent packages, where developers can continue to use an outdated major version and still receive the necessary updates to keep it secure. The Maven ecosystem tends to be conservative (it is the only major package manager that primarily uses fixed declarations [10], and works within the Java philosophy of maintaining backwards compatibility as much as possible. To quote Martin Buchholz: 'Every change is an incompatible change. A risk/benefit analysis is always required.'[4]). It also suffers from binary compatibility issues during updates that may dissuade developers from updating. Other package managers such as npm, on the other hand, inform developers at build time of vulnerabilities in dependencies, motivating them to update dependencies faster. Other generic tools, such as Snyk[5], also provide similar functionalities of warning developers of potential vulnerabilities in libraries.

TABLE IV
FREQUENCY OF DEPENDENCY UPDATES

Project A Update Type	Dependency Update Type		
	Major	Minor	Micro
Major	5.16%	5.43%	3.83%
Minor	0.96%	4.34%	3.15%
Micro	0.27%	0.92%	2.19%
Tag update	0.28%	2.12%	2.70%

RQ2 Summary: When lag exists, it is generally in small amounts. Most dependencies do not get more than 1-2 major or minor versions behind, or 3-5 micro versions, owing to that micro updates represent two-thirds of project updates. The time lag for most package managers is between 0.5-1.5 years.

RQ3: How often do developers update their dependencies, what type of updates are the most common, and what type of project releases contain dependency updates?

This research question investigates update frequencies for fixed declarations only, as discussed in Section III-C.

Fig. 5 shows how often a given dependency is updated, and the types of dependency updates made - whether the dependency is increased a micro, a minor, or a major version.

[4]https://tinyurl.com/y3avjslo
[5]https://snyk.io

The predominant type of dependency update is the micro update, where only the micro number is increased, e.g. `1.0.1` → `1.0.2`. In most package managers, micro declaration updates are two to three times more likely to occur than minor updates. Micro updates are less likely to cause compatibility issues. Semver mandates that they never cause compatibility issues, but this is not always true in practice [8]. Across all ecosystems, major declaration updates, where backward breaking changes are expected, occur in <1% of versions.

The amount developers update their dependencies varies significantly by package manager, with dependencies being updated less than 5% of the time in Pypi, npm and Atom, through to over 45% of the time in NuGet. The types of dependency updates that developers do varies by the sort of update in their own project, as shown in Table IV. Across package managers, major updates to dependencies most often coincide with major changes to the dependent project A. Minor updates are more likely in major or minor updates of project A, while micro updates tend to happen in micro or pre-release changes (the *Tag update* row, e.g. `1.4.0-beta.1` → `1.4.0-beta.2`). These results make sense within the semver construct, as major updates to dependencies are more likely to require a significant amount of refactoring or re-designing of a project if the external APIs are not well insulated from the internal logic, so it may be easier to make these disruptive dependency updates when the dependent project would be undergoing significant changes regardless.

RQ3 Summary: In most package managers, fixed version dependencies are not updated regularly. NuGet (46%), Cargo (33%) and Hex (20%) are the only package managers where fixed dependencies are updated in >10% of version releases. Developers tend to update fixed dependencies much more often in major changes to their own projects than in minor or micro updates. They are also more likely to update to a new major version of the dependency at this point as well.

RQ4: How often do developers update when they lag behind, and do the updates bring them up to date (or are they intentionally staying behind in some way)?

This question also focuses on fixed declarations. Table V shows how often declarations are updated, classified by if they are outdated. The final column shows declarations that were already up to date, so did not require developer intervention. The second-to-last column shows the percentage of time declarations are updated and are now current (without the developer's intervention, they would have become outdated), e.g. $B_{dec_{i-1}}$ is `3.7.2` and B_{dec_i} has been changed to `3.7.3` - the newest version of B available. Together these two columns show declarations that do not have technical lag. Technical lag is updated <10% of the time, with the exception of Cargo and NuGet, where over half of outdated declarations get updated.

The first two columns of Table V together show the dependencies that are outdated. The second column shows the ones that are out of date and did not have any changes made to them. Notice that in most package managers, over half of fixed dependencies fall within this category. The first column

TABLE V
UPDATES VS LAG

	Updated & Outdated	No Update & Outdated	Updated & Current	No Update & Current
	With any lag being considered as outdated			
Atom	0.6%	52.1%	2.9%	44.4%
Cargo	1.8%	23.0%	31.5%	43.7%
Hex	2.3%	35.4%	17.8%	44.5%
Maven	3.8%	62.7%	4.8%	28.7%
npm	0.7%	49.2%	2.3%	47.9%
NuGet	7.3%	28.9%	38.6%	25.2%
Packagist	2.9%	62.0%	2.8%	32.3%
Pypi	0.3%	54.0%	0.9%	44.9%
Rubygems	2.2%	50.7%	7.1%	40.1%
	Allowing for major lag to be classified as current[*]			
Atom	0.5%	41.9%	3.0%	54.6%
Cargo	1.8%	22.7%	31.5%	43.9%
Hex	2.2%	33.2%	17.9%	46.7%
Maven	2.8%	53.8%	5.8%	37.7%
npm	0.4%	38.8%	2.5%	58.2%
NuGet	4.5%	19.7%	41.5%	34.4%
Packagist	2.5%	57.3%	3.2%	37.0%
Pypi	0.2%	49.2%	0.9%	49.6%
Rubygems	1.7%	45.1%	7.5%	45.7%

[*] with only minor or micro lag being reported as *outdated*

are the dependencies that have had some type of change, but are still out of date. These ones are the most interesting, as they indicate that the developer is choosing an outdated version for some reason, rather than going to the newest version.

The definition of being outdated is quite coarse - if there is *any* lag (major, minor or micro), the dependency is considered outdated. However, a common situation in some ecosystems is to have a project with simultaneous development on two major versions (Python 2 and Python 3 being a very well known example). This analysis was rerun, considering major lag as not being outdated, and only considering minor and micro lag to be outdated, allowing for projects in a lagging major version to still be counted as up to date if they are at the newest minor and micro version. This is reported in the second half of Table V. Comparing the second half of Table V with the first half shows that over 10% of projects using fixed declarations in Atom, Maven, npm and NuGet are lagging overall but are up to date within that old major version.

RQ4 Summary: Developers in some package managers update fixed dependencies regularly to stay current, with NuGet and Cargo updating their lagging dependencies most of the time. In most package managers, however, < 10% of dependencies that are out-of-date get updated with a new version.

RQ5: How often do developers make a backwards change to their dependencies, and why?

In the fixed declarations analysed, backwards changes, where a project deliberately increases the technical lag in a dependency, were a rare case. Table VI reports the frequency of backwards changes across all package managers. The results range from zero found in Cargo, up to 0.33% found in Maven - representing 1 in 300 declarations. There are no major trends towards making a particular type of backwards change based on the dependency's version change.

Through the manual review of backward changes and

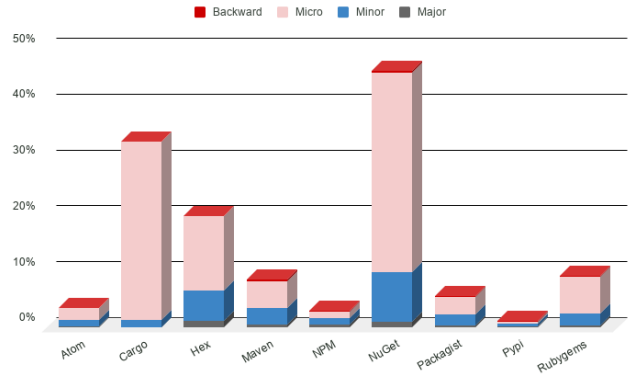

Fig. 5. Updates by package manager

Thematic Content Analysis described in Section III-H, we identified several ***reasons for backward changes***:

Project A goes stable. The most common time we saw backward changes in dependencies is when project A moves from an unstable release to a stable release (26 of the 66 backward changes). These downgrades were not explicitly discussed in the project repositories, but it is likely they tried to update their dependencies to the latest versions in the unstable release candidates, but experienced some problems and reverted when releasing the stable version.

Compatibility issues. The next most common reason for backward changes were compatibility issues (14 of 66). The project A developers often discussed the compatibility issues in the new version before downgrading the dependencies. Some examples include[6]:

> *"... many of our production systems use OTP20 still and this change is not backwards compatible."*
>
> *"Downgrade to [project B] 2.7.5 until [our project] is compatible with 2.8."*

Bug in project B. Another common reason (11 of 66) was that the backward change was made because of a bug introduced in the project B release. For example, after encountering issues after the project B dependency upgrade, one developer identified the problem stems from project B and explains[7]:

> *"I made the following change to [project B]'s source code ... can we push this issue upstream ... could we downgrade [project B] until an upstream fix can be applied?"*

Release not available. Another reason for backward changes (7 of 66) was that the project B release was no longer available in the dependency manager.

Performance issues. The new release of project B introduced performance issues in project A (2 of 66).

New stable version of project B. Project A depended on an unstable minor release of project B, and B released a new stable micro release of the previous minor version (2 of 66).

Consistency. One of the backward changes was made to ensure consistency across the components in project A.

[6]https://github.com/AdRoll/erlmld/pull/9

[7]https://github.com/renovatebot/renovate/issues/196

TABLE VI
FREQUENCY OF BACKWARDS CHANGES

PM	Micro		Minor		Major	
Atom	2	0.01%	3	0.01%	0	0.00%
Cargo	0	0.00%	0	0.00%	0	0.00%
Hex	0	0.00%	2	0.08%	0	0.00%
Maven	1357	0.05%	7453	0.25%	899	0.03%
npm	1561	0.02%	2828	0.04%	1890	0.03%
NuGet	5	0.02%	41	0.13%	37	0.12%
Packagist	176	0.15%	53	0.05%	49	0.04%
Pypi	17	0.03%	16	0.03%	13	0.02%
Rubygems	25	0.02%	33	0.03%	13	0.01%

> *"update [project B] version to match other components"*

Mistake. One of the backward changes was made by mistake. The project went from depending on version 1.0.1 to version 0.0.1. When this was questioned, one of the developers said *"Thank you; it's a typo. We'll fix it."*[8]. The change was reverted in the next release.

The remaining two backward changes with explanations did not provide enough details for us to categorize them into one of our themes. They both mentioned that the change was being made as a fix, but did not provide additional details on what was being fixed. It is likely these were compatibility issues or bugs in project B, but we did not include them in the categories above due to lack of available information.

RQ5 Summary Backwards updates are uncommon across package managers; the highest rate is 1 of 300 changes in Maven. Through qualitative analysis of a sample of backward changes, we found the common reasons for the backward changes include: 1) project A moved from an unstable to stable release, 2) project A is not compatible with the newer version of project B, and 3) project B introduced a bug.

V. THREATS TO VALIDITY

There are a number of threats that can affect the validity of this study.

Construct Validity. We developed a number of scripts that extracted and then processed data obtained from *libraries.io*. We built our scripts in iterations, first testing them on small samples size (from a selected set of package managers) that could be manually verified before employing them on the larger dataset in order improve precision. Our scripts and filtered dataset are publicly available for replication purposes [26].

External Validity. In this study we used a data dump from *libraries.io* that contains data from over 2.7 million projects from 37 package managers. We consider this to be a representative sample size for a large scale empirical study. However, we cannot claim that the results can be generalised for other package managers that are not investigated in this study. Each package manager has different approaches in handling dependencies, which might result in different conclusions.

The dataset contains some missing and incorrect data. We estimate that up to 5% of dependency information within

[8]https://github.com/pouchdb/pouchdb/issues/5430

a given project was missing. Some timestamps were also not correct - some excluded projects in NuGet had default timestamps (1900-01-01), while up to 3% of versions had incorrect timestamps, usually on the order of hours or a few days. This caused some issues of declarations being declared before version releases, but overall constituted a minor issue.

Conclusion Validity. To answer RQ5, we adopted a manual review procedure and Thematic Context Analysis of a selected sample set of pairs in order to identify reasons for backward changes. To improve accuracy, this process was done by two authors. Still, we cannot exclude potential precision issues (false classification), even if this is partially mitigated through the discussion between the authors involved in this process.

VI. Conclusion

In this paper, we showed that technical lag, which can cause security vulnerabilities and make software more brittle, is common. Many dependencies lag, but this varies widely by package manager, as declarations can be anywhere on a continuum from *fixed* (where lag is avoided by regular developer intervention), to *open ranges* (where the latest version can always be chosen by the package manager). When lag exists, it is generally in small amounts. Most dependencies do not get more than 1-2 major or minor versions behind, or 3-5 micro versions behind. It appears that moving from custom ranges and fixed declarations to semver compliant ranges would solve about a third of technical lag present.

Developers tend to update fixed dependencies much more often in major changes to their own projects than in minor or micro updates. In general, fixed declarations are not regularly updated, leading to fixed declarations lagging significantly.

We found backwards updates are uncommon - the highest rate is 1 in 300. When backward changes happen, the most common reasons are: 1) project A moves from an unstable to a stable release, 2) project A is not compatible with the new version of project B, or 3) project B introduces a bug.

Automated dependency tracking tools, such as Dependabot[9] and Renovatebot[10], offer another way to expedite updates and minimise technical lag. Both tools are provided as GitHub plugins and can automatically create pull requests to update dependencies. This allows developers to run tests against the updates to check for compatibility before merging. Widespread adoption of such tools could provide a safe and convenient alternative to flexible declarations for reducing technical lag.

References

[1] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," *arXiv preprint arXiv:1811.00918*, 2018.

[2] J. Cox, E. Bouwers, M. Van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *Proc. Int'l Conf. on Software Engineering*, 2015.

[3] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable open source dependencies: Counting those that matter," in *Proc. Int'l Sym. on Empirical Software Engineering and Measurement*, 2018.

[4] P. Salza, F. Palomba, D. Di Nucci, C. D'Uva, A. De Lucia, and F. Ferrucci, "Do developers update third-party libraries in mobile apps?" in *Proc. Int'l Conf. on Program Comprehension*, 2018.

[5] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Liu, and Y. Wu, "An empirical study of usages, updates and risks of third-party libraries in java projects," *arXiv preprint arXiv:2002.11028*, 2020.

[6] Owasp top ten. [Online]. Available: https://owasp.org/www-project-top-ten/

[7] T. Preston-Werner. Semantic versioning 2.0.0. [Online]. Available: semver.org

[8] S. Raemaekers, A. Van Deursen, and J. Visser, "Semantic versioning versus breaking changes: A study of the maven repository," in *Proc. Int'l Conf. on Source Code Analysis and Manipulation*, 2014.

[9] J. Dietrich, K. Jezek, and P. Brada, "Broken promises: An empirical study into evolution problems in java programs caused by library upgrades," in *Proc. Int'l Conf. on Software Maintenance, Reengineering, and Reverse Engineering*, 2014.

[10] J. Dietrich, D. J. Pearce, J. Stringer, A. Tahir, and K. Blincoe, "Dependency versioning in the wild," in *Proc. Int'l Conf. on Mining Software Repositories*, 2019.

[11] J. M. Gonzalez-Barahona, P. Sherwood, G. Robles, and D. Izquierdo, "Technical lag in software compilations: Measuring how outdated a software deployment is," in *Int'l Conf. on Open Source Systems*, 2017.

[12] 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 *Proc. Int'l Conf. on Software Analysis, Evolution, and Reengineering*, 2015.

[13] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages," in *Proc. Int'l Conf. on Software Maintenance and Evolution*, 2018.

[14] S. Raemaekers, A. Van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *Proc. Int'l Conf. on Software Maintenance*, 2012.

[15] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, "An empirical analysis of technical lag in npm package dependencies," in *Proc. Int'l Conf. on Software Reuse*, 2018.

[16] A. Zerouali, V. Cosentino, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the impact of outdated and vulnerable javascript packages in docker images," in *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering*, 2019.

[17] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the relation between outdated docker containers, severity vulnerabilities, and bugs," in *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering*, 2019.

[18] A. Decan, T. Mens, and M. Claes, "An empirical comparison of dependency issues in oss packaging ecosystems," in *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering*, 2017.

[19] T. Espinha, A. Zaidman, and H.-G. Gross, "Web api growing pains: Stories from client developers and their code," in *Proc. Int'l Conf. on Software Maintenance, Reengineering, and Reverse Engineering*, 2014.

[20] A. Decan and T. Mens, "What do package dependencies tell us about semantic versioning?" *IEEE Transactions on Software Engineering*, 2019.

[21] A. Decan, T. Mens, and E. Constantinou, "On the evolution of technical lag in the npm package dependency network," in *Proc. Int'l Conf. on Software Maintenance and Evolution*, 2018.

[22] F. Roseiro Côgo, G. Oliva, and A. E. Hassan, "An empirical study of dependency downgrades in the npm ecosystem," *IEEE Transactions on Software Engineering*, 11 2019.

[23] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an api: cost negotiation and community values in three software ecosystems," in *Proc. Int'l Sym. on Foundations of Software Engineering*, 2016.

[24] J. Katz, "Libraries.io Open Source Repository and Dependency Metadata," Dec. 2018. [Online]. Available: https://doi.org/10.5281/zenodo.2536573

[25] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative research in psychology*, vol. 3, no. 2, pp. 77–101, 2006.

[26] J. Stringer, A. Tahir, K. Blincoe, and J. Dietrich, "Technical lag of dependencies in major package mangers - replication package," 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4090644

[27] How nuget resolves package dependencies. [Online]. Available: https://docs.microsoft.com/en-us/nuget/concepts/dependency-resolution

[9] https://dependabot.com/

[10] https://renovate.whitesourcesoftware.com/