An Empirical Study of Dependency Downgrades in the npm Ecosystem

Filipe R. Cogo, *Member, IEEE* Gustavo A. Oliva, *Member, IEEE,* and Ahmed E. Hassan, *Fellow, IEEE*

Abstract—In a software ecosystem, a dependency relationship enables a *client* package to reuse a certain version of a *provider* package. Packages in a software ecosystem often release versions containing bug fixes, new functionalities, and security enhancements. Hence, updating the provider version is an important maintenance task for client packages. Despite the number of investigations about dependency updates, there is a lack of studies about dependency downgrades in software ecosystems. A downgrade indicates that the adopted version of a provider package is not suitable to the client package at a certain moment. In this paper, we investigate downgrades in the npm ecosystem. We address three research questions. In our first RQ, we provide a list of the reasons behind the occurrence of downgrades. Our manual analysis of the artifacts (e.g., release notes and commit messages) of a package code repository identified two categories of downgrades according to their rationale: reactive and preventive. The reasons behind reactive downgrades are defects in a specific version of a provider, unexpected feature changes in a provider, and incompatibilities. In turn, preventive downgrades are an attempt to avoid issues in future releases. In our second RQ, we investigate how the versioning of dependencies is modified when a downgrade occurs. We observed that 49% of the downgrades are performed by replacing a range of acceptable versions of a provider by a specific old version. This observation suggests that client packages have the tendency to become more conservative regarding the update of their providers after a downgrade. Also, 48% of the downgrades reduce the provider version by a minor level (e.g., from $2.1.0$ to $2.0.0$). This observation indicates that client packages in npm should be cautious when updating minor releases of the provider (e.g., by prioritizing tests). Finally, in our third RQ we observed that 50% of the downgrades are performed at a rate that is 2.6 times as slow as the median time-between-releases of their associated client packages. We also observed that downgrades that follow an explicit update of a provider package occur faster than downgrades that follow an implicit update. Explicit updates occur when the provider is updated by means of an explicit change to the versioning specification (i.e., the string used by client packages to define the provider version that they are willing to adopt). We conjecture that, due to the controlled nature of explicit updates, it is easier for client packages to identify the provider that is associated with the problem that motivated the downgrade.

Index Terms—downgrades, dependency management, npm, software ecosystems

1 Introduction

Prior research shows that code reuse is related to the improvement of developers productivity, software quality, and time-to-market of software products [1, 2]. In the last decade, software ecosystems arose as an important mechanism to promote and support code reuse. In this paper, we focus on ecosystems that are set around packaging platforms for a programming language [3, 4]. Such platforms are built upon the notion of *dependencies* between packages. A dependency relationship enables a *client* package to reuse a certain version of a *provider* package.

In several ecosystems, client packages can specify a dependency as either a dependency to a *specific version* or a *range of versions* of a provider. If a range of versions is specified, the provider is *implicitly updated* whenever a new version satisfying this range is released. If a specific version is used, an update can only happen by switching it to a newer specific version or by using an appropriate version range. In addition, packages in software ecosystems typically adopt a version numbering scheme to communicate changes that are introduced in new releases. A popular scheme is the Semantic Version specification, in which a version number comprises three digits separated by a dot (e.g., $1.0.0$). The first digit represents the major level of the version number, commonly incremented whenever a backward-incompatible API change is introduced in a new release. The second one represents the minor level, commonly incremented whenever a new backward-compatible feature is introduced, and the third one represents the patch level, commonly incremented whenever a bug fix is introduced.

The benefits and drawbacks of updating providers in software ecosystems have been extensively studied [5, 6, 7, 8, 9, 10, 11]. On the one hand, updating providers enables a client package to benefit from bug fixes, new functionalities, security enhancements, and novel APIs. On the other hand, updating providers also makes a client package more susceptible to potential problems in the new version of the provider. In the latter case, client packages might end up *downgrading* a provider, i.e., reverting it to an older version.

In this paper, we study *why* and *how* downgrades occur. Our motivation is based on the following observations:

• *Downgrades naturally indicate that one or more provider versions caused problems to the client package.* Despite being a natural indication of issues, downgrades can be a simple and rapid workaround for specific issues that arise when

• *Filipe R. Cogo, 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
• *Filipe R. Cogo is with the Universidade Tecnológica Federal do Paraná (UTFPR), Câmpus Campo Mourão, Departamento de Computação, Brazil.*

a provider package is updated. While prior research indicates that backward-incompatible changes (a.k.a., breaking changes) [12], bugs [13, 14], and vulnerabilities in the provider package [15] motivate downgrades, the actual reasons behind downgrades have not been thoroughly investigated. Our objective in this paper is to study the reasons that motivate downgrades, such that practitioners can be aware of the typical cases that lead to this type of workaround.

• *Many software ecosystems platforms allow client packages to accept implicit updates of provider packages.* Such automatic updates can hinder the identification of the provider package behind a certain issue. As a consequence, the downgrade of such a provider might be delayed, or even the downgrade of unrelated providers might occur. Moreover we hypothesize that such automated upgrading may lead to problematic updates which in turn might force client packages to abandon automated updating altogether.

• *Downgrades indirectly impact packages in a software ecosystem environment.* A package that downgrades one of its providers might affect some of its clients that indirectly depend on features of the downgraded provider. Package releases with downgrades and that are used by many client packages have a particularly large potential impact.

• *Downgrades increase the technical lag of client packages.* The technical lag represents the extent to which client packages are missing the latest features and bug fixes from a provider. Measuring the technical lag that is introduced when a downgrade is performed should help practitioners understand the side-effects of this workaround.

All of the aforementioned issues raise important questions about how to reduce the potentially harmful side-effects of a downgrade. Elucidating detailed reasons behind downgrades, as well as how client packages perform these downgrades, would provide a more in-depth understanding of downgrades and their consequences, ultimately fostering further research and tool development to support package developers.

To conduct our study, we collected data from npm[1], the largest ecosystem supporting the Javascript programming language, containing more than 600K reusable packages. According to a survey by StackOverflow[2], JavaScript was the most commonly used programming language in 2018, with 69.8% out of more than 100K respondents affirming their use of JavaScript. In addition, managing dependencies in npm is a growing business and a number of commercial tools to aid in this task are available nowadays [13, 16]. We addressed the following research questions:

RQ1. Why do downgrades occur? We observed two types of downgrades: *reactive* and *preventive*. The main reasons behind reactive downgrades are: defects in the provider version (during build-time, run-time, or development-time), unexpected feature changes in the provider, and incompatibilities (between provider versions, or with Node version). Preventive downgrades occur by pinning a provider package to a prior version in an attempt to avoid issues in future releases of this particular provider. Preventive downgrades can be triggered by recommendations from automated tools.

1. https://npmjs.com
2. https://insights.stackoverflow.com/survey/2018/

RQ2. How is the versioning of providers modified in a downgrade? Downgrades are commonly performed by choosing a specific old version of the provider (62%) instead of specifying a range of acceptable old versions (38%). In 75.5% of the client releases containing a downgrade, only a single provider is downgraded. In 48% of the downgrades, the provider version is reduced by a minor level (e.g., from 2.1.0 to 2.0.0). In addition, we calculated the technical lag induced by downgrades, i.e., the number of releases that are back skipped when a provider is downgraded. We observed that downgrades of major version levels (e.g., from 2.0.0 to 1.2.3) introduce more technical lag than downgrades of minor and patch version levels.

RQ3. How fast do downgrades occur? Half of the downgrades are performed at a rate that is 2.6 times as slow as the typical time-between-releases of their associated client packages. The median time to downgrade an implicitly updated provider is roughly 9 times higher than that for an explicitly updated provider. In specific, only 5.6% of the downgrades are performed in within 24 hours after the update of the provider.

Our key contribution is providing empirically-sound evidence from cross-linked data regarding why and how downgrades occur on npm, while also discussing the implications of our findings to client package developers. As an additional contribution, we provide an algorithm to recover a branch-based ordering of releases, which may be reused by other researchers studying downgrades (and updates) on npm. Finally, we provide a supplementary material with the data that is used in this study [3] as a means to bootstrap other studies in the area.

The remainder of this paper is organized as follows. Section 2 provides a background on how npm selects the provider version to be loaded by a client package. Section 3 describes our approach to detect downgrades. Section 4 explains how we collected and processed the data from npm. Section 5 presents the motivation, approach, and findings to the aforementioned RQs. Section 6 presents a discussion about our findings. Section 7 presents the related work and Section 8 presents the threats to the validity. Finally, Section 9 concludes our paper.

2 DEPENDENCY MANAGEMENT ON NPM

When developers publish a package on npm, they must include a metadata file called *package.json*. The *package.json* file contains a list of providers that are used in a given release of the published package. Each provider has a *versioning statement* associated with it. The versioning statement specifies the version(s) of the provider on which the client (i.e., published package) is willing to depend.

The *resolved version* is the actual version of the provider package that is going to be loaded as a dependency at the time of the installation of the client package. For example, if a client package C depends on a provider package P, the developer of C would include a versioning statement in its *package.json* file such as `"P": "1.2.3"`. This versioning statement informs npm that version 1.2.3 of package P should be loaded when package C is installed.

3. https://github.com/SAILResearch/replication-npm_downgrades

A versioning statement can be one of two types: a specific version (e.g., "P":"1.2.3") or a version range (e.g., "P":">1.2.3"). The specific version statement is satisfied by a unique version of a provider, defined by the right-hand side of the versioning statement (i.e., version 1.2.3). The version range statement is satisfied by a range of versions of a provider (i.e., any version greater than 1.2.3). Version range statements are used by client packages when they wish to implicitly update the version of a provider without having to change their versioning statement. When a provider releases a new version that is satisfied by the existing version range statement by a client package, this provider is implicitly updated. More specifically, the new version of the provider is loaded for all new installs of the client package.

A version range statement has three parts: the provider to which it refers ("P", in the previous example), an operator (">", in the previous example), and a numerical part ("1.2.3", in the previous example). The combinations of operator and numerical part define the range of provider versions that can be satisfied by the versioning statement. In fact, there is a grammar for defining a version range in npm. Such a grammar relies on a set of operator whose definitions and examples are provided on the Appendix A. When a version range is used, the resolved version corresponds to the largest provider version that satisfies the range. Pre-releases of the provider are not satisfied unless they are explicitly included in the versioning statement.

The mechanism to resolve a provider version relies on the precedence between version numbers, since npm needs to know if a particular version number is greater than, less than, or equal to another version number. npm adopts the Semantic Version[4] numbering scheme. A version number in this scheme is comprised of three levels: major, minor, and patch. Each level is separated by a "." (dot) sign, such as in 1.2.3, where 1 is the major level, 2 is the minor, and 3 is the patch. It is also possible to append a"–" (hyphen) sign to the version number, followed by a number or a string, in order to indicate a pre-release. Furthermore, it is possible to append a "+" (plus) sign, followed by a number or a string, in order to indicate that a version is not production ready (i.e., it is merely a new build). Similarly to decimal numbers, semantic version numbers are compared initially by the magnitude of their major level, then by their minor and patch levels. For example, version 3.2.1 is lower than versions 4.0.0 (by a major), 3.3.1 (by a minor), and 3.2.2 (by a patch), but greater than versions 2.2.1 (by a major), 3.1.1 (by a minor), and 3.2.0 (by a patch).

A client package can set a provider package as either a development or a production dependency. A provider package that is set as a development dependency (so-called development provider) is loaded only at the development environment (e.g., the source code repository to which developers commit changes). Consequently, development providers are not loaded when the client package is installed from npm. For instance, test frameworks are generally development providers, since they need to be loaded by the client package developers but not by the client users. As a consequence, issues that arise from development providers do not affect the deployed client package (i.e., in the production environment), making the reaction to such issues less urgent. In turn, provider packages that are set as production dependencies (so-called production providers) are loaded both at the production and development environments. When a client package is installed from npm, providers that are set as production dependencies are also installed with their respective resolved versions and loaded at runtime.

4. https://semver.org

3 Downgrade detection

A downgrade is detected whenever the resolved provider version decreases between two adjacent client releases. Considering the history of releases of a client package, the definition of a downgrade relies on how the logical order of client releases is defined. More formally, given a list of R_C releases of the client package C with the respective version numbers and timestamps, the definition of a downgrade of a provider P by the client C depends on how r_{i-1} and r_i, $\forall i \in \{1 \ldots |R_C|\}$, are defined. To detect downgrades, we sorted the releases of the client packages according to a branch-based ordering algorithm.

The conceived algorithm to derive the branch-based ordering works as follows: as the client releases are examined in a chronological order, we check if any of the previously visited releases are in the same branch as the current one (being a branch defined by the major and minor levels, e.g. the 1.0 branch contains versions 1.0.0 and 1.0.1). If so, then the release with the largest version number in the branch of the current release is deemed the predecessor of the current release. Otherwise, the release visited so far with the largest version number is deemed the predecessor of the current release. We detected 19,651 downgrades by comparing the resolved provider versions from adjacent client releases. Of all detected downgrades, 48% are from development providers and 52% are from production providers. Further details about the approach that we conceived to detect downgrades are described in Appendix B.

4 Data collection

We obtained the *package.json* metadata file of 461,548 packages from the npm registry within the period of December 20, 2010 to July 01, 2017. The *package.json* file lists, among other pieces of information, all the published releases by a client package, the name of the used providers in each release, the versioning statements associated with the providers in each client release, and the timestamp of each release.

We parsed the versioning statement of all dependencies in the *package.json* files according to the adopted grammar by npm (c.f. Section 2). Subsequently, we determined the resolved version of each provider according to the versioning statement used by the client in that release. Such information was used to detect the downgrades that were studied in RQ1, RQ2, and RQ3. We also collected the list of commits, issues, and release notes that are associated with a statistically representative sample of the releases that contain at least one downgrade, from which we obtained information that was used to answer RQ1. Further details about our data collection proccess are given in Appendix C.

5 Results

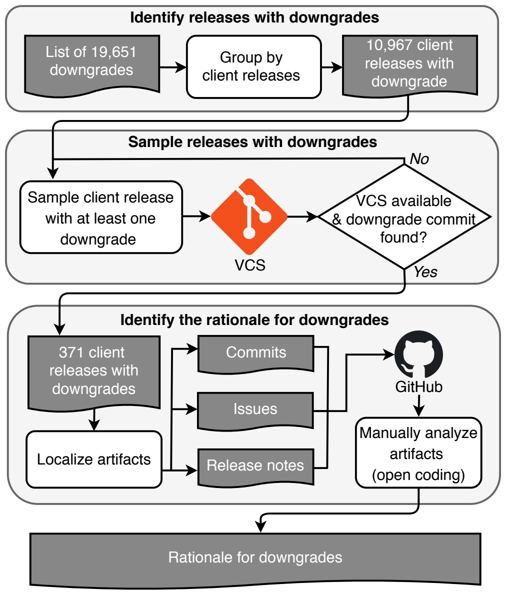
This section presents the results for each of our RQs. For each RQ, we discuss its motivation, the method that we used to address the RQ, and our findings.

5.1 RQ1. Why do downgrades occur?

Motivation: Downgrades indicate that one or more provider versions caused some problem to the client package. Prior studies have only provided limited explanation regarding what these problems are. Therefore, in this RQ, we investigate the rationale behind downgrades.

Approach: We manually examined a statistically representative random sample of client releases in which a provider downgrade occurred. We studied the various artifacts (e.g., release notes, commit messages, and modified files) associated with a client and its release in search for explicit mentions of the rationale for a downgrade. For example, in a commit message that says "*gulp-strip-comment 1.1.1 is broken. force to use an old version*", the rationale for the downgrade is that the package *gulp-strip-comment* at version 1.1.1 caused a failure in the client package.

Figure 1 depicts the approach that we used to identify the rationale for downgrades. We initially grouped the list of 19,651 downgrades into the 10,967 client releases in which at least one downgrade occurred. The reason for grouping the downgrades by the client releases is that the distribution of the number of downgrades per client release is skewed: 52% of the downgrades occur in 20% of the client releases with downgrades. Hence, a simple random sampling of the downgrades would be biased towards client releases with many downgrades.



Identify releases with downgrades
List of 19,651 downgrades
Group by client releases
10,967 client releases with downgrade
Sample releases with downgrades
No
Sample client release with at least one downgrade
VCS
VCS available & downgrade commit found?
Yes
Identify the rationale for downgrades
371 client releases with downgrades
Commits
Issues
Release notes
Localize artifacts
GitHub
Manually analyze artifacts (open coding)
Rationale for downgrades

Fig. 1 Our approach to identify the rationale for downgrades.

After grouping the downgrades by the client releases, we drew a statistically representative sample (95% confidence level and $\pm 5\%$ confidence interval) from these releases (371 cases out of 10,967). For each client package release in our sample of client releases with at least one downgrade, we checked whether the VCS that is used by the package was available and whether the exact commit in which the downgrade was performed could be identified. Whenever an observation in our sample did not meet any of these two requirements, we randomly drew another observation from the population of client releases containing downgrades.

Finally, we manually analyzed the sampled releases in order to identify the rationale for downgrades. The examined artifacts were obtained from the VCS and ITS of each client package. More than 98% of the examined packages used GitHub[5] as their ITS. We performed a thorough examination of the modified artifacts in a commit in which a downgrade was performed. The following artifacts were examined:

- Commit message;
- Artifacts that are modified in the commit, particularly the *package.json* file and the release notes (if available);
- Issues that reference the commit (if available);
- Pull requests that reference the commit (if available).

We performed an open coding [17] over the examined artifacts to categorize the rationale for the downgrades. The codes generated as part of the open coding process are included in our supplementary material.

Findings: Observation 1) ***Downgrades are performed by client packages either to cope with an issue in a specific provider version or in an attempt to avoid potential future issues.*** This observation led us to separate downgrades in two categories, according to their rationale: *reactive* and *preventive*. The motivation for reactive downgrades is to cope with an issue in a specific provider version that negatively affected the client package. Reactive downgrades are captured by quotes such as "*tar@2.2.1 breaks build*", "*Bluebird's 2.9.x branch has proven to be rather buggy and introduces more issues than it fixes, so let's stick with the stable version*". On the other hand, preventive downgrades are performed to avoid issues from recent provider releases. This latter category is represented by quotes such as "*Locks down package.json dependency versions to avoid build inconsistencies and variation across systems*", or "*We should consider pinning all dependencies to prevent issues like this in the future*".

Observation 2) ***There are three issues that motivate a reactive downgrade:*** defect in the provider version, unexpected feature changes in a provider, and incompatibilities. In the following, we describe each of these issues.

Issue 1) ***Defect in the provider version*:** The resolved provider version contains a defect that leads to a failure in the client package. The failure can manifest itself at three different times:

a) Build-time – occurs when the client package fails while it is being installed/built:

5. http://www.github.com

"tar@2.2.1 breaks build." [commit message from pull request #103 of package *urllib*],

"The most recent 0.1.x (0.1.15) broke the build, hence pin it to 0.1.13 for now until it is fixed." [message on commit #2b23764 of package *noise-search*],

"This library currently yield a warning on install" [discussion on issue #28 of package *ua-parser-json*],

"Fix version range for devDependencies (...) fixes #28" [message on commit #2dec1a8 of package *ua-parser-json*]

b) Run-time – occurs when the client package fails while it is running:

"The update has some breaking changes in how the CircularProgress is rendenred [sic] (...)" [message on commit #3273dae of package *d2-ui*],

"lock dependencies so specs run" [message on commit #1dfad5e of package *wunderbits.db* when downgrading provider *karma*],

"The way tokens are exposed seems to have changed fundamentally, which broke parsing." [message on pull request #832 of package *witheve* when downgrading provider *chevrotain*]

c) Development-time – occurs when the failure manifests itself during the development and in-house testing of the client package:

"fix versions of things in package.json to original known working versions (trying to get react datum tests working again)" [message on commit #7397630 of package *bumble-test*],

"Fixed package.json which for some reason was not allowing webpack and karma validate for my tests." [message on commit #5d10a53 of package *karma-styluspreprocessor*]

The defect in the provider might also occur due to degradation of non-functional requirements. The actually resolved provider version is not able to fully adhere to a non-functional requirement of the client package:

"Downgrade 'css-loader' to 0.14.5 to address superslow HMR builds (...)" [message on commit #67bec17 of package *brokerjs*],

"Rollback the Karma dependency version. 'karma' was taking a long time (∼30s) to exit the test suites (...)" [message on commit #6f49232 of package *packery-angular*],

"Freeze dependencies version to better use cache on Travis (...)" [message on commit #0d4633d of package *ember-cli-foreigner*]

Issue 2) ***Unexpected feature changes*****:** The current provider version behaves in an unexpected and/or undesired way compared to some prior version (however, the provider's behaviour is not considered defective).

"Reverting mysql to 2.1.1 (...) Unfortunately mysql has changed the way it handles the charset setting (...) We need to revert this upgrade until the issue is fixed or we have a way to handle it nicely for our users." [message on commit #1f17d5b of package *ghost*],

"Downgraded esdoc to 0.4.3 because they got rid of CLI options" [message on commit #67bec17 of package *brokerjs*]

Issue 3) ***Incompatibilities*****:** An incompatibility prevents the client package from operating properly. We identified two sources of incompatibilities.

a) Incompatibilities between provider versions – it occurs when the version of two (or more) providers that are used by the client package are not compatible with each other:

"release 1.3.2 fix fs-extras compability [sic]." [message on commit #3f0f6c4 of package *yog2-kernel*],

"Package version modification for compatibility." [message on commit #d3f42 of package *mozaik-ext-jira-2* when downgrading package *superagent*],

"reverted jquery version to 2 for jquery-ui compatability [sic]." [message on commit #926653a of package *yasgui-yasr*]

b) Incompatibilities with Node version – it occurs when the resolved provider version requires a specific Node version, which in turn is incompatible with the Node version that is used by the client. Node is the run-time engine for JavaScript, which is the language in which npm packages are written:

"npm versions changed to run the project with node 5.4 (...)" [message on commit #ee5c524 of package *d3-composite-projections*],

"Fix npm error on node 0.8.x." [message on commit #d5d3078 of package *grunt-wget*],

"remove caret to allow compatibility with node 0.8" [message on commit #f0b57e4 of package *nodo*],

"(...) lock to very specific version that works on node 0.10" [message on commit #76de1c0 of package *stack-utils-node-internals*]

Observation 3) ***A preventive downgrade is performed to avoid potential issues from future releases of the provider.*** This preventive action is often referred to in the examined commit messages as "pinning" or "locking" the provider version. It is done to avoid potential failures that might arise when a provider is updated to a new version. When pinning a provider version, the developer of the client package typically removes the range operator from a version range statement. Such a modification to the versioning statement might lead to a downgrade. For example, if the versioning statement is modified from `"P"`: `">=2.0.0"` to `"P"`: `"2.0.0"` and the newest version of the provider *P* is 2.0.1, then this provider will be downgraded from 2.0.1 to 2.0.0 when the range operator `">="` is removed. The following excerpts are examples that we observed in the manually examined downgrades:

"We have now had 2 issues where a "patch" upgrade in a dependency broke Parse Server. (...) We should consider pinning all dependencies to prevent issues like this in the future." [discussion on issue #2040 of package *parse-server*],

"Locks down package.json dependency versions to avoid build inconsistencies and variation across systems." [message on pull request #82 of package *lightstep-tracer-javascript*],

"Use exact versions in package.json: Because some of the new ones caused issue when calling npm install." [message on commit #c1e12fe of package *atom-keymap*],

"Lock broccoli-funnel to prevent rebuild error (...)" [message on pull request #237 of package *ember-engines*],

"please pin the moment dependency: By using >= you expose anyone using your package and installing via npm install to different versions of the package being installed." [discussion on issue #55 of package *emailjs*],

"Lock broccoli-funnel to prevent rebuild error (...)" [message on pull request #237 of package *ember-engines*]

Observation 4) ***Preventive downgrades can be triggered by recommendations from automated tools.*** Automated tools that manage dependency versioning can recommend that a client package converts all version ranges to specific versions. These recommendations are deployed through automatically created pull requests that simply remove the range operator from all the versioning statements listed in the client's *package.json* file.

"Hello! We're all trying to keep our software up to date, yet stable at the same time. This is the first in a series of automatic PRs to help you achieve this goal. It pins all of the dependencies in your package.json, so you have complete control over the exact state of your software." [message in pull request #16 of package *noflo-core*]

5.2 RQ2. How is the versioning of providers modified in a downgrade?

In this RQ, we investigate how the versioning of providers is modified when a provider is downgraded. Our investigation contemplates three different angles, namely: how the versioning statements are modified in releases containing a downgrade (Section 5.2.1), how many providers are downgraded in a release containing a downgrade (Section 5.2.2), and how the resolved provider's version changes when a downgrade occurs (Section 5.2.3).

5.2.1 Modification of versioning statements

Motivation: From a client package perspective, using version range statements has the advantage of reducing the overhead of keeping its providers up-to-date. On the other hand, the adoption of version range statements makes the client package susceptible to bugs in provider versions that are implicitly updated. We hypothesize that downgrades are associated with a transition from version range statements to specific versions, specially when a problematic implicit update triggers the downgrade. An evidence of such an association is the occurrence of preventive downgrades and the action of "pinning" dependencies, as we observed in RQ1. In fact, practitioners often advocate against the adoption of version range statements due to the possibility of being caught by surprise by a newly introduced bug in a provider version [18, 19, 20]. In this RQ, we investigate how downgrades are associated with changes in the versioning statements.

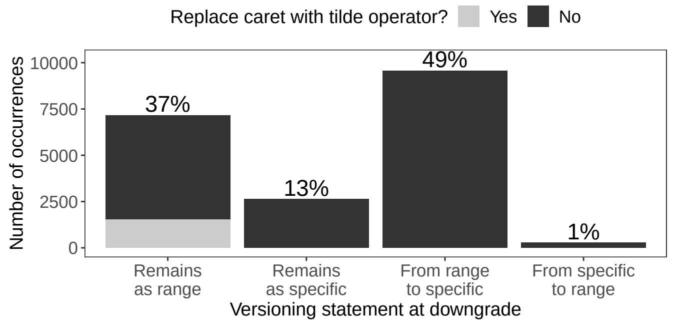


Fig. 2 Proportion of downgrades per type of versioning statement change.

Approach: We calculate the proportion of downgrades that resulted from replacing a version range statement with a specific version and vice-versa. In addition, for the cases in which a version range remains being used after a downgrade, we investigate how the operators and numerical part of the versioning statement are modified.

Findings: Observation 5) ***Almost half (49%) of the downgrades occur due to a replacement of a version range statement with a specific version.*** Figure 2 shows the proportion of downgrades per type of versioning statement change. The most common type of versioning statement change in downgrades (49%) is *from range to specific* (of which 49% are from production providers). In this subgroup, 68% of the cases were performed simply by removing the range operator and keeping the numerical part. As we observed in RQ1, this is how preventive downgrades are typically performed: instead of carefully choosing a provider version, developers simply remove the version range operator from versioning statements. Also, when the versioning statement remains as specific, the proportion of downgrades that are from production providers is 61%. In turn, the same proportion is 48% when the versioning statement changes from specific to range.

As also shown in Figure 2, the versioning statement *remains as a range* in 37% of the downgrades (of which 51% are from production providers). In this subgroup, 21.8% of the cases involved replacing a caret ($\wedge$) operator with a tilde ($\sim$) operator. The tilde operator resolves towards a patch update of the provider, while the caret operator resolves towards a minor update. In 58.1% of the caret-to-tilde replacements, the numerical part of the version range did not change. As a consequence, the range of provider versions that are accepted by the client is narrowed down. There are no cases for which the change from caret to tilde would be effectless (i.e., downgrades in which the versioning statement change from "`∧0.x.y`" to "`∼0.x.y`").

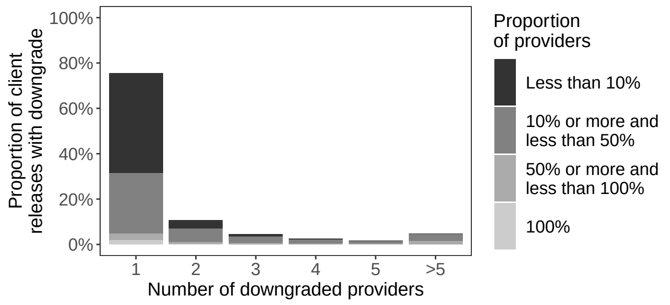


Fig. 3 Number of downgraded providers in the same client release.

5.2.2 Number of providers that are downgraded in a release

Motivation: When one or more providers cause an issue, client package developers might perform a reactive downgrade. However, detecting the specific troublesome provider might not be trivial and developers might end up downgrading unrelated providers. In addition, "pinning" a large number of providers might result in downgrading many providers at once. Hence, in this research question we investigate how localized downgrades are.

Approach: We categorize all client releases with downgrades as having one, two, three, four, five, and more than five providers downgraded. Afterwards, we count the number of client releases with downgrades that fit these categories. For each client release with a downgrade in one of these categories, we verified the proportion of providers (from the total number of used providers) that were downgraded.

Findings: Observation 6) ***In 75.5% of the client releases containing a downgrade, only a single provider is downgraded in these releases.*** This observation is depicted in Figure 3. In 58.3% of the releases in which a single provider was downgraded, more than 10 providers were being used by the client package (black portion of left-most stacked bar). On the other hand, in only 2.5% of all releases with downgrades all providers were downgraded at once (sum of lightest gray portion over all stacked bars). These results thus indicate that downgrades are often localized.

5.2.3 Introduced technical lag

Motivation: When a client package performs a downgrade, there is an increase in the technical lag regarding the resolved provider version [21, 22]. As such, downgrades naturally prevent client packages from leveraging the benefits brought by newer releases, including bug fixes, vulnerabilities fixes, and new features [8, 23, 24]. Thus, it is generally advised that client packages keep their providers up-to-date. The technical lag introduced in a downgrade can not only affect the client package itself, but also affect transitive dependencies. For this reason, it is important to evaluate the impact of downgrades in other packages in the ecosystem. In this RQ, we measure the impact of downgrades on the technical lag and the extent to which the introduced technical lag can impact client packages that use a release with downgrade. Finally, Zerouali et al. [25] show that there is a difference between the technical lag of development and production providers. Therefore, we study whether the technical lag introduced in a downgrade differs between these two types of providers.

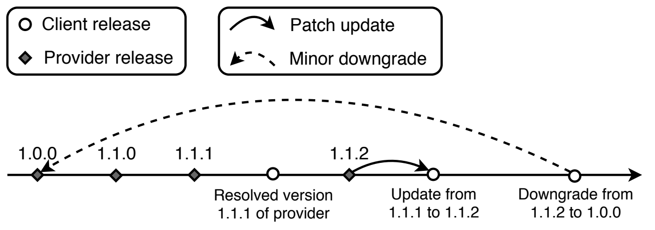

Client release
Provider release
Patch update
Minor downgrade
1.0.0
1.1.0
1.1.1
1.1.2
Resolved version 1.1.1 of provider
Update from 1.1.1 to 1.1.2
Downgrade from 1.1.2 to 1.0.0

Fig. 4 Number of back skipped provider versions in a minor downgrade following a patch update.

Approach: We calculate the proportion of provider versions that were reduced by a major, minor, and patch level in a downgrade. In addition, we compare the increased technical lag when a downgrade occurs with the decreased technical lag when the prior update occurred. The increase (or decrease) in technical lag is measured by the number of already published provider versions that are back (or forward) skipped in a downgrade (or update), according to the numerical ordering. Figure 4 depicts this calculation. The figure shows an update followed by a downgrade. A patch update changes the resolved provider version from 1.1.1 to 1.1.2, decreasing the technical lag by one patch release. On the next client release, the fourth version (1.1.2) of the provider is downgraded towards the first version (1.0.0). In this example, the downgrade back skipped three versions (1.1.2, 1.1.1, and 1.1.0). Hence, we say that the technical lag was increased by three versions (or two patch and one minor release). Finally, we calculate the proportion of releases with downgrade that have at least one client package.

We verify if the distribution of the number of back-skipped major, minor, and patch releases is different between downgrades of development and production providers. To compare the distributions, we test the null hypothesis that both distributions do not differ from each other using the Wilcoxon Rank Sum test ($\alpha = 0.05$) [26] and assess the magnitude of the difference with the Cliff's Delta (d) estimator of effect size [27]. To classify the effect size, we use the following thresholds [28]: *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.

Findings: Observation 7) ***13% of the downgrades induce an unnecessary increase in the technical lag.*** Table 1 shows the proportion of patch, minor, and major downgrades that follow a patch, minor, and major update. Downgraded version levels that are larger than the updated version level are shown in grey filled cells. Almost one fifth of the minor downgrades (18%) follow a patch update and a total of 18% of the major downgrades follow a patch or a minor update. For such cases, the downgrade not only nullifies the benefits of the prior update, but also increases the technical lag.

In 48% of the downgrades, the provider package version is reduced by a minor level. Patch and major downgrades represent, respectively, 27% and 25% of the downgrades. Interestingly, these proportions do not correspond to the proportion of patch, minor, and major releases of down-

TABLE 1 Proportion of major, minor, and patch downgrades that follow a major, minor, or patch update.

Downgraded version level	Update that precedes the downgrade: Patch	Minor	Major
Patch (27%)	95%	3%	2%
Minor (48%)	18%	80%	2%
Major (25%)	9%	9%	82%

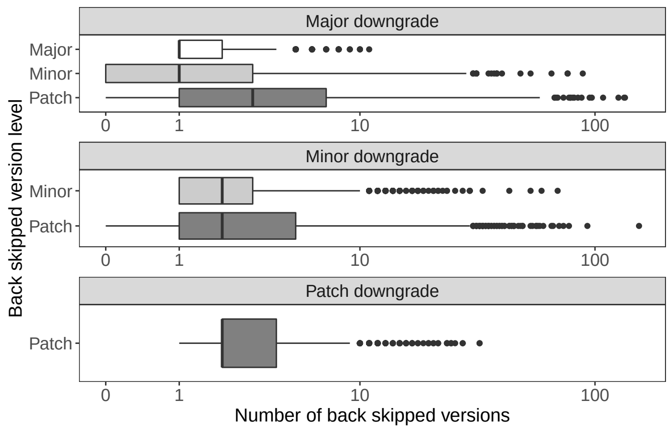


Fig. 5 Number of back skipped provider versions in a major, minor, and patch downgrade.

graded packages which are, respectively, 72%, 23%, and 5%. In addition, almost one fifth (19.4%) of the releases containing a downgrade have at least one client package, representing cases in which the technical lag can affect transitive dependencies.

Observation 8) ***Major downgrades back skip a median of 1 major, 1 minor, and 3 patch releases.*** Figure 5 shows the increase in technical lag for each modified version level in a downgrade. In 65% of the major downgrades, at least one minor release is also back skipped, while in 91% a patch release is also back skipped (59% back skip both minor and patch releases). Also, we verified that 85% of the minor downgrades back skip at least one patch release. Comparing downgrades of development and production providers, the difference between the distribution of the number of backskipped major and patch releases is not statistically significant (p-value > 0.05). While the difference for the number of back skipped minors is statistically significant, the effect size is negligible ($|d| = 0.114$).

Major downgrades represent one quarter of all downgrades and typically follow a pattern. 82% of the major downgrades are preceded by a major update. Also, 75% of the updates preceding a major downgrade are explicit, indicating that the versioning statement used at the update time generally does not satisfy the new major releases of the provider and that major updates tend to be a well-thought-out decision. Finally, 70% of the major downgrades are rollbacks (i.e., the target of the downgrade is the version that was originally used by the client package). These results suggest that major downgrades are likely the result of a failed attempt to update to a major version.

In 88% of the major updates that precede a major downgrade, the update was explicit. This observation shows that the majority of the major updates that precede a major downgrade are deliberated, suggesting that many issues that arise after a major update of a provider manifest themselves in-field (i.e., after deployment) but not in-house (i.e., at the development environment).

5.3 RQ3. How fast are downgrades performed?

Motivation: A downgrade indicates that one or more providers caused some issue to the client package. In particular, when the provider is implicitly updated (i.e., because the new provider version satisfies the specified version range), these issues can manifest themselves in a sudden manner, making it challenging to rapidly identify the provider that is associated with the issue. In fact, prior research has shown how unexpected defects impact the quality of a software product [29]. Measuring the time between the update of a provider version and the consequent downgrade can thus help to understand how fast client packages are able to react to the issues behind downgrades. Furthermore, client packages can react to issues that arise from development and production providers with a different degree of urgency. For example, issues from development providers should not impact the deployed client package and the contingency of such issues can be delayed without affecting the client package's users. In this RQ we also differentiate between the time to downgrade development and production providers.

Approach: To determine how fast a downgrade is performed, we calculate the ratio shown in Equation 1. The ratio is a means to compare the taken time to perform a given downgrade with the typical time between two releases of a client package. Values larger than 1 indicate that a downgrade D takes more time to occur than a typical release of the client package C. A typical time between releases of a client package C is calculated by $timeRel(C)$ in Equation 1. We verify whether there is statistical difference between the $speedRatio(C, D)$ of development and production providers. To do so, we used the Wilcoxon Rank Sum test ($\alpha = 0.05$) and the Cliff's Delta estimator of effect size (see Section 5.2.3 for a classification of the effect sizes).

$$speedRatio(C, D) = \frac{time(D)}{rel(D) \times timeRel(C)} \qquad (1)$$

where:

- $time(D)$ is the elapsed time (in days) between the update and the eventual downgrade D.
- $rel(D)$ is the number of spanned client releases between the update and the eventual downgrade D (inclusive).
- $timeRel(C)$ is the median elapsed time between the last half releases of a client package C (in days per release).

We also investigate whether downgrades of implicit updates take longer than downgrades of explicit updates. Given a downgrade, we determine the timestamp of the preceding update based on how this update occurred. When an update is explicit, i.e., it occurs because the versioning statement was modified, the timestamp is the date at which the client package publishes a release with the updated

TABLE 2 Summary of the variables calculated in Equation 1.

	1^{st} quart.	Median	Mean	3^{rd} quart.
$speedRatio(C, D)$	1.02	2.63	191.59	10.78
$time(D)$	10.73	34.84	79.76	95.86
$rel(D)$	1.00	2.00	5.05	4.00
$timeRel(C)$	1.24	5.55	19.14	18.14

provider (depicted as a shaded dot in the timeline on the left-hand side of Figure 6). On the other hand, when an update is implicit, i.e., it occurs because the actual version range satisfies the new version of the provider, the update timestamp is the date at which the provider released the version that was eventually downgraded (depicted as a shaded diamond in the timeline on the right-hand side of Figure 6).

We compare the distribution of the elapsed time between an explicit update and its eventual downgrade with the distribution of the elapsed time between an implicit update and the eventual downgrade. This comparison is controlled by the downgraded version level (i.e., whether it is a patch, minor, or major downgrade). We compared the distributions using the Wilcoxon Rank Sum test ($\alpha = 0.05$) and the Cliff's Delta estimator of effect size (see Section 5.2.3 for a classification of the effect size).

Findings: Observation 9) ***50% of the downgrades are performed 2.6 times slower than k typical releases, where k is the number of releases taken for the downgrade to occur.*** Figure 7 shows the distribution of the $speedRatio(C, D)$ (Equation 1) for development and production providers. The difference between the two distributions is statistical significant ($p < 0.05$) with a negligible effect size ($|d| = 0.026$). The first quartile of both distributions is greater than 1, indicating that more than 75% of the client releases published during the update and following downgrade of a provider are slower than the typical releases of the client package. In addition, Table 2 shows the median, mean, first, and third quartile for the variables used in Equation 1. The median time for a downgrade to occur is 34.8 days and 50% of the downgrades occur in one or two releases after the update of the provider.

Observation 10) ***Downgrades of an implicit update generally take longer than downgrades of an explicit update.*** The observed difference can be explained by the fact that implicit updates are not controlled by client package developers. Hence, an issue that arises after the provider is updated can appear unexpectedly. Thus, developers might need additional time to identify the provider(s) that is(are) associated with the issue that concerns the downgrade. On the other hand, when performing an explicit update, developers are aware of which providers were modified, making it easier to identify a provider that eventually needs to be downgraded.

Considering explicit updates that precede a downgrade, the median time to downgrade a provider is 35 days for major downgrades, 37 days for minor downgrades, and 46 days for patch downgrades. Considering implicit updates, the median time is, respectively, 36, 48, and 35 days for major, minor, and patch downgrades. Figure 8 depicts the distributions. All pairwise differences between the downgraded version levels (grouped by implicit and explicit updates) are statistically significant, but have negligible effect size. In turn, comparing the update types (implicit vs. explicit) for major, minor, and patch downgrades, we obtain, respectively, a statistically significant difference with large effect size ($|d| = 0.474$), medium effect size ($|d| = 0.454$), and large effect size ($|d| = 0.477$).

Observation 11) ***Urgent downgrades occur more often after an explicitly update than after an implicit update.*** Urgent downgrades refer to downgrades that occurred in less than one day after the update of the provider. Figure 8 depicts this observation. We found that 5.6% of all downgrades are performed in an urgent manner. Also, 37.1% of the downgrades of explicit updates are urgent downgrades. In contrast, only 3.8% of the downgrades of implicit updates are urgent updates. This observation corroborates our conjecture that, because explicit updates are controlled by the client packages, developers are more likely to react fast to the issues that were brought by these updates. In addition, 67% of the urgent downgrades are from production providers.

6 Discussion

In this section, we discuss the lessons that we learned from conducting this study (Section 6.1), as well as opportunities for future research (Section 6.2).

6.1 Lessons learned

The general lesson learned from our findings is that *package developers should manage their dependencies*. In particular, package developers should keep track of their dependencies over time and be cautious with provider updates. Ultimately, these practices should optimize the debug of troublesome updates (consequently reducing the time to fix issues that affect the packages' clients) and reduce the technical lag that is introduced when a downgrade occurs. In the following, we present specific lessons learned to help practitioners manage dependencies.

Learned lesson 1) ***Package developers can use automated tools to support early discovery of provider issues and thus decrease the time taken to downgrade.*** The issue that motivates a downgrade can take some time to manifest itself. In particular, downgrades associated with implicit updates take longer to occur (c.f., Observation 10), thus delaying the provision of the fix to the packages' users. Several tool-assisted approaches can be employed to support the *early* detection of troublesome provider updates. A simple approach can be employed using three different tools: (i) latest-version[6] checks what the latest version of a provider package is, (ii) next-update[7] runs the client's test suites, and (iii) npm-check-updates[8] automatically updates the versioning statements in the *package.json* file A step further in the degree of automation involves the usage of bots to manage dependency updates, such as Greenkeeper[9] and

6. https://www.npmjs.com/package/latest-version
7. https://www.npmjs.com/package/next-update
8. https://www.npmjs.com/package/npm-check-updates
9. https://greenkeeper.io

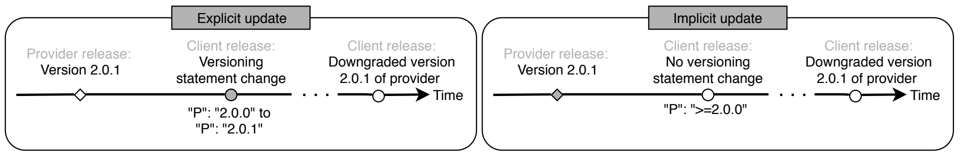

Explicit update
Provider release: Version 2.0.1
Client release: Versioning statement change
Client release: Downgraded version 2.0.1 of provider
Time
"P": "2.0.0" to "P": "2.0.1"
Implicit update
Provider release: Version 2.0.1
Client release: No versioning statement change
Client release: Downgraded version 2.0.1 of provider
Time
"P": ">=2.0.0"

Fig. 6 Downgrades preceded by explicit and implicit updates.

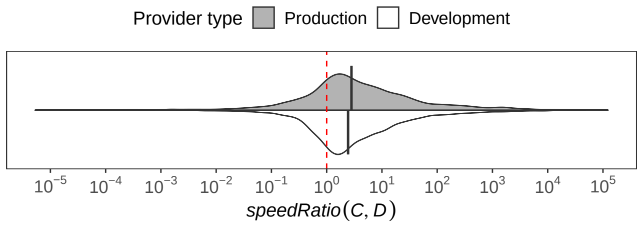


Fig. 7 Distribution of the $speedRatio(C, D)$. The red dotted line indicates a ratio value of 1.

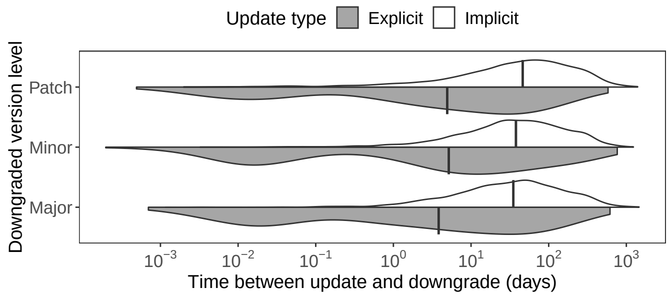


Fig. 8 Distribution of the elapsed time between the update and the eventual downgrade of a provider package.

Renovate[10]. These bots interact with client package developers through the package's ITS (e.g., GitHub). The following workflow is generally implemented: (i) the bot identifies an opportunity for an implicit update of a provider, (ii) the implicit update is performed in an isolated branch, (iii) the bot runs a suite of automated tests and attempts to rebuild the package, (iv) in case the test suite or the build fails, the bot opens an issue report with recommended actions.

Learned lesson 2) ***Client packages can log their dependency tree to debug troublesome providers.*** While the approach described in Lesson learned 1 ensures that tests pass and the build does not break, it is still possible that a provider package might lead to a problem. For instance, incompatibilities or a performance regression might not be captured by the package's test suite. In this scenario, package developers might need to debug the troublesome update. A lightweight approach would consist of simply keeping a log file containing the state of the client package's dependency tree (i.e., all the provider versions that are loaded at a given time). With such log files, developers can trace back the state of the dependency tree at a given time and determine the exact updates that potentially led to the problem. Logging updates can be achieved by setting an automated background routine that uses simple commands provided by npm and the VCS. Such a routine can be implemented by: 1) using the npm-update command in an isolated environment (e.g., in a new branch or within any folder created to this end), such that this isolated environment contains the client package with its providers updated to the latest version that satisfies the versioning statements, 2) using the npm-ls command to produce a report of the dependency tree with the loaded provider versions, 3) committing the dependency tree state on a daily basis to the VCS, and 4) using some VCS's tool (e.g., blame[11] from Git) to identify the update of a given provider.

Learned lesson 3) ***Client packages using a flexible dependency versioning strategy (i.e., extensive use of range statements) should emphasize testing the functionalities that involve provider packages.*** Reactive downgrades are caused by issues coming from the provider packages. Therefore, testing functionalities that rely on the providers should be intensified. In addition, testing corner cases for the provided functionalities by the providers can safeguard client packages, especially when those functionalities are not fully tested by the provider itself. Moreover, due to incompatibilities, client packages should, if possible, test scenarios where multiple providers are used together.

Learned lesson 4) ***Client packages should be mindful of the latest working provider version when pinning a dependency.*** Almost half (49%) of the downgrades are performed by pinning the provider version. Pinning is typically (68%) performed by removing the range operator and keeping the numerical part of the range statement (c.f., Observation 5). However, this downgrade pattern can lead to the adoption of a version of the provider that is older than the latest working provider version (thus increasing technical lag). Therefore, client packages should consider the latest working version of the provider instead of simply removing the version range operator from the versioning statement.

6.2 Avenues for future research

In the following, we list future research that can be leveraged from our results.

• *Further research must be carried out to understand how downgrades affect packages throughout the dependency network.* Although we observed that only 19% of the releases with downgrade have at least one client package, downgrades can still transitively impact packages in the ecosystem. For instance, a package A can depend on a package B which, in turn, might depend on a package C. Therefore, package A can transitively depend on some feature of package C. If C is downgraded by B, A will transitively depend on a downgraded version of C. However, in this paper, we do not investigate the impact of downgrades in transitive dependencies.

10. https://renovatebot.com

11. https://git-scm.com/book/it/v2/Git-Tools-Debugging-with-Git

• *Further research is necessary to understand why downgrades tend to take long to occur.* We conjecture that either the problem that triggered the downgrade takes long to manifest or tracing a problem back to a certain provider version is not trivial. Also, we conjecture that, due to the controlled nature of explicit updates, it is easier for client packages to identify the provider that is associated with the problem that motivated the downgrade.

• *Further research should be performed to understand the extent to which vulnerability and security advisories might be influencing client developers to downgrade a provider.* Different studies show the relevance of security vulnerabilities to the decision of updating a given provider [15, 30]. However, after manually analyzing a representative sample of downgrades (c.f. Section 5.1), we did not find any explicit mention of security vulnerabilities. We conjecture that client packages tend to wait for a vulnerability fix instead of performing a downgrade.

7 Related work

In this section, we describe related work concerning dependency downgrades, dependency management in npm, and dependency management in other ecosystems.

Downgrade of dependencies: Two prior studies mention the phenomenon of downgrades in the npm ecosystem. Decan et al. [15] analyzed the impact of security vulnerabilities in npm dependencies. The authors highlight that vulnerable providers impact the quality of their client packages. The authors also claim that vulnerability issues can be solved by "rolling back to an earlier version" of a provider. When analyzing a representative sample of the downgrade cases (Section 5.1), we did not identify any explicit mentions to downgrades being performed because of vulnerabilities in a provider. However, we did identify that one of the rationales for downgrades is the presence of defects in the provider. In this sense, it is possible that these defects encompass vulnerabilities issues.

Mirhosseini and Parnin [13] studied the effectiveness of automated tools (e.g., bots) to manage dependencies in npm. The authors show that providers are updated 1.6 times more often and 1.3 times faster when client packages use such bots, compared to clients that do not. By means of a survey, the authors demonstrate that the three most common developers concerns regarding the update of providers are backward-incompatible changes, understanding of the implications of changes, and migration effort. Furthermore, 24% of the builds fail when some provider version is changed. The authors also found that several provider updates performed by bots were downgraded in 2 or 3 days after being merged. In this paper, we found that defects in the provider, in particular the ones that affect the client package build, motivate downgrades. In addition, we identified that bot recommendation is one of the reasons behind preventive downgrades of a provider.

A few other papers approach the subject of downgrades in software ecosystems other than npm, such as the Apache[12] and the Android[13] ecosystems. Mileva et al. [14] performed an empirical study of over 250 Apache projects with the goal of understanding how the popularity of a package relates to its quality. They propose that the number of downgrades of a given provider is an indicator of the (lack of) quality of that package. Our findings corroborate this proposition.

Salza et al. [12] analyzed the categories of provider packages that were downgraded in mobile apps. They found that Graphical User Interfaces (GUI) and Utilities are the categories of providers with the highest number of downgrades. The authors show evidence that client packages want to follow look and feel tendencies, which explains the high number of updates of GUI-related providers. Also, utility packages support the development of applications in the ecosystem and are highly popular. Despite the high number of packages that depend on those providers in mobile apps, the authors did not explain *why* downgrades of these providers are often performed.

Dependency management in npm: Wittern et al. [7] studied the evolution of dependencies in npm. They found that from 2011 to 2015, the proportion of packages being used as dependencies (provider packages) increased from 23% to 81%. As the number of dependencies increases in an ecosystem, the likelihood of downgrades increases. Also, the authors present evidence that the Semantic Version specification is not being followed by provider packages when they increment a version number. This is likely problematic, since backward-incompatible changes could be introduced when minor or patch levels of the provider version are incremented. As a consequence, these backward-incompatibilities would manifest themselves unexpectedly on the client package side. Indeed, we identified that almost half of the downgrades reduce the provider version by a minor level.

Zerouali et al. [25] analysed the technical lag induced by direct dependencies in npm over seven years period. The authors found that the median technical lag is 1 major, 1 minor, and 4 patch releases. Our analysis (c.f. Section 5.2.3) shows that major downgrades introduce a similar technical lag compared to the technical lag that is typically induced by direct dependencies.

Vulnerabilities in npm were studied by Zapata et al. [31]. The authors conjecture that fixing a vulnerability from a provider package can cause a failure in the client package. Also, Zerouali et al. [30] identified that vulnerabilities are common in official Docker containers using npm packages as dependencies. When analyzing a representative sample of the downgrade cases (Section 5.1), we did not identify any explicit mentions to downgrades being performed because of vulnerabilities in a provider. However, we did identify that one of the reasons for downgrades is the presence of defects in the provider. In this sense, it is possible that these defects encompass vulnerabilities issues.

In a survey with developers from 18 ecosystems, Bogart et al. [32] show that provider packages in npm are often introducing changes that require modifications in the client code. This observation can partially explain the high number of downgrades found in npm: client package developers use downgrades as a workaround to postpone the need for change their code after a provider being implicitly updated. Mezzetti et al. [33] propose a technique called type regres-

12. https://www.apache.org
13. https://developer.android.com

sion that can detect changes in the provider package that cause a failure in a client package (a.k.a breaking change). Detecting breaking changes can be useful to avoid issues that cause a downgrade to occur. However, this technique is only accurate when providers are used by a large number of client packages. Our results show that the number of client packages that depend on a release with downgrade is, in general, small, making the type regression technique impractical for such cases.

Decan et al. [34] studied the dependency network of seven software ecosystems (including npm) and concluded that developers face issues when updating providers. Because packages in those ecosystems heavily depend on each other, the authors highlight the fact that a change in one package can affect many others. Kikas et al. [35] also performed a similar study over the dependency network of the npm, RubyGems, and Cargo[14] (for the Rust language) ecosystems. The authors identified a key set of packages that, if modified, could impact over 30% of the other packages in the ecosystems. In Section 5.2.3, we verified that 19.4% of releases with downgrade have more than one client package, representing cases in which the technical lag introduced by a downgrade can affect transitive dependencies.

Decan and Mens [36] studied whether the Semantic Version specification is respected by packages in four different ecosystems (including npm). The authors observed that versioning statements used by client packages generally accept implicit updates of providers that comply with the Semantic Version specification, suggesting that packages in this ecosystem tend to respect this specification.

Dependency management in other ecosystems: Prior studies focused on the management of dependencies versioning in different software ecosystems. In the scope of the Android ecosystem, McDonnell et al. [6] analyzed the relation between the adoption of API versions and the API stability. The authors show that client packages avoid to update towards unstable APIs (i.e., APIs that change frequently). In our findings, we uncovered that defects in the provider are one of the causes for providers to be downgraded. Those defects might be associated with unstable APIs of the provider packages. Ruiz et al. [37] analyzed the rationale for Android applications (client packages) to update their ad libraries (provider packages). Some of the reasons that they identified are related to the rationale for downgrades in npm. In particular, they found that fixing a bug and improving performance are some of the reasons to update an ad library. Derr et al. [8] performed a survey with Android developers to understand how they update provider packages. They observed that preventing incompatibilities is the second-ranked reason why developers prefer to use outdated providers. In our study, we also observed that incompatibilities are one of the rationale for downgrading a provider package.

The management of dependencies versioning was also studied in some ecosystems for packages written in Java. Bavota et al. [23] identified some factors associated with updates that intersect with the factors that we identified for downgrades. In particular, bug fix was identified as a factor influencing the update of provider packages. Also, the authors observed that packages that have common dependencies or common developers are more likely to be updated. Kula et al. [10] show that vulnerability and security advisories play a role in the decision to update a provider version. As we observed in our study, preventing errors is associated with downgrades.

Robbes et al. [38] studied how changes in the API of a provider package propagate back to the client package in the Pharo[15] ecosystem (for the Smalltalk language). Changing the API signature of a provider is one of the actions that causes backward-incompatible changes. Normally, the client package cannot avoid such changes when they are introduced in a minor or patch release of the provider. We also found that the majority of the downgrades involve patch and minor version levels. A possible explanation is that provider packages are changing API signatures in minor and patch releases.

Zerouali et al. [39] analyzed the technical lag of outdated installed packages in Docker containers. The authors found that the technical lag on such containers is one to two versions, which is similar to the technical lag that is typically introduced by an npm downgrade.

8 Threats to validity

Internal validity: When identifying the reasons for a downgrade, we searched for the specific commit in which the provider versioning was changed in the *package.json* file. However, it is possible that the downgrade reason was revealed in some prior commit. We likely missed those cases in our analysis. Furthermore, in our manual analysis, we did not inspect every file in the commit but merely searched for an explicit mention for a downgrade in the examined artifacts (see method in Section 5.1). Also, we acknowledge that different classifications of the downgrades (e.g., based on prior theories about maintenance categories or derived from interviews with developers) would likely yield a complementary view to our results.

The proportion of published major, minor, and patch releases are different across npm packages. Such non-uniform distribution of releases types has an impact on the interpretation of our results. We mitigate this threat by controlling for release type (i.e., major, minor, and patch) when appropriate (Sections 5.2.1, 5.2.3, and 5.3).

External validity: Because we collected data exclusively from npm, our findings might not be generalizable to other ecosystems. Although npm is representative in size, in fact, each software ecosystem has its own intrinsic characteristics. The goal of this paper is not to build theories around downgrades that would apply to all software ecosystems. Rather, our study is only a first step towards a deeper understanding of why and how packages are downgraded. Therefore, we acknowledge that additional studies are required in order to further generalize our results. Nonetheless, to the best of our knowledge, this is the first paper to thoroughly investigate the phenomenon of downgrades. In addition, our approach can be replicated in other ecosystems. Structures similar to versioning statements (i.e., that allows one to set a specific version or a range of versions to a given

14. https://crates.io

15. http://catalog.pharo.org

provider) and version numbering schemes can be found in several other software ecosystems, such as Bundler (for Ruby), Cabal (for Haskell), pip (for Python) and Maven (for Java). Downgrades of provider versions can also be found in all those ecosystems platforms.

Construct validity: We identified the downgrades in npm based on a heuristic for ordering the client releases, which we call branch-based order (see Section 3). However, although this heuristic is a best-effort to capture the logical ordering of the releases, the actual ordering adopted by a package can be arbitrary. Therefore, it is not guaranteed that the use of such heuristic captures all downgrades performed in npm. Nevertheless, we consider that, with respect to the identification of the downgrades, the branch-based ordering represents the actual order of releases more accurately than either the numerical or chronological orderings. The reasons for this consideration are explained in Section 3. Furthermore, our manual analysis served as a sanity-check for the reliability of our approach to detect downgrades. Finally, in Section 5.3, we calculate the typical inter-release time of a package as the median time between the last half releases. Although considering the last half releases is an arbitrary decision, this is a reasonable form of representing the current inter-release time of packages that have different release schedules.

Swanson [40] proposes four dimensions of maintenance activities, namely corrective, adaptive, perfective, and preventive. The classification of downgrades into reactive and preventive (RQ1) is related with the dimensions of maintenance proposed by Swanson [40]. Conceptually, reactive downgrades can be understood as a combined form of corrective, adaptive, and perfective maintenance, while preventive downgrades can be understood as a preventive maintenance activity. However, for some of the manually investigated cases, it was impossible to determine, given the available evidence, whether a reactive downgrade was corrective, adaptive, or perfective. Hence, driven by the constraints of the data that we investigated, we simply classify downgrades into reactive and preventive.

9 Conclusions

The benefits of having up-to-date provider versions is extensively studied in the literature [5, 10, 37]. Prior studies also point to the reasons why developers might prefer not to update provider packages [8, 23, 41]. On the other hand, only a few papers examine aspects related to downgrades in software ecosystems [12, 14, 15]. Using historical data from package releases, we empirically investigate downgrades in npm. In particular, we study why provider packages are downgraded, how they are downgraded, and how fast the downgrade occurs. Our results show that downgrades are a facet of the management of dependencies in software ecosystems, being used as a workaround to deal with issues coming from provider packages. In Section 6 we discuss a set of procedures that practitioners can implement to better cope with the need for downgrading a provider. We make the following observations:

• *Downgrades are performed because of issues that arise from the provider, but are also performed for preventive purpose*. We identified two types of downgrades as reactive and preventive (Observation 1). Three issues motivate reactive downgrades, namely defects in the provider version (during build-time, run-time, or development-time), sudden feature changes in the provider, and incompatibilities (between provider versions, or with Node version) (Observation 2). On the other hand, preventive downgrades are originated from the preventive action often called by client developers as "pinning" the provider version (Observation 3). Also, preventive downgrades can be triggered by bot recommendations (Observation 4).

• *Downgrades are associated with a change to a more conservative versioning of providers*. We observed that 49% of the downgrades change the versioning statement from range-to-specific. In addition, when the versioning statement remains as range after a downgrade, the range of acceptable versions is often narrowed down (Observation 5). In 75.5% of the client releases with downgrade, a single provider is downgraded (Observation 6). 13% of the downgrades induce an unnecessary increase in the technical lag (Observation 7). Nonetheless, downgrades of major versions also occur and they normally introduce a larger technical lag on the client package compared to minor and patch downgrades (Observation 8). An explanation is that client packages often delay the integration of major releases of a provider due to its inherent increased difficulty compared to minors and patches.

• *The speed with which a downgrade occurs is associated with how the provider was formerly updated*. Client releases published between the update and the following downgrade of a provider take 2.6 times longer than the typical time-between-releases of the same client package (Observation 9). More than half of the downgrades that follow an explicit update are performed almost 10 times faster than half of the downgrades following an implicit update (Observation 10). Also, we observed the occurrence of urgent downgrades, i.e., those that occur up to 24 hours after the prior downgrade. There are almost 9 times more urgent downgrades following an explicit update (36%) than urgent downgrades following an implicit update (3.8%) (Observation 11).

Our findings contribute to the advance of the research concerning dependency management in software ecosystems. In particular, it complements prior studies that relate downgrades to issues in the provider packages, but that did not describe what those issues were. Based on the identified causes for downgrades and on the understanding of how downgrades are commonly performed, we derived a set of lessons learned to help client packages mitigate the side-effects of downgrades. Lastly, we contribute to the field by listing future research opportunities.

References

[1] W. C. Lim, "Effects of reuse on quality, productivity, and economics," *IEEE Software*, vol. 11, no. 5, pp. 23–30, 1994.

[2] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? An empirical case study on npm," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, 2017, pp. 385–395.

[3] K. Manikas and K. M. Hansen, “Software ecosystems – a systematic literature review,” *Journal of Systems and Software*, vol. 86, no. 5, pp. 1294 – 1306, 2013.

[4] A. Serebrenik and T. Mens, “Challenges in software ecosystems research,” in *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ser. ECSAW ’15. New York, NY, USA: ACM, 2015, pp. 40:1–40:6.

[5] G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella, “The evolution of project inter-dependencies in a software ecosystem: The case of Apache,” in *IEEE International Conference on Software Maintenance, ICSM*, 2013, pp. 280–289.

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

[7] 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.

[8] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, “Keep me updated: An empirical study of third-party library updatability on Android,” in *Proceedings of the ACM Conference on Computer and Communications Security (CCS ’17)*, 2017, pp. 2187–2200.

[9] R. G. Kula, D. M. German, T. Ishio, A. Ouni, and K. Inoue, “An exploratory study on library aging by monitoring client usage in a software ecosystem,” in *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2017, pp. 407–411.

[10] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, “Do developers update their library dependencies?: An empirical study on the impact of security advisories on library migration,” *Empirical Software Engineering*, pp. 1–34, 2017.

[11] A. Ihara, D. Fujibayashi, H. Suwa, R. G. Kula, and K. Matsumoto, *Understanding when to adopt a library: A case study on ASF projects*. Cham: Springer International Publishing, 2017, pp. 128–138.

[12] P. Salza, F. Palomba, D. Di Nucci, C. D’Uva, A. De Lucia, and F. Ferrucci, “Do developers update third-party libraries in mobile apps?” *Proceedings of the 26th Conference on Program Comprehension - ICPC ’18*, pp. 255–265, 2018.

[13] S. Mirhosseini and C. Parnin, “Can automated pull requests encourage software developers to upgrade out-of-date dependencies?” in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 84–94.

[14] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, “Mining trends of library usage,” in *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, ser. IWPSE-Evol ’09, 2009, pp. 57–62.

[15] 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)*. Gothenburg, Sweden: ACM, New York, NY, USA, 2018, p. 11.

[16] C. Bogart, C. Kastner, J. Herbsleb, and F. Thung, “How to break an API: cost negotiation and community values in three software ecosystems,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, 2016, pp. 109–120.

[17] 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*, ser. ICSE ’16. New York, NY, USA: ACM, 2016, pp. 120–131.

[18] N. Bevacqua, “Keeping your npm dependencies immutable,” https://ponyfoo.com/articles/immutable-npm-dependencies, 2015, accessed: 2018-01-07.

[19] P. Draper, “Package management: Stop using version ranges,” https://www.lucidchart.com/techblog/2017/03/15/package-management-stop-using-version-ranges/, 2017, accessed: 2018-05-28.

[20] K. C. Dodds, “Why semver ranges are literally the worst?” https://blog.kentcdodds.com/why-semver-ranges-are-literally-the-worst-817cdcb09277, 2015, accessed: 2018-01-07.

[21] J. M. Gonzalez-Barahona, P. Sherwood, G. Robles, and D. Izquierdo, “Technical lag in software compilations: Measuring how outdated a software deployment is,” in *Open Source Systems: Towards Robust Practices*, F. Balaguer, R. Di Cosmo, A. Garrido, F. Kon, G. Robles, and S. Zacchiroli, Eds. Cham: Springer International Publishing, 2017, pp. 182–192.

[22] A. Decan, T. Mens, and E. Constantinou, “On the evolution of technical lag in the npm package dependency network,” in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 404–414.

[23] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, “How the Apache community upgrades dependencies: an evolutionary study,” *Empirical Software Engineering*, vol. 20, no. 5, pp. 1275–1317, 2015.

[24] J. Cox, E. Bouwers, M. C. J. D. van Eekelen, and J. Visser, “Measuring dependency freshness in software systems,” in *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, ser. ICSE’15, vol. 2, 2015, pp. 109–118.

[25] 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.

[26] D. F. Bauer, “Constructing confidence sets using rank statistics,” *Journal of the American Statistical Association*, vol. 67, no. 339, pp. 687–690, 1972.

[27] N. Cliff, *Ordinal methods for behavioral data analysis*. New-York, USA: Psychology Press, 1996.

[28] 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.

[29] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan, "High-impact defects: A study of breakage and surprise defects," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 300–310.

[30] A. Zerouali, V. Cosentino, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the impact of outdated and vulnerable javascript packages in docker images," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 619–623.

[31] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 559–563.

[32] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an api: Cost negotiation and community values in three software ecosystems," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, 2016, pp. 109–120.

[33] G. Mezzetti, A. Møller, and M. T. Torp, "Type regression testing to detect breaking changes in node.js libraries," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, 2018.

[34] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019.

[35] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *IEEE International Working Conference on Mining Software Repositories*, 2017, pp. 102–112.

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

[37] I. J. M. Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. E. Hassan, "Analyzing ad library updates in android apps," *IEEE Software*, vol. 33, no. 2, pp. 74–80, 2016.

[38] R. Robbes, M. Lungu, and D. Rothlisberger, "How do developers react to API deprecation? The case of Smalltalk ecosystem," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE'12, 2012, pp. 56:1–56:11.

[39] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the relation between outdated docker containers, severity vulnerabilities, and bugs," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 491–501.

[40] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the 2nd International Conference on Software Engineering*, ser. ICSE '76. IEEE Computer Society Press, 1976, pp. 492–497.

[41] R. G. Kula, D. M. German, T. Ishio, and K. Inoue, "Trusting a library: A study of the latency to adopt the latest Maven release," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*, 2015, pp. 520–524.

Filipe R. Cogo is a PhD student at the School of Computing of Queen's University in Canada, under supervision of Dr. Ahmed E. Hassan. His research at the Software Analysis and Intelligence Lab (SAIL) focuses on empirical studies about dependencies in software ecosystems. Filipe is also a professor at the Universidade Tecnológica Federal do Paraná (UTFPR), in Campo Mourão, Brazil. He received his BSc and MSc in Computer Science from the Universidade Estadual de Maringá (UEM), Brazil.

Gustavo A. Oliva is Post-Doctoral 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
Version range operators

Table 3 gives the definition and examples for the operators in the version range grammar used by npm. The grammar in Backus-Naur form can be found at https://www.npmjs.com/package/semver.

TABLE 3 Operators in the grammar of npm version range.

Operator(s)	Definition	Example
>, <, >=, <=	Allows, respectively, any version greater, smaller, greater or equal, or smaller or equal to a given semantic version number.	"P": ">1.0.0" is satisfied by any version of *P* greater than 1.0.0 (e.g., 1.0.2, 1.2.0, or 2.0.0).
∼ (tilde)	Allows changes to the least precedent (left-most) level of the semantic version number. Intuitively, the tilde operator resolves towards a patch update of the provider.	"P": "∼1.2.3" is satisfied by any version of *P* greater than or equal to 1.2.3 and less than 1.3.0. Still, "P": "∼1.2" is satisfied by any version of *P* greater than or equal to 1.2.0 and less than 1.3.0.
∧ (caret)	Allows changes that do not modify the non-zero least precedent level in a semantic version number. Intuitively, the caret operator resolves towards a minor update of the provider.	"P": "∧1.2.3" is satisfied by any version of *P* greater than or equal to 1.2.3 and less than 2.0.0. Still, "P": "∧0.2.3" is satisfied by any version of *P* greater than or equal to 0.2.3 and less than 0.3.0.
— (hyphen)	Allows an inclusive set of versions.	"P": "1.2.3 — 2.3.4" is satisfied by any version greater than or equal to 1.2.3 and less than or equal to 2.3.4.
omit a semantic version level or replace it by "x"	Allows changes in the omitted/replaced semantic version level.	Both "P": "1.x" or "P": "1" are satisfied by any version of *P* greater than or equal to 1.0.0 and less than 2.0.0. Still, both "P":"1.2" or "P":"1.2.x" are satisfied by any version greater than or equal to 1.2.0 and less than 1.3.0.
"*", "latest", "last", ""	Resolves to the largest version available.	Both "P": "*" or "P": "" are satisfied by the largest version of *P*.
||	Combines two or more versioning statements in a logic 'OR'.	"P": "∧2.0.0 || ∼3.0.0" indicates that any of the statements "∧2.0.0" or "∼3.0.0" are satisfied by any version of *P* in accordance with the statement definition.

Appendix B
Downgrade detection

In this appendix, we describe our approach for identifying downgrades on npm. We start by discussing the problems that occur when either the chronological or the numerical ordering of the client releases is used to detect downgrades. Subsequently, we describe our solution in the form of an algorithm that derives a branch-based ordering of client releases.

The problem with chronological versus numerical ordering of versions: A downgrade is defined as an event that occurs between two adjacent releases $\langle r_{i-1}, r_i \rangle$ of a client package C. When the resolved version of a provider package P in r_i is smaller than the resolved version of this same provider in r_{i-1}, we say that P was downgraded by C in r_i. Hence, the definition of a downgrade relies on the definition of the logical order of client releases. Analogously, the definition of an update also relies on such a definition.

Prior studies use either a chronological [11, 12] or a numerical ordering [7, 35] to recover the release history of packages. However, assuming those orderings leads to inconsistencies in how the resolved provider version changes from one client release to another. In the remainder of this section, we show that none of these orderings are suitable to detect updates and downgrades in the resolved provider packages.

When analyzing the order of package releases in npm, we observed that several releases can be actually maintained in parallel. Releases are developed in parallel because, even with the existence of releases with a higher numerical order, a release with a lower numerical order might need to be patched. For example, in Figure 9, even though the release 2.0.0 was already available, the release 1.1.2 had to be published in order to patch the release 1.1.1. Hence, the version 1.1.1 is considered adjacent to both versions 1.1.2 and 2.0.0. Because the numerical and chronological ordering are linear, they are not suitable to represent the parallel releases of npm.

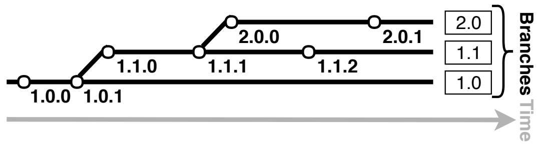


Fig. 9 Development of parallel versions in npm.

Applying the chronological and numerical orderings to the releases that are shown in Figure 9 would yield the following results ($\prec$ denotes a precedence relation):

Chronological:
$1.0.0 \prec 1.0.1 \prec 1.1.0 \prec 1.1.1 \prec 2.0.0 \prec 1.1.2 \prec 2.0.1$

Numerical:
$1.0.0 \prec 1.0.1 \prec 1.1.0 \prec 1.1.1 \prec 1.1.2 \prec 2.0.0 \prec 2.0.1$

Branch-based:
$1.0.0 \prec 1.0.1 \prec 1.1.0 \prec 1.1.1 \prec 1.1.2$
$\succ 2.0.0 \prec 2.0.1$

For a client package that releases according to Figure 9, analyzing the changes in the resolved provider version from one client release to another would produce different results depending on the assumed ordering. Figure 10 illustrates the inconsistencies that arise when assuming either the chronological or the numerical orderings to detect downgrades or updates. The timeline at the top of the Figure depicts a sequence of releases from the provider and the client packages. For the client package releases, the Figure also shows the used versioning statement and the resolved provider version.

When the *chronological order* of the client releases is assumed (see Figure 10), an incorrect downgrade from version 2.0.0 to 1.1.2 is detected due to a *version inconsistency*. Logically, version 1.1.2 does not succeed version 2.0.0, since

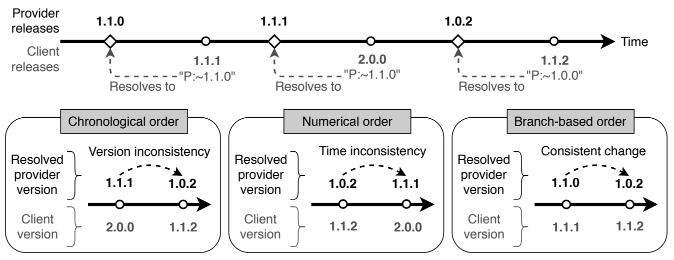
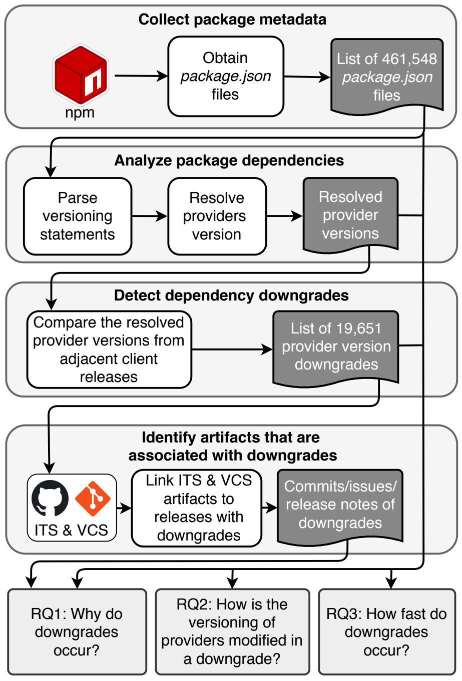


Fig. 10 Numerical, chronological, and branch-based ordering to evaluate the provider version changes over the client releases.

these versions belong to different branches. Similarly, when the *numerical order* of the client releases is assumed (see Figure 10), an incorrect update from version 1.1.2 to 2.0.0 is detected due to a *time inconsistency*. The provider version resolved at the time that the client version 1.1.2 was released (i.e., provider version 1.0.2) did not exist at the time of client version 2.0.0. Hence, this update is invalid. However, when the *branch-based order* of the client releases is assumed, the changes in the resolved provider version from one client release to another are consistent regarding both version and time.

An algorithm for branch-based ordering: The collected data from npm records only the chronological and numerical orderings of the package releases. Therefore, we conceived an algorithm to derive the branch-based ordering from these two orderings, which works as follows: as the client releases are examined in a chronological order, we check if any of the previously visited releases are in the same branch as the current one. If so, then the release with the largest version number in the branch of the current release is deemed the predecessor of the current release. Otherwise, the release visited so far with the largest version number is deemed the predecessor of the current release. If the releases in chronological and numerical order, shown in Figure 9, are given as input to our algorithm, then the branch-based ordering of the releases (as shown in the same figure) is returned as output.

Algorithm 1 gives the pseudo-code for the algorithm that we conceived to recover the branch-based ordering. The parameters for the procedure BRANCH-RELEASE-ORDERING are R_{T_C}, the list of all releases of a client package C in chronological order, and R_{N_C}, the list of all releases of a client package C in numerical order. This procedure manages sets R and U. The set R stores the pairs of adjacent releases $\langle r_{i-1}, r_i \rangle$ that are identified. In turn, U stores the visited releases from R_{T_C}. The procedure BRANCH(r_i) returns the set of releases that were added to the same branch as that of release r_i. If there are no releases that are added to a given branch, this procedure returns $\varnothing$ (the empty set). In turn, the procedure UPDATE-BRANCH(r_i, R_b) adds a release r_i to its respective branch (R_b). The procedure LARGEST-VERSION-SMALLER-THAN takes a release r_i and a set L of release versions and returns the largest release version $r_j \in L$ that is smaller than r_i. In case there is no version smaller than r_i in L, the procedure returns a null reference. A given release r_i with a null reference as its predecessor means that r_i has no release preceding it. The procedure LARGEST-VERSION-SMALLER-THAN obtains the



Collect package metadata
npm
Obtain package.json files
List of 461,548 package.json files
Analyze package dependencies
Parse versioning statements
Resolve providers version
Resolved provider versions
Detect dependency downgrades
Compare the resolved provider versions from adjacent client releases
List of 19,651 provider version downgrades
Identify artifacts that are associated with downgrades
ITS & VCS
Link ITS & VCS artifacts to releases with downgrades
Commits/issues/ release notes of downgrades
RQ1: Why do downgrades occur?
RQ2: How is the versioning of providers modified in a downgrade?
RQ3: How fast do downgrades occur?

Fig. 11 Overview of our data collection approach.

precedence between the version number of the releases from the numerical order (R_{N_C}).

Algorithm 1 Sort releases according to a branch-based ordering

```
Input: Releases in chronological ($R_{T_C}$) and numerical ($R_{N_C}$) orders
Output: Releases in branch-based order
  procedure BRANCH-RELEASE-ORDERING($R_{T_C}$, $R_{N_C}$)
      $R \leftarrow \varnothing$
      $U \leftarrow \varnothing$
      for all $r_i \in R_{T_C}$ do
          APPEND($U, r_i$)
          $R_b \leftarrow$ BRANCH($r_i$)
          if $R_b \neq$ then
              $r_{i-1} \leftarrow$ LARGEST-VERSION-SMALLER-THAN($r_i, R_b, R_{N_C}$)
          else
              $r_{i-1} \leftarrow$ LARGEST-VERSION-SMALLER-THAN($r_i, U, R_{N_C}$)
          APPEND($R, \langle r_{i-1}, r_i \rangle$)      ▷ Store $r_{i-1}$ as the predecessor of $r_i$
          UPDATE-BRANCH($r_i$, $R_b$)
      return $R$
```

Algorithm 1 Branch-based release ordering algorithm.

APPENDIX C DATA COLLECTION

Four main steps were performed in our data collection: collection of package metadata, analysis of package dependencies, detection of dependency downgrades, and identification of artifacts associated with downgrades. Figure 11 depicts an overview of our data collection approach.

Collect package metadata: ***Obtain package.json files*** – We crawled the registry[16] of npm and obtained the *package.json* metadata file of 461,548 packages. The metadata file of each package lists, among other pieces of information, all the

16. https://github.com/npm/registry-follower-tutorial

published releases by a client package, the name of the used providers in each release, the associated versioning statements with the providers in each client release, and the timestamp of each release. An example of a *package.json* file can be seen on Appendix D. We keep in our database only package releases containing at least one provider. Our data collection encompassed the period of December 20, 2010 to July 01, 2017.

Analyze package dependencies: ***Parse versioning statements*** – We parsed the versioning statement of all dependencies in the *package.json* files according to the adopted grammar by npm (c.f. Section 2). Two types of dependencies were considered: (i) dependencies that are required for the installation of a package and that are loaded at runtime and (ii) dependencies that are not required for the installation of a package and that are loaded only at development time. These two types of dependencies are listed separately in the package.json file (as "dependencies" and "devDependencies" respectively).

Resolve providers version – The *package.json* file contains the date on which the releases of a package were published on npm. Given a client package release and the name of a provider that is used in such a release, we initially obtained the list of published versions by the provider before the client release date. Subsequently, we determined the resolved version of each provider according to the versioning statement used by the client in that release (c.f. Section 2). Dependencies with a versioning statement that did not satisfy any provider version were discarded. The output of this step is the resolved provider version for all client dependencies to providers.

Detect dependency downgrades: ***Compare the resolved provider versions from adjacent client releases*** – We sorted the releases of the client packages according to the branch-based ordering algorithm that we discussed in Section 3 (c.f. Algorithm 1). Afterwards, we detected downgrades by comparing the resolved provider versions from adjacent client releases. The output of this step is a list of 19,651 downgrades, which were used as input to our RQs.

Identify artifacts that are associated with downgrades: ***Link ITS & VCS artifacts to releases with downgrades*** – From the list of downgrades, it is possible to identify the releases of a client package in which at least one downgrade occurred. In particular, for the packages whose Version Control System (VCS) and Issue Tracker System (ITS) were publicly available, one can also identify which artifacts (e.g., committed files, commit messages) were produced during the releases with downgrades. We identified and examined the artifacts of a statistically-representative sample of client releases with downgrades (more details in Section 5.1). The output of this process is thus a list of commits, issues, and release notes that are associated with a sample of the releases that contain at least one downgrade of a provider.

APPENDIX D
Package.json FILE EXAMPLE

Listing 1 shows an excerpt of a hypothetical *package.json* file. The client package named *client_package* released two versions: 1.0.0 and 1.0.1. In the former release, the client package assigned the versioning statements "`2.0.0`" and "`<1.2.3`" to the providers named *provider_1* and *provider_2*, respectively. In the latter release, the client package assigned the versioning statements "`2.0.1`" and "`<1.3.0`" to the providers named *provider_1* and *provider_2*, respectively. The bottom of the *package.json* file shows the timestamp of each client release.

```
"name": "client_package",
"versions": {
    "1.0.0": {
        "dependencies": {
            "provider_1": "2.0.0",
            "provider_2": "<1.2.3"
        }
    },
    "1.0.1": {
        "dependencies": {
            "provider_1": "2.0.1",
            "provider_2": "<1.3.0"
        }
    }
},
"time": {
  "1.0.0": "2016-11-24T00:48:15",
  "1.0.1": "2017-02-08T13:26:38"
}
```

Listing 1 Excerpt of a *package.json* file