

A Comprehensive Study of the Lifecycle of Dormant npm Packages

Ahmed Zerouali  · Valeria Pontillo  ·
Coen De Roover 

Received: date / Accepted: date

Abstract Open-source software, with its extensive array of reusable components, is the foundation of modern software development. Package managers like npm simplify the integration of numerous modules and dependencies. While previous research has examined dependency management, security vulnerabilities, and package updates, little is known about the effects of extended inactivity on these packages. This study aims to investigate the phenomenon of “*dormant*” packages and explore the implications of this dormancy on software projects and the broader open-source community. We propose a two-step investigation of the dormant packages in npm. First, we quantitatively investigate the duration and timing of release gaps, the nature of updates following dormancy, and the state of dependencies during these periods of inactivity. Second, we qualitatively examine the activities of maintainers and users through an analysis of commit messages, pull requests, and issues. The key findings of our study reveal significant risks associated with dormant packages, including outdated dependencies, unpatched security vulnerabilities, and potential erosion of community trust. While some packages are eventually updated to address these issues, others remain neglected, posing challenges in using open-source software. Our study highlights the potential benefits of proactive maintenance and suggests that community involvement may play a role in mitigating the risks associated with dormant packages.

Ahmed Zerouali
Vrije Universiteit Brussel, Brussels, Belgium
E-mail: ahmed.zerouali@vub.be

Valeria Pontillo
Vrije Universiteit Brussel, Brussels, Belgium
Gran Sasso Science Institute (GSSI), L'Aquila, Italy
E-mail: valeria.pontillo@gssi.it

Coen De Roover
Vrije Universiteit Brussel, Brussels, Belgium
E-mail: coen.de.roover@vub.be

Keywords npm, Dormant Packages, Vulnerabilities, Updates.

1 Introduction

The use of open-source software has become essential for modern development, offering a wealth of reusable components that accelerate innovation and reduce development costs [12]. Among these, package managers like npm play a pivotal role, providing developers with easy access to a vast library of modules and dependencies [26]. However, the health and maintenance of these packages are critical to ensuring the stability and security of the software that relies on them [13, 17, 25, 34]. Security issues are even more severe due to the phenomenon of “*dormant*” packages—those that experience prolonged gaps in releasing new versions from their maintainers. Dormant packages can pose significant risks, including outdated dependencies, unpatched security vulnerabilities, and the erosion of community trust [20]. Despite these risks, many dormant packages continue to be widely used, raising important questions about the consequences on software development practices and the overall resilience of open-source projects. This study aims to investigate this phenomenon within npm and examines the implications of this dormancy on software projects and the broader open-source community. Through a comprehensive analysis, we aim to explore the prevalence of these packages in npm, the versions released after the gap, and the patterns of their revival.

The study is organized into two parts: A quantitative analysis that is structured around several research questions aimed at uncovering the characteristics of dormant packages, the status of dependencies during the gap, and the static vulnerabilities that affect packages’ dependencies; and a qualitative analysis that manually examines commit messages, pull requests, and issues to explore the experiences of maintainers and users with dormant packages. Both analyses shed light on the lifecycle of these packages, the factors that lead to their dormancy, and the circumstances under which they are revived. More specifically, the quantitative analysis addresses the following research questions:

- **RQ_1 : How long are the release gaps in software packages?** This question investigates the duration of the periods in which packages do not release any new versions. Understanding the length of these gaps helps to identify how long packages remain dormant and the potential risks that accumulate during these periods.
- **RQ_2 : At what point in packages’ lifecycle do release gaps most commonly occur, and how do these attributes relate to the duration of dormancy?** This question investigates when release gaps arise and whether certain package attributes are linked to longer dormancy periods. By identifying patterns in the timing and duration of gaps, we aim to uncover early indicators of dormancy.
- **RQ_3 : What is the type of the first version released after the gap?** This question examines the nature of the updates made when a package resumes activity after a period of dormancy. Understanding the first changes

can reveal how maintainers address the issues that may have accumulated during the gap.

- **RQ₄: What files were modified between the versions before and after the gap?** This question explores the specific changes made to the package’s codebase when it is revived. By analyzing which files are modified, we can identify the focus areas of the updates, such as bug fixes, security patches, or new features.
- **RQ₅: Do packages modify their dependencies upon resuming activity?** This question investigates whether packages change, remove, or add their dependencies when they become active again. Analyzing these modifications can reveal how maintainers address the package’s external dependencies and whether they prioritize updating or altering these connections when reviving the package.
- **RQ₆: How outdated were the dependencies during the gap?** This question examines the extent to which the dependencies of dormant npm packages become outdated during the period of inactivity. Understanding the level of obsolescence helps in assessing the potential risks posed by these dependencies.
- **RQ₇: How do security vulnerabilities affect a package’s dependencies before, during, and after a gap?** This question assesses the security of package dependencies throughout the lifecycle of a dormant period. By examining the security status before, during, and after the gap, the study aims to understand the potential risks that may have been introduced or mitigated during these periods of inactivity.
- **RQ₈: To what extent do dependents continue using dormant packages during and after the gap?:** This question investigates the behavioral patterns of dependent packages when their dependencies become dormant, examining whether and when they abandon these relationships during or after release gaps. Understanding these dependency abandonment and loyalty patterns reveals package manager resilience and provides insights into how developers respond to maintenance uncertainty in their dependency chains.

By understanding the dynamics of dormant npm packages, this research provides valuable insights into challenges faced in open-source development and point toward the potential benefits of proactive maintenance and stronger community engagement for improving package reliability.

The results of this study reveal some patterns in how dormant packages are managed and revived. While some packages are eventually updated to address security vulnerabilities or adapt to evolving software environments, others remain neglected, with maintainers offering little or no explanation for their prolonged inactivity. Our findings underscore the need for further research to understand the factors that contribute to dormancy and the strategies that can be employed to mitigate its effects.

Structure of the paper. Section 2 overviews the background and the related work, positioning our work within the current body of knowledge. Sec-

tion 3 elaborates on the research method employed to create our dataset. In Section 4, we analyze the results achieved from the quantitative analysis, while Section 5 reports the results of the qualitative analysis. Section 6 further discusses the main findings of our work, emphasizing the implications for research and practice. The potential limitations of our study are discussed in Section 7. Finally, Section 8 concludes the paper and outlines our future research agenda.

2 Background and Related Work

This section describes the background and the related work we build upon.

2.1 Terminology

This section introduces the terminology used throughout this article. All main terms are highlighted in **boldface**.

In the realm of software development, **software packages** serve as essential building blocks, encapsulating specific functionalities or tasks that can be easily reused across various projects. These packages are typically open source and are distributed through **package distributions** such as npm or Maven. These distributions act as centralized repositories where developers can find, manage, and integrate a wide array of packages into their software projects. Each package can have different versions, with each version being referred to as a **package release**. These releases are denoted by unique version numbers that follow a sequential order, often adhering to **semantic versioning**¹ to communicate the nature and significance of the changes made.

A key aspect of software packages is their **dependencies**, which define the relationships between different packages. When a package relies on another to function properly, it establishes a dependency relationship. These dependencies are crucial for maintaining the integrity and compatibility of software, as they specify the range of acceptable versions that can be installed alongside a given package release. However, when no new versions of a package are released for an extended period, a **release gap** occurs. This gap might not necessarily indicate a lack of activity in the package’s development but rather a pause in the release cycle. If such a gap is intentional or goes unaddressed, it can lead to a **release pause**, as shown in Figure 1, which may cause concerns about the maintenance and future viability of the package. Packages that exhibit one or more of these release gaps are called **dormant package**.

During these periods of inactivity, security risks can emerge, particularly if the package or its dependencies contain known **vulnerabilities**. A vulnerability is a reported security threat that affects certain versions of a package, potentially exposing any dependent software to exploitation. These **vulnerable dependencies** can have far-reaching consequences, affecting not just the directly dependent packages but also any software that indirectly relies on

¹ <https://semver.org/>

them. Addressing these vulnerabilities requires careful monitoring of package releases and dependencies to ensure that updates are applied promptly and that the overall security of the software is maintained.

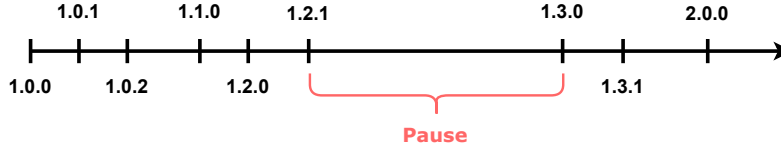


Fig. 1 An example of a package release timeline showing a release pause between versions 1.2.1 and 1.3.0.

2.2 Related work

2.2.1 On package distributions

Software package distributions have been the focus of numerous research studies across software development. Wittern et al. [26] conducted a comprehensive analysis of the npm packages, exploring aspects such as package descriptions, interdependencies, download metrics, and the usage of these packages in publicly accessible repositories. Their study revealed that the number of npm packages and their updates is growing at a superlinear rate. Additionally, they found that packages are becoming increasingly interconnected through dependencies, with over 80% of npm packages having at least one direct dependency.

Kikas et al. [14] analyzed the structure and evolution of the dependency networks in the JavaScript, Ruby, and Rust package distributions. They discovered that the number of transitive dependencies increased by 60% in 2016. Additionally, they highlighted the growing negative impact of removing a popular package, as evidenced by incidents like the left-pad removal.

Bogart et al. [4] carried out case studies on three distinct package managers – Eclipse, CRAN, and npm – each with its tooling and philosophy toward change. Their findings revealed substantial differences in practices, policies, and community values, offering insights into how developers make decisions about change and its associated costs. In a similar vein, Decan et al. [10] conducted an empirical comparison of dependency issues across the npm, CRAN, and RubyGems package distributions. In a subsequent study [9], this comparison was extended to include four additional distributions: CPAN, Packagist, Cargo, and NuGet. Their findings revealed notable differences among these package managers, which they attributed to variations in characteristics and maturity levels across distributions.

2.2.2 On package outdatedness and security

Many researchers have studied the impact of outdated and vulnerable dependencies. Kula et al. [15] conducted an empirical analysis of thousands of Java libraries distributed on Maven to investigate their latency in adopting the latest versions of dependencies. Their findings revealed that maintainers are initially hesitant to upgrade to the latest library versions at the start of a project. However, they observed that maintainers are more likely to use the latest version when introducing a new dependency. In a subsequent study, Kula et al. [16] examined library migration across over 4,600 GitHub software projects and 2,700 library dependencies. They observed that four out of five projects had outdated dependencies. A survey of project maintainers revealed that a significant majority were unaware of these outdated dependencies.

Zerouali et al. [28, 31] introduced the notion of technical lag to quantify the degree of outdatedness of packages and dependencies along different dimensions: time, version, and vulnerability lags. Their findings on different package distributions highlighted a significant prevalence of outdated package dependencies and technical lag, reflecting a tendency to avoid updates due to concerns about backward incompatibility. Later, the same authors carried out a study on security vulnerabilities affecting npm and RubyGems packages and their dependencies. They found that vulnerabilities in npm are both increasing and being disclosed more rapidly than in RubyGems. On average, npm vulnerabilities affect 30 releases, compared to 59 for RubyGems [30].

Cox et al. [8] examined 75 software systems and introduced various metrics to assess their usage of recent dependency versions. They discovered that systems with outdated dependencies were four times more likely to encounter security issues compared to those that were current.

Cogo et al. [6] investigated the impact of package downgrades in npm on technical lag. They found that one-fifth of all downgraded packages contributed to an increase in technical lag for client packages. Notably, downgrades involving major versions were found to introduce more technical lag compared to downgrades of minor and patch versions, often reverting to a version earlier than the most recent functional version. This resulted in an avoidable increase in technical lag for 13% of the downgrades.

Zhong et al. [33] analyzed the deprecation level of packages in Python, showing the consequences of making deprecation declarations. Additionally, they have investigated the challenges that package developers and users face, as well as their expectations for the future deprecation pattern. Finally, they provided guidelines for developing package-level deprecation mechanisms.

2.2.3 On package activity and abandonment

Other researchers focused on package abandonment, deprecation, and activity decline. Avelino et al. [2] investigated the phenomenon of project abandonment in open-source software, focusing on how often projects are abandoned or survive and the factors influencing these outcomes. Analyzing 1,932 popular

GitHub projects, they found that 16% were abandoned, but 41% of these survived due to new core developers taking over. Their survey revealed that new maintainers were often motivated by their own use of the software, while social factors and access issues were significant challenges.

Cogo et al. [5] examined the use of the deprecation mechanism in npm, focusing on how often package releases are deprecated and the implications for client packages. They found that 3.7% of npm packages have at least one deprecated release, with 66% of these packages deprecating all their releases, leaving no replacement options. Additionally, 31% of partially deprecated packages lacked replacement releases. The study identified five main reasons for deprecation: withdrawal, supersession, defect, test, and incompatibility. They also discovered that 27% of client packages directly adopt deprecated releases, and 54% do so transitively.

Mujahid et al. [21] investigated the automatic identification of declining packages in distributions and suggested alternatives for developers to migrate their dependencies. Their method, based on observed dependency migration patterns, was evaluated on npm, showing 96% of accuracy.

Miller et al. [19] conducted a large-scale analysis of widely used npm packages to investigate the prevalence and impact of package abandonment. They found that abandonment is common, leaving many projects exposed and often unresponsive. In addition, the authors highlighted that proactive responses to abandonment are linked to better dependency management practices, and practitioners remove abandoned packages more quickly when their end-of-life status is clearly communicated.

2.3 Novelty of our Contribution

Our work builds on top of the aforementioned research and contributes to the broader software engineering community by raising awareness of the prevalence of dormant npm packages. While our analysis is focused on npm, the findings and research method can inform the practices of other package managers as well. For instance, community managers can use our dormancy indicators to more effectively monitor package activity and take action to prevent long-term inactivity. Similarly, library maintainers and users can use our findings to assess the risks associated with depending on dormant packages. Our study thus provides a foundation for further research and development of tools aimed at improving ecosystem health and enhancing visibility in maintenance.

More specifically, our research contributes to the study of package dormancy by addressing several unexplored dimensions within npm. Through an extensive analysis of release gaps, our work provides a comprehensive understanding of the duration and timing of dormancy periods in software packages (**RQ₁**, **RQ₂**). Then, we analyze the types of updates released after periods of inactivity, offering insights into how maintainers prioritize changes (**RQ₃**). We also investigate specific file changes and the extent of dependency modifi-

cations upon a package’s return to activity, revealing hints into how technical debt and security issues are managed post-dormancy (**RQ**₄, **RQ**₅).

An important aspect of our study is the investigation of dependency, outdatedness, and security risks throughout the dormant periods. By examining how dependencies age and become insecure before, during, and after these gaps (**RQ**₆, **RQ**₇), our work highlights the potential vulnerabilities that can accumulate in dormant packages. Finally, we investigated the effect of dormant npm packages on the dependent packages in order to extract possible behavioral patterns (**RQ**₈). This analysis is complemented by a qualitative exploration of the experiences and decision-making processes of maintainers during these dormant periods, offering valuable insights into the social factors that influence package maintenance and revival.

3 Dataset and Research Method

This paper studies reusable software packages that have been distributed via package managers and that have stopped releasing updates for a long period of time² before resuming. We decided to focus on the well-established and mature package manager npm that represents the most used package manager of JavaScript libraries with a large and active developer community [7]. We focused on npm and JavaScript for two main reasons. First, numerous researchers in the empirical software engineering field have used npm as a case study [1, 3, 6, 9]. Then, JavaScript represents one of the most popular programming languages on GitHub in the last decade.³

3.1 Data Selection

To study npm, we relied on a PostgreSQL data dump from Ecosyste.ms.⁴ This is an open API service that provides metadata about packages from many open-source software and registries. For our empirical study, we used the data dump version that was released on 2023-10-22,⁵ which is available as open access under the CC BY-SA 4.0 license. We only extracted npm data from this dump, creating a new dataset comprising 3.5M npm packages, 38.9M releases, and 587.3M direct (runtime, development, or optional) dependencies with unresolved dependency requirements [9].

For vulnerability analysis, we used the GitHub Advisory Database.⁶ We retrieved all vulnerabilities reviewed by GitHub and impacting npm packages. The vulnerability extraction was done on 2023-10-22. Our dataset comprises

² This period will be defined later in Section 3.2.

³ <https://github.blog/2023-11-08-the-state-of-open-source-and-ai/>

⁴ <https://ecosyste.ms/>

⁵ <https://packages.ecosyste.ms/open-data>

⁶ <https://github.com/advisories>

3,774 vulnerabilities with different degrees of severity. Given the different information in vulnerability reports, we only extracted data that we required for our study, including the affected package's name, vulnerability ID, severity, affected version range, CVE and CWE identifiers, and publication date.

3.2 Inclusion Criteria

After extracting npm data from Ecosyste.ms, the next step was the identification of packages that had a release gap in their process. Our strategy was performed in three different steps.

We started by identifying all versions of all npm packages and computing the time between each two consecutive version releases of the same package. In our study, we aim not only to identify packages that experienced a gap in their release schedule but also to explore how this gap impacted their dependents. For instance, a recently created package with minimal adoption will likely have less influence if it stops its releases. Conversely, a package with a longer history and widespread usage is likely to have a more significant impact. For this reason, we established several assumptions. First, we focus on packages with at least two years of consistent updates since their initial release, indicating they have gained trust and adoption over time. Additionally, we focus on packages with at least three versions, indicating a pattern of at least two regular pauses between releases. To identify the release gap, we compare the time between each pair of consecutive releases with the average pause time between previous releases. If the current pause exceeds this average by at least one year, we classify the package as having experienced a release gap.

The choice of the threshold was driven by two different works. Avelino et al. [2] explicitly performed a sensitivity analysis across five different thresholds (3 months, 6 months, 1 year, 1.5 years, and 2 years) to assess how threshold selection affects the classification of developers as abandoners. Their results show that the one-year threshold offers the best trade-off between precision and error sensitivity, achieving the highest harmonic mean (66%) of those two measures. Based on these findings, the authors concluded that the one-year threshold was the least sensitive to error and therefore the most appropriate for identifying project abandonment. We aligned our study with this empirically validated choice to ensure methodological consistency and comparability.

To further justify this threshold, we focused on one additional study. The work presented by English and Schweik [11] examined the institutional designs of FLOSS commons, arguing that projects must clear two minimal thresholds, sustained developer contribution activity and a baseline level of user adoption, to avoid collective-action failure. In their framework, the authors operationalize project abandonment as having no (or very few) code commits, forum posts, or mailing-list messages for a full year, using this one-year inactivity window as the critical threshold for diagnosing a non-active project.

Following the research method proposed by Avelino et al. [2] to identify all packages that experienced a release gap, we filtered out those that were

not used by any other package. This step allows us to focus on the impact of releasing gaps in packages that are more interconnected and potentially more consequential for the broader software.

Finally, we identified 11,970 packages that experienced at least one release gap, accounting for 0.34% of all npm packages. These packages collectively released 718,507 versions between December 18, 2010, and October 22, 2023.

Considering all versions, we found that 1,280,518 packages have required these dormant packages in at least one version and collectively have used 25,534 packages as direct dependencies. The dormant maintained packages have been predominantly used as development dependencies in 56.6% of cases and as runtime dependencies in 43.3%.

For all these packages, we analyze their dependencies and dependents by tracking them at various release dates. Since packages specify dependencies using version constraints, which allow for a range of dependency versions instead of one strict version, resolving these constraints can lead to different package versions being installed depending on the installation time [9]. To ensure accurate resolution of dependency constraints at specific time points, we only consider package releases available at that time. Then, using the dependency constraint resolver proposed by Decan and Mens [9], we determine the appropriate package version that will be installed for each direct dependency according to its version constraints.

4 Quantitative Analysis

Using the datasets prepared in Section 3, this section answers the research questions introduced in the introduction.

RQ₁: How long are the release gaps in software packages?

With this research question, we measure the duration of release gaps in software packages. By understanding the length of these gaps, we can assess their significance and impact on software maintenance and development processes.

Figure 2 illustrates the distribution of the number of days between two consecutive versions, grouped by versions of a release gap, versions before the gap, and all versions of all 11,970 packages that experienced a release gap. We observe that the distribution of the time between two versions before the gap is heavily skewed. The median time between consecutive versions before the gap is 3 days, while the average is 25.5 days, with 36% of the pairs of consecutive versions being released on the same day. The fact that there is a short median time between consecutive versions before a gap suggests that these packages were actively maintained and updated frequently. However, once a gap occurs, the duration varies significantly. The minimum gap duration observed is 12.3 months, while the maximum is 7.1 years. The median release gap for dormant packages is 18.7 months. This is critical since users of these packages rely on

regular updates to ensure the software’s functionality, security, and compatibility with other components in their projects. On the one hand, the absence of new updates for a long time could indicate that new bugs or vulnerabilities are present, but are not being addressed, thereby jeopardizing the project. On the other hand, the wide range of gap durations highlights the unpredictability and potential challenges in maintaining consistent release schedules.

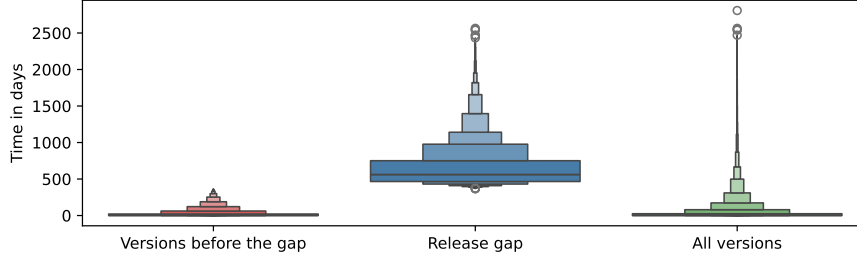


Fig. 2 Distribution of the number of days between two consecutive versions for the packages with a releasing gap.

We also observe that the distribution for all versions is skewed. This might be impacted by the versions before the gap. In that case, we would expect packages to have fewer versions after the gap. We will verify this hypothesis in the next research question.

► **Answer RQ₁.** *The median time between consecutive versions before a gap is 3 days, indicating active maintenance, while the median release gap itself is 18.7 months. Gaps vary widely, ranging from 12.3 months to 7.1 years. The unpredictability of these gaps can lead to potential security risks and maintenance challenges, as long periods without updates may leave packages vulnerable. This highlights the critical need for consistent update schedules to ensure the reliability and security of software dependencies.*

RQ₂: At what point in packages’ lifecycle do release gaps most commonly occur, and how do these attributes relate to the duration of dormancy?

With this research question, we aim to shed light on when gap events tend to occur and whether certain characteristics, such as package maturity, activity level, or popularity, are associated with longer or shorter gaps. By identifying temporal patterns and package attributes that correlate with the start and duration of release gaps, we seek to uncover signals that could help developers, users, and platform maintainers anticipate future release gaps.

First, we quantify the number of regular versions that packages released before experiencing a release gap: Table 1 shows the number of versions that

packages had before the gap. We notice that, on average, packages released 44.9 versions before experiencing a release gap, while the median number of versions before the gap is 28, reflecting a sustained period of active maintenance before experiencing release pauses.

We have also looked at the age of the packages when they paused releasing versions—Figure 3 shows the age of packages before the gap. On average, packages were 3 years and one month old at the release date of the last version before the gap. The median age of these packages before the gap is 2 years and 9 months. This confirms our previous observation that packages had enough time to gain user trust before the occurrence of the gap.

Table 1 Distribution of the number of versions before and after the release gap.

	Min	Mean	Median	Max	Std
Before the gap	4	44.9	28	3,907	75.2
After the gap	1	38.5	4	2,123	32.3
All	5	51.8	33	5,000	88

Looking at the number of versions released after a gap, we found that 4,297 (30.5%) of the packages that had a release gap created just one version after the gap. This means that around one in three packages with a release gap resumed releases but only issued a single version. Therefore, we filtered out this subset of packages and focused solely on those with at least two versions after the gap (69.5%). Their median number of versions after the gap is 4, while the average is 38.5. This big difference between the average and the median is due to some packages that released thousands of versions after their release gap (i.e., the maximum is 2,123 versions). For instance, we found that 80% of these packages (which had more than one version after the gap) released less than 10 versions, confirming our assumption in the previous question.

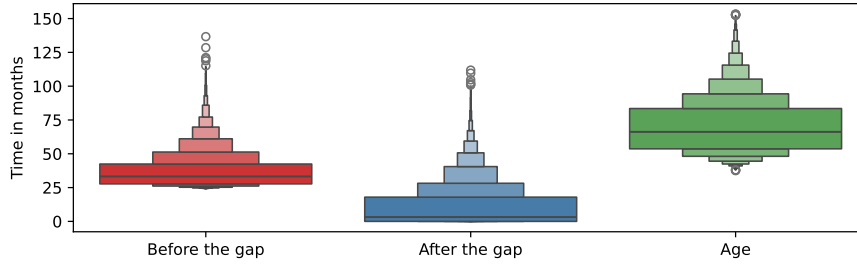


Fig. 3 Distribution of the time before and after the release gap.

In terms of time between the first and the latest version after the gap, Figure 3 shows our results. We observe that this distribution is skewed towards shorter periods, particularly near 0. On average, this subset of packages

continued releasing updates after a gap of approximately 466 days. However, since some packages may still be in the process of preparing new versions, this last information does not give us much insight. Therefore, it is more important to look at the time between two consecutive versions after the gap. We found that the median is only 24 days to release the second version after the gap. Considering all versions after the gap, the median is 3 days to release a new version. Since 50% of the packages had less than 2 versions after the gap, we hypothesize that many of the packages release few versions after the gap and then stop releasing again. This led us to investigate whether the gap happens again. We found that 2,197 of the packages (18.3%) have another release gap. In addition, 5,510 (46%) of the packages have not been updated for more than one year, and 803 of them have more than one release gap.

We also carried out correlation tests for three different aspects. First, we looked at the relationship between the package age (in terms of the number of versions) when the gap starts and the duration of the gap. Then, we analyzed the length of the release gap and the number of versions after it to see whether packages with a longer release gap have fewer or more versions after it. Finally, we performed two other correlation analyses between the number of downloads and gap duration, and the length of the gap and the time between consecutive versions after the gap. In each case, we applied both Pearson and Spearman correlation tests [23, 24]: the former is ideal for uncovering a linear relation. The latter is more tolerant of skewed data and will reveal any consistent order. The results show that there is no correlation between the packages' age and the gap duration (Pearson coefficient = -0.068 and Spearman coefficient = -0.061). Even for the length of the release gap and the number of versions after it, we did not find any significant correlation (Pearson coefficient = -0.05 and Spearman coefficient = -0.17). Similarly, we did not find a correlation between the number of downloads of a package and the gap duration (Pearson coefficient = -0.007 and Spearman coefficient = 0.005). Finally, we found a very weak correlation when testing the relation between the length of the gap and the median time between consecutive versions after the gap (Pearson coefficient = 0.15 and Spearman coefficient = 0.09). The statistical results reinforce the idea that dormancy duration is driven by factors beyond simple package profile metrics.

► **Answer RQ₂.** *Before experiencing a release gap, the median number of versions released was 28, suggesting packages were actively maintained and trusted by users. The median age of packages before the gap is around 3 years. After a gap, 30.5% of packages released only one version, while the rest had a median of 4 versions. Short intervals were observed between post-gap releases, with a median of 24 days for the second version. Notably, 18.3% of packages experienced a second gap, and 46% remained inactive for over a year. Finally, looking at the correlation analyses, there was no significant correlation between the package age when the gap starts and the duration of the gap, the length of the gap and the number of versions released after it, nor the number of downloads and gap duration*

We observed a weak correlation for the time between consecutive versions following the gap.

RQ₃: What is the type of the first version released after the gap?

This research question aims to dissect the first version released following a release gap. By examining these versions, we can gain insights into the development team’s priorities and focus areas upon resumption. With this analysis, we can reveal whether the post-gap versions primarily address technical debt accumulated during the hiatus, introduce significant new features, fix reported bugs, or just make minor edits.

Thus, the first step is to classify the type of the first release based on version numbers using as heuristic the semantic versioning scheme.⁷ We focused on **SemVer** because, as previously demonstrated, npm relies on this scheme to maintain a healthy environment, where bug-fixes are reliably delivered to downstream packages as quickly as possible, while breaking changes require manual intervention by downstream package maintainers [22]. By examining whether the first version after the gap is a major, minor, or patch release, we can infer the scale and impact of the changes:

- Major: Significant changes that may include new features, architectural changes, or backward-incompatible updates.
- Minor: Incremental improvements and additions are backward-compatible.
- Patch: Small updates focused on bug fixes and minor improvements.

Considering all packages that experienced a release gap (i.e., 11,970 packages), we found that the first version released after the gap was a major version for 2,613 (21.83%) packages, a minor version for 3,403 (28.43%) packages, and a patch version for 5,954 (49.74%) packages.

These results suggest some interesting trends in how practitioners could prioritize their work after a period of inactivity. Notably, nearly half of the packages released a patch version first, even after a long pause. This suggests that many package maintainers could focus on fixing urgent bugs and minor improvements immediately after resuming activity, possibly to stabilize the software and address any critical issues that might have arisen during the inactivity period.

The fact that 28.43% of the packages released a minor version could indicate that a significant portion of the maintainers resumed activity with incremental improvements and new features that are backward-compatible. This approach allows for enhancing functionality without disrupting existing users with backward-incompatible changes.

Interestingly, despite the long pause, only 21.83% of packages released a major version first. These should reflect cases where significant changes were made, possibly including new features, architectural changes, or incompatible

⁷ <https://semver.org/>

updates. This could indicate a substantial shift in the project’s direction or a major overhaul that had been planned for some time.

This last finding might seem counterintuitive, as one might expect a major release to be more appropriate given the extended development time. However, this could indicate that many maintainers prefer to ensure stability and address smaller issues before rolling out substantial changes. It might also suggest that the work required for a major release was started but not completed before the gap, and developers chose to focus on immediate fixes and improvements upon resumption. Note that our interpretation of the results is based on the assumption that maintainers of npm packages adhere to semantic versioning principles [22]. For instance, major versions should take more time to release since they require more effort compared to minor or patch releases.

Next, we analyze the gap period by categorizing it according to release type. Table 2 presents the distribution of gap periods in days grouped by release type. We observe that major releases tend to have slightly longer average and median gap periods compared to minor and patch releases, but the differences are relatively minor. This might suggest that the duration of the gap period does not significantly depend on the type of release.

Table 2 Distribution of gap periods in days grouped by release type.

Release type	Mean	Std	Min	Median	Max
major	670.19	296.42	371	577	2,564
minor	652.70	283.23	371	560	2,550
patch	659.72	284.91	369	557	2,470

We also observed that 306 (2.6%) of the packages were initially in the early development phase (i.e., version number $0.x.y$) before the release gap. Subsequently, when they released their first version after the gap, they transitioned to a non-zero major version number (e.g., $1.x.y$).

These initial findings provide valuable insights; however, to gain a more comprehensive understanding of what transpired in the first release after the gap, further analysis will be conducted in **RQ₄**.

➤ **Answer RQ₃.** *Nearly half of the dormant packages resumed with a patch release, indicating a focus on fixing urgent bugs and minor improvements. Minor releases accounted for 28.4%, while only 21.8% of packages released a major version, reflecting significant changes or incompatible updates. Interestingly, gap periods were similar across release types, suggesting that other factors, such as project planning or resources, may influence the timing of updates. Additionally, 2.6% of packages transitioned from early development ($0.x.y$) to a stable major version ($1.x.y$) upon resumption.*

RQ₄: What files were modified between the versions before and after the gap?

Understanding the nature of changes in software packages between the last version before a release gap and the first version after resumption can provide valuable insights into the development practices and strategies employed by software maintainers. This research question focuses on identifying and analyzing the specific files that were modified during this period.

By examining which files were touched, we can gain a clearer picture of the scope and focus of updates implemented after the gap. This includes identifying whether changes were predominantly in source code files, configuration files, documentation, or other components of the software package. Such an analysis helps us understand how development teams prioritize different aspects of the software and uncover their patterns.

For 96.2% (11,516) of the packages that experienced a release gap, we downloaded the last release before the gap and the first release after the gap and analyzed their differences to identify the files that were changed during this period. We had to exclude from our analysis 3.8% (454) of the packages due to download timeouts. Table 3 shows the files that are frequently modified as well as the most common file formats (and their categories) that were changed.

Table 3 Distribution of categories, formats and specific files modified after the release gap.

Category	Packages %	Format	Packages %	File	Packages %
configuration	88.2	json	84.6	package.json	81.9
Source Code	84.6	js	75.4	readme.md	50.9
Documentation	63.2	md	62.1	license	18.9
build/meta	17.3	ts	28	changelog.md	15.4
UI/presentation	12.4	yml	14.6	index.js	15.2
Script	1.9	html	6.7	.npmignore	10.8
IDE/project file	1.4	css	6.1	index.d.ts	10.4
misc	1.2	txt	3.5	.travis.yml	9.2
security/patch	0.35	lock	3.3	lib/index.js	7.6
other	25.1	xml	3	src/index.js ⁸	3.4

We found that JSON files were modified in 84.6% of the packages, indicating that configuration and metadata updates, such as those in *package.json* (excluding mandatory version number changes), are essential and consistently addressed after a gap. JavaScript files were changed in 75.4% of the packages, reflecting updates to the core logic and functionality of the software.

⁸ We also observed `dist/index.js` with 6.2%, but did not include it in the table since files in the `dist/` directory are typically auto-generated by the build process. Their modifications are more likely to reflect changes in the source code or build configuration rather than direct developer edits.

This suggests that significant code changes, whether for adding features or fixing bugs, are the primary focus upon resuming development. Markdown files, including *readme.md*, were updated in 62.1% of the packages, underscoring the importance of maintaining current documentation to support users and provide information on new features or changes. The presence of changes in Typescript (28%), YAML files (14.6%), HTML files (6.7%), etc., further illustrates the diverse range of files touched during these updates, indicating that various aspects of the software, from configuration and build processes to user interfaces and styles, are considered during the post-gap release.

Looking at the specific files modified after the release gap, we observe that specific files such as *package.json* (81.9%⁹), *readme.md* (50.9%), *license* (18.9%), and *changelog.md* (15.4%) were frequently touched. These results emphasize the need to keep metadata, documentation, licensing information, and changelogs up to date. Notably, *package.json* is expected to be modified after each release, primarily to update the version number. However, even when excluding cases where changes were limited to version number updates, we found that *package.json* was of the most modified, indicating frequent updates to other metadata. These changes likely pertain to dependencies and their versions, which play a crucial role in ensuring compatibility and functionality across software versions. Further investigation into the nature of these dependency changes will be discussed later, aiming to provide deeper insights into how maintainers manage and update dependencies to enhance software stability and performance after the release gap. In addition, other files like *index.js* (15.2%), *.npmignore* (10.8%), *index.d.ts* (10.4%), and *.travis.yml* (9.2%) were also commonly changed, pointing to updates in entry points, package management configurations, type definitions, and CI/CD configurations, respectively.

To provide a clearer overview of the functional nature of changes, we grouped files into broader categories based on their roles. As expected, we found that **configuration files** (e.g., *package.json*, *.yaml*) were the most frequently updated, appearing in 88.2% of the packages. **Source code files** (e.g., *.js*, *.ts*) followed closely at 84.6%, while **documentation files** (e.g., *readme.md*, *license*) were updated in 63.2% of the packages. Other categories such as **build/meta** (17.3%), **UI/presentation** (12.4%), and **scripts** (1.9%) were also touched, albeit less frequently.

While this analysis provides a descriptive foundation for understanding the types of files that are frequently modified, a deeper investigation into whether the changes adhere to semantic versioning (**SemVer**) conventions would require more sophisticated methods. In JavaScript, due to its dynamic nature and lack of static types, detecting true breaking changes or API evolution through simple file diffs or AST analysis is unreliable. Moreover, packages often involve complex build pipelines that obfuscate actual developer-facing APIs. As such, a rigorous **SemVer** compliance check would demand reconstructing module interfaces, tracing exports, and potentially using dynamic or test-based anal-

⁹ All packages modify *package.json* during a release. This percentage only considers packages that modified *package.json* beyond the mandatory version number change.

ysis—an effort we identify as an important direction for future work. At this stage, we treat semantic versioning labels as informative heuristics and focus our efforts on cataloging change patterns to support future audits.

Concerning the size of changes that occurred during the gap, Figure 4 shows the distribution of the number of changes grouped by file formats. We observe that JavaScript files experience the highest median number of changes (94), reflecting its central role in application logic and functionality. This suggests that significant updates in the codebase are common when development resumes after a gap. TypeScript files also see substantial changes (median of 32), highlighting its importance in modern development practices for adding type safety and improving code quality.

JSON files, with a median of 14 changes, are frequently updated to reflect changes in configuration and dependencies. Markdown files, often used for documentation (median of 21 changes), show the importance of keeping documentation current with the latest changes. YAML files have lower median changes (12), but their updates are still significant.

The overall median of 147 changes per release suggests that resuming development often involves important updates across various parts of the project. The average of 2,999 changes indicates that some packages undergo very large-scale modifications, possibly involving major refactoring or significant new feature additions. This result highlights the effort and complexity involved in revitalizing a software package after a period of inactivity.

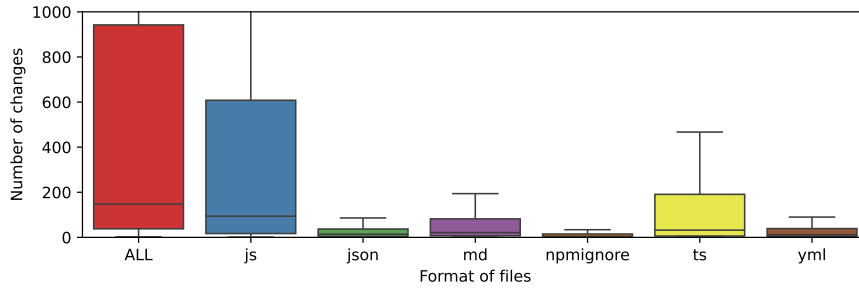


Fig. 4 Distribution of the number of changes grouped by file formats after the release gap.

➤ **Answer RQ₄.** *JSON files were modified in 84.6% of the packages, indicating frequent updates to configurations and metadata, particularly package.json. JavaScript files were changed in 75.4% of the packages, reflecting substantial updates to the core logic. Documentation files, such as readme.md, were updated in 62.1%, highlighting the importance of keeping user-facing information current. Concerning the size of changes, the average of 2,999 indicates that some packages are involved in big modifications. Finally, the median of 147 changes suggests that resuming packages often involves extensive modifications across configuration, code, and documentation to ensure software stability after a gap.*

RQ₅: Do packages modify their dependencies upon resuming activity?

In **RQ₄**, we found that *package.json* is the most frequently changed file after a release gap, indicating that maintainers often update metadata upon resuming activity. This file typically includes crucial information about the package, such as its version, scripts, and dependencies. Given the centrality of dependencies in ensuring a software package functions correctly and stays up-to-date with security patches and feature improvements, it is reasonable to hypothesize that maintainers frequently modify their dependencies when updating *package.json*.

In this research question, we aim to investigate whether packages modify their dependencies upon resuming activity. By examining the extent and nature of these changes, we can confirm or refute the assumption that dependency updates are a common part of the post-gap release process. Additionally, we quantify the changes in dependencies, providing insights into how maintainers manage external packages to ensure the stability and improvement of their software. Understanding these patterns is essential for comprehending the broader maintenance strategies employed by open-source project maintainers.

Initially, we found that 11,337 (94.7%) of the packages had at least one dependency either before or after the gap, with 11,260 packages having dependencies before the gap and 11,285 after the gap. Collectively, these packages used 25,534 unique packages as dependencies. Of these, 66.06% were for development purposes, 33.91% for run-time, and 0.02% for optional use. Both before and after the gap, the median number of dependencies per package was 10, with a similar average of approximately 14 dependencies. This consistency suggests that, despite the development pause, the overall dependency footprint of the packages remained stable.

Examining the dependencies before and after the gap, we found that 62.3% of the dependencies remained unchanged, indicating a significant proportion of stability in dependency management. However, 27.5% of the dependencies had changed their version requirements, reflecting efforts to upgrade or downgrade dependency versions to maintain compatibility and leverage new features or fixes. Additionally, 10.2% of the dependencies were removed, suggesting some level of refactoring or optimization. Notably, after the release gap, packages

Table 4 Distribution of dependency changes in packages after a release gap, showing the number and proportion of added, removed, changed, and unchanged dependencies.

Dependencies	# packages	mean # (%)	median # (%)
Added	3,645	5 (23.6)	2 (16.7)
Removed	3,172	5.2 (23.6)	2 (16)
Changed requirement	6,976	6.3 (44.4)	4 (40)
No change	3,172	10.3 (68.9)	6 (83.3)

added about 11.3% new dependencies, indicating ongoing development and integration of new functionalities. Table 4 provides a detailed breakdown of the changes in dependencies across the packages that experienced a release gap, with proportions calculated by merging both pre-gap and post-gap dependencies for each package. For example, if a package initially had three dependencies, then added two new ones, modified one, retained one, and removed one, the total number of dependencies would be five. The proportions for each category are then derived from this total. The data shows that, on average, 5 new dependencies were added per package (23.6%), 5.2 dependencies were removed (23.6%), and 6.3 dependencies had their version requirements changed (44.4%), with medians of 2 and 4 for each of these categories, respectively. Additionally, on average, 68.9% of dependencies remained unchanged, with a median of 6. These findings highlight that many packages actively change their dependencies when resuming releases, reflecting efforts to stay current and secure. However, the proportion of unchanged dependencies indicates that many packages do not update their dependencies, potentially leaving them out of date and possibly vulnerable. Furthermore, we noticed that run-time dependencies are less prone to change compared to development dependencies, suggesting that maintainers prioritize stability for run-time components while being more flexible with development tools and libraries.

► **Answer RQ₅.** While 62.3% of dependencies remained unchanged, 44.4% of the changed dependencies had their version requirements modified, and 23.6% of packages added new dependencies, reflecting active efforts to update or improve functionality. Another 23.6% of packages removed dependencies, indicating refactoring or optimization. Run-time dependencies were less frequently changed compared to development dependencies, highlighting a focus on maintaining stability in production environments. These findings reveal that maintainers frequently adjust dependencies post-gap to ensure compatibility and security, though many dependencies remain unchanged, posing potential risks.

RQ₆: How outdated were the dependencies during the gap?

In the previous research question, we observed that many packages modify their dependencies upon resuming releases, while a significant portion do not,

potentially leaving their dependencies outdated. This finding is particularly concerning for packages with extended release gaps, as they can result in significantly outdated dependencies, increasing the risk of security vulnerabilities, performance issues, and incompatibility with newer technologies.

Given the long release gaps, it is plausible that dependencies may have fallen behind on critical updates during this period. This situation can pose risks not only to the inactive packages but also to the broader software development. Therefore, it is essential to understand how outdated the dependencies were during the gap to assess the potential impact on software security.

In this research question, we investigate the extent to which dependencies became outdated during the release gap. By analyzing the version histories, we understand how far these dependencies fell relative to their latest versions available during the gap. This analysis provides insights into the challenges maintainers face when reactivating their projects and highlights the importance of keeping dependencies up-to-date even during periods of inactivity.

After identifying the exact dependency version installed by each package at three different time points – before the gap (i.e., the last version before the gap), during the gap (i.e., right before resuming releases), and after the gap (i.e., the first version after the gap) – we compared these versions to their latest available versions.

Table 5 Statistics about outdated dependencies before, during, and after the release gap.

	Before the gap	During the gap	After the gap
Proportion of outdated dependencies	32.4%	55.2%	37.8%
Number of packages with outdated dependencies	6,933	8,757	7,184
Mean number of outdated dependencies	6.6	9	7.7
Median number of outdated dependencies	4	6	5

Table 5 reports our results. We found that 32.4% of the dependencies were out-of-date before the gap. However, during the pause and right before resuming releases, this proportion increased to 55.2%. After the gap, maintainers updated their dependencies, reducing the proportion of outdated dependencies to 37.8%. Additionally, the number of packages with outdated dependencies increased from 6,933 before the gap to 8,757 during the gap, but then decreased to 7,184 after the gap. The mean number of outdated dependencies per package increased from 6.6 before the gap to 9 during the gap, then decreased to 7.7 after the gap. A similar trend was observed for the median.

This result shows that many dependencies become outdated during the release pause, which might be harmful to the packages’ dependents as they inherit outdated dependencies beyond their control. Moreover, the fact that

37.8% of dependencies remain outdated even after the gap suggests that maintainers might be less meticulous in updating dependencies than they were before the gap. This could happen for several reasons, e.g., a shift in priorities, limited resources, or a focus on addressing immediate critical issues first.

Next, we quantify the extent to which these dependencies were outdated, in terms of missed versions and the time elapsed between the latest available releases and the versions used by the dormant packages. Figure 5 illustrates the distribution of these two metrics. We observe that during the gap, packages had the highest number of missed releases for their dependencies. Specifically, the median number of versions behind the latest available release for outdated dependencies was 9 versions before the gap and 13 during and after the gap.

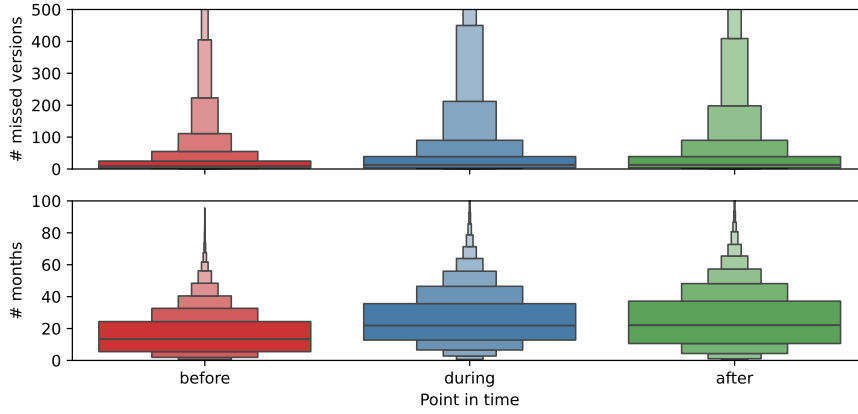


Fig. 5 Distribution of the number of missed versions and time of outdated dependencies in three points in time of dormant packages.

Similarly, in terms of time, we observe that outdated dependencies became progressively more outdated during and after the gap. Specifically, before the gap, outdated dependencies lagged behind the latest available version by a median of 13 months. During the gap, this increased to 22 months, and it remained at 22 months after the gap. This indicates that the period of inactivity significantly contributes to the staleness of dependencies, with maintainers potentially facing more substantial technical debt upon resuming activity.

To statistically confirm our observation, we conducted a non-parametric statistical test using the Mann-Whitney U test [18] to compare the distribution of missed versions across different time periods. The null hypothesis (H_0) states that there is no difference between the distributions being compared. We established a global confidence level of 99%, corresponding to a significance level of $\alpha = 0.01$. Our analysis revealed significant differences in the number of missed versions for outdated dependencies before the gap compared to other periods ($p - \text{value} < 0.01$). However, for outdated dependencies during and

after the gap, we could not reject the null hypothesis ($p = 0.16$). This suggests that outdated dependencies after the gap exhibited a similar number of missed versions as those during the gap. A possible explanation for this finding is that the outdated dependencies after the gap may be those that accumulated more lag compared to before and during the gap periods. When maintainers resume releasing, they may prioritize updating some dependencies but overlook others that have fallen significantly behind in versions. This selective updating behavior could explain the observed similarity in the number of missed versions during and after the gap periods.

► **Answer RQ₆.** *Before the gap, 32.4% of dependencies were outdated, increasing to 55.2% during the gap. After the gap, this number dropped to 37.8%, indicating that many packages remained outdated even after resuming activity. Dependencies lagged behind the latest versions by a median of 13 months before the gap, which grew to 22 months during and after the gap. This suggests that maintainers often face significant technical debt upon resuming development and may not fully update all dependencies, potentially leaving some outdated.*

RQ₇: How do security vulnerabilities affect a package’s dependencies before, during, and after a gap?

Building on our previous analysis of outdated dependencies, this research question explores the security implications associated with these dependencies. Specifically, we aim to assess the number of vulnerabilities affecting the dependencies of packages before, during, and after the gap in their release activity. Understanding the security posture of dormant packages is crucial for several reasons. Outdated dependencies often harbor known vulnerabilities that can be exploited if not addressed. For dormant packages, the risk is amplified as the period of inactivity may lead to an accumulation of security issues, potentially putting dependent packages and systems at risk. By analyzing the security status of dependencies across these three time points, we can gain insights into how periods of inactivity impact the overall security of software packages. This will also give us an idea of whether maintainers effectively mitigate these risks when they resume releasing new versions.

To collect the vulnerabilities, we employed a static approach, considering all vulnerabilities affecting a package, regardless of when they were discovered or publicly disclosed. In other words, a vulnerability is considered present from the moment it is introduced in the package, even if it remains undisclosed until reported later, rather than counting the vulnerabilities that had been officially disclosed by that time. Our choice of a static analysis is motivated by the need to accurately capture the security risk accumulated during dormancy. If we were to restrict our analysis to a dynamic view, counting only vulnerabilities as of their disclosure date, we potentially underestimate the exposure that latent, yet-to-be-disclosed flaws present to downstream users during a gap. By

attributing each vulnerability to the point at which its code first appears, we ensure that our risk assessment reflects the full window of potential exploitation, rather than just the period after public disclosure.

Table 6 Overview of vulnerabilities affecting dependencies before, during, and after the release gap, including the number of vulnerabilities, affected dependencies, and the proportion of exposed dormant packages.

Time point	# vulnerabilities	# affected dependencies	% exposed packages
Before	1,162	837	6.6
During	4,759	2,920	19.7
After	3,095	1,897	12.3

Table 6 provides a detailed overview of the number of vulnerabilities affecting dependencies at three key time points. We observe a significant increase in the number of vulnerabilities and affected dependencies during the gap period. Specifically, the number of vulnerabilities jumps to 4,759 during the gap from 1,162 before the gap, while the number of affected dependencies rises from 837 to 2,920. This substantial increase during the gap underscores the risk associated with periods of inactivity, as dependencies become more prone to accumulating known security vulnerabilities. The percentage of exposed packages also more than doubles during the gap, rising from 6.6% before the gap to 19.7% during the gap. This indicates that a larger proportion of packages are at risk during the period of inactivity, potentially exposing dependent projects and systems to security threats.

After the gap, there is a notable decrease in the number of vulnerabilities and affected dependencies, with the figures dropping to 3,095 and 1,897, respectively. Despite this decrease, the percentage of exposed packages remains higher than before the gap, at 12.3%. This suggests that while maintainers address some vulnerabilities when they resume activity, a significant number of dependencies remain unsecured, leaving a substantial portion of packages exposed to potential security risks. This highlights the importance of regular maintenance and updates to dependencies, especially for dormant packages, to mitigate the risks associated with security vulnerabilities.

We also observed that run-time dependencies consistently have a higher proportion of being affected by vulnerabilities compared to development dependencies across all time points. For example, before the gap, 1.05% of run-time dependencies were affected by vulnerabilities, whereas only 0.34% of development dependencies were affected. During the gap, the proportion of affected runtime dependencies increased to 2.99%, while 1.54% of development dependencies were affected. After the gap, the trend persisted, with 2.02% of runtime dependencies being affected compared to 0.94% of development dependencies. This suggests that run-time dependencies are more vulnerable to security issues than development dependencies throughout the release cycle.

Finally, Tables 7, 8, and 9 report some statistics on the security issues present in dormant packages. Specifically, Table 7 presents the most common

Table 7 Top 5 common weakness enumerations (CWE).

CWE	# vulnerabilities
400	1,333
79	1,149
20	904
1321	780
1333	630

Table 8 Top 5 dependencies affecting dormant packages.

Dependency	# affected packages
<code>debug</code>	257
<code>lodash</code>	185
<code>node-sass</code>	163
<code>eslint</code>	151
<code>semantic-release</code>	147

Table 9 Number of vulnerabilities grouped by severity.

Severity	# vulnerabilities
Moderate	4,882
High	2,856
Critical	732
Low	602

types of vulnerabilities affecting packages. We can observe that CWE-400, i.e., *Uncontrolled Resource Consumption* leads the list with 1,333 occurrences, followed closely by CWE-79, i.e., *Cross-Site Scripting* with 1,149. These vulnerabilities highlight the prevalence of both resource management and input validation issues in modern software development. In Table 8, we see that the `debug` library tops the list, impacting 257 dormant packages, followed by `lodash` with 185. These libraries are widely used in various projects, and their vulnerabilities have wide-ranging consequences, especially for packages that experience prolonged inactivity. Table 9 categorizes vulnerabilities based on their severity, revealing that most of the vulnerabilities are classified as *Moderate* (4,882) and *High* (2,856). Although *Critical* vulnerabilities are fewer in number (732), their potential impact on security cannot be overstated.

► **Answer RQ₇.** *During the gap, the number of vulnerabilities increased from 1,162 to 4,759, affecting 19.7% of packages, compared to 6.6% before the gap. After the gap, vulnerabilities decreased but remained concerning, with 12.3% of packages still exposed. Run-time dependencies were consistently more affected than development dependencies across all periods. The most common vulnerabilities included CWE-400 and CWE-79. Despite addressing some issues post-gap, many dependencies remained outdated, posing ongoing security risks.*

RQ₈: To what extent do dependents continue using dormant packages during and after the gap?

When a dependency becomes dormant, dependent packages face uncertainty about future support, bug fixes, and security updates (*RQ₇*). This creates a complex decision-making scenario for developers who must balance the risks of continuing to use potentially outdated dependencies (*RQ₆*) against the costs and risks of migrating to alternatives.

This research question investigates the behavior of packages that depend on dormant npm packages, specifically examining whether and when they abandon these dependencies during or after a release gap. Understanding how dependent packages respond to such dormancy is crucial for assessing the broader impact of inactivity within the npm ecosystem and for evaluating ecosystem resilience to dependency maintenance gaps.

To analyze this response, we tracked 19.2 million release-level dependency relationships involving 6,854 dormant packages and 94,621 dependent packages from our comprehensive npm dataset. We identified 160,799 cases where a dependent was using a dormant package prior to the start of its dormancy gap.¹⁰ Using a breakpoint analysis framework, we examined the behavior of these dependents across five critical timepoints: before the gap began, at 25% through the gap, at 75% through the gap, at gap end, and after gap conclusion. Based on dependency usage patterns across these breakpoints, we classified packages into eight distinct behavioral categories: loyal throughout, various abandonment patterns (early, mid, late, and post-gap), and re-adoption patterns (quick, mid-gap, and late re-adoption).

Our analysis revealed distinct and compelling behavioral patterns in how packages respond to dependency dormancy. The overwhelming majority of dependent packages (81.2%) exhibit loyal behavior throughout, continuing to use dormant dependencies throughout the entire dormancy period without interruption. This substantial loyalty rate suggests strong inertia in dependency management decisions, indicating that many dormant packages continue to function adequately despite a lack of active maintenance.

However, a significant minority (18.3%) of dependents eventually abandoned the dormant package, demonstrating various timing patterns in their abandonment decisions. Figure 6 shows these patterns. The most common abandonment pattern is abandoned mid-gap (7.0%), where packages cease using the dormant dependency approximately halfway through the dormancy period. This is followed by abandonment early in the gap (4.7%), where dependents make relatively quick decisions to discontinue usage, and abandonment after the gap ends (3.6%), where abandonment occurs only after the dormancy period has concluded. Additionally, 3.0% of packages show abandoned late in gap behavior, suggesting that some dependents wait until near the end of dormancy before making abandonment decisions.

¹⁰ One dependent package may have used more than one dormant package

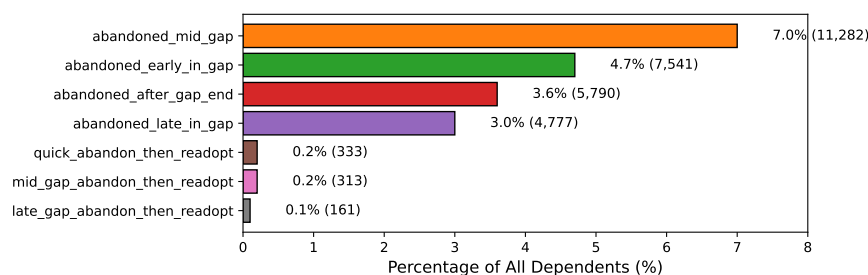


Fig. 6 Distribution of behavioral patterns at different breakpoints.

Notably, only a small fraction of dependents exhibited re-adoption behavior, where they initially abandoned the dormant dependency but later resumed its use. Specifically, 0.2% showed quick abandonment then re-adopt behavior, 0.2% demonstrated mid-gap abandonment then re-adopt, and 0.1% exhibited late gap abandonment then re-adopt patterns. This extremely low overall re-adoption rate (0.5%) contrasts sharply with the abandonment rate, indicating that abandonment decisions are typically permanent and that developers rarely reconsider their dependency choices once they have decided to abandon.

The timing analysis reveals important insights into the decision-making process surrounding dependency abandonment. Figure 7 shows the distribution of abandonment timing from the start of the gap. Packages that abandon dormant dependencies do so after an average of 281.5 days from gap start (median: 251.5 days), indicating that abandonment decisions are not immediate responses to dormancy but occur after substantial waiting periods. This delay suggests that developers adopt a cautious approach to dependency changes, potentially waiting to assess whether the dormancy is temporary or represents a permanent shift away from active maintenance.

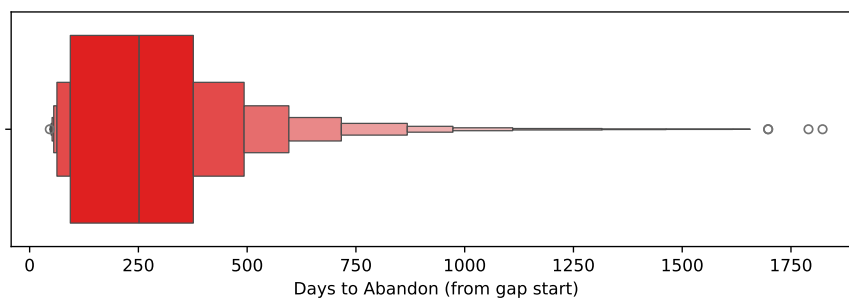


Fig. 7 Distribution of abandonment timing.

For the small subset of packages that re-adopt dormant dependencies, the temporal patterns are even more extended. The average time to re-adoption is approximately 670.2 days (median: 610 days), indicating that re-adoption typically occurs well into or after the dormancy period has concluded. This extended timeline suggests that re-adoption is often driven by the dependency’s eventual return to active maintenance rather than by urgent functional needs during the dormancy period itself.

► **Answer RQ₈.** *Out of 160,799 dependent packages using dormant dependencies before gaps, 81.2% remained loyal throughout the entire dormancy period, while 18.3% eventually abandoned these dependencies. Abandonment occurred gradually, with dependents waiting an average of 281.5 days before discontinuing usage, most commonly abandoning mid-gap (7.0%) or early in the gap (4.7%). Only 0.5% of packages showed re-adoption behavior, indicating that abandonment decisions are typically permanent.*

5 Qualitative Analysis

In Section 4, we explored the quantitative aspects of dormant packages, such as dependency management and security vulnerabilities. To gain a more comprehensive understanding of these packages, we now turn to a qualitative analysis. In particular, starting from a random sample of 18 dormant packages, we manually inspected the commit messages, pull requests, and issues to analyze the dynamics of package development during the release gap and the decisions made by maintainers when resuming release. We repeated the same process until saturation to find new patterns related to package dormancy and subsequent revival. After the third iteration, we did not find new patterns, so we stopped our analysis with a final sample of 54 dormant packages—more details on the analyzed packages are reported in our online appendix [32].

This qualitative analysis aims to reveal the dynamics of package development during release gaps and the decisions made by maintainers when resuming release. By examining commit messages, we can identify the types of changes implemented and the motivations behind them. Pull requests offer insights into the collaboration between contributors and maintainers, highlighting the review processes and discussions. Issues provide a glimpse into the challenges and priorities faced by the maintainers, as well as the feedback and concerns raised by the user community.

After manually inspecting these packages, we observed several recurring themes and patterns regarding their package dormancy and subsequent revival.

Author Interaction and Maintenance Patterns

A common observation is that many packages had minimal or no commits during their dormancy period. However, some authors remained intermittently active, addressing critical issues or merging pull requests. For instance,

`grunt-compile-handlebars`¹¹ saw some activity during the gap, primarily for updating dependencies to avoid vulnerabilities, indicating sporadic yet essential maintenance. Similarly, `sax`¹² had commit activity and merged pull requests during its dormant period, showing that even during inactivity, some level of maintenance continued. On the other hand, packages like `copy-paste`¹³ and `bitcoin`¹⁴ had no activity during the gap but were revived with significant updates, demonstrating a renewed interest or need for maintenance. The package `bitcoin` was notably revived by a different author who deprecated all previous versions, signaling a significant overhaul and a fresh start.

Reasons for Revival and Updates

Several packages were revived to address accumulated technical debt, such as updating dependencies, fixing security vulnerabilities, or making minor patches. For example, `js-md5`¹⁵ and `dox`¹⁶ were revived with updates focused on dependency management and security enhancements. The package `grunt-string-replace`¹⁷ came back with updates to dependencies and added continuous integration (CI) workflows. The `auto-reload-brunch`¹⁸ package was updated to avoid vulnerabilities, highlighting the importance of security in reviving dormant packages. In some cases, the revival was driven by specific needs or changes in the development environment, such as `componentjs`,¹⁹ which added new features upon its revival.

Impact on Users and Community Interaction

User complaints and interactions often played a crucial role in the revival of dormant packages. Many packages saw user inquiries about the status of the project, offers for help, or even forks and new projects being created due to the lack of updates. For instance, `spritesheet-js`²⁰ saw users asking about the state of the project and offering to help, while the `semantic-ui-*` packages²¹ experienced community-driven forks like `fomantic-UI`.²² Still, the `ip`²³ package had users questioning its status and the original owner eventually offering

¹¹ <https://www.npmjs.com/package/grunt-compile-handlebars>

¹² <https://www.npmjs.com/package/sax>

¹³ <https://www.npmjs.com/package/copy-paste>

¹⁴ <https://www.npmjs.com/package/bitcoin>

¹⁵ <https://www.npmjs.com/package/js-md5>

¹⁶ <https://www.npmjs.com/package/dox>

¹⁷ <https://www.npmjs.com/package/grunt-string-replace>

¹⁸ <https://www.npmjs.com/package/auto-reload-brunch>

¹⁹ <https://www.npmjs.com/package/componentjs>

²⁰ <https://www.npmjs.com/package/spritesheet-js>

²¹ <https://semantic-ui.com/>

²² <https://www.npmjs.com/package/fomantic-ui>

²³ <https://www.npmjs.com/package/ip>

to transfer the package to experienced maintainers, illustrating how community pressure can lead to package revival.²⁴

Changes and Management Post-Revival

The approach to managing the package post-revival varied. Some authors continued with minor updates and patches, while others undertook major overhauls. For example, the package `a`²⁵ introduced a new major version and created a funding file upon its revival, indicating an attempt to sustain the package financially. The package `js-md5` not only updated dependencies but also deprecated elements and added new features, showing a comprehensive update strategy. These overall observed patterns can be attributed to several factors:

Resource Constraints. Open-source maintainers often face time and resource constraints, leading to periods of inactivity or dormancy. When critical issues accumulate, or community demand resurges, maintainers or new contributors step in to revive the package.

Community Influence. User feedback and community pressure significantly influence the revival of dormant packages. Persistent user interest, offers of help, and the creation of forks highlight the community’s role in sustaining open-source projects.

Technical Debt and Security. The need to address technical debt, such as outdated dependencies and security vulnerabilities, is a strong motivator for reviving dormant packages. Security concerns, in particular, drive maintainers to update and patch packages to ensure their safety for users.

Changing Development Needs. As the development environment evolves, previously dormant packages may become relevant again, necessitating updates and new features to keep pace with current standards and practices.

6 Discussion

In this section, we elaborate on the main insights coming from our analyses.

6.1 Contextualizing Our Findings

In this section, we compare our findings on dependency changes, outdatedness, security, and use of dormant packages with those from prior studies.

Starting from the activity to resume a dormant package (**RQ₅**), we found that only about 38% of dependencies change upon revival, while 62% remain exactly the same. This behavior closely mirrors the formal technical-lag framework for npm proposed by Zerouali et al. [31], which shows that packages updating after falling behind typically address only a subset of their outdated

²⁴ <https://github.com/indutny/node-ip/issues/128>

²⁵ <https://www.npmjs.com/package/a>

dependencies rather than performing a full upgrade. Always within npm, Mujahid et al. [21] demonstrated that when clear migration paths are offered, maintainers will adopt them. Yet absent such guidance, packages often stick with existing dependencies. Together, these studies underscore that maintainers balance stability and modernization after dormancy, rather than pursuing all-or-nothing updates. Finally, Cogo et al. [5] observed that among packages that partially deprecate, nearly 30% never supply direct replacements, suggesting maintainers’ cautious approach to sweeping dependency shuffles.

Concerning **RQ₆**, our findings show that the use of outdated dependencies jumps from 32% to 55%, with a median version lag from just over a year to nearly two years. This trend confirms the one revealed by Zerouali et al. [31], especially for transitive dependencies. Kula et al. [16] reported that more than 80% of systems have outdated dependencies, and that security patches take on average 14 months to be applied. Furthermore, Miller et al. [19] showed that once popular npm packages are flagged as abandoned, downstream projects struggle to respond, highlighting how the lack of clear deprecation warnings allows stagnation. These patterns confirm that without active maintenance or reporting, dependencies are severely delayed.


Moving to **RQ₇**, we observed an increase in vulnerability from around 1,100 before the gap to nearly 4,800, then receding only partially once releases resume. The analysis performed by Kula et al. [15] on Maven shows a median delay of a year in adopting the latest Maven release, creating a latent window in which bug fixes and security patches go unused. Again, Miller et al. [19] demonstrated that when maintainers explicitly declare end-of-life, downstream projects remove vulnerable dependencies roughly 1.6 times faster, pointing to communication as the key lever in mitigating security debt. Complementing these observations, Zerouali et al. [30] quantified a median of 31 months to disclosure lag and 55 months to fix lag for npm vulnerabilities, resulting in 42% of packages and 79% of external projects remaining exposed for years before remediation is available. Thus, the security-debt accumulation we observe during npm dormancy is part of a wider challenge.

Finally, **RQ₈** examines how dependents react to dormancy: we found that 81% of dependents continue to declare and install the inactive package throughout its gap. At the same time, only 18% eventually abandon it, doing so after a median of 252 days. This result recalls the work by Miller et al. [19]: about 18% of downstream projects remove an abandoned dependency. In addition, the removal is typically delayed unless a clear sunset signal is provided.

6.2 Quantitative analysis


The analysis of release gaps in **RQ₁** reveals that while packages are often actively maintained with frequent updates, they can enter extended periods of dormancy, with gaps lasting from 12.3 months to over 7 years. This unpredictability underscores the importance of developers and users maintaining awareness of the activity levels of the packages they rely on. Long gaps can

lead to significant technical debt and potential security risks, especially for critical dependencies. To mitigate these risks, developers should consider implementing automated monitoring tools to track the update frequency of their dependencies and proactively plan for alternatives or forks when a package shows signs of entering a dormant phase. Additionally, communities could benefit from fostering a culture of shared responsibility, where dormant but widely used packages receive community-driven maintenance to ensure their continued reliability and security. Furthermore, at the package managers level, npm itself could introduce an “inactive-package” badge, so practitioners can immediately know when a dependency is experiencing a dormant period.

 **Lesson₁.** *Automated tools or badges can help track update frequency and signal when a package enters dormancy, enabling developers to plan for alternatives or forks proactively. Additionally, fostering a community-driven approach to maintaining widely used, dormant packages can mitigate the risks of technical debt and security vulnerabilities, ensuring the continued reliability of critical software components.*

The results for **RQ₂** indicate that release gaps often occur after packages have built a significant history of regular updates, typically around three years of age. This suggests that while packages may initially maintain user trust through frequent updates, they are still vulnerable to periods of dormancy. Notably, when packages resume activity after a gap, a significant portion releases only a single version, implying that many may not fully recover and are at risk of future inactivity.

Rather than assuming that any revival signals a return to normal, consumers should place revived packages on a “short watchlist”, re-evaluating their health each time a new release appears. From a practical perspective, packages that have experienced a release gap should be scrutinized closely, and users should consider contingency plans, like alternative packages, to mitigate the impact of future gaps. Users are also encouraged to contribute to maintain momentum, especially after a gap. Additionally, given the weak correlation between gap length and subsequent version activity, developers should not assume that a longer gap will necessarily result in fewer updates afterward.

 **Lesson₂.** *Practitioners should prepare for potential disruptions by identifying alternatives and monitoring packages that have resumed activity post-gap, as many may not fully recover. Community engagement and consistent contributions are essential to sustaining package vitality and preventing future dormancy, regardless of the length of the initial gap.*

The findings from **RQ₃** reveal that the majority of packages (49.74%) release a patch version immediately after a release gap, emphasizing a strong focus on stabilizing the software.

This trend suggests that maintainers prioritize immediate stability and user confidence after a period of inactivity. We observed that only 21.83% of packages resume with a major release despite the potential for substantial

changes during a prolonged development gap. This may indicate a strategic decision to re-establish software stability before implementing more disruptive or large-scale updates. It also suggests that practitioners often adopt a phased approach to significant changes, making incremental updates first to lay a solid foundation for future major releases.

From a practical standpoint, developers should prioritize stability and minor improvements post-gap. Users can plan a gradual adoption of updates to avoid disruptions, while community managers should encourage maintainers to communicate their post-gap plans and address critical issues early.

While 49.74% of dormant packages resume with a patch release, this proportion is relatively low when compared to general npm trends, where 68.8% of all releases are patches [31]. This suggests that dormant packages tend to resume activity with more substantial changes (often major or minor versions) than regularly maintained packages. This contrast highlights that post-gap releases may not simply reflect routine maintenance but instead signal important functional or architectural updates that accumulated during the dormant period. For this reason, maintainers should accompany major or minor post-gap changes with detailed migration guides and semantic-version checklists to ease downstream upgrades.

📌 **Lesson₃.** *Developers and users should anticipate the focus on stability post-gap by scheduling the updates. Clear communication from maintainers about post-gap plans can further manage expectations and ensure that critical issues are addressed early, fostering smoother transitions.*

RQ₄ reveals that after a release gap, developers consistently focus on updating key files like configuration files, core code files, and documentation. This indicates a priority on stabilizing the software, ensuring compatibility, and keeping users informed through updated documentation. For practitioners, this suggests the importance of systematically addressing technical debt and potential dependency vulnerabilities that may have accumulated during the gap. Prioritizing updates to configuration and dependency management files is important to ensure that the software remains secure and functional. Furthermore, clear and comprehensive documentation updates are essential for maintaining user trust and easing the transition to new versions. Maintainers can streamline this work by enabling automated dependency bots (e.g., Dependabot), so that every update triggers a pull request with a human-readable summary and documentation automatically refreshed. Such automation ensures configuration, code, and docs remain in sync out of the box.

📌 **Lesson₄.** *Developers should systematically address technical debt and dependency vulnerabilities. Moreover, implementing automated tools for dependency management can further enhance the efficiency of post-gap updates, ensuring high-quality releases and maintaining user trust through clear documentation and communication.*

In **RQ₅** we found that most software packages maintain stability in their dependencies after resuming activity, with 62.3% of dependencies unchanged.

However, a non-negligible portion of packages actively manage their dependencies, reflecting an effort to maintain compatibility, incorporate new features, and address security concerns. The data also shows that runtime dependencies are less frequently altered compared to development dependencies, indicating a priority on maintaining stability for end-users. These findings suggest that maintainers should focus on updating critical dependencies, particularly those affecting security and core functionality, while ensuring the stability of runtime environments. To help maintainers prioritize these updates, projects should adopt a “risk-scoring” metric in their dashboards that weights dependencies by runtime impact and known vulnerabilities, automatically surfacing critical patches first.

📌 **Lesson₅.** *The active management and update of critical dependencies, especially those related to security and core functionality, is crucial. By balancing stability with necessary updates, maintainers can address potential risks while preserving the reliability of their software packages.*

The analysis in **RQ₆** highlights that even after resuming activity, 37.8% of dependencies remain outdated, suggesting that maintainers may only partially address the technical debt accumulated during the pause. This underscores the risks associated with prolonged inactivity, as outdated dependencies can affect both the dormant packages and their dependents by introducing compatibility issues and security vulnerabilities. Furthermore, the number of missed versions for outdated dependencies shows little improvement after the gap, indicating that maintainers often prioritize updating a subset of critical dependencies while reducing the priority of others due to resource constraints or immediate development needs.

📌 **Lesson₆.** *To mitigate these risks, maintainers should adopt a targeted yet systematic approach to dependency management. By prioritizing updates for critical and high-risk dependencies while leveraging automated tools and community support to address non-critical dependencies over time, maintainers can better prevent security vulnerabilities and ensure long-term compatibility, especially for packages with extended-release gaps.*

RQ₇ shows an increase in security vulnerabilities during periods of inactivity, with the number of affected dependencies and vulnerabilities more than quadrupling during the gap. This confirms the significant security risks posed by dormant packages, as outdated dependencies can accumulate known vulnerabilities that compromise the security of dependent systems. Even after resuming activity, a portion of dependencies remains vulnerable, with 12.3% of packages still exposed, compared to 6.6% before the gap. This underscores the critical need for continuous maintenance of dependencies, particularly in runtime environments, which are more prone to exposure to security issues than development dependencies. Upon resuming releases, addressing these vulnerabilities should be a top priority to ensure the software remains secure.

✚ **Lesson₇.** *The increase in vulnerabilities during periods of inactivity highlights the critical need for maintainers to prioritize updating and securing dependencies, especially in runtime environments, which are more vulnerable. Users of dormant packages should be vigilant and conduct their own security assessments to mitigate risks. Proactive, continuous maintenance is essential to minimize the security risks posed by outdated dependencies.*

RQ₈ reveals that the majority of dependent packages (81.2%) remain loyal to dormant dependencies throughout the entire dormancy period, demonstrating significant ecosystem resilience to maintenance gaps. However, 18.3% of dependents eventually abandon dormant packages, with abandonment decisions occurring after careful consideration (average 281.5 days) rather than immediate responses to dormancy. During that window, we observed that 160,799 dependent-package relationships persisted through dormancy. Hence, these packages remain connected to a dormant dependency that can no longer receive updates (including security patches). In other words, every dependent inherits any preexisting vulnerabilities of its dormant dependency, and no new version is available to fix them. Thus, even if a dormant package has known security flaws, its dependents remain exposed until they choose to abandon it or a new release appears. The extremely low readoption rate (0.5%) indicates that abandonment decisions are typically permanent, suggesting that dormancy periods can result in lasting ecosystem fragmentation. These patterns reveal that while ecosystems can tolerate dependency dormancy in the short term, extended periods of inactivity gradually erode dependency relationships and may lead to permanent loss of dependent packages. To prevent this, maintainers should publish a concise “revival roadmap” upon first returning from dormancy, outlining which vulnerabilities will be patched, which dependencies will be upgraded, and a tentative schedule for further minor and major releases.

✚ **Lesson₈.** *The high loyalty rate during dormancy demonstrates ecosystem resilience, but the gradual abandonment and low readoption rates highlight the long-term costs of maintenance gaps. Maintainners should be aware that extended dormancy may result in permanent loss of dependents, even if they later resume development. For users of dormant packages, the data suggest that most dependencies continue to function during gaps; however, critical evaluation of alternatives becomes increasingly important as dormancy periods extend.*

6.3 Qualitative analysis

The qualitative analysis reveals several key insights into the dynamics of dormant package management and revival, offering practical recommendations for maintainers, users, and developers.

Maintainers should recognize the importance of continuous engagement with their projects, even during periods of low activity. As observed in packages like `sax` and `grunt-compile-handlebars`, sporadic yet critical maintenance, such as updating dependencies to address security vulnerabilities or merging pull requests, can prevent the accumulation of technical debt and ensure that packages remain available for future improvement. In contrast, packages like `bitcoin` and `copy-paste` demonstrated that a revival after complete dormancy often requires significant updates. This highlights the importance of adopting a clear fixing strategy that first addresses core dependencies and security issues, and then considers whether a fresh start is necessary. Additionally, proactive communication could help maintainers prioritize issues and set realistic expectations, improving the transition from dormancy to active maintenance and ensuring long-term sustainability.

✚ **Lesson₉.** *Sporadic maintenance, such as addressing security vulnerabilities and merging pull requests, helps keep packages stable. When reviving a package, maintainers should adopt a clear fixing strategy to prioritize issues and manage expectations for smoother revival and long-term sustainability.*

Users play a crucial role in the lifecycle of open-source projects, particularly in prompting the revival of dormant packages. The case of `bitcoin`, in which a new maintainer took over and made a significant revision, demonstrates that external contributions can give new life to a previously inactive project. Similarly, sporadic maintenance activity observed in packages like `sax` and `grunt-compile-handlebars` indicates that reporting issues or pull requests may have influenced maintainers to address critical updates during dormancy. In cases where maintainers are unresponsive, users may consider forking the project or collaborating with the community to sustain it. However, as seen in packages with prolonged inactivity like `copy-paste`, reliance on dormant packages without active maintenance can expose users to risks from outdated dependencies or unpatched vulnerabilities.

✚ **Lesson₁₀.** *Users are vital in sustaining open-source projects by engaging with maintainers and offering contributions. If maintainers are unresponsive, users can fork or collaborate to maintain the project.*

Finally, developers considering the adoption of open-source packages should, therefore, assess their maintenance history, community activity, and responsiveness to updates. If signs of dormancy appear, such as outdated dependencies or unresolved issues, developers should carefully weigh the risks, explore alternative packages, or contribute directly by submitting patches or dependency updates. Moreover, community involvement, as seen in the cases of packages revived in response to user interest or forks, e.g., `spritesheet-js`, can play a crucial role in maintaining package relevance.

📌 **Lesson₁₁.** *Developers should assess a package’s maintenance history before adoption, weighing risks of dormancy. Contributing to dormant projects by updating dependencies, fixing vulnerabilities, and engaging with the community can help ensure continued relevance and security.*

7 Threats to Validity

Given the empirical nature of our study, there are several potential threats to validity that we need to mention. We discussed these threats following the classification and recommendations provided by Wohlin et al. [27].

Construct Validity arises from potential imprecision or incompleteness in our data sources used to identify packages and their associated vulnerabilities. We relied on the open dataset provided by Ecosyste.ms.²⁶ Although we conducted a manual inspection of a sample of packages and did not find missing data, there is no guarantee that the dataset is exhaustive – some packages may be missing. Additionally, the dataset may reference packages that have been removed from the registry.

Another possible limitation comes from our focus exclusively on dormant packages that are used by other packages distributed via npm. Our filtering strategy provides a clean basis for measuring cascading delays within the npm. Still, it may underestimate the broader influence of dormant packages that are used in software projects that are not published on npm. As part of our future work, we plan to extend our analysis to include GitHub projects that depend on dormant packages but are not themselves distributed via npm [30].

Another concern is our reliance on the GitHub vulnerability database, which we assume to be a comprehensive source of vulnerability reports for third-party packages. This may result in underestimations, as some vulnerabilities may not yet be disclosed and thus not included in the database. Still, we computed the number of *known vulnerabilities* before, during, and after a release gap statically. However, this approach does not account for whether the vulnerable functionality is actually used by the package. Due to the dynamic and flexible nature of JavaScript, performing reliable dynamic analysis to trace the usage of vulnerable code is not trivial. As a result, our vulnerability counts may overestimate the actual security risk faced by downstream users.

Internal Validity pertains to factors within the study that could influence the observed results. A primary threat is our method for identifying dormant packages. We defined a release pause as occurring when the gap between releases exceeds one year more than the average gap. This choice, while consistent with prior studies [2, 11], lacks a strong theoretical justification. Additionally, our focus on the first release pause, rather than subsequent pauses, could impact the results. To partially mitigate this threat, we computed the selection of dormant packages with a different gap, i.e., the gap as the average pause between two releases + 18 months. We found 9,114 packages that experienced

²⁶ <https://ecosyste.ms/>

a release gap, of which 76.14% were the same packages analyzed in the initial analysis. This result suggests that our identification of dormant packages is generally stable across different thresholds. Additionally, we corroborated our quantitative results through a qualitative analysis of a subset of packages.

Concerning **RQ₃**, we used a semantic versioning scheme as a heuristic because a previous study has shown its use in the context of npm [22]. Nonetheless, we recognize that not every package adheres to **SemVer**, so further analysis is necessary to validate these inferences through qualitative analysis of commit messages or issue trackers.

Another internal validity threat is that our analysis in *RQ₄* reports percentages of packages modifying specific file types relative to all packages. Still, we lack data on which packages actually contain those file types. Thus, modification rates may reflect frequency across all packages rather than among projects using the files. For example, YAML files were modified in 14.6% of packages, though only some contain YAML. This limitation means our findings indicate modification prevalence but not precise usage-based rates. Addressing this would require detailed project metadata.

Conclusion Validity addresses whether the conclusions drawn from our data analysis are reasonable. Given that our findings are primarily based on empirical observations, we believe this validity threat is minimal. To further corroborate the conclusions drawn in the study, we applied the Pearson and Spearman correlation coefficients [23, 24], and the Mann-Whitney test [18], which allowed us to report our findings from a statistical perspective. Our conclusions about maintenance motivations and decision processes rely on the analysis of public artifacts such as commit messages, pull requests, and issue discussions; therefore, they may not accurately reflect the decisions that drive dormancy, reactivation, or versioning choices. However, conducting such an analysis, for instance, through a survey, is beyond the scope of this paper for two reasons. First, dormant packages account for only 0.34% of the entire npm package manager (11,970 out of 3.5 million modules). While this focus makes our dataset both novel and valuable, it also means that the maintainers who can speak of “release-gap” motivations are few and dispersed. Recruiting and engaging that specific sample would require substantial outreach and tailored incentives, lest we end up surveying mostly active-only maintainers who cannot address dormancy. Second, designing a survey that fairly captures reasons for inactivity, dependency update strategies, and reactivation triggers across projects of wildly different sizes and domains would require a research method groundwork to ensure valid and unbiased questions. As part of our future agenda, we plan to design and conduct a survey with the maintainers of npm dormant packages. Specifically, we will leverage the dataset of 11,970 known gaps to identify and recruit the right contributors to explain the packages’ dormancy, reactivation, and versioning choices.

As for **RQ₁**, we computed our analysis by counting every individual change. Our decision could potentially include “burst” sequences of same-day or next-day patch releases addressing a single defect. While **RQ₂** already breaks down patch vs. minor/major timing, aggregating those rapid patch clusters into sin-

gle “logical” release events could indeed filter out noise and provide a cleaner view of true maintenance rhythms. Unfortunately, such clustering would require a deeper pre-processing step, e.g., defining per-package time windows, identifying sequences of patch-only bumps, and collapsing them into one update, which is not feasible in a timely manner and goes beyond our current scope. Nonetheless, we recognize that the current analysis may overemphasize clustered patch activity. Further study should aggregate the multiple changes performed for the same issue in a single patch to assess how such clustering shifts the median inter-release time and gap detection.

External Validity concerns the extent to which our results and conclusions can be generalized beyond the scope of our study. While our current study focuses exclusively on JavaScript packages (from npm), early signs from other package managers suggest that dormancy is a cross-language phenomenon rather than an npm peculiarity. For instance, the analysis performed by Zerouali et al. [29] on both npm and RubyGems shows that vulnerabilities accumulate rapidly in unmaintained releases across these registries. Still, Zhong et al. [33] highlighted that packages in the PyPI ecosystem often remain non-updated for extended periods. Such findings suggest that prolonged maintenance gaps and the resulting security and trust concerns may be common in modern package management systems. At the same time, each ecosystem has its conventions and tooling that shape how dormancy is detected and mitigated. RubyGems, for instance, exposes commands and transfer mechanisms, enabling community members to adopt and revive abandoned projects. The “security contact” field in package metadata on PyPI can serve as an early warning. npm lacks an analogous feature today, but learning from these examples could help us design registry-level signals, such as required metadata fields or clearer ownership handoff workflows, to reduce long-term abandonment. We can conclude that while our results cannot be generalized beyond npm, the design of our study can easily be replicated for other package distributions.

8 Conclusion

The ultimate goal of our study was to explore the lifecycle and impact of dormant npm packages, with a focus on understanding the risks and challenges posed by prolonged periods of inactivity. Using a comprehensive dataset of dormant packages and dependencies, we conducted quantitative and qualitative investigations to uncover patterns in dormancy, maintenance, and security.

The key findings of our research reveal that dormant packages often accumulate technical debt, outdated dependencies, and security vulnerabilities, posing risks to dependent systems. We found that while some packages successfully revive with patches and minor updates, a significant portion do not fully recover, leaving them vulnerable to future inactivity. Additionally, runtime dependencies tend to be less frequently updated, leading to an increased risk to end-users.

Our investigation into dependent package behavior during dormancy periods revealed that while the majority of packages (81.2%) demonstrate remarkable loyalty by continuing to use dormant dependencies throughout gaps, nearly one in five packages eventually abandon these relationships.

To summarize, our paper has made the following contributions:

1. A detailed investigation into the lifecycle of dormant npm packages, including the timing and duration of release gaps.
2. An analysis of the type and focus of updates following periods of dormancy, emphasizing the challenges of dependency management and security risks.
3. A comprehensive study of the security implications associated with outdated dependencies, highlighting the critical need for proactive dependency management.
4. An examination of dependent package behavioral patterns during dormancy periods, revealing ecosystem resilience mechanisms and the long-term consequences of maintenance gaps on dependency relationships.
5. An online appendix [32] in which we provide all material and scripts employed to address the goals of the study.

The main considerations and conclusions of the study represent the input for our future research agenda. We plan to develop automated tools to help maintainers and users track dormant packages and proactively address potential risks. Additionally, we aim to investigate the long-term impacts of dormancy on software sustainability and explore strategies for community engagement in maintaining widely used, but neglected, packages. In future work, we also plan to investigate and understand the behavioral patterns of dependent packages during dormancy periods, which could inform the development of early warning systems to help maintainers anticipate and mitigate the erosion of their package’s user base.

Another promising direction involves a follow-up study that samples revived releases to conduct diff-based and behavioral analyses, with the goal of validating **SemVer** adherence across a representative subset of packages.

To better understand the human dimension of package dormancy, we plan to conduct a targeted survey of maintainers from different ecosystems, including npm, PyPI, and RubyGems, to investigate the factors influencing dormancy, reactivation, and versioning practices. Finally, we propose to generalize our definitions of release gap and dormant package across multiple ecosystems, enabling a direct comparison of dormancy patterns, recovery strategies, and vulnerability accumulation rates.

Acknowledgement

The authors would like to thank the handling editor and the anonymous reviewers for the insightful comments raised during the review process. These have improved the quality of our manuscript.

Funding

Not applicable. The authors declare that they do not have any funding applicable to the work reported in this paper.

Declaration of Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data Availability Statement

The manuscript includes data as electronic supplementary material. In particular, datasets generated and analyzed during the current study, detailed results, scripts, and additional resources useful for reproducing the study are available as part of our online appendix on Zenodo: <https://doi.org/10.5281/zenodo.14733205>.

Credits

Ahmed Zerouali: Formal analysis, Investigation, Data Curation, Validation, Writing - Original Draft, Visualization. **Valeria Pontillo:** Writing - Review & Editing. **Coen De Roover:** Supervision, Writing - Review & Editing.

References

1. Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? an empirical case study on npm. In *Int'l Symp. Foundations of Software Engineering (FSE)*, pages 385–395. ACM, 2017.
2. Guilherme Avelino, Eleni Constantinou, Marco Tulio Valente, and Alexander Serebrenik. On the abandonment and survival of open source projects: An empirical investigation. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12. IEEE, 2019.
3. Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. When and how to make breaking changes: Policies and practices in 18 open source software ecosystems. *ACM Trans. Softw. Eng. Methodol.*, 30(4), July 2021.
4. Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an API: Cost negotiation and community values in three software ecosystems. In *Int'l Symp. Foundations of Software Engineering (FSE)*, pages 109–120. ACM, 2016.
5. Filipe R Cogo, Gustavo A Oliva, and Ahmed E Hassan. Deprecation of packages and releases in software ecosystems: A case study on npm. *IEEE Transactions on Software Engineering*, 48(7):2208–2223, 2021.
6. Filipe Roseiro Cogo, Gustavo A Oliva, and Ahmed E Hassan. An empirical study of dependency downgrades in the npm ecosystem. *IEEE Transactions on Software Engineering*, 47(11):2457–2470, 2019.

7. Eleni Constantinou and Tom Mens. An empirical comparison of developer retention in the rubygems and npm software ecosystems. *Innovations in Systems and Software Engineering*, 13:101–115, 2017.
8. Joël Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. Measuring dependency freshness in software systems. In *International Conference on Software Engineering*, pages 109–118. IEEE Press, 2015.
9. Alexandre Decan and Tom Mens. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*, 2019.
10. Alexandre Decan, Tom Mens, and Maëllick Claes. An empirical comparison of dependency issues in OSS packaging ecosystems. In *International Conference on Software Analysis, Evolution and Reengineering*, pages 2–12. IEEE, 2017.
11. Robert English and Charles M Schweik. Identifying success and tragedy of floss commons: A preliminary classification of sourceforge. net projects. In *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS’07: ICSE Workshops 2007)*, pages 11–11. IEEE, 2007.
12. Oscar Franco-Bedoya, David Ameller, Dolores Costal, and Xavier Franch. Open source software ecosystems: A systematic mapping. *Information and software technology*, 91:160–185, 2017.
13. Fang Hou and Slinger Jansen. A systematic literature review on trust in the software ecosystem. *Empirical Software Engineering*, 28(1):8, 2023.
14. Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *International Conference on Mining Software Repositories (MSR)*, pages 102–112. IEEE, 2017.
15. R. G. Kula, D. M. German, T. Ishio, and K. Inoue. Trusting a library: A study of the latency to adopt the latest Maven release. In *Int’l Conf. on Software Analysis, Evolution, and Reengineering*, pages 520–524, March 2015.
16. Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, 2017.
17. Fabio Massacci and Ivan Pashchenko. Technical leverage in a software ecosystem: Development opportunities and security risks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1386–1397. IEEE, 2021.
18. Patrick E McKnight and Julius Najab. Mann-whitney u test. *The Corsini encyclopedia of psychology*, pages 1–1, 2010.
19. Courtney Miller, Mahmoud Jahanshahi, Audris Mockus, Bogdan Vasilescu, and Christian Kästner. Understanding the response to open-source dependency abandonment in the npm ecosystem. In *2025 47th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2025.
20. Courtney Miller, Christian Kästner, and Bogdan Vasilescu. “we feel like we’re winging it:” a study on navigating open-source dependency abandonment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1281–1293, 2023.
21. Suhaib Mujahid, Diego Elias Costa, Rabe Abdalkareem, and Emad Shihab. Where to go now? finding alternatives for declining packages in the npm ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1628–1639. IEEE, 2023.
22. Donald Pinckney, Federico Cassano, Arjun Guha, and Jonathan Bell. A large scale analysis of semantic versioning in npm. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 485–497. IEEE, 2023.
23. Philip Sedgwick. Pearson’s correlation coefficient. *Bmj*, 345, 2012.
24. Philip Sedgwick. Spearman’s rank correlation coefficient. *Bmj*, 349, 2014.
25. Alexandros Tsakpinis. Analyzing maintenance activities of software libraries. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 313–318, 2023.
26. Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the JavaScript package ecosystem. In *Int’l Conf. Mining Software Repositories (MSR)*, pages 351–361. IEEE, 2016.
27. C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering - An Introduction*. Kluwer, 2000.

28. Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. An empirical analysis of technical lag in npm package dependencies. In *International Conference on Software Reuse*, pages 95–110. Springer, 2018.
29. Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles, and Jesus M Gonzalez-Barahona. On the impact of outdated and vulnerable JavaScript packages in Docker images. In *International Conference on Software Analysis, Evolution and Reengineering*, pages 619–623. IEEE, 2019.
30. Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. On the impact of security vulnerabilities in the npm and rubygems dependency networks. *Empirical Software Engineering*, 27(5):107, 2022.
31. Ahmed Zerouali, Tom Mens, Jesus Gonzalez-Barahona, Alexandre Decan, Eleni Constantinou, and Gregorio Robles. A formal framework for measuring technical lag in component repositories—and its application to npm. *Journal of Software: Evolution and Process*, 2019.
32. Ahmed Zerouali, Valeria Pontillo, and Coen De Roover. A comprehensive study of the lifecycle and impact of dormant npm packages — online appendix. "<https://doi.org/10.5281/zenodo.14733205>", 2025.
33. Zhiqing Zhong, Shilin He, Haoxuan Wang, Boxi Yu, Haowen Yang, and Pinjia He. An Empirical Study on Package-Level Deprecation in Python Ecosystem . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 66–77, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.
34. Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *USENIX Security Symposium*, pages 995–1010, 2019.