Deprecation of packages and releases in software ecosystems: A case study on npm

Filipe R. Cogo, Gustavo A. Oliva, Ahmed E. Hassan

Abstract—Deprecation is used by developers to discourage the usage of certain features of a software system. Prior studies have focused on the deprecation of source code features, such as API methods. With the advent of software ecosystems, package managers started to allow developers to deprecate higher-level features, such as package releases. This study examines how the deprecation mechanism offered by the npm package manager is used to deprecate releases that are published in the ecosystem. We propose two research questions. In our first RQ, we examine how often package releases are deprecated in npm, ultimately revealing the importance of a deprecation mechanism to the package manager. We found that the proportion of packages that have at least one deprecated release is 3.7% and that 66% of such packages have deprecated all their releases, preventing client packages to migrate from a deprecated to a replacement release. Also, 31% of the partially deprecated packages do not have any replacement release. In addition, we investigate the content of the deprecation messages and identify five rationales behind the deprecation of releases, namely: withdrawal, supersession, defect, test, and incompatibility. In our second RQ, we examine how client packages adopt deprecated releases. We found that, at the time of our data collection, 27% of all client packages directly adopt at least one deprecated release and that 54% of all client packages transitively adopt at least one deprecated release. The direct adoption of deprecated releases is highly skewed, with the top 40 popular deprecated releases accounting for more than half of all deprecated releases adoption. As a discussion that derives from our findings, we highlight the rudimentary aspect of the deprecation mechanism employed by npm and recommend a set of improvements to this mechanism. These recommendations aim at supporting client packages in detecting deprecated releases, understanding their impact, and coping with them.

Index Terms—Software ecosystem, Deprecation, Release deprecation, Dependency, npm, JavaScript, Node.js

1 Introduction

Deprecation is a mechanism used by developers to communicate that a software's feature is obsolete and its usage should be avoided [1]. Traditionally, deprecation is done at the source code level, allowing developers to deprecate any function from an API [2]. When a function is deprecated, a compile-time warning is typically issued whenever a call to such a function is performed [3]. In the context of software ecosystems, in which client packages depend on a specific release of a provider package, deprecation can be offered at the release level by a package manager. Therefore, developers can deprecate the entire release of a package in a software ecosystem. In such cases, an install-time warning is issued whenever a client package installs a deprecated provider release using the package manager.

Although many different aspects of API deprecation have been studied [2, 4–8], the deprecation of releases in a software ecosystem has never been studied. Simple characteristics such as the frequency with which releases are deprecated remain unknown. The complex network of package dependencies typically found in software ecosystems raises important concerns, such as how often deprecated releases are adopted and whether such an adoption occurs directly (by means of direct dependencies) or indirectly (by means of transitive dependencies). Client packages that directly depend on a deprecated provider release can migrate to a replacement release, i.e., a newer provider release that is not deprecated. Nevertheless, this migration is not always straightforward, since replacement releases might not always exist or be easily discoverable. Indeed, the provision of a proper replacement release and its communication depend entirely on the maintainers of provider packages. Furthermore, when a deprecated provider release is transitively adopted, the client package has no control over the migration to a replacement release. When client packages decide to continue using a deprecated release (e.g., because a migration to replacement release would incur a costly change to the codebase or lead to some incompatibility), they should be aware of the risks of doing so. For instance, a release might be marked as deprecated because it contains a defect. Yet, the rationale behind release deprecations in a software ecosystem has not been investigated by prior literature.

In this paper we study the deprecation of releases in the npm ecosystem, which is the largest software ecosystem to date.[1] In the following, we list our research questions and the key results that we obtained:

RQ1. How often are releases deprecated? The proportion of releases that are deprecated in a software ecosystem shed light about the popularity of the deprecation mechanism, ultimately demonstrating the importance of such a mechanism in an ecosystem. We study the proportion of packages that are fully deprecated (i.e., have all releases deprecated) or partially deprecated (i.e., have some releases deprecated).

- *Filipe R. Cogo is with the Centre for Software Excellence at Huawei, Canada.*
- *Gustavo A. Oliva, and Ahmed E. Hassan are with the Software Analysis and Intelligence Lab (SAIL) in the School of Computing at Queen's University, Kingston, Canada.*
  E-mail: {cogo,gustavo,ahmed}@cs.queensu.ca

We also analyze whether partially deprecated packages offer a follow-up replacement release to their clients. Finally, we perform a manual analysis over a representative random sample of the deprecation messages to understand the rationale behind the deprecation of a release. Our results indicate that:

Deprecation is performed by almost 4% of the packages in npm, with two-thirds of these packages being fully deprecated (i.e., all their releases are deprecated). Almost one-third (31%) of the partially deprecated packages do not offer any follow-up replacement release and 15% of the existing follow-up replacement releases are major releases. Also, withdrawal (i.e., terminating the development of the deprecated package) is the most common rationale for fully deprecating a package (49%) and defect is the most common rationale for deprecating a specific release (63%).

RQ2. How do client packages adopt deprecated releases? Although approximately 4% of the packages have at least one deprecated release, the rate at which deprecated releases are directly or transitively adopted by client packages is also important to determine the typical scope of the deprecation. In this RQ, we study the extent to which deprecated releases are directly and transitively adopted by client packages. Our results indicate that:

At the time of our data collection, 27% of all client packages in npm directly adopt at least one deprecated release. Half of these adoptions target a specific set of 40 provider releases. All these 40 deprecated releases report a replacement release in their deprecation message. In addition, more than half (54%) of all client packages in the ecosystem transitively adopt at least one deprecated release. The median number of direct provider packages that result in the transitive adoption of at least one deprecated release is 1.

The main contribution of this paper is to build a body of knowledge and provide insights into how releases are deprecated in a large software ecosystem, as well as into how client packages adopt such deprecated releases. Our results provide information to client packages regarding how often deprecated releases are published and the common reasons for deprecation, which help in understanding the risks associated with adopting a deprecated release. Also, client packages will find relevant information regarding the identification of transitively adopted deprecated releases and the associated challenges with migrating away from deprecated releases. Finally, we highlight the rudimentary aspect of the deprecation mechanism employed by npm and recommend a set of improvements to this mechanism. Our recommended improvements aim to support client packages in detecting and reasoning about deprecated releases. A supplementary package with our preprocessed data is available online.[2]

Paper organization. The remainder of this paper is organized as follows. Section 2 introduces key concepts that are employed throughout this paper. Section 3 explains the data collection procedures that we followed to conduct our study. Section 4 presents the motivation, approach, and findings from our two research questions. Section 5 discusses the implications of our findings. Section 6 presents the different perspectives from which prior research has investigated the notion of deprecation. Section 7 discusses threats to the validity of our study. Finally, Section 8 concludes the paper by summarizing our observations.

2 Background

In this section we present the key concepts employed throughout our study about deprecation of releases in npm. In Section 2.1, we describe how packages manage dependencies in npm. In Section 2.2, we describe how the deprecation mechanism of npm works and how packages use this mechanism.

2.1 Dependency management in npm

In a software ecosystem, dependencies are set by a *client package*, so that the features of a *provider package* can be reused by this client package. To set a dependency, client packages use a *versioning statement*, which determines the provider package and the respective provider's release that is going to be adopted. Such versioning statements are typically annotated in a configuration file, which in npm is called `package.json` [9]. For example, a client package C that depends on a provider package P can include the following versioning statement in its configuration file: "`P: 1.2.3`". As a result, whenever the package C is installed, the release 1.2.3 of package P is also installed (and eventually loaded at run-time) as a dependency for package C. When a package is installed from npm, the provider packages that are used by means of *transitive dependencies* are also installed [10]. An example of transitive dependency is a client package C that depends on package P_1 that, in turn, depends on package P_2. In this example, C directly depends on P_1 and transitively depends on P_2. Therefore, when C is installed from npm, P_1 and P_2 are also installed.

The release numbering of an npm package follows the semantic version specification.[3] According to this specification, a version number of a release is comprised of three levels, namely: major, minor, and patch. For instance, in release 1.2.3, the number 1 stands for the major level, the number 2 stands for the minor level, and the number 3 stands for the patch level. The semantic version also specifies simple change-related rules for developers to determine how one of the three levels should be incremented when a release is published. In summary, a *major release* should be published whenever a backward-incompatible change is introduced (e.g., an API change). A major release must yield the increment of the major version level, for example, from 1.2.3 to 2.0.0. A *minor release* should be published when some new backward-compatible change is introduced. A minor release must yield the increment of the minor level of the version number (e.g., from 1.2.3 to 1.3.0). Finally, a *patch release* should be published when a bug fix is introduced. A patch release must yield the increment of the patch level of the version number, such as from 1.2.3 to 1.2.4. Although the adoption of the semantic version specification is not mandatory, a prior study shows that, in general, packages in npm comply with this specification [11]. Throughout this paper, we use the increment of the version level between two releases as an estimator for the complexity of the introduced changes in the newer release.

Versioning statements can be of two types, namely a *version range* or a *specific version*. When a version range

statement is used, the installed provider release is the latest provider release that satisfies the stated version range. For example, the versioning statement "`P: >1.2.3`" yields the installation of the latest release of P that is greater than release 1.2.3, which is eventually loaded at run-time by the client package. We refer to the provider release that is effectively installed as the *resolved provider release* (which, in our prior example, is the latest release of P that is greater than 1.2.3). Version range statements in npm have a *range operator*, which determines the restrictiveness vs. the permissiveness of the range of satisfied provider releases. Essentially, the *caret operator* ($\wedge$) satisfies any patch releases that are equal to or larger than 0.1.0 and any minor provider releases that are equal to or larger than 1.0.0. Also, the *tilde operator* ($\sim$) satisfies any patch provider releases, and the *latest operator* satisfies the latest provider release (including major releases). Other operators are also possible,[4] however the caret, tilde, and latest operators are the most commonly used operators [11]. Version range statements need to comply to a specific grammar.[5] In turn, when a specific version statement is used, the installed provider release is determined by the specific stated version. For example, the versioning statement "`P: 1.2.3`" yields the installation of release 1.2.3 of the provider package P.

Client packages can *migrate* from one provider release to another. A migration occurs when the resolved provider release changes from one client release to another. A migration can occur even when the versioning statement used by the client package does not change. For example, suppose that a provider package P at release 1.2.3 eventually publishes a new release with version 1.2.4. A client package C that uses the versioning statement "`P: >1.2.3`" will perform an *implicit migration*, i.e., the resolved provider release will be updated from 1.2.3 to 1.2.4 without any modification of the used versioning statement by the client package. In this case, the migration is an *implicit update*, since the resolved provider release was updated. Client packages can also perform a *downgrade migration* of the resolved provider release by restricting the versioning statement to an older provider release [12]. An implicit update contrasts with an *explicit update*, in the sense that the latter requires a modification of the versioning statement by the client package so that the resolved provider release is updated. For example, if a client package C uses a versioning statement "`P: 1.2.3`", then an update will only be performed after the modification of the versioning statement to "`P: 1.2.4`" (or some version range statement that satisfies the release 1.2.4).

2.2 The deprecation mechanism of npm

The npm deprecation mechanism allows provider packages to communicate to their client packages that the usage of a certain release should be avoided. In contrast with the deprecation mechanism offered by programming languages, in which a certain method can be deprecated, the deprecation mechanism offered by a package manager allows developers to deprecate an entire release. For instance, to deprecate release 1.2.3 of a package P, a developer can use the following command:[6] `npm deprecate P@1.2.3 "this release contains a bug that is fixed in version 1.2.4"`. When a release is deprecated, the npm registry is modified and the release is recorded as being deprecated (to date, the timestamp at which the deprecation is performed is not recorded in npm, making it impossible to analyze the deprecation history of a package). Information about the deprecation of a release can be stated in the *deprecation message*. In the prior example, the deprecation message states that the reason for the deprecation is the presence of a bug in release 1.2.3. Whenever a deprecated release is installed by some user, a warning is issued (at the installation time) and the deprecation message is displayed. For example, when the command `npm install P@1.2.3`[7] is used to install release 1.2.3 of package P, the installation will succeed and the deprecation message will be displayed. The deprecation mechanism adopted by npm also allows one to deprecate a range of versions or, alternatively, the entire package (which essentially deprecates all versions of the package). In such cases, the installation of any version that satisfies the deprecated version range will issue a warning and the deprecation message will be displayed.

Provider packages use the deprecation mechanism to maintain backward compatibility of prior releases. Instead of removing the deprecated release and causing a failure in the client packages that adopt the removed release, provider package developers opt for the deprecation. Whenever a client package that adopts a deprecated provider release is installed, a warning is issued and the provider's deprecation message is displayed. For example, suppose that a client package C adopts the deprecated release 1.2.3 of provider P. Whenever the command `npm install C@latest` is used to install the latest release of package C, a warning with the deprecation message of release 1.2.3 of package P will be displayed. Any deprecated provider release that is adopted by means of a transitive dependency also yields a warning. Deprecation warnings that come from a transitive dependency are difficult to trace,[8] since the issued warning does not explicitly indicate the dependency depth. By dependency depth, we mean the number of downstream dependencies from one client package to a transitively adopted provider. For example, if a client package C depends on a provider P_1 that, in turn, depends on a provider P_2, then the transitive dependency between C and P_2 has a depth of 2.

Client packages might want to migrate away from a deprecated provider release towards some *replacement release* (i.e., a non-deprecated provider release that follows the deprecated one). Client packages that set a dependency using a version range statement will eventually perform an implicit update to the replacement release, as long as the version range is satisfied by the replacement release. Nevertheless, in many cases, the update of the provider to a replacement release requires modifying the versioning statement. Because the installation of a client package yields the installation of all transitive dependencies, updating a deprecated provider package that is transitively adopted can also be desired.[9] However, identifying whether the transitively adopted deprecated release has a follow-up replacement release (and should consequently be updated) might not be trivial. In particular, npm provides the `npm outdated`[10] tool that checks which providers can be updated. To check

transitive dependencies, this tool requires an argument that determines the maximum dependencies depth to be checked. For example, `npm outdated --depth 1` will check whether the providers of the direct providers can be updated. To date, the value for this argument needs to be determined by a trial-and-error approach, i.e., iteratively increasing the value of the `--depth` argument until all transitive dependencies are checked.

3 Data Collection

In this section, we summarize our data collection approach (a detailed description is given in Appendix A). Our data collection approach entails three main steps: collect package metadata, analyze package releases, and analyze package dependencies. In the following, we describe each of these steps.

1) Collect package metadata. In this step, we collect the metadata of npm packages from the `package.json` files. We obtained all `package.json` files that were stored in the npm registry as of May, 2019. Each collected `package.json` file corresponds to a distinct package P in npm. The `package.json` file of a given package P contains release-related metadata for all releases of P. The release-related metadata include the release version number, the timestamp at which the release was published, the dependencies that are set in the release (i.e., the provider package name and the respective versioning statement), and the deprecation message in case the release is deprecated. The output of this procedure is a list of 976,631 `package.json` files, each one corresponding to a unique npm package. We only analyze release activities that can be found in the npm data and we do not consider activities performed on the packages' code base (e.g., commits of `package.json` files). Our rationale is that there is no trivial approach to link npm releases with commits of a package. Even if the link between npm releases and the commits could be accurately determined, performing a large-scale analysis of the `package.json` files at the commit level (in particular, resolving the provider versions revision by revision) becomes an infeasible task. Hence, to consider release activities at the code base level, we would have to study a smaller sample of packages, thus not portraying the npm ecosystem as a whole, which we find important for a first paper in the area.

2) Analyze package releases. In this step, for each package obtained in the previous step, we sort the adjacent package releases and determine how the version numbers change between two adjacent releases (e.g., whether the latter release is a major, minor, or patch release). Because some packages adopt parallel development branches (e.g., release 1.2.3 is published after the existence of release 2.0.0), releases are sorted according to a branch-based ordering (in contrast to a chronological- or a numerical-based ordering). A branch-based ordering schema allows a release to be considered the predecessor of more than one release. For example, if releases 1.0.0, 2.0.0, and 1.0.1 are published in chronological order, then the branch-based ordering would allow 1.0.0 to be the predecessor of both releases 1.0.1 and 2.0.0. More details about such an algorithm can be found in our prior work [12]. The output of this procedure is a list of 7,829,364 releases of the packages (clients and providers) obtained in the previous step.

3) Analyze package dependencies. In this step, we select the dependencies that are set in the latest client releases (6,178,019 dependencies, corresponding to 6.3% of all dependencies from all releases). We choose the latest client release because npm does not store the date at which a package release was deprecated. Hence, we can only correctly assume that the client package is using a deprecated provider release at our data collection date. Finally, we resolve the version of the providers that are used by each client package in their latest release.

4 Results

In this section, we present the results of our two RQs. For each RQ, we discuss its motivation, the approach that we used to address it, and the results that we obtained.

4.1 RQ1: How often are releases deprecated?

Motivation. Deprecation mechanisms are employed to discourage the usage of a certain piece of code. Deprecation can occur at different levels of granularity within a software system. Traditionally, prior studies focus on API deprecation, which typically operate at the function level [1, 3, 5, 6, 13]. However, the offered deprecation mechanism by the npm package manager operates at a higher level, allowing one to deprecate either a release, a range of releases, or the entire package (i.e., all releases of the package). Maintainers of npm have drawn attention to features that can discourage developers to use unmaintained packages.[11] The deprecation mechanism is the main existing tool that allows developers to communicate that a certain package or release is no longer maintained. Yet, to the best of our knowledge, no prior studies have investigated how often deprecation occurs at the release-level within a software ecosystem.

Ideally, the deprecation of a release should occur after a replacement release is published (i.e., some newer release that is not deprecated). In such cases, client packages that are using a deprecated provider release can perform an update targeting the replacement release. Therefore, it is important for client packages to know how often follow-up replacement releases are available. Also, to assess the likelihood of an implicit migration to a replacement release, client packages can compare their versioning statements against the proportion of replacement releases that are patch, minor, or major releases. Nonetheless, when a replacement release is not available, client packages that want to migrate away from a deprecated release need to perform a downgrade (i.e., migrate to an older release that is not deprecated). The drawback of performing a downgrade is that client packages can miss new features and bug fixes from newer releases [12, 14]. Hence, it is interesting for client packages to understand how many patch, minor, and major releases are back skipped when a downgrade from a deprecated release to an older non-deprecated release is performed. For example, a downgrade that changes the resolved provider release from 1.0.1 to 1.0.0 is back skipping one patch release.

Approach. To determine the prevalence of release deprecation in npm (**Result 1**), we calculate the proportion of packages that have at least one deprecated release and the proportion of all releases that are deprecated in the ecosystem. We also determine the prevalence of packages that are *fully deprecated* (i.e., all releases are marked as deprecated) or *partially deprecated* (i.e., some releases are marked as deprecated, but not all). To this end, for each package, we calculate the proportion of deprecated releases over all its published releases.

Next, we study how often deprecated releases have a follow-up *replacement release* (**Result 2**). More specifically, we calculate the proportion of partially deprecated packages that have some non-deprecated release that follows the *highest deprecated release* (i.e., the deprecated release with the largest version number within a package's sequence of releases). We focus on the highest deprecated release because we want to understand the availability of replacement releases as of the data collection date (since we do not have historical deprecation information). As an example, Figure 1 shows a hypothetical sequence of releases of a partially deprecated package with a replacement release. The 2.1.1 release is the highest deprecated release and the following 3.0.0 release is the replacement release. We also investigate how often a partially deprecated package that does not have a replacement release is actually intended to be fully deprecated. To do so, we manually analyze a representative random sample of 364 ($\pm 5\%$ C.I., 95% C.L.) deprecation messages of the partially deprecated packages that do not have a replacement release.

In **Result 4**, we calculate the proportion of replacement releases that are major, minor, or patch releases (in the example shown in Figure 1, the replacement release is a major release). We also measure the number of provider releases with the highest version level (i.e., major > minor > patch) that are published in between the older non-deprecated and the highest deprecated releases (this measure is called technical lag [15]). For example, in Figure 1, in between the older non-deprecated release and the highest deprecated release, one major release is published (2.0.0), one minor release is published (2.1.0), and three patch releases are published (2.0.1, 2.0.2, 2.1.1). In this scenario, the technical lag is one major provider release.

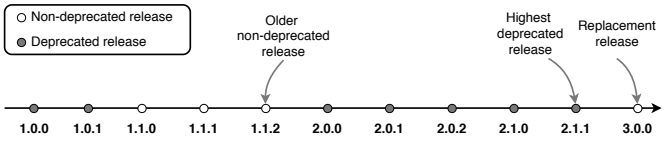

Fig. 1: Older non-deprecated and replacement release of a hypothetical partially deprecated package.

Lastly, in **Result 3** we determine the common rationales behind the deprecation of packages and releases and how often such rationales are associated with the deprecation of all releases of a package in contrast to the deprecation of a specific release (or range of releases) of a package. To this end, we manually analyzed a statistically representative sample ($\pm 5\%$ C.I, 95% C.L) of the deprecation messages used by npm packages. We sampled a total of 381 out of the 44,112 unique deprecation messages used by different packages. We performed an open coding [16] to categorize the rationale behind release-level deprecation. We also differentiate deprecation messages that are associated with the deprecation of all releases of a package (full package deprecation), in contrast to the deprecation of a specific release of a package (partial package deprecation). Finally, we calculate how often a deprecation message reports a replacement package or a replacement release.

Result 1) ***3.7% of the npm packages have at least one deprecated release.*** There are 253,501 deprecated releases in npm (3.2% of all releases) and 31,810 packages with at least one deprecated release in npm, representing 3.7% of all packages in the ecosystem. Two-thirds (66%) of the packages with a deprecated release are fully deprecated (i.e., have deprecated all releases). Figure 2 shows the proportion and the total number of deprecated releases per package with at least one deprecated release. A total of 29% of the fully deprecated packages have a single release (represented by the darkest portion of the largest bar in Figure 2) and 20% have ten or more deprecated releases (represented by the sum of the two lightest portions of the largest bar in Figure 2). Among the partially deprecated packages, 69% have more than one deprecated release. That is, partially deprecated packages tend to either deprecate a range of releases or apply the deprecation mechanism more than once over time. We cannot distinguish between these two cases because the `package.json` files do not record historical information about deprecation (i.e., one can only know whether a certain release is deprecated or not).

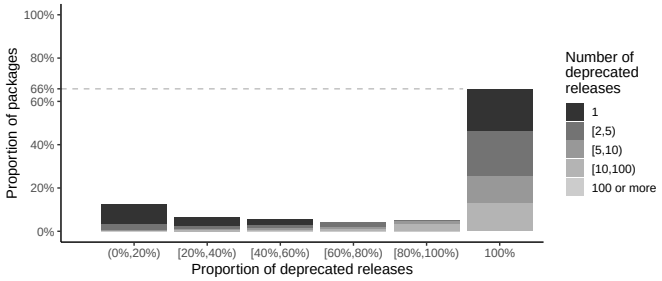

Fig. 2: Number and proportion of deprecated releases per package (excluding packages without deprecated releases). The dashed line shows the proportion of fully deprecated packages.

Result 2) ***31% of the partially deprecated packages do not have a replacement release.*** By manually analyzing the deprecation messages of a representative sample of the partially deprecated packages without a replacement release, we observe that 65% of such deprecation messages report that either the package maintenance was abandoned or that the package was superseded by another package. That is, we estimate that 20.15% (65% of 31%) of all partial deprecations are actually intended to be full deprecations. This estimated percentage corresponds to 1,987 packages. We conclude that, although these packages have been partially deprecated (including their latest release), the package was in fact intended to be fully deprecated. Examples of deprecation messages that indicate that packages were intended to be fully deprecated include: "The functionality

of these package is now directly integrated in 'sql-pg"' (deprecation message of package sql-pg-helper) and "Use unitejs-webdriver-plugin instead" (deprecation message of package unitejs-polymer-webdriver-plugin).

Result 3) ***Withdrawal is the most common rationale for the deprecation of a package (49%) and defect is the most common rationale for the deprecation of a release (63%).*** Almost two-thirds (64%) of the deprecation messages report the rationale behind the deprecation of a package or release. We found five rationales: withdrawal (e.g., the development of a package is no longer maintained in npm), supersession (e.g., a deprecated release is replaced by a newer, improved release), defect (e.g., a certain functionality is discovered to be buggy), test (e.g., package is published for test purposes only), and incompatibility (e.g., dependency incompatibility). More thorough descriptions and examples can be found in Appendix B.

The proportion with which a given rationale is associated with the deprecation of either a package or release is shown in Table 1. In total, 80% of the deprecation messages are for deprecated packages, whereas 20% of the deprecation messages are for deprecated releases. In the deprecation messages for deprecated packages, 51% report a replacement package. Similarly, in the deprecation messages for replacement releases, 51% report a replacement release. From the deprecation messages that regard the supersession of a package, 84% state that the package was moved or renamed. In turn, from the deprecation messages that regard the supersession of a specific release, 13% states that the release was superseded by another.

TABLE 1: The proportion with which each rationale is associated with the deprecation of a package or a release.

Rationale for deprecation	Package deprecation (80%)	Release deprecation (20%)	Example
Withdrawal	49.0%	12.0%	*"This module is no longer maintained"*
Supersession	45.0%	20.0%	*"Version 1.x branch of Iridium has been superseded by v2.x."*
Defect	0.5%	63.0%	*"Buggy implementation of class mixins."*
Test	5.0%	2.5%	*"Use version 1.0.0, this was a prerelease (...)"*
Incompatibility	0.5%	2.5%	*"Old versions not compatible with sqb >0.7.0."*

Result 4) ***68% of the replacement releases are patch releases, 17% are minors, and 15% are majors.*** Although the majority of the replacement releases introduce simpler changes (patch and minor releases), a non-negligible number of the replacement releases introduce more complex changes (i.e., major releases, which might introduce backward incompatible changes). Our analysis shows that even client packages that set a restrictive range for their versioning statements (e.g., version ranges that accept only patch updates) will likely perform an implicit update to the replacement release. However, client packages must be aware of the possibility of having to integrate major releases of the providers when they are willing to adopt a replacement release. Furthermore, client packages might prefer to downgrade the provider from the highest deprecated release to the older non-deprecated release, instead of integrating a major replacement release. In this case, the median number of provider releases that need to be backskipped with the downgrade (i.e., the introduced technical lag) is either one patch, one minor, or one major release of the provider.

RQ1: How often are releases deprecated?

From all npm releases, 3.2% are deprecated.

- 3.7% of the npm packages have at least one deprecated release.
- 66% of the packages with deprecated releases are fully deprecated.
- 31% of the partially deprecated packages do not publish a replacement release.
- 68% of the existing replacement releases are patches, 17% are minors, and 15% are majors.
- *Withdrawal* is the most common rationale for the deprecation of a package (49%) and *defect* is the most common one for the deprecation of a release (63%).

4.2 RQ2: How do client packages adopt deprecated releases?

In this RQ, we investigate how client packages adopt deprecated releases. We differentiate between direct and transitive adoptions (Section 2.1). Direct adoptions are under the control of the client package, since they originate from direct dependencies (i.e., those specified in the `package.json` file). For these direct adoptions, we determine the frequency with which they happen and how they relate to the type of versioning statements that are employed by the client packages. Transitive adoptions of deprecated releases happen indirectly and thus are not under the control of client packages. As part of this RQ, we also determine the frequency with which clients transitively adopt deprecated releases, as well as how deep these adoptions happen in the dependency tree. Direct adoptions are discussed in Section 4.2.1 and transitive adoptions are discussed in Section 4.2.2.

4.2.1 Direct adoption of deprecated releases

Motivation. In this section, we investigate how frequently client packages directly adopt deprecated releases. Such an investigation will give insights into how effective the deprecation mechanism is in pushing client packages away from deprecated provider releases. In particular, we investigate whether there are deprecated releases that are still adopted by a large number of clients and whether a replacement release (or replacement package) exists for them. The latter is particularly relevant for client packages that value keeping their providers up-to-date.

Approach. We calculate the proportion of client packages that, in their latest release, directly adopt at least one deprecated provider release (**Result 5**). We then investigate whether there are deprecated releases that are adopted by more client packages than other deprecated releases (**Result 6**). To this end, for each deprecated release d, we calculate a_d, which is the number of times that d is directly adopted by a client package. We then divide a_d by the total number of direct adoptions of deprecated releases, obtaining the proportion p_d of direct adoptions of the deprecated release d. We define a *popular deprecated release* as any release d that belongs to the smallest subset of deprecated releases for which p_d sums up to 50% (i.e., the deprecated releases that concentrate half of all adoptions). The adoption of deprecated provider releases was assessed only at the latest

client release and, as a consequence, our results denote such adoptions at the time of our data collection (see Section 3). For the sake of simplicity, we will refer to a "client package release" simply as a "client package" (implicitly referring to the latest client package release).

Next, we determine how often popular deprecated releases have a replacement release (**Result 7**). Because a popular deprecated release can be a release of either a partially or a fully deprecated package (see Section 4.1), we employ a different method for those two cases. Basically, for popular deprecated releases of fully deprecated packages, we search for a *replacement package* instead of a replacement release. In the following, we describe the two employed methods:

• *Partially deprecated packages:* To determine whether a popular deprecated release of a partially deprecated package has a replacement release, we search for the existence of any non-deprecated release whose version number is larger than the newest deprecated release (see Figure 1).

• *Fully deprecated packages:* To determine whether a popular deprecated release of a fully deprecated package has a replacement package, we perform a manual analysis over the deprecation messages. We search for replacement packages by reading the deprecation message of the popular deprecated releases and any documentation that is mentioned in such deprecation messages (e.g., tutorials that explain how client packages should perform changes to migrate away from the deprecated release).

After determining the replacement releases and packages, we estimate the date at which a popular deprecated release was deprecated (**Result 8**). The motivation for performing such an estimate is twofold. First, we want to safeguard the validity of our analysis, since a given deprecated release might be massively adopted because client packages did not have enough time to migrate away from that release (e.g., releases that were deprecated at a date that is close to our data collection). Second, we want to analyze the time taken as well as the frequency at which client packages migrate away from a popular deprecated release (to either a replacement package or a release). We estimate the deprecation date by gathering two pieces of evidence. First, we verify whether any mentioned documentation in the deprecation message includes the date at which the release was deprecated or some event that drove the deprecation and whose date can be obtained (e.g., the first stable release of a reported replacement package). Second, when none of these pieces of information are reported in the existing documentation, we assume that the deprecation occurred at the package's latest release date.

We use survival analysis [17] to study how the probability of a client package to migrate away from a popular deprecated release changes over time. Survival analysis is a statistical technique that is suitable for estimating the probability of the occurrence of an event over time (i.e., a migration away from a popular deprecated release by a client package), even when there are instances in the data for which the event is not observed (i.e., right-censored data). For each adoption of a popular deprecated release by a client package, we verify whether the adoption is followed by a migration away from the deprecated release. For the cases in which the migration occurs, we calculate the time spanned between the deprecation date and the migration date (see Figure 3). Similarly, we treat as right-censored data the adoptions for which we did not observe a migration until the date of our data collection. We only consider adoptions of popular deprecated provider releases that occur before the deprecation. Also, we separate the migrations from popular deprecated releases of a fully deprecated provider package from those of partially deprecated packages. For example, the popular deprecated release babel-preset-es2015@6.24.1 does not have a replacement release, but rather a replacement package (babel-preset-env). In this case, we detect a migration whenever a client package of babel-preset-es2015@6.24.1 eventually starts adopting babel-preset-env. On the other hand, the popular deprecated release react-dom@16.2.0 does have a replacement release and, therefore, we detect a migration whenever a client adopting the deprecated release starts adopting any of the replacement releases of react-dom@16.2.0 (i.e., react-dom@>=16.2.1). Finally, we visualize the migration probability over time by plotting the complement of the Kaplan-Meier survival curve. While Kaplan-Meier survival curves are plotted as an interpolation of the survival estimates $S(t_i)$, for each time t_i, we plot survival curves as an interpolation of the complement of the survival estimate, $1 - S(t_i)$, for each time t_i.

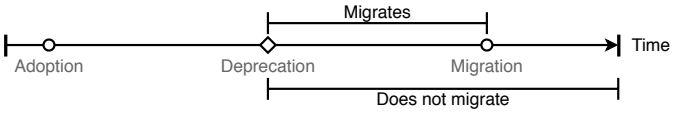

Fig. 3: Survival analysis modelling for the time to migrate away from a popular deprecated release.

Result 5) ***27% of the client packages directly adopt at least one deprecated provider release.*** In RQ1, we observed that only a small proportion of npm packages (3.7%) deprecated at least one release. Yet, when analyzing client adoption of deprecated releases, we note that 27% of all client packages in npm adopt at least one deprecated release.

Result 6) ***A remarkably small proportion of the deprecated releases are massively adopted by client packages.*** More specifically, 75% of all adoptions of deprecated releases concentrate on only 2.6% of all deprecated releases (Figure 4). The top 40 most frequently adopted deprecated releases account for 50% of all adoptions of deprecated releases. We call these 40 deprecated releases as popular deprecated releases. In total, 80% (32 out of 40) of the popular deprecated releases are from a fully deprecated package and 20% (8 out of 40) are from a partially deprecated package.

Result 7) ***All popular deprecated releases have a deprecation message that indicates a replacement package or release.*** This result indicates that popular provider package developers always support clients in determining candidate releases (or packages) to be migrated to in face of release deprecation. Furthermore, we estimate that all popular deprecated releases have been marked as such for at least 6 months since our data collection date, showing that client packages had a reasonable amount of time to migrate away from them. In Appendix C, we list all the 40 popular deprecated

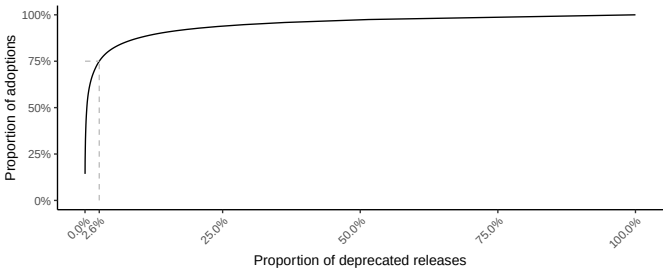

Fig. 4: Cumulative histogram for the proportion of client packages that depend on a deprecated provider release.

releases, their respective replacement release or package, and our estimate of their deprecation date.

Result 8) ***More than half (59%) of the adoptions of a popular deprecated release of a partially deprecated package are not followed by a migration, even though a replacement release is available.*** Moreover, more than 94% of the adoptions of a popular deprecated release of a fully deprecated package are never followed by a migration to a replacement package. Figure 5 shows the probability of migration over time. The probability of an earlier migration to a replacement release is significantly larger than the probability of an earlier migration to a replacement package. Moreover, all migrations to a replacement release occur in less than 285 days after the deprecation, while migrations to a replacement package occur in up to 877 days. Similarly, half of all migrations to a replacement release take less than 22 days, while half of all migrations to a replacement package take less than 315 days.

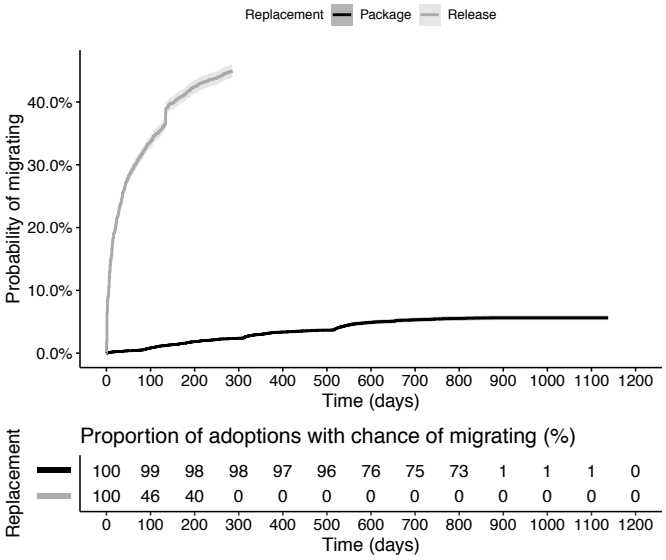

Fig. 5: Survival curve for the migration away from a popular deprecated release.

4.2.2 Transitive adoption of deprecated releases

Motivation. In a software ecosystem, a deprecated provider release can be transitively adopted by a client package (Section 2.1). While client packages can choose the provider packages that are directly adopted, provider packages that are transitively adopted are out of the scope of client packages. As a consequence, tracking the transitive adoption of a deprecated release can be challenging to client packages. Even when providers that are transitively adopted can be tracked, the provided tools by npm that help client packages to check whether such providers can be updated require the specification of the dependency depth parameter (see Section 2.2), which is usually unknown. Also, there is no trivial approach to update transitive providers. These issues show how challenging dealing with the transitive adoption of deprecated provider releases is. By knowing how often deprecated provider releases are transitively adopted, client packages can estimate the likelihood of having to deal with a transitively adopted deprecated release. Also, transitive providers can be updated as a consequence of the update of a direct provider. Hence, client packages can benefit from estimating how often a directly adopted provider package results in the transitive adoption of a deprecated release. In this RQ, we determine how often client packages transitively adopt a deprecated provider release, what the typical dependency depth is, and how often a provider that is directly adopted results in the transitive adoption of a deprecated release.

Approach. We study the transitive adoption of deprecated releases by analyzing the *dependency tree* of the latest release of client packages. Such dependency trees contain the provider package releases that are directly and transitively adopted by each client in their latest release. To obtain these dependency trees, we run the `npm install` command (to install the client package release and its dependencies) followed by the `npm ls` command (to obtain the dependency tree in a parsable format). The dependency trees represent the resolved provider releases at the time that we run the `npm install` tool (February 2020), whereas the timestamp of the latest client package releases were obtained from our data set (which was collected on May 2019). For simplicity, in this subsection we refer to the "latest client package release" simply as a "client package". Similarly, we refer to the provider releases that are directly and transitively adopted by the client packages as "provider packages". Furthermore, given the total number of client packages to be analyzed (595,052), we draw a statistically representative random sample of size 16,641 (99% confidence level, $\pm 1\%$ confidence interval). Our sampling method preserves the distribution of the number of provider packages per client package that is found in the population. In total, the studied dependency trees have 71,663 installed packages (among client and provider packages) and 5,796,506 dependencies. With the usage of version range statements by client packages, the installation of an npm package is not reproducible, since installations that are run at a different time can also produce different results. To mitigate this limitation regarding the reproducibility of our results, in our supplementary material we provide the data that was obtained during the client package installations.

After obtaining the dependency trees, we estimate the proportion of client packages that transitively adopt at least one deprecated provider release (**Result 9**). In addition, to understand the relation between the direct and transitive

adoptions of deprecated releases, we calculate how often client packages that directly adopt at least one deprecated provider release also transitively adopt at least one deprecated provider release (and vice-versa).

We also calculate the distribution of the *deepest dependency depth of a deprecated provider release* for each client package (**Result 10**). The deepest dependency depth of a deprecated provider release is the largest distance from the client package to any adopted deprecated release in the dependency tree. For instance, suppose that a client package C transitively adopts a deprecated provider release at depths 2 (i.e., the provider of a provider is deprecated) and 3. For client package C, the deepest dependency depth of a deprecated provider is 3. We use the deepest dependency depth of non-deprecated releases as a baseline for assessing the deepest dependency depth of deprecated releases. We use the Wilcoxon Signed Rank test ($\alpha = 0.05$) to verify the hypothesis that the distribution of the deepest dependency depth of deprecated and non-deprecated releases differ. We also assess the effect size using the Cliff's Delta estimator with the following classification [18]: negligible for $|d| \leq 0.147$, small for $0.147 < |d| \leq 0.33$, medium for $0.33 < |d| \leq 0.474$, and large otherwise.

Finally, we analyze the distribution of the number and the proportion of directly adopted providers that result in the transitive adoption of at least one deprecated release (**Result 11**). As an example, suppose that a client package C directly adopts three providers, namely P_1, P_2, and P_3. Suppose that adopting P_1 results in the transitive adoption of deprecated release d_1 and that adopting P_2 results in the transitive adoption of deprecated releases d_2 and d_3. In such hypothetical scenario, the number of directly adopted providers of C that result in the transitive adoption of at least one deprecated release is 2 (P_1 and P_2), while the proportion is 66% (2 out of 3 directly adopted providers).

Result 9) ***54% of all client packages transitively adopt at least one deprecated release.*** For client packages that adopt a deprecated provider release, the majority of such adoptions is exclusively transitive (see Table 2). In particular, the number of client packages that adopts a deprecated provider exclusively by means of a transitive dependency (5,406) is almost 5 times larger than the number of client packages that adopts a deprecated provider by means of a direct dependency (1,107). This scenario has the potential to confuse client packages. In fact, a common theme of discussion between developers evidences this situation: developers get confused when deprecation messages come exclusively from transitively adopted deprecated providers.[12,13,14,15,16]

TABLE 2: The number of client packages that directly and transitively adopt at least one deprecated provider release (only client packages that adopt a deprecated provider release are shown).

Client packages		**Directly adopt deprecated releases?**	
		No	*Yes*
Transitively adopt deprecated releases?	*No*	–	1,107 (11%)
	Yes	5,406 (54%)	3,461 (35%)

Result 10) ***In 90% of all adoptions of a deprecated provider release, the deepest dependency depth is no larger than 6.*** Figure 6 depicts a histogram for the deepest dependency depth of deprecated and non-deprecated provider releases. While the median dependency depth for non-deprecated releases is 6, the median for deprecated releases is 4. The difference between the distributions is statistically significant (p-value < 0.05) with a medium effect size ($|d| = 0.46$). This observation leads us to conclude that client packages should be able to trace the transitive adoption of a deprecated release with relative easiness (compared with the transitive adoption of non-deprecated releases) and to take actions to update the transitively adopted deprecated release. More specifically, it is plausible to client packages to either update a direct provider as an attempt to transitively update the deprecated release or to suggest developers of transitively adopted providers to update their own providers. Nonetheless, in practical terms, it might be still challenging for client packages to trace transitive dependencies. Therefore, our conclusion is limited to the comparison between deprecated and non-deprecated releases.

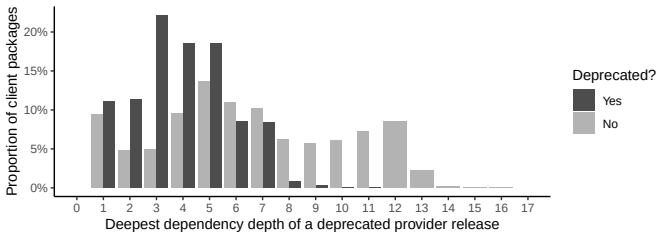

Fig. 6: Histogram of the deepest dependency depth of deprecated and non-deprecated releases.

Result 11) ***The median number of direct providers that result in the transitive adoption of at least one deprecated release by a client package is 1.*** In other words, for 50% of the client packages, one single direct provider would need to be updated or replaced as an effort to cease the transitive adoption of a deprecated release. In addition, Table 3 shows that the median proportion of direct providers that account for the transitive adoption of at least one deprecated release is 25%.

TABLE 3: Descriptive statistics for the total number and the proportion of direct providers that result in the transitive adoption of at least one deprecated release.

Providers	**Min.**	**Q1**	**Median**	**Mean**	**Q3**	**Max.**
Total	1.0	1.0	1.0	1.98	2.0	60.0
Proportion	2.0%	16.6%	25.0%	32.5%	42.8%	100.0%

RQ2: How do client packages adopt deprecated releases?

The direct adoption of deprecated releases is highly skewed, with the top 40 popular deprecated releases accounting for more than half of all deprecated releases adoption.

- All the top 40 popular deprecated releases have a deprecation message that reports a replacement package or release, which eases the migration from such deprecated releases.
- 54% of all client packages transitively adopt at least one deprecated release.
- In 90% of the cases where a deprecated provider release is adopted, the deepest dependency depth is no larger than 6.
- A median of one in each four providers that are directly adopted result in the transitive adoption of at least one deprecated release.

5 Discussion

In this section, we discuss the findings presented in Section 4. We divide our discussion in two topics: improvements to the npm deprecation mechanism (Section 5.1) and an assessment of the impact of deprecated releases on client packages (Section 5.2).

5.1 Improving the deprecation mechanism

Although a small proportion of npm packages make use of the deprecation mechanism (Result 1), a noteworthy proportion of client packages directly adopt deprecated releases (Result 5). Therefore, we consider release-level deprecation to be a relevant aspect of the npm ecosystem. Despite such relevance, the deprecation mechanism provided by npm is fairly rudimentary (see Section 2.2 for a description on how the npm deprecation mechanism works). In the following, based on our observations from Section 4, we propose specific improvements to the deprecation mechanism.

Implication 1) ***The deprecation mechanism should encourage developers to provide more meaningful deprecation messages.*** The rationale for the deprecation is not reported in 36% of the deprecation messages (Result 3). As presented in Result 3, as well as in prior studies regarding API deprecation in the Maven ecosystem for Java [2], there is a diversity of rationales behind a deprecation that should be explained to client packages. Informing the rationale behind a deprecation allows client packages to assess the trade-off between the risk of adopting the deprecated release and the effort to migrate away from this release. Therefore, npm should encourage developers to provide better reasons for the deprecation of a release (e.g., by providing standardized deprecation messages based on the five identified rationales for deprecation).

In addition, the replacement release is not reported in approximately half (49%) of the deprecation messages (Result 3). Reporting a replacement release supports client packages in migrating away from the deprecated release (i.e., a clear and explicit option is given to them). Hence, we argue that the npm deprecation mechanism should support package developers in informing what the replacement release is (e.g., by automatically detecting the existence of a newer non-deprecated release and, also, by warning developers when the latest package release is being deprecated). A feature request endorsed by npm maintainers suggests the creation of an "override" field on the `package.json` file,[17] which would be useful to store replacements of deprecated releases. In addition, the npm deprecation mechanism should record a deprecation timestamp (to date, this information is not recorded in the npm registry) and a severity level for the deprecation. These information would help client packages on 1) evaluating for how long a release is considered deprecated and 2) assessing the risks of adopting a deprecated release, based on the perspective of the developer of the provider package.

Nonetheless, simply reporting a replacement release might not be sufficient to allow client package developers to evaluate the difficulty to migrate to the replacement release. In fact, deprecation messages should point to a migration guide documentation that facilitates client packages to understand the impact of changing from a deprecated to the replacement release. For example, a migration guide could provide information on which features or set of functionalities are impacted by the deprecation and, therefore, were modified by the replacement release.

Implication 2) ***The deprecation mechanism should interactively prompt for contextual information about the deprecation.*** To reason about what lessons can be learned from the deprecation mechanism of other ecosystems, we analyzed the features of seven other package managers, chosen according to the size of their ecosystem (i.e., the number of packages),[18] namely CPAN for Perl, Crates.io for Rust, Maven for Java, Nuget for .NET, Packagist for PHP, PyPi for Python, and RubyGems for Ruby. We observe that four out of the seven package managers do not implement a deprecation mechanism (CPAN, Maven, PyPi, and RubyGems). In Crates.io, the deprecation mechanism was implemented after an explicit request of this feature by the package manager users,[19,20,21] indicating the relevance of the mechanism to the package manager users. In this package manager, developers have the option to explicitly set the maintenance status of their packages and the notion of deprecation is incorporated into this feature. In particular, when the maintenance status of a package or release is set as deprecated, a status indication badge is displayed in the package's description (package manager user interface). However, no option for describing the rationale behind the deprecation and its potential impact on client packages, nor the explicit indication of a replacement package or release, is provided by this deprecation mechanism. Also, there is no option to deprecate a range of releases. Both Packagist and Nuget have an integrated user interface for the deprecation mechanism, which might help users to understand how the deprecation mechanism works and to provide better information regarding the deprecation. However, Packagist does not contain any official documentation for the deprecation mechanism, which can be detrimental to users. The most complete set of features can be observed in Nuget. The deprecation mechanism of Nuget allows users to select between one out of two pre-established deprecation rationales or, alternatively, to input a user-defined rationale. In addition, Nuget explicitly prompts users for a replacement release. The features that are implemented by the deprecation mechanism of each of the aforementioned package managers are summarized in Table 4.

Based on our observations regarding the deprecation

TABLE 4: Summary of the features implemented by four package managers with a deprecation mechanism.

Package manager	Official document.	Support for describing rationale	Graphical user interface	Prompts for replacement release	Deprecation of range of releases
npm	✓				✓
Crates.io					✓
Packagist			✓		
Nuget	✓	✓	✓	✓	✓

mechanism of npm and other package managers, we highlight the following set of best practices for the implementation of a deprecation mechanism for software ecosystems:

1) Official documentation about the deprecation mechanism should be provided as part of the own package manager documentation.
2) Documentation should make explicit the common scenarios for which deprecation is encouraged. In specific, a pre-specified set of the common rationale for deprecation should be displayed.
3) The deprecation mechanism should have a user interface integrated with the package manager, both to display and to input information about a deprecated release or package.
4) The deprecation mechanism should prompt for a replacement package or release and, ideally, prevent a release to be deprecated without having a replacement.
5) Deprecation should be allowed for a single release, a range of releases, or the whole package.

Implication 3) ***The deprecation mechanism should periodically warn client packages about the adoption of a deprecated provider release.*** Even though all popular deprecated releases report a replacement release in their deprecation message (Result 7), such releases are still massively adopted by client packages (on their latest release, as of our data collection date) (Result 6). The migration away from popular deprecated releases of partially deprecated packages should be performed at low cost by client packages, since the majority of the replacement releases are patch releases. However, less than half of the client packages of a popular deprecated release eventually migrate away from such releases (Result 8). We hypothesize that the lack of migrations is associated with the fact that deprecation messages are displayed only when a deprecated provider release is installed. When the provider release is deprecated while it is already installed, the client package developer does not become aware of such a deprecation. In fact, this issue has been a subject of discussion in at least two issue reports.[22,23] We argue that npm should provide an easy way for client package developers to check the adoption of a deprecated provider release. Preferably, the deprecation mechanism should proactively (and periodically) warn client packages when a deprecated release is adopted. Another hypothesis that can be stated to explain the lack of migrations from a deprecated release is that the set of functionalities of the provider that is used by the client package is not affected by the deprecation. Another plausible hypothesis is that client packages do not have a strict policy against the adoption of deprecated provider releases. A fruitful future research endeavour is to survey client package developers to understand the reasons for not migrating to an existing replacement release (particularly, when the replacement is a patch release).

Implication 4) ***The deprecation mechanism should provide built-in features for assessing the transitive adoption of deprecated provider releases.*** The current support by the standard npm tools for verifying the transitive adoption of a deprecated release is limited, as is the support for migrating away from a transitively adopted deprecated release. For example, when the deprecation message for a transitively adopted deprecated release is displayed, there is no information on whether the deprecation message refers to a direct or a transitive dependency. Moreover, there is no trivial solution to update a transitively adopted provider package. Yet, more than half (54%) of the client packages transitively adopt a deprecated provider release (Result 9). Hence, a significant proportion of the client packages would benefit from having better support for dealing with transitively adopted provider packages. In contrast to the current scenario, in which separated tools need to be used to detect the adoption and to update transitive deprecated releases (see Section 4.2.2), the own deprecation mechanism should implement such functionalities. In particular, the deprecation mechanism should 1) differentiate between deprecation messages from directly and transitively adopted deprecated providers, 2) help client packages to identify how deep in the dependency tree a certain transitively adopted deprecated release is, 3) assist client packages in the identification of which (and how many) directly adopted provider results in the transitive adoption of a deprecated release. In Result 10, we observe that the typical dependency depth of a transitively adopted deprecated release is relatively shallow, compared with the depth of non-deprecated releases. Also, in Result 11, we observe that the median number of direct providers that result in the transitive adoption of a deprecated release is one. Therefore, the set of suggested features can be implemented by the deprecation mechanism without overloading client packages with too much information regarding the transitive adoption of deprecated releases.

Implication 5) ***The deprecation mechanism should prevent the implicit adoption of deprecated releases when an older non-deprecated release is available.*** If a provider package does not have a replacement release (i.e., the latest release is deprecated) but has some older non-deprecated release, then the deprecation mechanism should interact with the provider version resolution mechanism to make client packages adopt the older non-deprecated release in detriment to the latest deprecated release. For example, suppose a provider package P that has version 1.2.3 deprecated and version 1.2.2 non-deprecated. A client package C that adopts P using a version range statement such as "`P: >1.0.0`" should load version 1.2.2 of P, instead of version 1.2.3.

5.2 Assessing the impact of deprecated releases

Although the migration away from a deprecated release depends on the client package's willingness, we found evidences that such a migration might not be trivial (e.g., by the lack of a replacement release or the need to perform changes to migrate). For this reason, client packages that are willing to migrate away from deprecated releases can benefit from

understanding whether a replacement release is available and how difficult such a migration typically is. In turn, client packages that still adopt a deprecated provider release should evaluate the impact and risks associated with such an adoption. In the following, we discuss the risks that client packages can face when adopting a deprecated release, as well as the challenges to migrate away from a deprecated release.

Implication 6) ***Client packages should be especially careful about the usage of deprecated releases of partially deprecated packages.*** The deprecation of a release can occur for different reasons. The most common reason for the deprecation of all releases of a package in npm is withdrawal (i.e., terminating the maintenance of a package), whereas the deprecation of one or more specific releases of a package usually occurs due to a defect (Result 3). Defective provider releases might be risky to client packages and, therefore, should be addressed with proper attention (e.g., by migrating away from the deprecated release). Another common reason for deprecation is when a release is superseded. This rationale for deprecation does not indicate any issue that needs to be urgently addressed, however provider packages can have some specific reason (e.g., the deprecated release use some obsolete feature) to communicate a deprecation to client packages instead of simply publishing a newer release. Therefore, client packages should update a superseded provider release whenever it is possible, especially when the newer provider release is backward compatible with the deprecated release. Incompatibility is also identified as a rationale for the deprecation of a package's release, however the deprecation of a release for this reason is significantly less common than for other reasons. Nevertheless, client packages are exposed to incompatibilities in very specific circumstances (i.e., when incompatible versions of two providers are used at once).

Implication 7) ***Client packages should be aware that providers misuse release deprecation.*** A total of 31% of the partially deprecated packages do not have a follow-up replacement release. However in Result 2, by manually analyzing the deprecation message of such packages, we observed that 65% of the deprecation messages refer to a withdrawal or a supersession of a whole package (not a specific release). Therefore, client packages might be inadvertently adopting a release of an unmaintained provider package. Nonetheless, we observed that 84% of the superseded packages were simply moved to another repository (e.g., codebase) or renamed (Result 3). Therefore, for fully deprecated packages that were superseded, client packages should expect to be able to find a respective replacement package.

Implication 8) ***Client packages should reason about the characteristics of the replacement release when migrating away from a deprecated release.*** 80% of the popular deprecated releases belong to fully deprecated packages that only provide a replacement package (in lieu of a replacement release) (Result 6). The migration to a replacement package requires the adoption of a different provider package, potentially requiring client packages to perform changes in order to cope with a new design implemented by this provider (e.g., new APIs). When a replacement release is available, in 15% of the cases client packages will need to integrate a major provider release (Result 4), which is assumed to be backward incompatible with the deprecated release, likely requiring client packages to perform changes to integrate the new major provider release. Also, we observed that 32% of the replacement releases are minor (17%) or major releases (15%). In some of these cases, a patch downgrade to an older non-deprecated release is a viable option (for instance, when the release 2.1.0 of the provider package is non-deprecated, the release 2.1.1, which is currently adopted by the client package is deprecated, and the release 2.2.0 is non-deprecated). In such a scenario, client packages should take several factors into consideration to decide whether a patch downgrade or a minor update is the appropriate migration choice. Some of these factors include the presence/absence of features, bugs and security vulnerabilities, as well as the estimated required effort to perform the migration.

Implication 9) ***Client packages should be attentive to the transitive adoption of deprecated provider releases.*** More than half (54%) of the client packages in the ecosystem transitively adopt at least one deprecated release. Interestingly, the majority of these client packages (54%) do not directly adopt deprecated releases (Result 9). Client packages that want to avoid the adoption of deprecated releases by all means should be attentive to their transitive providers. When a deprecated release is transitively adopted, client packages have two main options to cease the adoption:

1) Updating the direct provider that is responsible for the transitive adoption of a deprecated release. This option does not necessarily guarantee that the adoption of the deprecated release will cease, however it serves as a best-effort approach. We verified that the median number of direct providers that result in the transitive adoption of at least one deprecated release is 1 (Result 11). This number thus serves as an estimate of how many direct providers would need to be updated when this best-effort approach is chosen by the client package as an attempt to cease the transitive adoption of deprecated release.

2) Performing workarounds. The deeper a transitively adopted deprecated release is in the dependency tree of a client package, the less control the client package has over such an adoption. Indeed, we verified that the median of the deepest dependency depth of a deprecated provider release is 4 (Result 10), showing that, in general, client packages need to cope with the transitive adoption of deprecated provider releases that are far down the dependency tree. To manually update transitive adoptions, client packages often rely on workarounds, such as manually modifying build files that are automatically generated and reinstalling all provider packages.[24,25]

A feature that can help client package developers to assess the impact of a transitively adopted deprecated release is to explicitly display the dependency tree when these deprecated releases are installed.[26] The implementation of this feature can help client developers to understand the transitive adoption of deprecated releases (e.g., which direct provider causes the adoption of a deprecated release and how deep this deprecated release is in the dependency tree). Moreover, the deprecation mechanism should offer client

packages a built-in feature to update transitive dependencies.[27,28,29]

6 Related Work

In this section, we describe prior studies about deprecation in software ecosystems. We initially present related work that discusses how client packages use deprecated APIs and how such clients react to the deprecation of these APIs. Then, we discuss studies that report how often deprecation messages report a replacement API. Finally, we discuss studies that describe the rationale behind the deprecation of APIs. All the presented related work discusses the phenomenon of *API deprecation*, whereas our paper is the first to present a study about *release deprecation* in a software ecosystem. In an attempt to bridge the two related areas, we compare literature results regarding API deprecation with our results about release deprecation. Nonetheless, we remind the reader that such comparisons should be taken with a grain of salt. The reason is twofold: 1) API and release deprecation operate at different levels, the former at the function- or method-level and the latter at the package's release level, and 2) the mechanisms for API deprecation are provided by the programming language, whereas the mechanisms for release deprecation are provided by the package manager.

Usage of and migration away from deprecated releases. Henkel and Diwan [19] discuss that the usage of a certain provider's API by client packages can have an impact on the decision of deprecating this API. The authors argue that provider packages do not want to drive complex changes in the client packages and the decision about the deprecation of an API should consider this assumption. Robbes et al. [3] analyzed the deprecation of packages' API in the Pharo ecosystem (for the Smalltalk language) and found that a small proportion (14%) of the deprecated methods triggered a client package reaction (e.g., a method replacement). Sawant et al. [13] analyze the reaction of clients to the deprecation of Java APIs and found that a small proportion of the client packages migrate away from the deprecated methods. The authors found that the majority of the client packages do not use any deprecated provider API. In a follow-on study, Sawant et al. [4] derive the following patterns of reaction to Java API deprecation: deletion of call to deprecated API, replacement by third-party API, replacement by in-house API, and downgrade of API version. Although the authors also found that the majority of the client packages do not migrate away from the deprecated methods, when the migration takes place client packages usually replace the deprecated method with a call to another third-party API. Li et al. [5] report that 38% of a random sample of 10,000 Android apps are using a deprecated API. Our results show that 27% of all client packages in the ecosystem adopt a deprecated release.

Prior studies show that the rate at which client packages replace an adopted deprecated API is low. In contrast, we found that 85% of the replacement releases in npm are patch and minor releases, which are often implicitly adopted by client packages.

Replacement releases in deprecation messages. Zhou and Walker [1] found that 51% of the studied packages in the Maven ecosystem (for the Java language) have a deprecation message that reports a replacement API. Brito et al. [8] show that 59.5% of the API elements (types, fields, and methods) of 661 Java projects are deprecated with a message that reports a replacement API element. Ko et al. [20] reveal that 61% of 260 deprecation messages of eight Java packages have a replacement API. In turn, Li et al. [5] found that, among a set of 20 Android releases, the median proportion of deprecated APIs that do not report a replacement on the deprecation message is approximately 30%. In a study about API deprecation messages of the top 50 most popular client packages in npm, Nascimento et al. [21] points that 67% of the deprecation messages report a replacement API. Our results show that 51% of the deprecation messages in npm report a replacement release.

In general, the difference between the proportion of API and release deprecation messages that report a replacement is at most 16%.

Rationale behind deprecation. According to a survey by Sawant et al. [2], there are seven rationales for the usage of the deprecation mechanism by client developers in Java. Their study examined deprecation at the method level (i.e., API deprecation). With the exception of one out of the seven stated rationales (namely "old interface encourages bad practices"), all of them have commonalities with the rationales for the deprecation of a release in npm (see Table 5). In turn, Sawant et al. [7] manually investigate the deprecation messages of 374 Java APIs and propose 12 categories for the rationale of deprecation. Even focusing on Java API deprecation, many of the proposed categories agree with the rationales for the deprecation of npm packages and releases. Mirian et al. [22] studied the reasons for the deprecation of APIs provided by the Chrome web browser. The authors identified six different categories for the rationale behind the deprecation of an API. Four out of the six categories are related with the identified rationales for the deprecation of a release in npm. The "inconsistent implementation" and "security" categories by Mirian et al. [22] are related with defects, the "updated standard" category is related with supersede, and the "removed from standard" category is related with withdrawal. Raemaekers et al. [23] studied the Maven ecosystem and found that deprecation is rarely used to communicate that a given API has introduced backward incompatible changes (a.k.a. breaking changes). Decan et al. [14] suggest that package developers in npm should deprecate releases that can potentially suffer from vulnerabilities.

There are a number of commonalities between the rationale for deprecating an API and a release. All rationales for release deprecation can also be associated with API deprecation, although the opposite does not hold.

7 Threats to Validity

In this section, we discuss the threats to the validity of our study about release deprecation in npm. We discuss the

TABLE 5: Comparison between the identified rationales behind API and release deprecation.

Reference	Identified rationales for method deprecation	Identified rationales for release deprecation
Sawant et al. [2]	Feature is unnecessary; no longer provide a feature	Withdrawal
	New feature supersedes existing one	Supersede
	Functional issue; non-functional issue	Defect
	Mark as beta	Test
	Old interface encourages bad practices	—
Sawant et al. [7]	Redundant methods, Renaming of feature	Withdrawal
	Merged to existing method; new feature introduced; separation of concerns	Supersede
	Functional defects; security flaws	Defect
	Temporary feature; dissuade usage	Test
	No dependency support	Incompatibility
	Avoid bad coding practices; design pattern	—
Mirian et al. [22]	Removed from standard	Withdrawal
	Updated standard	Supersede
	Security; inconsistent implementation	Defect
	Clean experience; never standardized	—

threats related to construct validity, internal validity, and external validity.

Construct Validity. The npm registry does not record the date of a release deprecation. Therefore, when our data was collected from the npm registry, we only knew that a given release was deprecated *some* time before our data collection. The lack of knowledge about the deprecation date makes it impossible for us to perform a reliable historical analysis about the adoption of deprecated releases. To mitigate this threat, we do not perform a historical analysis of deprecated release adoption (e.g., an analysis on how client packages migrate away from deprecated releases). Rather, we consider how deprecated releases were adopted at the latest client release (i.e., at the snapshot of our data collection). Even though, we can still rely on cases for which the provider release was deprecated at the time of our data collection, but was not deprecated when the client package started adopting this release.

When analyzing the adoption of the top 40 popular deprecated releases using survival analysis (Section 4.2.1), we manually search for any documentation that allows us to estimate the deprecation date. For 30 out of the top 40 popular deprecated releases, the deprecation date is stated either in the deprecation message or in some associated documentation. Nonetheless, for the other 10 releases, we assumed the deprecation date to be that of the date of the latest package release. For these 10 deprecated releases, we might have overestimated the deprecation time. In Appendix C, we show more detailed information about how we estimate deprecation dates.

The analyses of how client packages adopt deprecated provider releases (RQ2) are performed at the package dependency level. As a consequence, they abstract how client packages are used by providers. The set of features of a provider package that are used by different client packages range from an isolated functionality to a full set of dependent features. Although the circumstances under which a provider package is used by its clients are abstracted, they can influence how client packages adopt and migrate away from a deprecated provider release. For example, we observed that many deprecated releases are defective. In those cases, a client package might be compelled to migrate away from a deprecated release.

External Validity. Our study is limited to data from the npm ecosystem, therefore our results might not be generalized to other ecosystems. Our study is the first to analyze the deprecation phenomenon at the release- and package-levels in a software ecosystem and to provide knowledge about such a phenomenon. Nevertheless, we identified that the rationales for the deprecation of releases have commonalities with the deprecation of APIs.

Our study does not have the objective of elucidating general theories about the deprecation phenomenon and further research is needed to provide more comparisons and an eventual generalization. Furthermore, release-level deprecation mechanisms are provided by other package managers ecosystems (e.g., Packagist for the PHP language or Nuget for .NET) and our approach can be replicated in those other ecosystems. Although the deprecation mechanisms of different ecosystems have different characteristics, the learned lessons discussed in Section 5 can be useful to reason about release-level deprecation in other ecosystems.[30,31,32]

8 Conclusion

Deprecation is a mechanism employed by software developers to discourage the use of a particular piece of code. Prior empirical studies focused on the deprecation of API elements (e.g., methods and functions) and investigated several topics, such as how frequently deprecated APIs are adopted by clients [3, 13], the provisions of replacement APIs [1, 5, 21], and the rationales behind the deprecation of APIs [2, 7, 22]. In this paper, we studied deprecation from a software ecosystem perspective, which entails the deprecation of releases. More specifically, we conducted a case study of release deprecation in npm.

To understand the relevance of the deprecation mechanism in npm, we analyzed how often releases are deprecated by provider package developers and the impact of the deprecated releases over the client packages. We found that the rate at which the deprecation mechanism is used by provider packages is small, with approximately 3% of the releases being deprecated. However, 27% of the client packages directly adopt at least one deprecated release and 54% of all client packages transitively adopt at least one deprecated release in their latest release. We assessed the risks brought by the usage of deprecated releases by studying the rationales behind the deprecation. We verified that the deprecation of all releases of a package is usually associated with withdrawals (i.e., terminating the package maintenance) and supersession (i.e., the substitution of a release by another). Also, we verified that the deprecation of one specific package release is usually associated with the presence of defects in that release.

Based on our results, we concluded that, despite the importance of the deprecation mechanism to the npm ecosystem, such a mechanism is still fairly rudimentary. For instance, to date, there is no simple approach that enables client packages to check whether any installed provider

release is deprecated. We proposed a series of improvements to the npm deprecation mechanism. We also concluded that is not straightforward for client package to assess the impact (e.g., risk) of using a deprecated release. For instance, the rationale behind a deprecation is not always provided and client packages can unwittingly adopt deprecated releases by means of transitive dependencies. We proposed ways in which client packages can better assess the impact of using deprecated releases.

REFERENCES

[1] J. Zhou and R. J. Walker, "API deprecation: A retrospective analysis and detection method for code examples on the web," in *Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE'16)*, 2016, pp. 266–277.

[2] A. A. Sawant, M. Aniche, A. van Deursen, and A. Bacchelli, "Understanding developers' needs on deprecation as a language feature," in *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, 2018, pp. 561–571.

[3] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation? The case of a Smalltalk ecosystem," in *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE'12)*, 2012, pp. 1–11.

[4] A. A. Sawant, R. Robbes, and A. Bacchelli, "To react, or not to react: Patterns of reaction to API deprecation," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3824–3870, 2019.

[5] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Characterising deprecated android APIs," in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR'18)*, 2018, pp. 254–264.

[6] 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, 2018.

[7] A. A. Sawant, G. Huang, G. Vilen, S. Stojkovski, and A. Bacchelli, "Why are features deprecated? An investigation into the motivation behind deprecation," in *Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME'18)*, 2018, pp. 13–24.

[8] G. Brito, A. Hora, M. T. Valente, and R. Robbes, "Do developers deprecate APIs with replacement messages? A large-scale analysis on java systems," in *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, 2016, pp. 360–369.

[9] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the JavaScript package ecosystem," in *Proceedings of the 13th International Workshop on Mining Software Repositories (MSR'16)*, 2016, pp. 351–361.

[10] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *Proceedings of the 14th International Working Conference on Mining Software Repositories (MSR'17)*, 2017, pp. 102–112.

[11] A. Decan and T. Mens, "What do package dependencies tell us about semantic versioning?" *IEEE Transactions on Software Engineering*, pp. 1–15, 2019.

[12] F. R. Cogo, G. A. Oliva, and A. E. Hassan, "An empirical study of dependency downgrades in the npm ecosystem," *IEEE Transactions on Software Engineering*, 2019, preprint.

[13] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of clients of 4+1 popular java APIs and the JDK," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2158–2197, 2018.

[14] A. Decan, T. Mens, and E. Constantinou, "On the evolution of technical lag in the npm package dependency network," in *Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME'18)*, 2018, pp. 404–414.

[15] A. Zerouali, T. Mens, J. Gonzalez-Barahona, A. Decan, E. Constantinou, and G. Robles, "A formal framework for measuring technical lag in component repositories and its application to npm," *Journal of Software: Evolution and Process*, 2019.

[16] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: A critical review and guidelines," in *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*, 2016, pp. 120–131.

[17] I. Samoladas, L. Angelis, and I. Stamelos, "Survival analysis on the duration of open source projects," *Information and Software Technology*, vol. 52, no. 9, pp. 902–922, 2010.

[18] J. Romano, J. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys?" in *Annual Meeting of the Florida Association of Institutional Research*, 2006.

[19] J. Henkel and A. Diwan, "Catchup! Capturing and replaying refactorings to support API evolution," in *Proceedings of the 27th International Conference on Software Engineering (ICSE'05).*, 2005, pp. 274–283.

[20] D. Ko, K. Ma, S. Park, S. Kim, D. Kim, and Y. L. Traon, "API document quality for resolving deprecated APIs," in *Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC'14)*, vol. 2, 2014, pp. 27–30.

[21] R. Nascimento, A. Brito, A. Hora, and E. Figueiredo, "Javascript API deprecation in the wild: A first assessment," in *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'2020)*, 2020.

[22] A. Mirian, N. Bhagat, C. Sadowski, A. Porter Felt, S. Savage, and G. M. Voelker, "Web feature deprecation: A case study for Chrome," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'19)*, 2019, pp. 302–311.

[23] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning versus breaking changes: A study of the Maven repository," in *Proceedings of the 14th International Working Conference on Source Code Analysis and Manipulation (SCAM'14)*, 2014, pp. 215–224.

[24] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR'18)*, 2018, p. 181–191.

Filipe R. Cogo is a Software Engineering Researcher at Huawei, Canada. His research focuses on empirical software engineering and mining software repositories. He received his BSc and MSc in Computer Science from Universidade Estadual de Maringá (UEM), Brazil, and his PhD from Queen's University, Canada.

Gustavo A. Oliva is Research Fellow at Queen's University in Canada under the supervision of professor Dr. Ahmed Hassan. His research focuses on understanding the rationale of software changes and their impact on Software Maintenance and Evolution. As such, his studies typically involve Mining Software Repositories (MSR) and applying static code analysis, evolutionary code analysis, and statistical learning techniques. Gustavo received his MSc and PhD from the University of São Paulo (USP) in Brazil under the supervision of professor Dr. Marco Aurélio Gerosa.

Ahmed E. Hassan is an IEEE Fellow, an ACM SIGSOFT Influential Educator, an NSERC Steacie Fellow, the Canada Research Chair (CRC) in Software Analytics, and the NSERC/BlackBerry Software Engineering Chair at the School of Computing at Queen's University, Canada. His research interests include mining software repositories, empirical software engineering, load testing, and log mining. He received a PhD in Computer Science from the University of Waterloo. He spearheaded the creation of the Mining Software Repositories (MSR) conference and its research community. He also serves/d on the editorial boards of IEEE Transactions on Software Engineering, Springer Journal of Empirical Software Engineering, and PeerJ Computer Science. Contact ahmed@cs.queensu.ca. More information at: http://sail.cs.queensu.ca.

APPENDIX A
DATA COLLECTION

In this section, we describe our data collection method. Figure 7 depicts an overview of the method. Three main steps are performed: collect package metadata (Section A.1), analyze package releases (Section A.2), and analyze package dependencies (Section A.3). In the following, we discuss each of these steps.

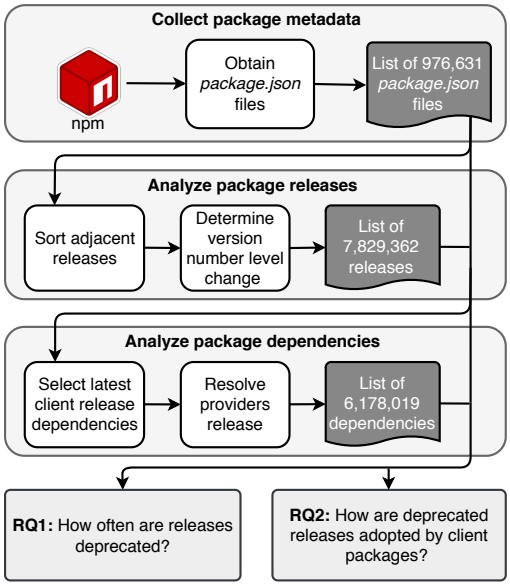

Fig. 7: An overview of our data collection method.

A.1 Collect package metadata

In this step, we collect the metadata of npm packages from the `package.json` files (see Section 2.1 for an explanation of the `package.json file`).

Obtain `package.json` files: We obtained all `package.json` files that were stored in the npm registry as of May, 2019. The `package.json` file of a given package P contains release-related metadata for all releases of P. The release-related metadata include the release version number, the timestamp at which the release was published, the dependencies that are set in the release (i.e., the provider package name and the respective versioning statement), and the deprecation message in case the release is deprecated. The output of this procedure is a list of 976,631 `package.json` files.

When the release of a package P is deprecated, the `package.json` file of P contains a `deprecated` field that stores the deprecation message that is associated to that release (in contrast, non-deprecated releases do not have such a field). We observed that 8% of all deprecated releases have the "False" string as a deprecation message. After performing a manual analysis (see Section 5 for further details about this analysis), we decided to classify such releases as non-deprecated. Also, 1% of the deprecated releases have an empty deprecation message. According to the npm documentation on deprecation,[33] a developer can remove the deprecation of a release by setting the deprecation message to an empty string. Therefore, releases whose deprecation message is an empty string were classified as non-deprecated.

A.2 Analyze package releases

In this step, we sort the adjacent package releases and determine how the version numbers change between two adjacent releases (e.g., whether the latter release is a major, minor, or patch release). See Section 2.1 for a definition of the release version levels.

Sort adjacent releases: We sort the releases of a package according to a *branch-based ordering* (in contrast with a chronological- or numerical-based ordering). The motivation for using a branch-based ordering is the adoption of parallel development branches by some packages, for which chronologically interleaved releases are published (e.g., release 1.2.3, a patch for release 1.2.2, being published after the existence of release 2.0.0). With a chronological-based ordering, release 2.0.0 would be considered the predecessor of release 1.2.3, which is incongruous from the numerical point of view. Similarly, with a numerical-based ordering, release 1.2.3 would be considered the predecessor of release 2.0.0, which is incongruous from the chronological point of view (after all, release 1.2.3 was published *after* release 2.0.0). A branch-based ordering schema allows a release to be considered the predecessor of more than one release. In our example, release 1.2.2 would be considered the predecessor of both releases 1.2.3 and 2.0.0. More details about the branch-based ordering algorithm can be found in our prior work [12].

Determine version level change: After sorting the releases of a package according to our branch-based ordering, we analyze how the version level changes between two adjacent releases. For each pair of adjacent releases, we classify the version level change into a major, minor, or patch release. For example, if release 1.2.2 is considered the predecessor of release 2.0.0, then release 2.0.0 is classified as a major release. Similarly, if release 1.2.2 is considered the predecessor of release 1.2.3, then release 1.2.3 is classified as a patch release. The same logic applies to minor releases. The output of this procedure is a list of 7,829,364 release changes.

A.3 Analyze package dependencies

In this step, we select a subset of all dependencies, namely the dependencies that are set in the latest client release. We also resolve the release of the providers that are used by each client package (see Section 2.1).

Select latest client release dependencies: The date at which a provider package release was deprecated is not available in the npm registry. This limitation in the npm data prevents an analysis about the adoption history of deprecated releases by client packages. For example, Figure 8 depicts a scenario in which the provider release is deprecated after it is adopted by a client package. In such case, we can only correctly assume that the client package is using a deprecated provider release at our data collection date (since the deprecation date is unknown). Therefore, we select only the latest client release (i.e., the current client release at our data collection date) to analyze how deprecated provider releases are adopted by client packages.

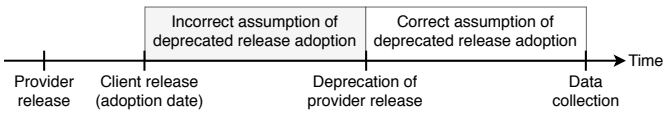

Fig. 8: Illustration of a scenario in which the provider release is deprecated after it is adopted by the client package.

Resolve providers release: We resolve the release of the providers that are used in the latest release of each client package. For the latest release of each client package, we parse the used versioning statements according to the grammar provided by npm. The resolved release is the latest provider release (at the time of the client release) that is satisfied by the versioning statement. For instance, suppose that a client package C, in its 2.0.0 release, sets a dependency using the versioning statement "`P: >` 1.2.3". In this case, the resolved provider release will be the latest release of P (the provider package) that is greater or equal 1.2.3 and that is published before the client package release 2.0.0. Invalid versioning statements or versioning statements that do not satisfy any existing provider release are ignored. The output of this procedure is a list of 6,178,019 dependencies that are set in the latest client release with the respective resolved release of each provider.

Data collection summary

- Data collection day: May 5th, 2019.
- Data source: the npm registry.
- Collected data: `package.json` files, from which 7,829,362 releases and 6,178,019 dependencies are extracted and analyzed.

APPENDIX B
CONTENT OF DEPRECATION MESSAGES

Documentation is an important aspect of deprecation. With proper deprecation messages, client packages can understand the reason for the deprecation and evaluate the risk of adopting a given deprecated release. Furthermore, it is important that deprecation messages report the replacement packages and releases, therefore client packages can perform an easier migration.

We manually analyzed a statistically representative sample (95% confidence level, with $\pm$5% confidence interval) of the deprecation messages used by npm packages. We sampled a total of 381 out of the 44,112 unique deprecation messages used by different packages. The performed analysis resulted in a categorization of the rationale behind the deprecation of a release, as well as an estimate of the proportion of deprecation messages that report a replacement package or release. We performed an open coding [16] to categorize the rationale behind release-level deprecation. We also classify the deprecation messages between messages that refer to a deprecated package (e.g., "*This package is no longer supported*") from messages that refer to a specific release of the deprecated package (e.g., "*This version has a bug. Use version 1.0.1 instead*"). We perform such a classification to calculate how often each identified rationale is associated with the deprecation of all releases of a package (full deprecation), in contrast to the deprecation of a specific release of a package (partial deprecation). Finally, we calculate how often a deprecation message reports a replacement package or a replacement release.

Almost two-thirds (64%) of the deprecation messages report the rationale behind the deprecation of a package or release. This result suggests that, when installing a deprecated package or release, client packages in many cases will be able to evaluate the risk of adopting a deprecated release. From the deprecation messages that report the rationale behind the deprecation, 86% are a customized message (i.e., they are different from the standard message that is provided by npm).[34] This latter observation suggests that the typical rationale for the deprecation of a package or release goes far beyond the rationale stated in the standard message (which is "*Package no longer supported. Contact support@npmjs.com for more info.*" at the time this paper was written).

We also note that a total of 8% of all deprecation messages are the string "False". To understand the usage of the "False" string as a deprecation message, we manually analyzed the revisions (i.e., the history of versions) of the `package.json` files from packages with such deprecation message. We identified that some packages have the "False" deprecation messages since its creation, having never been in fact deprecated. We hypothesize that some developers might not understand the meaning of the `deprecated` field in the `package.json` file. In such cases, the developer manually edit the `package.json` file and set the `deprecated` field as "False", with the intent of communicating that the release is not deprecated. This result suggests that the deprecation mechanism is not intuitive, leading some package developers to misunderstand how the mechanism works.

We identified five rationales for the deprecation of a package or release: withdrawal (43%), supersession (37%), defect (13%), test (6%), and incompatibility (1%). Each category is described below:

• Rationale 1) *Withdrawal:* The deprecation message indicates that the package or release was deprecated because its development was terminated. However, the package is left on the registry, such that the actual client packages are not affected. An analysis of such deprecation messages shows that withdrawals occur for different reasons. The following deprecation messages indicate that the withdrawal might occur because the package/release is no longer maintained in npm:

> "*This module is no longer maintained.*" [deprecation message of package `kilt`],
>
> "*This project is no longer a npm-package. Checkt [sic] our github at https://github.com/Server-Eye/bucket-collector*" [deprecation message of package `bucket-collector`],
>
> "*Package unsupported. Please use the rws-compile-sass package instead.*" [deprecation message of package `custom-rws-compile-sass`]

The following deprecation messages indicate that the withdrawal might occur because the package/release is a de-

pendency that is no longer required (e.g., its features were incorporated into another package/release):

> "*Deprecated as it's now the default reporter in ESLint.*" [deprecation message of package `eslint-stylish`],

> "*This is a stub types definition. p-limit provides its own type definitions, so you do not need this installed.*" [deprecation message of package `@types/p-limit`],

> "*No longer needed for grunt-vows-runner. Use grunt-vows-runner instead.*" [deprecation message of package `vows-reporters`]

Also, as shown in the following deprecation messages, the withdrawal might occur because the package/release became obsolete.

> "*Very old and unmaintained module. Don't recommend using this anymore.*" [deprecation message of package `grunt-copy-mate`],

> "*Since Catberry@4 this package is not supported due to architecture changes.*" [deprecation message of package `catberry-lazy-loader`],

> "*Do not use this package to update globally installed CLIs anymore.*" [deprecation message of package `npm-update-module`]

• Rationale 2) *Supersession:* The deprecation message indicates that the deprecated package or release was replaced by another one. The following deprecation messages indicate that the deprecated release was replaced by a newer, improved release:

> "*Version 1.x branch of Iridium has been superseded by v2.x.*" [deprecation message of package `iridium`],

> "*Still using old declarative binding syntax? Please, update to its latest version: 0.5.102.*" [deprecation message of package `pacem`],

> "*API changed: then() to on(), catch() to onerror(), finally() to oncancel().*" [deprecation message of package `rnr`]

In addition, the following deprecation messages indicate that the deprecated package features were incorporated into another package:

> "*This has been merged back into express-batch, which you should now use.*" [deprecation message of package `express-batch-deep`],

> "*This module is now a part of babel-preset-steelbrain@2.x.x.*" [deprecation message of package `babel-preset-steelbrain-async`],

> "*@appnest/focus-trap has moved to @a11y/focus-trap. Please uninstall this package and install @a11y/focus-trap instead.*" [deprecation message of package `@appnest/focus-trap`],

> "*All Pivotal UI components & styles have been moved to the 'pivotal-ui' package. Install that package for all future updates.*" [deprecation message of package `pui-react-checkbox`],

> "*This library has been renamed to flum. Please install flum to get the lastest [sic] version.*" [deprecation message of package `react-basic-forms`]

• Rationale 3) *Defect:* The deprecation message indicates that the package or release was deprecated due to the presence of a known defect. An analysis of such deprecation messages shows that the source of defect can be either in the source code or in the deployed artifact (built package). The following deprecation messages indicate that the package or release was deprecated due to a defect in the source code:

> "*This patch version has breaking changes. Please use 0.23.0 instead.*" [deprecation message of package `@devexperts/react-kit`],

> "*Buggy implementation of class mixins.*" [deprecation message of package `@zenparsing/skert`],

> "*windows posix socket bug*" [deprecation message of package `node-ipc`],

> "*Sending Blob body using XMLHttpRequest polyfill may cause incorrect result with this version, please use 0.9.1 instead.*" [deprecation message of package `react-native-fetch-blob`]

Also, the following deprecation messages indicate that the package or release was deprecated due to a defect in the built package:

> "*wrong build.*" [deprecation message of package `dc-webapi`],

> "*critical dir missing.*" [deprecation message of package `angular-html5`],

> "*incorrect main field in package.json, fixed in 1.0.1*" [deprecation message of package `eslint-config-r29`],

> "*error in version number.*" [deprecation message of package `react-native-aerogear-ups`],

> "*Main script path incorrect. Only ES6 module is working.*" [deprecation message of package `defy`]

• Rationale 4) *Test:* The deprecation message indicates that the deprecation occurs because the package or release was published for test purposes or by accident:

> "*This is package is just for testing. don't install it.*" [deprecation message of package `reactmanishbot`],

> "*Not a usable package.*" [deprecation message of package `glarce-combo`],

> "*not meant to be published sorry.*" [deprecation message of package `chat-engine`]

The following deprecation messages indicate that prereleases, which are used for in-field testing, are also deprecated:

> "*Development versions have been deprecated.*" [deprecation message of package `@servicensw/page`],

> "*outdated prerelease.*" [deprecation message of package `@dandi/common`],

> "*Use version 1.0.0, this was a prerelease and is no longer maintained.*" [deprecation message of package `vue-cli-plugin-git-describe`],

> "*still in beta.*" [deprecation message of package `chat-engine`]

• Rationale 5) *Incompatibility:* The deprecation message indicates that the package or release was deprecated due to incompatibility. The following deprecation messages indi-

cate incompatibility between client and provider packages (dependency incompatibility):

> "*Old versions not compatible with sqb >0.7.0.*" [deprecation message of package `sqb-serializer-oracle`]

Also, the following deprecation messages indicate incompatibility between the package and some specific browser version:

> "*Incompatible with modern browsers.*" [deprecation message of package `yahoo-shapes`]

Limitations. To understand the rationales behind the deprecation of a release, we manually analyzed a representative sample of unique deprecation messages. We choose to sample *unique deprecation messages* instead of *unique deprecated releases* because each package has a different number of releases and the same package can deprecate a range of releases (perhaps all releases) with the same message. Therefore, by sampling unique deprecation messages instead of unique deprecated releases, we are avoiding a selection bias towards packages with a large number of deprecated releases.

Our sample size ensures a confidence level of 95% and a confidence interval of $\pm 5\%$. Therefore, the reported prevalence for each rationale is bound to the properties of such sample. Also, we might have not sampled messages that refer to rationales that rarely appear. For example, prior studies indicate that vulnerabilities can potentially drive a deprecation [24], however our sample did not include any deprecation messages that explicitly refers to a vulnerability (although vulnerabilities would be part of the *defect* category).

APPENDIX C
TOP 40 POPULAR DEPRECATED RELEASES

In this appendix section, we provide information regarding the popular (top-40 frequently used) deprecated releases (see Section 4.2 to more details about the popular deprecated releases). Table 6 shows the popular deprecated releases in npm at the time of our data collection. The column "Replacement package or release" shows the name of the replacement package, when the deprecation message provides a replacement package (e.g., babel-preset-env), or the replacement release, when the deprecation message provides a replacement release (e.g., gulp@>= 4).

TABLE 6: Popular deprecated releases and their replacement package or release.

Package name	Deprecated release	Replacement package or release	Deprecation date estimate	Evidence for date estimate
babel-preset-es2015	6.24.1, 6.18.0, 6.9.0, 6.6.0, 6.22.0, 6.3.13, 6.14.0, 6.24.0, 6.16.0, 6.13.2, 6.5.0	babel-preset-env	December, 2016	Documentation reports[35] that deprecation of babel-preset-es2015 occurred after release 1.0.0 of babel-preset-env (December 9, 2016).
istanbul	0.4.5, 0.3.22, 0.4.2, 0.4.3, 0.4.4	nyc	May, 2015	Istanbul and nyc packages were merged when nyc released version 2.0.0.[36]
gulp-util	3.0.8, 3.0.7, 3.0.6	vinyl, replace-ext, ansi-colors, date-format, fancy-log, lodash.template, minimist, beeper, through2, multipipe, list-stream, plugin-error	December, 2017	Deprecation message reports documentation with deprecation date information.[37]
babel	6.23.0, 6.5.2	babel-cli	February, 2017	Deprecation message reports that deprecation occurs on release "6.x". We assume the date of the latest release on 6.x branch (6.23.0).
react-dom	16.2.0	react-dom@>= 16.2.1	August, 2018	Vulnerability that drove deprecation was reported on August 01, 2018.[38]
core-js	2.4.1, 2.5.1	core-js@latest or core-js@>= 3	February, 2019	Deprecation message reports that deprecation occurs at release 2.6.5.
coffee-script	1.10.0, 1.6.3, 1.7.1, 1.8.0	coffeescript	February, 2017	Latest package release and release 1.0.0 of replacement package.
react-dom	16.4.1	react-dom@>= 16.4.2	August, 2018	Vulnerability that drove deprecation was reported on August 01, 2018.[39]
core-js	2.5.7	core-js@latest or core-js@>= 3	February, 2019	Deprecation message reports that deprecation occurs at release 2.6.5
jade	1.11.0	pug	April, 2016	jade becomes pug on version 2.0.0, released on April 1, 2016.[40,41]
react-dom	16.3.2	react-dom@16.3.3	August, 2018	Vulnerability that drove deprecation was reported on August 01, 2018.[42]
validate-commit-msg	2.14.0	commitlint	October, 2017	Latest package release and release 4.2.0 (first release) of replacement package.
babel-preset-es2017	6.24.1	babel-preset-env	September, 2017	Latest package release.
node-uuid	1.4.7	uuid	March, 2017	Latest package release.
core-js	2.5.3	core-js@latest or core-js@>= 3	February, 2019	Deprecation message reports that deprecation occurs at release 2.6.5.
rollup-watch	4.3.1	rollup	August, 2017	Documentation reports that rollup-watch package was deprecated at release 0.46.0 of rollup package (August 11, 2017).[43]
isparta-loader	2.0.0	istanbul-instrumenter-loader	November, 2011	Latest package release.
gulp-minify-css	1.2.4	gulp-clean-css	December, 2015	Package has deprecation commit.[44]
react-dom	16.0.0	react-dom@>= 16.0.1	August, 2018	Vulnerability that drove deprecation was reported on August 01, 2018.[45]

APPENDIX D
NOTES

[1]https://insights.stackoverflow.com/survey/2019#technology
[2]https://bit.ly/2wKO3se. For the final version of the paper, the contents will be made available on GitHub.
[3]https://semver.org
[4]https://docs.npmjs.com/misc/semver#advanced-range-syntax
[5]https://docs.npmjs.com/misc/semver#range-grammar
[6]https://docs.npmjs.com/cli/deprecate
[7]https://docs.npmjs.com/cli/install
[8]https://stackoverflow.com/questions/36329944/how-to-determine-path-to-deep-outdated-deprecated-packages-npm
[9]https://stackoverflow.com/questions/35236735/npm-warn-message-about-deprecated-package
[10]https://docs.npmjs.com/cli/outdated.html
[11]https://github.com/nodejs/package-maintenance/issues/93
[12]https://stackoverflow.com/questions/60735307/how-to-resole-this-npm-installation-problem
[13]https://stackoverflow.com/questions/34840153/npm-deprecated-warnings-do-i-need-to-update-something
[14]https://stackoverflow.com/questions/35236735/npm-warn-message-about-deprecated-package
[15]https://stackoverflow.com/questions/61174805/problems-trying-to-install-using-npm-core-js3-is-no-longer-maintained
[16]https://stackoverflow.com/questions/38889519/how-to-deal-with-deprecation-warnings-from-npm
[17]https://github.com/npm/rfcs/blob/latest/accepted/0009-package-overrides.md
[18]http://www.modulecounts.com
[19]https://github.com/rust-lang/crates.io/issues/549
[20]https://github.com/rust-lang/crates.io/issues/542
[21]https://github.com/rust-lang/crates.io/issues/704
[22]https://github.com/npm/npm/issues/15536
[23]https://github.com/npm/npm/issues/18023
[24]https://stackoverflow.com/questions/56634474/npm-how-to-update-upgrade-transitive-dependencies
[25]https://stackoverflow.com/questions/15806152/how-do-i-override-nested-npm-dependency-versions
[26]https://github.com/npm/rfcs/issues/155
[27]https://github.com/npm/rfcs/blob/latest/accepted/0019-remove-update-depth-option.md
[28]https://github.com/npm/rfcs/blob/latest/accepted/0027-remove-depth-outdated.md
[29]https://github.com/npm/rfcs/blob/latest/accepted/0006-shallow-updates.md
[30]https://devblogs.microsoft.com/nuget/deprecating-packages-on-nuget-org/
[31]https://www.tomasvotruba.com/blog/2017/07/03/how-to-deprecate-php-package-without-leaving-anyone-behind/
[32]https://api.rubyonrails.org/classes/ActiveSupport/Deprecation.html
[33]https://docs.npmjs.com/deprecating-and-undeprecating-packages-or-package-versions
[34]https://docs.npmjs.com/deprecating-and-undeprecating-packages-or-package-versions
[35]https://github.com/babel/babel-preset-env/pull/65
[36]https://github.com/istanbuljs/nyc/issues/524#issuecomment-280979372
[37]https://medium.com/gulpjs/gulp-util-ca3b1f9f9ac5
[38]https://reactjs.org/blog/2018/08/01/react-v-16-4-2.html
[39]https://reactjs.org/blog/2018/08/01/react-v-16-4-2.html
[40]https://www.npmjs.com/package/pug
[41]https://github.com/pugjs/pug/commit/ab26404b880a1db0b44269354d11d9c55ab4862f
[42]https://reactjs.org/blog/2018/08/01/react-v-16-4-2.html
[43]https://github.com/rollup/rollup-watch
[44]https://github.com/scniro/gulp-clean-css/commit/438a4faf27f134c30e3e94024a83951c21fdc5cc
[45]https://reactjs.org/blog/2018/08/01/react-v-16-4-2.html